Automatically Generating Complex Test Cases from Simple Ones

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
Konstantin Rubinov

under the supervision of
Prof. Mauro Pezzè

October 2013

Dissertation Committee

Prof. Matthias Hauswirth	Università della Svizzera Italiana, Switzerland
Prof. Mehdi Jazayeri	Università della Svizzera Italiana, Switzerland
Prof. Mark Harman	University College London, United Kingdom
Prof. Gregg Rothermel	University of Nebraska-Lincoln, USA

Dissertation accepted on October 2013

Prof. Mauro Pezzè
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Igor Pivkin
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis 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 thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Konstantin Rubinov
Lugano, October 2013

Abstract

While source code expresses and implements design considerations for software system, test cases capture and represent the domain knowledge of software developer, her assumptions on the implicit and explicit interaction protocols in the system, and the expected behavior of different modules of the system in normal and exceptional conditions. Moreover, test cases capture information about the environment and the data the system operates on. As such, together with the system source code, test cases integrate important system and domain knowledge.

Besides being an important project artifact, test cases embody up to the half the overall software development cost and effort. Software projects produce many test cases of different kind and granularity to thoroughly check the system functionality, aiming to prevent, detect, and remove different types of faults. Simple test cases exercise small parts of the system aiming to detect faults in single modules. More complex integration and system test cases exercise larger parts of the system aiming to detect problems in module interactions and verify the functionality of the system as a whole. Not surprisingly, the test case complexity comes at a cost – developing complex test cases is a laborious and expensive task that is hard to automate.

Our intuition is that important information that is naturally present in test cases can be reused to reduce the effort in generation of new test cases. This thesis develops this intuition and investigates the phenomenon of information reuse among test cases. We first empirically investigated many test cases from real software projects and demonstrated that test cases of different granularity indeed share code fragments and build upon each other. Then we proposed an approach for automatically generating complex test cases by extracting and exploiting information in existing simple ones. In particular, our approach automatically generates integration test cases from unit ones. We implemented our approach in a prototype to evaluate its ability to generate new and useful test cases for real software systems. Our studies show that test cases generated with our approach reveal new interaction faults even in well tested applications. We evaluated the effectiveness of our approach by comparing it with the state of the art test generation techniques. The evaluation results show that our approach is effective, it finds relevant faults differently from other approaches that tend to find different and usually less relevant faults.

Contents

Figures

Tables

Chapter 1

Introduction

This thesis develops an approach to automatically generate complex test cases starting from simple ones. In particular, it automates the generation of integration test cases from unit ones.

Software testing is the dominant practice to check correctness of software. Software testing improves software products by preventing, detecting, and removing faults, while assessing and improving the overall quality of software product.

Most successful software projects produce large amount of test cases manually and automatically. They produce test cases of different kind and granularity, that are essential to thoroughly check the functionality of the system. Simple unit test cases ensure that methods correctly implement the specified and implied pre- and post-conditions. Most commonly, individual unit test cases exercise only small parts of the system, for example, a single class, and check that the state of the module on which the methods are invoked is as expected. In contrast, complex integration and system test cases ensure that modules correctly follow interaction protocols, they test class interactions or complex system behavior. Such test cases exercise larger parts of the system working through long sequences of method calls and checking the state of all involved modules. As such, integration and system test cases are usually more complex, more expensive to develop, and harder to generate and maintain than unit test cases.

Modern software development processes emphasize early testing activities [SB02]. They encourage the early development of unit test cases to test the basic functionality of the system modules. Early testing activities follow test-first practices and are supported by tools for automated test execution [Bec02]. This is why a large amount of unit test cases is often available early in the software development process. In the current industrial practice, complex integration and system test cases are developed after unit ones.

Unit, integration and system test cases are often developed manually. At the same time, software systems become increasingly complex, although more modularized and distributed. The functionality of such systems results from multiple complex module

interactions and requires careful integration and system testing that involve complex scenarios of module integration with a variety of potential interactions and corner cases. Writing test cases manually is difficult and expensive, and both development effort and cost grow with the complexity of the test cases.

A popular approach to reduce the manual effort in designing test cases and reduce cost is *automated test case generation*. Most existing approaches to automated test case generation focus on generating test cases to increase structural code coverage [dPX+06; PLEB07; ZSBE11; BBDP11; TXT+09b; TXT+11]. These approaches for automatic test case generation tend to generate rather simple test cases or parts of test cases. Some approaches focus on automatic generation of test inputs, others on execution scenarios or test oracles. While generating simple test cases is a rather simple process that can often be automated, generating complex test cases is a laborious process that is hard to automate.

This thesis inspires from the practice of *software reuse* – a process of creating software systems from existing software rather than building software systems from scratch. Software reuse has been widely studied and applied helping developers to reduce software development effort [Kru92]. By analogy with software reuse, we aim to reduce software development effort and cost by reusing information from existing test cases to generate new test cases.

In this thesis we propose a technique to automatically generate complex test cases starting from simple ones. We start from the observation that by looking at test case implementation, for example, *JUnit* code, one can notice that complex test cases share a lot of code with simple test cases. In particular, integration test cases share code with unit test cases.

Not surprisingly, developers reuse acquired domain knowledge to evolve the software system. Test suites evolve together with the software in the same way as software evolves – building on available information [HO08; XR09; MPP12]. Simple test cases are developed first to test single modules, then software is incrementally changed and integrated, and test suites are augmented with more complex test cases. These simple and complex test cases share information about the system under test and build on it and on each other.

Starting from this observation we develop an approach to leverage a large amount of unit test cases to generate new integration test cases automatically. We aim to extract meaningful information from unit test cases and use it to drive the generation of more complex ones.

Our approach derives from our investigation of test case structure and interrelation of many real-life test cases of different granularity. The goal of our investigation is twofold. First, it supports our observation that test cases share information and build upon each other. Second, it investigates the test case structure to enable test case analysis and extraction of important information available in test cases.

This investigation lays foundation to our approach. We identify the relevant pieces of information in the test cases (class instantiation and initialization, method call sequences and arguments, use of return values) and create more complex test cases suitably assembling these fragments. We first identify class dependencies within the system. We then compute the data flow information within the input test cases, and use this information to segment the test cases into useful fragments (initialization and execution). Finally, we generate new more complex test cases from the fragments extracted from unit test cases using the class dependence and data flow information.

The approach aims to drive test generation towards complex module interactions. It can serve to generate fresh integration test suites or to augment test suites with behaviors that are not yet present in the integration test suites.

1.1 Research hypothesis and contributions

The main **research hypothesis** of this thesis is:

> *Test cases contain important and meaningful information. This information can be automatically captured and exploited to construct new more complex test cases.*

The first part of the hypothesis describes the main intuition of this thesis that test cases build on domain knowledge and contain important information about the system under test. Recent research suggests that test cases contain important information: A number of techniques have analyzed test cases for system comprehension [QODL10], creating formal software behavior models [TEL11], automating test case repair [DGM10; MPP12], and others.

The second part of the hypothesis states that it is possible to capture and exploit information in test cases to generate more complex ones. There is little work that exploited information in test cases for automating test case generation and test suite augmentation [Ton04; JED08; XR09; BPdM09], and to the best of our knowledge, none of these techniques have exploited information in test cases to generate more complex ones.

This thesis explores several directions in the area of software testing and analysis: analysis of software artifacts, program analysis, and automatic test case generation. The two **main contributions** of the thesis are:

1. *A study of the relations between simple and complex test cases that discovered important correlations.* We experimentally investigate the structure of test cases and test case complexity to characterize simple and complex test cases. We use test case characterization to investigate the phenomenon of code reuse and information sharing between simple and complex test cases. We demonstrate that

different fragments of test cases can be reused and build upon to construct new test cases.

2. *An approach to exploit identified correlations to automatically generate new complex test cases.* We propose a general approach to identify relevant pieces of information in the test cases and generate more complex test cases using these fragments. The approach aims to find new faults related to integration problems and complex usage patterns. We evaluate our approach through a prototype implementation *Fusion* that we apply on a number of open-source software projects in *Java* with available test suites. The approach generates test cases that can find relevant faults even in well tested applications. A comparison with state of the art test case generation techniques shows that our approach is effective and it finds different kinds of faults and is thus complementary to other approaches.

By tackling the problem of automatically generating complex test cases this thesis makes contributions in the area of software testing and analysis that aid design, maintenance and evolution of quality software systems.

1.2 Scope of research

Our approach is general and explores the possibility to move up between testing levels by generating more complex test cases from simpler ones. To develop our approach we focus on generation of integration test cases from unit ones. This choice is motivated by several factors.

First, as we discussed in the introductory section, software projects that follow modern software development processes produce large amount of unit test cases. Second, integration testing level (adjacent to unit level) is important for early detection of integration faults. At the same time, existing approaches for automated generation of integration test cases are few, and often rely on information extracted from the system execution or the system code to generate test cases [MOP02; YX06; ECDJ09]. Relying on system executions to monitor the behavior of a software product at the system level can capture both correct and faulty behaviors as pointed out by Xie [Xie09], while approaches relying on source code analysis suffer from the limitations of the code-based techniques used for the underlying analyses, data flow analysis and symbolic execution [PV09]. Differently from existing approaches we do not rely on system executions nor expensive code analyses, and we use test cases and the domain knowledge they encapsulate to generate new test cases.

Definitions of unit, integration and system testing vary in the literature. Definitions also change depending on the program domain and the software development model. The many interpretations of testing terms cause some confusion in determining the precise frontier between different testing levels, and unit and integration testing levels in particular. We investigate this issue in our study of test case structure and test case

interrelation. We show that in practice test cases spread over a continuum of test case complexity and we develop a taxonomy for simple and complex test cases based on real examples of unit and integration test cases.

In this thesis we refer interchangeably to *unit* and *simple* test cases, as well as *integration* and *complex* test cases.

Our approach targets the development and maintenance phases of the software development process. To be applicable, the approach requires test cases and application source code. We expect an initial set of test cases to be generated by developers manually or automatically. With this prerequisite, our approach automatically generates new more complex test cases.

We describe the approach referring to object-oriented software. The approach uses software written in *Java* and test cases in *JUnit* format[1] as proof of concept; however, our approach is applicable in general to other typed languages and test case formats. The approach works with the Abstract Syntax Tree (AST) representation of the source code and test source code, and does not require the source code of the dependent libraries of the system under test.

We evaluated the approach on open-source software. We did not target specific application domains and our approach should be generally applicable for functional testing of applications from different domains. Being general, our approach does not address particular challenges posed by characteristics of the specific application domain. For instance, it may not be suited for dealing with specific memory constraints in embedded systems, interaction protocols in Web applications, or special integration frameworks in large-scale distributed systems.

Moreover, specific domains introduce different classes of faults. The approach is primary concerned with verifying functional behavior. We do not aim to verify non-functional system properties such as resource consumption, response time, and security.

Currently we do not analyze oracle information. We focus on generation of instantiation and execution parts of test cases. In the experimental evaluation of our approach we use exceptions to detect failures. We discuss the implications of this choice in Chapters 5 and 6.

1.3 Structure of the dissertation

The remainder of this dissertation is organized as follows:

- Chapter 2 overviews the state of the art techniques for automating test case generation. It identifies and discusses the main classes of techniques, namely,

[1] *JUnit* – state of the art test automation framework for *Java* applications `http://www.junit.org`

specification and model-based testing, random testing, search based test generation, approaches relying on symbolic and concolic execution, and hybrid techniques. Chapter 2 presents also the techniques applicable in the context of software evolution, and discusses the approaches that reuse information from existing test cases in test generation process.

- Chapter 3 focuses on the investigation of test case structure and information sharing between simple and complex test cases. It illustrates the composition of test cases in real software projects and describes the indicators of test case complexity that we develop. The chapter characterizes test cases of different granularity and describes the effects of the phenomenon of testing reuse.

- Chapter 4 introduces the approach for automatically generating complex test cases from simple ones. In particular, it explains the underlying techniques for analyzing and extracting information from source code and test cases, and describes the test case generation process. The chapter presents an example that illustrates the step-wise application of the approach.

- Chapter 5 describes the prototype implementation that we used to evaluate our approach. It presents the results of the empirical evaluation and comparison of our approach with the state of the art techniques that demonstrates the effectiveness of the approach.

- Chapter 6 summarizes the contributions of this dissertation, and discusses the future research directions opened by this work.

Chapter 2

State of the Art

Automating test case generation is a widely explored approach to reduce testing costs and improve software quality. This chapter overviews techniques to automatically generate complete test cases from simpler ones. We start by describing the state of the art approaches to generate test cases. We then survey techniques applicable in the context of software evolution and present the challenges that emerge in this context. Finally, we focus our discussion on the approaches that share with ours the idea of reusing information from existing test cases to generate new test cases.

This thesis explores automatic generation of test cases both to reduce the cost of software development process and increase the quality of the software. The thesis shares its goal with a number of approaches that aim to augment test suites with automatically generated test cases. Differently from many state of the art approaches the technique proposed in this thesis leverages information available in existing test cases and it does not rely on software specifications, nor expensive code analyses.

The main research related to this work lies in the area of automating test case generation. In particular, related approaches tackle the problem of test case generation in the absence of specifications. These include random testing, search based test generation, symbolic and concolic execution based approaches, and hybrid approaches. We also briefly survey the work on specification and model based testing that it is loosely related to our work. We overview related work in the area of test suite augmentation and test suite evolution.

Finally, because this thesis inspires from a practice of software reuse, testing reuse for automating test case generation is an important part of related work. We discuss test generation approaches that apply testing reuse, and share with our approach the idea of leveraging existing test cases.

2.1 Automating test case generation

Software projects rely on software testing to verify software and improve its quality. A large body of work reduces the cost of expensive testing activities through automation. In particular, automating test case generation is an active research topic that produces many important results [Har00; Ber07; PY07; ABC+13].

Different techniques automate the generation of complete test cases, while other focus on automatic generation of parts of test cases like test inputs, execution scenarios, or test oracles.

Specification and model based testing

Formal specifications and models can be used as the basis for automating parts of the testing process [BJK+05; PY07; HBB+09]. A considerable amount of work automates test case generation from formal and informal specifications [HBB+09; MSK09; BKM02]. Various formal specifications have been proposed for generating test inputs, execution scenarios and test oracles, for instance, descriptions of test suites from Z specifications [SC96], test sequences from finite state-based languages [GGSV02; HKU02; GOC06], test data and oracles from algebraic specifications [DF94].

Some of these techniques are very powerful, but test case specifications drawn automatically from system specifications are often not well connected with the final code. Automating test generation from formal models and specifications brings its own issues. For instance, for many systems detailed finite state machines cannot be produced due to the large state space. Reducing the state space through abstraction may lead to inconsistency between feasible paths in the finite state machines and the program they model. In addition, automating test generation from model-based languages requires powerful analysis tools [HBB+09].

Well-designed informal specifications suitable for automatic test generation are seldom available. Defining formal specifications for test generation is a time-consuming and demanding activity. Recent work addressed this issue by designing user-friendly specifications suitable for automating test case generation. For instance, Gligoric et al. proposed a Java-based specification language for automating test generation, however limited to the generation of test inputs [GGJ+10].

In practice, availability of specifications varies depending on the software development process and the type of software system being developed. For instance, agile software development processes produce little documentation and consider existing test cases as system specifications [Bec00; SB02]. Heterogeneity of specifications hinders application of some specification-based approaches.

Random testing

Random testing is a viable solution for producing a large number of test cases automatically [CMWE04; CS04; GKS05; dPX+06; CLOM07; PLEB07]. This type of testing may compensate for the absence of domain knowledge and specifications in generating test cases. Along with valid and useful test cases, random testing produces invalid test cases and test suites that unevenly sample the input domain. Recent research mitigates these issues by improving the effectiveness of random testing in several ways.

A family of feedback-directed approaches improves random testing by directing test generation process [AEK+06; PLEB07; REP+11]. The input space is explored with randomly generated class instantiation and method call sequences. Valid sequences are built incrementally, while intermediate sequences are selected after executing and filtering them. Execution provides intermediate feedback in test generation process and guides generation of valid method invocation sequences that do not raise unexpected exceptions. The original technique is implemented in tools like *Palulu* [AEK+06] and *Randoop* working with Java [PLEB07] and .Net [PLB08].

Adaptive random testing (ART) directs test case generation to cover contiguous failure input regions [CMWE04; CKMT10]. The underlying idea stems from empirical studies that indicate that failure-causing inputs tend to form contiguous failure regions. ART aims to generate test cases that evenly spread across the input domain and thus enhance the failure detection effectiveness of random testing.

ART approaches define distance metrics that allow to select the best candidate test cases from an initial set of randomly generated inputs. Chen et al. developed distance metrics for numerical data and demonstrated the effectiveness of the approach for small numerical programs [CMWE04]. Ciupa et al. defined distance metrics for object oriented software [CLOM08]. The distance is defined between the direct values of the objects, between the types of the objects, and recursive distances between the fields of the objects. The approach selects as inputs objects that have the highest average distance to those already used as test inputs. A further improvement of the approach guides object selection strategy to select objects satisfying method preconditions [WGMO10]. Despite a number of improvements, a general limitation of ART approaches relates to their low scalability and high computational costs.

Random testing is fairly successful in detecting certain types of faults, but is ineffective in generating valid method call sequences when facing a large search space of possible sequences due to its random nature. Random testing remains a viable solution for test case generation in the absence of specifications. However, lack of knowledge about a system under test makes random testing approaches prone to produce a large amount of false positives [dPX+06].

Search based test generation

Software testing research is a predominant application area of search based approaches in software engineering [HMZ12]. Search based testing (SBT) applies search-based optimization to the problem of automating test data generation seeking cost-effective solutions for combinatorial problems [McM04; Har07; HM10]. SBT approaches automatically generate test data guided by adequacy criteria that are encoded as fitness functions. The search process aims to generate test data that maximize the number of program structures executed, for instance, maximizing branch coverage.

Search based techniques applied for test generation mostly use hill climbing, simulated annealing, and evolutionary/genetic algorithms [McM04]. The algorithms vary in their strategies and methods to explore the solution search space, avoid sub-optimal results, and seek globally optimal solutions.

A large body of SBT work explores different solutions [MMS01; MHBT06; AH11; KHS11; BHH+11; LHG13]. The approaches mostly generate test inputs. Here we survey the work that is closest to ours with respect to generated data. In particular, we are interested in approaches that produce test data for complex data structures and test scenarios as method invocation sequences [Ton04; FA11].

Tonella proposed a test case generation technique *eToc* based on genetic algorithm to automatically produce test cases for the unit testing of classes [Ton04]. *eToc* implements a more sophisticated definition of the individuals' chromosome than the ones used for generating test cases for procedural programs. Chromosomes encode sequences of statements for class instantiations and method invocations including sequences of operations to be performed and the associated parameter values. The technique is applicable for test case generation for single classes aiming to maximize branch coverage.

An search based technique *EvoSuite* by Fraser and Arcuri improves over *eToc* in several respects [FA11; FA12]. *EvoSuite* test suite generation is based on evolutionary algorithms and mutation-based oracle generation. The test suite generation focuses on an entire coverage criterion, instead of individual coverage goals. Test cases are mutated by adding, deleting, or changing individual statements and parameters. To help the generation of appropriate input data, *EvoSuite* uses focused local searches and dynamic symbolic execution. *EvoSuite* generates oracles using mutation testing. It executes the generated test cases on original and mutated programs and calculates a reduced set of assertions that is sufficient to kill all the mutants. These assertions represent suggested and potential oracles for generated test cases, and are relevant in the context of regression testing, because they capture information from the program source code.

Symbolic and concolic execution

Symbolic execution is a method for characterizing program behavior through symbolic expressions in the form of predicates on program paths and program states [Kin76]. Symbolic execution abstracts over potentially infinite set of normal program executions. It has been proposed for many activities, for instance, in verification techniques to build operational models of the software [CPDGP01], and in test generation techniques to derive constraints on test input data [PV09]. The constraints are solved with a constraint solver to generate test inputs enabling systematic exploration of target program paths. Examples of approaches applicable for data structures in object-oriented code include *Java PathFinder*, *Symstra* and *Symclat* [VPK04; XMSN05; dPX+06]. Despite recent advances, constraint solving still represents a major bottleneck for scalability. The size of constraints grows quickly with the complexity of the program under analysis impacting on the performance of constraint solving.

Dynamic symbolic execution, also known as *concolic* execution, combines symbolic and concrete program executions to overcome some of the limitations of the former one, such as availability of decision procedures and handling calls to native libraries [GKS05; SMA05; TDH08; ADTP10]. Original concolic approach by Godefroid et al. executes a program on random inputs and at the same time collects the path constraints along the executed path [GKS05]. These path constraints are then used to compute new inputs that drive the program along alternative paths. More precisely, branch conditions are negated to guide the test generation process towards executing alternative branches.

Recent work by Artzi et al. uses concolic execution to aid fault localization by generating test cases with execution characteristics similar to the given fault-exposing execution [ADTP10]. The work develops similarity criterion to measure the similarity of the executions of two test cases. This criterion is then used to direct concolic execution towards generating test cases whose execution characteristics are similar to those of a given failing execution. The approach generates minimal test suites effective in localizing faults compared to concolic approaches.

Latest research attempts to improve symbolic and concolic execution. For example, Kim et al. develop a distributed concolic testing framework to scale concolic execution enabling distributed test case generation [KKR12]. Denaro et al. improve symbolic execution by built-in term rewriting and constrained lazy initialization [BDP13]. The improvement allows efficient verification of programs with complex data structures and helps to avoid exploration of invalid traces.

Recent research widely applies symbolic and concolic execution in combination with other approaches. In the following section we discuss hybrid approaches that improve original test case generation and analysis techniques by combining them in a mutually beneficial manner.

Hybrid approaches

A wide range of approaches for automating test case generation combine different analysis techniques. Here we describe in some detail the most representative approaches that share with ours a minimal set of requirements for being applied.

MSeqGen by Thummalapenta et al. statically mines sequences of method calls from code bases [TXT+09b] and assists other automatic test generation approaches such as *Randoop* and *Pex* [TDH08] by providing them with method call sequences for instantiating complex objects. Extracting sequences from code bases limits *MSeqGen*, because in practice code bases exist only for few specific types of applications, and code bases are often incomplete.

Seeker by Thummalapenta et al. synthesizes method sequences combining static and dynamic analyses to achieve desired object states and thus to cover branches and intra-class DU pairs [TXT+11]. *Seeker* performs dynamic symbolic execution and combines it with the results of static analysis on method-call graphs.

The test generation approach by Martena et al. aims to generate integration test cases automatically [MOP02]. They generate complete test cases by deriving a set of test case specifications for inter-class testing through data-flow analysis, and automatically generate test cases that satisfy the derived specifications using symbolic execution and automated deduction.

Andrews et al. [AML11] propose a hybrid approach using genetic algorithm to find good parameters for randomized unit testing that optimize test coverage of a randomly generated test suite. This way the approach aims to reduce the size of randomized test suite, while achieving the same coverage. The approach has been shown to perform well on small studies, however it has not been applied to larger systems yet.

Hybrid approach by Baluda et al. integrates test case generation and infeasibility analysis to improve structural code coverage [BBDP11]. The approach combines concolic execution with abstraction refinement and coarsening technique. It identifies infeasible branches that can be eliminated from the computation of the branch coverage and it steers the generation of new test cases toward uncovered branches. When it does not find a test case that covers a target branch, it investigates the feasibility of the branch using an analysis based on abstraction refinement of the control flow graph and backward propagation of preconditions for executing the uncovered branches. Although evaluated on the programs of limited size, the approach produces test suites that cover all feasible branches and correctly identifies all infeasible ones, thus reporting a 100% branch coverage of the code in most cases.

RecGen and *Palus* improve the effectiveness of random testing with static and dynamic analyses aiming to achieve higher structural coverage [ZZLX10; ZSBE11]. The approaches share the static analysis step to identify intra-class method dependences that derive from accesses to common fields. Differently from *RecGen*, *Palus* uses a dynamic step and generates more complex test cases extending sequences of methods calls captured from system traces with new dependent method calls [ZSBE11]. In the

dynamic step, *Palus* traces system executions, captures sequences of method calls from the traces, and generalizes them in the call sequence model. It then extends these sequences with new calls to methods whose invocation depends on methods already in the traces, and generates test cases from the extended sequences by means of directed random test generation.

Despite suffering from the scalability issues due to the code-based techniques used for the underlying analyses, such as data flow analysis and symbolic execution, many hybrid approaches largely improve over original techniques.

2.2 Test suite evolution

A number of approaches automate test case generation in the context of software evolution. In this context, evolving software not only needs to be retested after changes, but test suites have to be augmented and optimized to reflect the changes in software; obsolete test cases have to be repaired and maintained [HO08].

In the context of software evolution automatic test case generation aims to produce test cases that can stress the effect of a given program change [QRL10]. Many test suite augmentation techniques characterize the introduced changes by relying on source code information to identify differences between versions of software. These approaches identify the impact of changes using program dependence analysis and symbolic execution combined with constraint solving [TXT+09a; QRL10; XKK+10]. Slicing on program dependence graphs is used to identify data and control dependencies of the code affected by changes that require testing. Symbolic execution and constraint solving generate test inputs that reach and execute identified code fragments.

Different granularity of changes represents a challenge for test augmentation and test adaptation techniques. These techniques tackle simple changes and consider them in isolation. However, some changes cannot be tested in isolation and lead to incorrect program configurations [Zel99; RST+04].

Combinations of multiple syntactic program changes may have different impact on program entities and program output. While change impact analysis for individual changes is well studied, multiple change impact is still a challenge for test augmentation approaches [QRL10]. When multiple changes are considered new challenges arise related to change interactions and change inter-dependence. Combined such changes may impact program output differently than the individual changes. To determine whether changes are inter-dependent program semantics need to be examined. Moreover, new methods are needed to detect feasible change combinations to be covered. Generated test cases should propagate the effects of changes to the program output and thus make their semantic effects observable [TXT+09a; QRL10]. This remains an open area of research [SHO10].

A currently active research area of *Mining Software Repositories* (MSR) offers systematized information for decision-making processes in software development. The

data is used for making predictions, finding commonly occurring patterns, finding data instances for given patterns, grouping data into clusters, and predicting labels of data based on already-labeled data [Has08; XTLL09; HX10].

Recent work in software evolution leveraged MSR for change characterization. In particular, MSR has been applied for change impact analysis and for collecting bug relation information [RST+04; ZZWD05], and has been used to collect information about patterns of changes in software [BMZ+05; KCM07; KR11]. However, to date such information has not been used for software testing.

The challenge for most of the current approaches for test suite evolution is to find a good compromise between expensive analyses and formality. Scalability of symbolic execution is a major problem [PV09]. This problem is reflected in studies applying symbolic execution that are generally performed on small subjects and investigate simple changes in software. Such studies may be not representative. At the same time approaches that do not rely on formal methods lack soundness and precision.

2.3 Reuse in automating software testing

Reuse in automating software testing, or *testing reuse* as we refer to it in this thesis, is a phenomenon similar to *software reuse*. Software reuse has been widely studied and applied helping developers to reduce software development effort [Kru92]. Despite substantial progress in the area, reuse in automating software testing has attracted much less attention from research community [TG13].

Previous studies mostly focussed on identifying and reducing reuse in test cases. The main motivation behind these work came from the need to satisfy time constraints on test execution for ever-growing test suites for evolving software. Since test suites are often extended by copying, modifying and reusing parts of existing test cases, code reuse across test cases may lead to overlap in functionality that test cases execute. Many techniques have been proposed to eliminate possible drawbacks of code reuse by identifying and removing redundant and repeating test cases and to reduce test suites for regression testing [RH96; GHK+01; RH97].

Testing reuse pursues the same goals as software reuse does in software development: to reduce effort in software development and maintenance. Reuse can be employed in different testing activities including test case generation, test case execution, test report generation and analysis [QODL10; TEL11; PG12; CGS13]. In this section we focus on testing reuse for test case generation, when domain knowledge in existing test cases is build upon to create new test suites. This kind of reuse is closely related to our work.

The process of reusing parts of test cases can help automating test case generation. The idea of reusing parts of test cases can be traced back to the work of Leung and White [LW90]. Leung and White proposed to reuse parts of existing unit test cases to augment test suites for modified modules for regression testing.

Recent research investigates the possibility of reuse of testing artifacts in test case generation [Ton04; KRH+08; XR09; BPdM09; JED08; MPP12]. Some techniques reuse testing artifacts through abstracting on test input data [TDH08; FZ11b] and abstracting on various configurations for software product line testing [PM06; KM06; DK06; RMP07; CCR10].

Testing reuse underlies many search based approaches. An approach by Tonella uses randomly generated set of test cases described as chromosomes to mutate them and maximize given coverage measure [Ton04]. Test cases associated to each chromosome are executed using the instrumented version of the system to determine the targets (branches) covered by each individual. Fraser and Zeller propose an evolutionary approach to improve the readability of automatically generated test cases [FZ11a]. The approach collects object usage information from Java bytecode of the system under test and available test cases, and evolves randomly selected portions of observed sequences of method calls to shorten them without altering the overall coverage. The search-based approach by Yoo and Harman reuses and regenerates existing test data for primitive data types [YH10]. Yoo and Harman reuse test input data to generate additional test input data applying meta-heuristic search algorithm and executing test cases on the instrumented program to evaluate the fitness of a candidate solution.

Recent research in test suite augmentation leverages existing test data in different ways. Xu and Rothermel use test cases to determine the program changes (branches) not covered during test execution [XR09]. Authors use test executions to identify uncovered changes and drive concolic execution to cover remaining branches by negating the path conditions. The subsequent work empirically studied factors affecting effective and efficient test case reuse for test case generation of test data-based techniques based on concolic and genetic approaches and their combination [XKK+10; XCR10; XKKR11]. Mirzaaghaei et al. study relations of system source code and test code to automate test case repair and test generation for sibling classes [MPP12]. They analyze compilation errors in test code and apply data-flow analysis on test cases to identify argument values suitable for repairing test cases. This way they leverage data from existing test suites to augment them with new repaired test cases and new test cases for sibling classes.

Jorge et al. reuse system test executions to derive test cases for differential unit testing. Their approach captures parts of the system test execution related to individual methods and program states into abstract unit tests that they later aggregate into tests of a coarser granularity [JED08]. These new tests replicate larger parts of system execution and serve to detect regression faults based on the behavior differences between versions of software. The data from system test execution is “carved” by instrumenting the application to capture, serialize, and store run-time program states [ECDD06]. Although this technique could be extended to generate new test cases with method sequences not observed in system tests, so far it has only been proposed to replay the observed behavior.

Mariani et al. propose *AutoBlackTest*, an approach to generate test cases at the system level [MPRS11; MPRS12]. Differently from the approach by Jorge et al., they produce system executions with Q-learning agent. The agent incrementally executes the system recording the system responce and calculates the cumulative "reward" measure based on the amount of triggered system state changes and relevant computations. The approach automatically generates test cases that combine actions with high cumulative reward and thus exercise relevant portion of the statements in the target applications.

Testing reuse has been applied in the context of GUI testing. Atoms Framework technique by Bertolini et al. uses small fragments of existing GUI test cases and data observed in these test cases to generate new test cases that explore the state space of application GUI and detect GUI crashes [BPdM09]. The test case fragments are selected manually, while the data provided to these fragments is selected randomly from the pool of data observed in existing and in newly generated test cases.

Automated test case generation, test suite augmentation and test suite evolution are active research topics that establish a sound foundation for future research. Many techniques focus on generation of simple test cases. Hybrid techniques gradually improve the state of the art providing solutions to generate more complex test cases turning to sophisticated analyses and new sources of information.

This dissertation tackles a problem of automating generation of complex test cases during development and maintenance, when some test cases are available. Differently from existing techniques, the proposed approach leverages test cases in a new way enabling the generation of complex test cases automatically.

Chapter 3

Test Case Interrelation

Test cases of different kind and granularity are essential to test software systems thoroughly. In this chapter we empirically investigate the structure and complexity of test cases of different granularity. This is important, because existing methods for characterizing test cases do not represent well their structure hindering test code comprehension and analysis. Based on the results of the investigation, we develop a taxonomy for characterizing test cases and argue that simple and complex test cases are interrelated and share important information. We propose to exploit the interrelation between test cases of different granularity for test case generation.

In this thesis we argue that some information in test cases can be beneficially reused to generate new test cases automatically in the same way as test suites evolve by copying, modifying and reusing parts of existing test cases. We aim to identify such information and develop an approach to use this information to generate new complex test cases.

In this chapter we discuss two main issues associated with the relation among test cases: (1) defining a taxonomy of test cases and (2) understanding information sharing and reuse among test cases. With respect to the first issue, we investigate test case structure and test case complexity to characterize simple and complex test cases. We rely on test case characterization to investigate the second issue, that is, how simple and complex test cases share information. By investigating these issues we aim to support our hypothesis that important information in test cases can be identified and reused, and thus lay a foundation for our approach to automatically generating complex test cases from simple ones.

Commonly test cases are differentiated by the goal of the testing phase they belong to [PY07]. For instance, unit test cases check module functionality against specifications or expectations with isolated and focussed test cases; integration test cases check module compatibility and module interactions; and system test cases verify functionality of the integrated parts of the system with respect to specifications and user needs.

In this chapter we show that such characterization of test cases is insufficient to differentiate them automatically. We also show that characterizing the complexity of test cases based on traditional intrinsic code complexity measures does not aid automatic differentiation either. We present our analysis of test case structure that allows us to define test case complexity through indicators based on the amount of system entities used in the test cases.

We study the amount of code reuse between test cases to support our hypothesis that test cases often build upon the knowledge from existing test cases. We measure how much information test cases share, in particular, we are interested in distinguishing the kinds of information shared among test cases. We use our measures of complexity of test cases to understand the interrelation between simple and complex test cases, and we exploit it for test case generation.

In the following sections we characterize test cases empirically aiming to define and detect meaningful information encapsulated in test cases (Sections 3.1 and 3.2). In Section 3.3 we describe our investigation of the complexity of real-life test cases. Based on this investigation we characterize simple and complex test cases. We show how test cases of different granularity spread along the complexity scale. In Section 3.4 we investigate test cases from real-life software systems to understand the strategies and amount of reuse in real test cases.

We conclude the chapter by discussing the important information available in test cases and we present a core idea for automatically generating complex (integration) test cases from simple (unit) ones that stems from our intuition and the results of our investigation.

Empirical framework

We draw our intuition from the state of the practice. Our empirical framework is based on static analysis and inspection. We analyze real-life test suites and we focus on test cases that are built during the software development phase. We investigate the structure of test cases, the program entities test cases are build from, and the common test case organization. We use these data to identify meaningful components of test cases that are essential for generating new test cases.

Our investigation includes more than thirty different software projects. In this chapter we report experimental data for a subset of the most representative projects that belong to distinct program domains. Our analysis encompasses 7041 *JUnit* test cases[1] from seven small and medium size software systems developed in *Java* (Table 3.1):

- *TestabilityExplorer*[2], a source code analyzer;

[1] We refer to a *test case* as an individual test method in *JUnit* format. In *JUnit* test methods are structurally aggregated in test classes with shared scaffolding and setup procedures.

[2] `http://code.google.com/p/testabilityexplorer/`

Program (version)	*LOC*	*unit test cases, LOC*	*test coverage (line - branch)*
TestabilityExplorer (1.3.2)	8214	5596	81% - 66%
JGraphT (0.8.3)	12207	5637	70% - 63%
Apache Ant (1.8.4)	104307	24384	48% - 42%
JFreeChart (1.0.14)	93460	49644	56% - 46%
JodaTime (2.1)	27213	51715	60% - 55%
PMD (5.0)	60060	13922	58% - 45%
PureMVC (1.1)	666	706	53% - 69%

Table 3.1. Characteristics of subject programs

- *JGraphT*[3], a library that supports graph theory;
- *Apache Ant*[4], a Java library and a state of the art build tool;
- *JFreeChart*[5], a professional chart generation library;
- *JodaTime*[6], a calendar system library;
- *PMD*[7], a source code analyzer;
- *PureMVC*[8], a lightweight framework for creating applications based on the Model View Controller pattern.

3.1 Test entities

Our analysis of unit and integration test cases indicates that both types of test cases operate on the same program entities. We have inspected a large number of unit and integration test cases from the selected case studies and we have observed that both kinds of test cases use method calls as atoms to construct test cases from. For example, consider unit and more complex integration test cases from *PureMVC* shown in Figure 3.1. Both unit and integration test cases instantiate classes, invoke class methods and verify the states of classes under test and dependent classes.

Unit test case `testGetInstance` in the figure exercises small part of the system (single method `getInstance()` in this example) and checks that the return value

[3] `http://jgrapht.org/`
[4] `http://ant.apache.org/`
[5] `http://jfree.org/`
[6] `http://joda-time.sourceforge.net/`
[7] `http://pmd.sourceforge.net/`
[8] `http://puremvc.org/`

```
// Simple unit test case
public void testGetInstance() {
  // Test Factory Method
  IFacade facade = Facade.getInstance();
  // test assertions
  assertNotNull("Expecting instance not null", facade);
  assertNotNull("Expecting instance implements IFacade", (IFacade) facade);
}
```

```
// More complex integration test case
public void testRegisterAndRemoveProxy() {
  // register a proxy, remove it, then try to retrieve it
  IFacade facade = Facade.getInstance();
  IProxy proxy = new Proxy("sizes", new String[] { "7", "13", "21" });
  facade.registerProxy(proxy);
  // remove the proxy
  IProxy removedProxy = facade.removeProxy("sizes");
  // assert that we removed the appropriate proxy
  assertEquals("Expecting removedProxy.getProxyName() == 'sizes'", removedProxy.getProxyName
     (), "sizes");
  // make sure we can no longer retrieve the proxy from the model
  proxy = facade.retrieveProxy("sizes");
  // test assertions
  assertNull("Expecting proxy is null", proxy);
}
```

Figure 3.1. Unit and integration test cases for the `Facade` component in *PureMVC*

and the state of the class on which the method was invoked are as expected. More complex integration test case `testRegisterAndRemoveProxy` exercises larger part of the system instantiating several classes `IFacade` and `IProxy`, and executing longer method call sequence. Both test cases are composed of homogeneous fragments with similar constructors, method invocations and assertions. This observation witnesses our experience on test suites for over thirty different software applications.

3.2 Test case structure

In principle, test cases on all abstraction levels, from unit to integration and system testing, have a common structure. In the initial study we identified four test cases parts: (1) test scaffolding, (2) setup of interacting components with a correct data, (3) execution of the functionality under test, and (4) oracles to verify the mutual state of interacting components.

Our investigation indicates that unit test cases use little or no scaffolding and usually have only *three parts*: test initialization, which corresponds to the mutual setup of objects, unit execution and verification of the results through assertions that serve as oracles. Unit test cases often use mock objects to simplify the setup procedure and focus testing efforts on a single class under test.

In contrast, integration test cases often require instantiating real objects rather than mock objects and more complex interactions between them. Integration *test scaffolding* requires external dependencies to be resolved and the environment to be set up for interacting objects. The *setup* of interacting objects requires information about object dependencies and the data for their initialization and wiring. The *execution order* for interacting objects must represent meaningful combinations of actions on integrated objects. The *oracle* for the mutual state of interacting objects requires considering the state transitions of these objects.

Our study of test cases shows that despite clear logical differentiation of test case parts each having its own purpose, it is difficult to distinguish them in practice. First of all, test cases operate on the same entities and parts of test cases invoke class constructors and methods in scaffolding, setup, execution, and oracle parts. Consider, for example, test case `testRegisterAndRemoveProxy` in Figure 3.1. It contains setup, execution and oracle parts in the context of a single test method. It is not discernible whether method invocation `facade.registerProxy(proxy)` in line 6 belongs to the setup or execution parts of the test case.

The oracle part is easier to distinguish because it is often associated with `assert()` methods. Nevertheless, some method invocations may belong to the oracle part providing access to class states to be verified and can be confused with method invocations belonging to the test execution part. Figure 3.1 contains an example of such situation. The statement in line 12 may belong to both test execution and test oracle parts.

Our observations have been also confirmed in a recent study by Greiler et al. [GvDS13]. The difficulty to differentiate parts of test cases automatically hinders test code comprehension and analysis. The same difficulty prevents distinguishing test cases from distinct testing levels. Without clearly distinguishing the parts of test cases that operate on the same program entities, it is difficult to distinguish test cases of different granularity levels automatically. We propose to distinguish test cases from different testing levels along the complexity scale. We characterize the complexity of test cases in the following section and we show the distribution of test cases for real software projects along the scale.

3.3 Test case complexity

In this section we characterize test cases of different granularity and we derive representative complexity indicators based on the study of real-life test cases. Using these indicators we develop a test case taxonomy for simple and complex test cases. To study the complexity of real-life test cases we statically analyze and inspect test suites.

We investigate application of classical code complexity measures for understanding the complexity of test code, and we conclude that classical complexity measures are not applicable as complexity indicators of test cases. We hypothesize that test case structure and, in particular, quantities of system entities used in test cases are representative

indicators of test case complexity. In the following we present our intuition behind this hypothesis and we support it with our study of real-life test cases.

Intuitively, simple test cases instantiate few classes and invoke one or few methods, while complex test cases instantiate several classes and work through longer method invocation sequences. We thus consider the number of classes involved in a test case and the length of method invocation sequences as the *primary indicators* of test case complexity.

Oracles in test cases implicitly indicate test case complexity. Oracles that verify the state of a single class suggest that test case checks only the state of a single unit. Oracles that verify states of several classes after test execution suggest that a test case checks interactions between several objects. We use this information as an *indirect indicator* of test case complexity.

Intrinsic code complexity measure

Traditional code complexity measures aid understanding the code complexity through intrinsic structural properties of source code. For example, McCabe cyclomatic complexity measure counts a number of linearly independent paths in a program to indicate program complexity [McC76]. We have analyzed the cyclomatic complexity of test code for our case studies using *Google CodePro AnalytiX*[9]. The results are summarized in Table 3.2.

Program (version)	*Average cyclomatic complexity*
TestabilityExplorer (1.3.2)	1.16
JGraphT (0.8.3)	1.19
Apache Ant (1.8.4)	1.32
JFreeChart (1.0.14)	1.33
JodaTime (2.1)	1.28
PMD (5.0)	1.19
PureMVC (1.1)	1.00

Table 3.2. Cyclomatic complexity of test cases

Despite slight variation of complexity in the test code, more than 97% of test code has cyclomatic complexity of *one*. In other words, the majority of test cases have a linear structure. Manual inspection of the test cases indicated that test cases with higher cyclomatic complexity correspond to either test scaffolding involving loops for reading data from file system or test cases for stress and performance testing.

There are two important implications of the observed results. First, little variation of code complexity prevents distinguishing unit and integration test cases based on this measure. For this reason we study and define other possible measures of complexity for

[9] `https://developers.google.com/java-dev-tools/codepro/doc/`

test code. Second, the linear structure of test code enables simplified analysis of this code with other analysis techniques. For instance, the data flow analysis of test cases can be substantially simplified and exclude analysis control flow information.

Primary test case complexity indicators

In this section we systematically explore the primary indicators of test case complexity: the number of system classes involved in a test case and the length of method invocation sequences.

To measure the indicators of test case complexity we implemented static data flow analysis. The analysis captures a number of system classes instantiated in each test case without considering external and library classes. The analysis captures the length of method invocation sequences in each test case taking into account method invocations contributing to test setup and cleanup procedures.

Since most of test cases do not rely on inter-procedural nor inter-class dependencies, our analysis does not deal with these dependencies without significantly affecting the accuracy of the analysis. It does not capture method invocations from auxiliary methods and method invocations from external or inherited classes.

Number of system classes Unit test cases are designed for testing individual classes of the system, while integration test cases – for testing aggregations of classes and their interactions. Although unit test cases focus on a single class under test, they often need to instantiate auxiliary or dependent classes, or to use mock objects to substitute them. The higher the coupling of the unit, the more classes the test case needs to instantiate or mock for test execution.

In practice, developers tend to instantiate the least necessary number of classes for test execution to make unit test cases lightweight and quickly executable. For this reason, unit test cases often use partial class instantiation with `null` object references instead of the required class instances.

We have analyzed the distribution of number of classes in test cases. The resulting frequency of number of classes in test cases is shown in Figure 3.2.

> Our analysis indicates that on average test cases instantiate *two classes* (statistical mean). However, most often (in 40% of cases) test cases instantiate only *one class* (statistical mode). Further we observed that 93% of test cases contain up to *four classes* and 99% of all test cases do not operate on more than *seven classes*. Less than one percent of test cases instantiate 8–14 classes.
>
> Manual inspection of test cases operating on four and more classes indicated that such test cases exercise complex scenarios of class instantiation and integration. The number of system classes is a clear indicator of test case complexity.

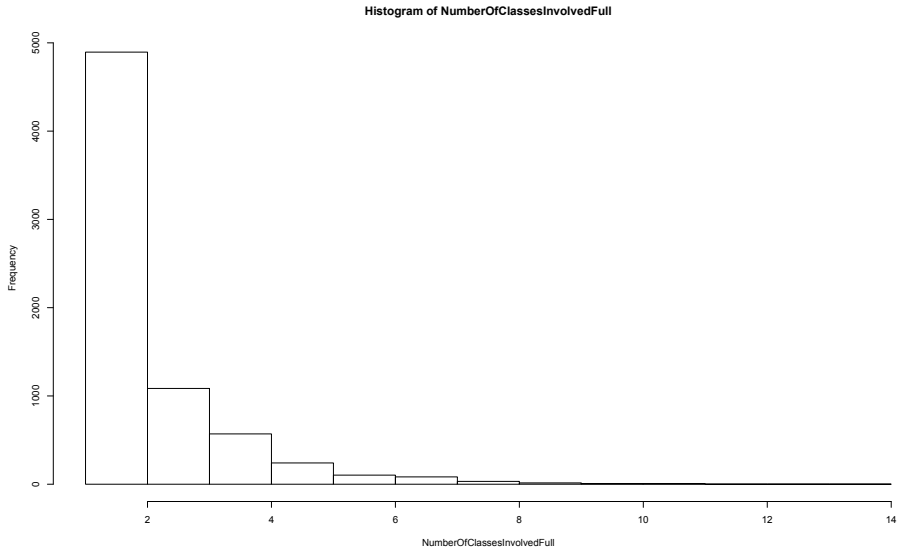

Figure 3.2. Frequency of class number in test cases

A fraction of test cases that instantiate more than four classes do not test complex test scenarios, but rather represent simple test cases aggregated together with repetitive checking of the same class property for different class instances. These are anomalous test cases that ideally should be developed as isolated simpler test cases.

Length of method invocation sequences Unit test cases usually contain few method invocations. Short sequences of method invocations either aim to verify the functionality of a single method invocation or represent a single class state transition that is checked. Additionally, some unit test cases aim to check the wiring of classes. These test cases instantiate and wire (integrate) several classes invoking their constructors, however check the wiring with few or no method invocations. This can be illustrated with the composition of unit test suites for our case studies. 95% of data in test cases is distributed as shown in Figure 3.3. The other 5% contain atypical test cases with long method invocation sequences that we discuss in the next paragraph.

A fraction of the test cases contain very long method invocation sequences: 4% of test cases contain from 20 to 43 method invocations and less than 1% of test cases have up to 230 method invocations. Manual inspection of these test cases indicated that they do not exercise complex test scenarios, but rather aggregate multiple test cases together.

The frequency of the number of method invocations in test cases is shown in Figure 3.4.

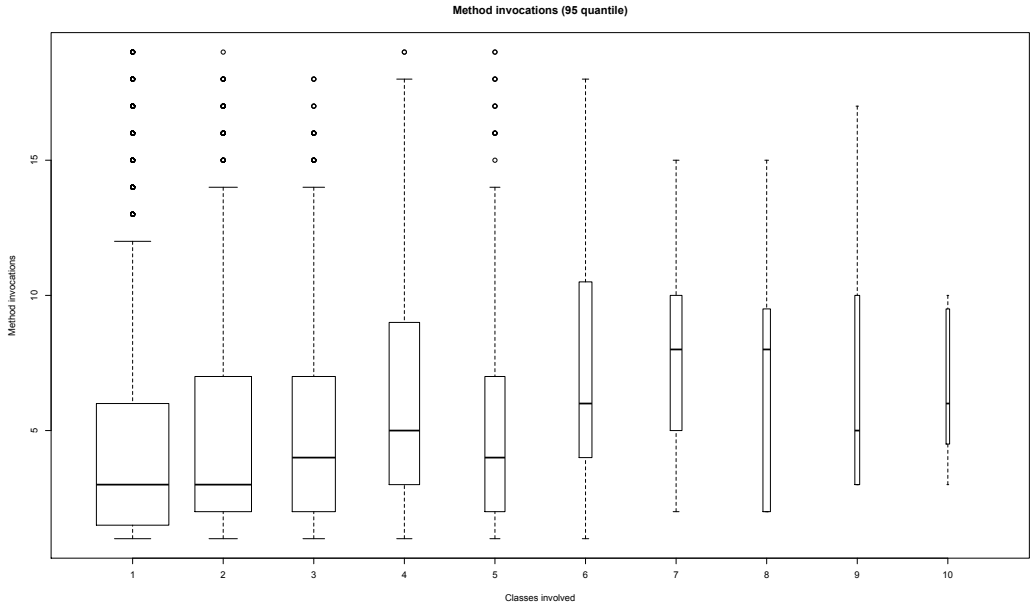

Figure 3.3. Distribution of the number of method invocations in test cases vs. number of instantiated classes

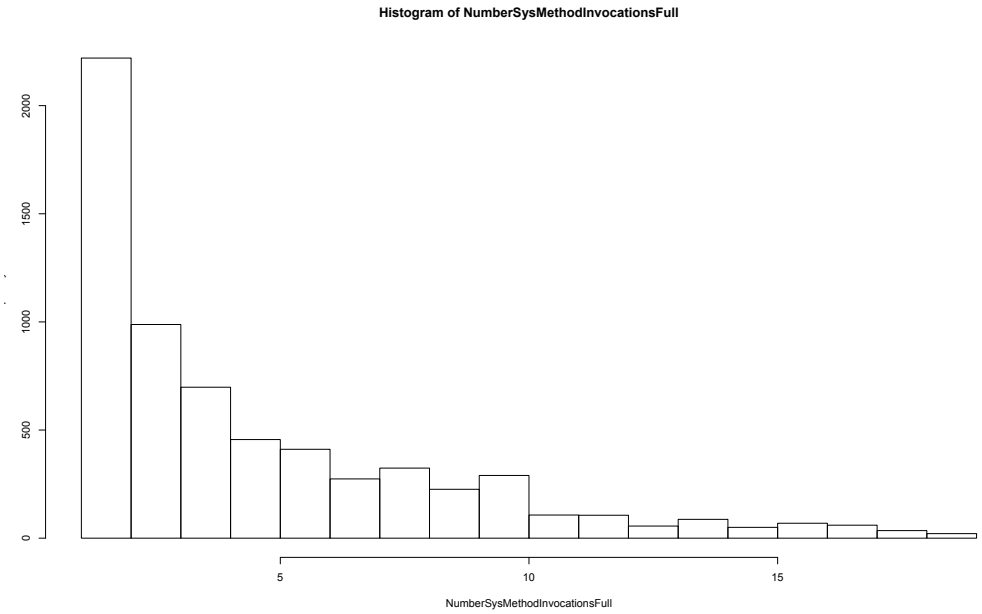

Figure 3.4. Frequency of number of method invocations in test cases

Average length of method invocations is 6.35 (statistical mean), 17% of test cases include *one method invocation* (statistical mode). While 80% of test cases have up to *eight method invocations*.

We have manually inspected test cases with method invocation sequences longer than the mean value. Test cases with long method invocation sequences correspond to two main categories. The first category contains test cases that are associated with the same complex test cases that instantiate more than four classes and test complex class interactions. The second category contains test cases that explore large amount of method invocations with different parameter values for single classes. Together with the number of classes, the length of method invocation sequences is a good indicator of the test case complexity.

Indirect test case complexity indicators

Oracle is an essential part of a test case that represents human knowledge about expected system behavior. Our intuition is that oracles can indirectly indicate whether test cases verify simple class properties or complex interactions and thus indicate test case complexity.

Oracle can indicate an interaction between several classes that changes their mutual states. Such oracle accesses and checks the state of several classes. In contrast, oracles in simple test cases usually check the state of a single class under test. Simple test cases often instantiate several classes and keep unchanged the state of auxiliary classes, while the state of the class under test changes and is controlled in the oracle.

We statically analyzed and categorized assertions in test cases according the number of classes assertions operate on considering a total number of assertions in each test case. We recorded the data for assertions that directly access class instances of different classes. We did not record indirect references to the class instances (for instance, assertion accessing a variable assigned as a result of a method invocation). We also did not count instances of anonymous classes, nor did we consider the class types returned from method invocations. The results of analysis are summarized in Figure 3.5.

37% of test cases predicate on *one class*. 51% of test cases predicate on *two classes*. 10% of test cases predicate on *three classes*, while less than 2% of test cases predicate on more than *three classes*.

Manual inspection of test cases operating on three and more classes revealed that most of these test cases (with some exceptions) check complex interaction scenarios and verify states of complex objects. In few exceptional cases, test cases contain large amount of assertions (more than ten assertions) because

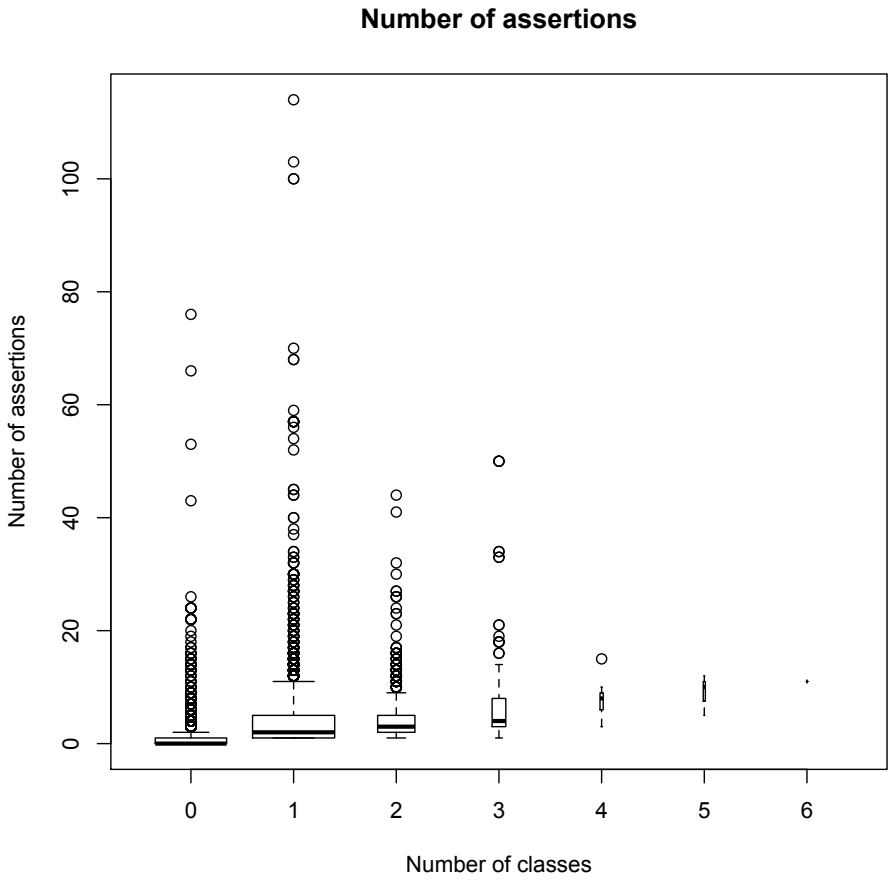

Figure 3.5. Distribution of the number of classes in assertions in test cases

they aggregate simple test cases and do not explore complex scenarios of class integration. These results suggest that the number of classes used in assertions in test cases is a good indirect indicator of test case complexity.

The analysis does not capture indirect references to the class instances in assertions when assertions comparing test execution results with expected values without accessing class instances under test. This is why the leftmost box plot in Figure 3.5 designates distribution of a number of assertions in test cases, where assertions operate on zero classes. The precision of the analysis can be improved to extract information for assertions that indirectly check the states of the classes under test or computation results.

Taxonomy of simple and complex test cases

We now define simple (unit) and complex (integration) test cases based on representative complexity indicators: C_T – a number of system classes a test case operates on, M – a length of method invocation sequence in a test case, and C_A – a number of system classes test assertions operate on.

Definition. A *simple test case* is a test method operating on less than four system classes in less than six method invocations with assertions on less than three classes ($C_T < 4$ and $M < 6$ and $C_A < 3$).

Conversely, a definition of a complex test case is as follows:

Definition. A *complex test case* is a test method operating on four or more system classes in six or more method invocations with assertions on three or more classes ($C_T \geq 4$ or $M \geq 6$ or $C_A \geq 3$).

In this thesis aim to generate complex test cases from simpler ones. In particular we focus on generating integration test cases from unit test cases. Using the presented taxonomy we associate *unit* test cases with the *simple* test cases and *integration* test cases with the *complex* ones. The taxonomy is generalizable and can be extended for characterizing test cases of different granularity.

Remarks

The presented taxonomy is based on a thorough investigation of test suites from many software projects. It reflects our intuition and supports our hypotheses. We have shown that test cases can be hardly distinguished automatically by their goal and that intrinsic code complexity measure is not applicable as a complexity indicator for test cases, because the intrinsic complexity of the *JUnit* code does not change significantly between simple and complex test cases.

Our investigation indicates that the test case structure and, in particular, the amount of system entities used in the test cases are representative indicators of test case complexity. In the next section we use the described characterization to study the overlap and reuse of information between unit and integration test cases.

Limitations and threats to validity

We selected the case studies according to the availability of test suites and not randomly, and this involves a threat to the validity of the experiment. The discovered indicators of test case complexity are specific to the object-oriented software. Case studies used for investigation are open source software. Projects and test suites extensively supported by software developers. Some projects, like *JodaTime*, *PMD*, *JGraphT*, *JFreeChart* are used in many research papers as case studies. Analyzed software projects come with mature

test cases with from medium to high coverage of system functionality (Figure 3.1). A threat to validity concerns generalization of the results to industrial systems that employ different testing standards and procedures than object-oriented open source software.

Another threat to validity concerns the generalization of the results to systems using special integration frameworks and to systems from domains not covered in the study. Large component-based systems and distributed enterprise applications often employ special integration frameworks, such as *Camel*[10]. These frameworks integrate system components according to the integration rules defined in a domain-specific language. Throughout our study we focus on applications that do not rely on special integration frameworks. We study the applications with class integration incorporated in the system. For instance, applications that integrate classes through dependency injection relying on class constructor parameters.

In this study we apply well known statistical techniques that are robust to violations of their assumptions. One general threat to validity is the number of test cases analyzed, which may reduce the ability to reveal patterns in the data.

Finally, the chosen analysis techniques may represent a threat to validity. Precision of static analysis is limited. The static analysis of number of classes used in assertions does not capture indirect references to the class instances in assertions. The static analysis of primary complexity indicators does not deal with inter-procedural nor inter-class dependencies in test cases. It does not capture method invocations from auxiliary methods and method invocations from external or inherited classes. This limitation should not significantly affect the accuracy of the analysis since most of the test cases in the study do not rely on inter-procedural nor inter-class dependencies.

3.4 Structural overlap

The study presented in this section explores the phenomenon of information sharing and reuse among test cases. The study aims to support our hypothesis that important information in test cases can be identified and reused. We investigate the extent of the phenomenon of testing reuse in real-world test suites. We hypothesize that different fragments of test cases are reused and build upon, and we investigate what fragments of test cases are reused. We aim to detect interrelation between simple and complex test cases with the goal to exploit it for test case generation.

The study of code reuse builds on results of our exploration of test case structure and common parts of test cases (Sections 3.1 and 3.2), and indicators of test case complexity (Section 3.3). In this section we investigate how simple and complex test cases share information and we present our analysis of the amount of code reuse in manually developed test cases taken from open-source projects.

[10] `https://camel.apache.org/`

Reuse in simple and complex test cases

Figures 3.6 and 3.7 show the excerpt of some unit test cases for the `Controller` and the `View` components of *PureMVC*. These test cases exercise short sequences of method invocations on individual classes aiming to check the behavior of the single classes.

In particular, the two test cases in Figure 3.6 exercise the `Controller` functionality to register, execute and remove `Commands` (the code corresponding to these method invocations is underlined in the figures), but they do not check the interactions with the `View` where the corresponding `Observers` are registered. These unit test cases focus on the effects of the method invocations in the scope of the class `Controller`.

Similarly, the unit test case for the `View` class shown in Figure 3.7 does not check the interactions between the `Controller` and `View` classes.

```
1  // Complexity indicators: C_t = 3; M = 2; C_a = 1;
2  public void testRegisterAndExecuteCommand( ) {
3   IController controller = Controller.getInstance();
4   controller.registerCommand("ControllerTest", new ControllerCommand());
5   ControllerTestVO vo = new ControllerTestVO(12);
6   Notification note = new Notification("ControllerTest", vo, null);
7   controller.executeCommand(note);
8   assertTrue("Expected result == 24", vo.result == 24);
9  }
```

```
1  // Complexity indicators: C_t = 3; M = 4; C_a = 1;
2  public void testRegisterAndRemoveCommand( ) {
3   IController controller = Controller.getInstance();
4   controller.registerCommand("ControllerRemoveTest", new ControllerCommand());
5   ControllerTestVO vo = new ControllerTestVO(12);
6   Notification note = new Notification("ControllerRemoveTest", vo, null);
7   controller.executeCommand(note);
8   assertTrue("Expected result == 24", vo.result == 24);
9   vo.result = 0;
10  controller.removeCommand("ControllerRemoveTest");
11  controller.executeCommand(note);
12  assertTrue("Expected result == 0", vo.result == 0);
13 }
```

Figure 3.6. Two unit test cases for the `Controller` component

Figure 3.8 shows a complex test case from the integration test suite of *PureMVC*. This test case exercises the interactions between the classes `Controller` and `View`. Class `Controller` registers, removes and registers again the same `Command`, and class `View` notifies the `Observers` associated with the registered `Command`.

By comparing the simple test cases for classes `Controller` and `View` shown in Figures 3.6 and 3.7 with the test case for the integrated classes in Figure 3.8, we can

```
1  // Complexity indicators: C_t = 3; M = 2; C_a = 0;
2  public void testRegisterAndNotifyObserver() {
3   IView view = View.getInstance();
4   Observer observer = new Observer(this, this);
5   view.registerObserver(ViewTestNote.NAME, observer);
6   INotification note = ViewTestNote.create(10);
7   view.notifyObservers(note);
8   assertTrue("Expected var == 10", viewTestVar == 10);
9  }
```

Figure 3.7. A unit test case for the `View` component

```
1  // Complexity indicators: C_t = 4; M = 5; C_a = 1;
2  public void testReregisterAndExecuteCommand() {
3   IController controller = Controller.getInstance();
4   controller.registerCommand("ControllerTest2", new ControllerTestCommand2());
5   controller.removeCommand("ControllerTest2");
6   controller.registerCommand("ControllerTest2", new ControllerTestCommand2());
7   ControllerTestVO vo = new ControllerTestVO(12);
8   Notification note = new Notification("ControllerTest2", vo, null);
9   IView view = View.getInstance();
10  view.notifyObservers(note);
11  assertEquals("Expected result == 24", vo.result, 24);
12  view.notifyObservers(note);
13  assertEquals("Expected result == 48", vo.result, 48);
14 }
```

Figure 3.8. An integration test case for *PureMVC*

notice that all the code of the integration test case belongs also to the unit test cases (reused statements are underlined in the figure).

We measure the amount of code reuse in test cases by identifying code sections that are equivalent except for a systematic change of parameters. We use *Google CodePro AnalytiX* to collect these data and we select reuse information for code fragments containing more than two lines of code. The results of the analysis are summarized in Table 3.3.

Program (version)	*Test cases, LOC*
TestabilityExplorer (1.3.2)	5596
JGraphT (0.8.3)	5637
Apache Ant (1.8.4)	24384
JFreeChart (1.0.14)	49644
JodaTime (2.1)	51715
PureMVC (1.1)	706
PMD (5.0)	13922
Total:	151604
Original code reuse	15008 (9.8%)
Total code reuse	44942 (29%)

Table 3.3. Code reuse in test source code

We have observed that for seven subjects a total of 29% of code is reused. In other words, one third of test code is shared across test suites. We also observed that test code is reused repeatedly. For the analyzed subjects, a total of 15008 LOC of unique code is repeatedly reused. Average length of reused code pieces is of *five to six lines of code* (Mean 5.45), while on average these pieces of code are reused *three times* (Mean 3.04).

We have manually analyzed reused code in test cases. Our analysis indicates that reused code involves *all* parts of test cases, including test setup and scaffolding, test execution and oracle. Moreover, some blocks of reused code correspond to several parts of test cases. For instance, different test cases share combinations of test execution and oracle parts, and different test setup.

These results suggest that different parts of test cases can be reused to construct new test cases.

3.5 Important information in test cases

Test cases are built using knowledge about system under test and they carry important information. As source code captures information about design and objectives of the system, test code captures information about various scenarios of system assembly and execution.

A test case is an instance of a test specification that derives from various sources including: system specification, module specification, interface specification, system design and architecture models, source code specification, source code structure, historical software project data, previous faults found in the system, and domain knowledge of application developer and tester.

Automatically derived test cases also carry meaningful information. Such test cases capture certain heuristics and assumptions about software system that allow them to explore system executions and detect specific structural and functional problems.

Different aspects of test cases and test case structure can be mapped to the important system information they capture. Correct instantiation of classes requires valid sets of inputs and parameter values for constructor and method invocations in test cases. The absence of input parameters, for instance, `null` class references in place of class constructor and method parameters indicate what information can be omitted in a certain context. In addition to valid inputs and parameters, dependent classes shall be instantiated in correct order. Instantiation order and mutual order of class instantiation and method invocations in test cases represent implicit protocols of class setup and class usage.

Information about external class dependencies, operational environment dependencies, and external system interactions are captured in test setup and test scaffolding. Scaffolding may include test drivers that substitute calling programs and thus contain information about how the system can be used or interacts with its environment. In addition, scaffolding contains information about dependencies to auxiliary and library classes used by the system.

Test oracles capture expected system behavior, effects of class interactions and method invocations. They indicate data that should change as a result of test execution, as well as invariant properties and data. A scope of an oracle check indicates the scope of changes and side-effects caused by test execution. Test oracles indicate what information is relevant in a given context in the same way as omitted parameters indicate their irrelevance for specific method invocations.

Intuition and core idea

We argued that test cases capture important system information and domain knowledge. Some of this information can be automatically identified and exploited.

In the discussion of the related work in Section 2.3 we have shown that different approaches analyze and leverage information from test cases and test executions to support software engineering tasks and system comprehension, facilitate and enable test case repair, test suite evolution, and test case generation.

Our **intuition** is that information from test cases can be exploited not only to support system comprehension, test case repair and test suite evolution, but also to generate new test cases automatically.

Differently from existing approaches, we intend to leverage important information in test cases to automatically generate more complex ones and to drive test generation towards complex module interactions. We aim to generate fresh integration test suites or to augment test suites with behaviors that are not yet present in the integration test suites.

We draw our intuition from the study of many test cases presented in this chapter. Our observations indicate that many test cases build on the information from other test cases. In the same way, we aim to reuse information in test cases to support automatic test case generation exploiting the common program entities used for test case construction that we identified in Section 3.1 and the similarities in test case structure that we identified in Section 3.2.

This study motivates our **core idea**: to reuse meaningful information in test cases to generate new test cases. We aim to identify fragments of meaningful information in test cases and capture relationships between these fragments to combine them into new complex test cases. In the next chapter we describe our approach to automatic test case generation.

Chapter 4

Generating Integration Test Cases Automatically

In this chapter we present a novel approach to generate test cases by automatically reusing information from existing test cases. The core idea is to automatically extract relevant fragments of test cases and combine them into meaningful and complex test cases. We present the key challenges in extracting and combining information from test cases, and we introduce our solutions that enable test case generation from the information in the system source code and in the corresponding test suites. We illustrate the approach on a real example by generating a new test case that detects a previously unknown integration fault.

In this thesis we introduce a technique to automatically generate complex test cases from simple ones. Our approach stems from the two key observations that derive from the study of many test cases from popular open-source software systems (Chapter 3).

First observation: both unit and integration test cases operate on the same program entities, and use method calls as main components of test cases. Both kinds of test cases exercise parts of the system functionality and check the resulting state of the system. The difference lies in the scope of those checks. Unit test cases exercise small parts of the system, for example, a single class. They invoke few methods and check that the method return values and the state of the class are as expected. In contrast, integration test cases exercise large parts of the system by replacing mock objects with real module implementations, working through long method call sequences, and checking the state of all modules involved.

Second observation: simple (unit) and complex (integration) test cases share considerable amount of information. We have observed that fragments of complex test cases are identical to the fragments of simpler unit test cases. This is not surprising, because integration testing usually follows unit testing and, according to our *first observation*,

integration testing operates on the same entities as unit testing. This is why integration test cases tend to share fragments with unit test cases.

Moreover, unit test cases contain information on how to instantiate classes in meaningful ways, how to construct arguments for method calls, and what the resulting system state should be after calling methods with those arguments. By reusing fragments of existing unit test cases developers reuse and consult human-made, meaningful and relevant information about a system under test.

These observations motivate the *core idea* behind our approach: *using simple unit test cases to automatically generate complex integration test cases that can reveal interaction faults.*

4.1 Approach

Figure 4.1 shows a general overview of the approach. The approach works by analyzing software system to identify dependent system components that can be integrated and tested together. It identifies dependencies among these components to stress them in generated test cases. Our approach analyzes test cases to extract relevant fragments. It then selects, reuses and combines relevant test case fragments into new test cases guided by the dependencies between the system components. Once new test cases are generated, they can serve as input to the process to generate even more complex ones.

We use information from existing test cases to construct more complex test cases that focus on class interactions rather than on individual state transformations. We aim to detect faults that depend on the interaction between several modules, and as such can be hardly detected with simple test cases that check the behavior of single units.

In the following section we describe the challenges of generating complex integration test cases, and we frame our approach for automatic test case generation that effectively tackles these challenges.

Challenges

Our approach relies on the hypotheses that we can identify automatically the test fragments that can be used to build new test cases, and that we can automatically assemble the identified fragments to generate feasible test cases. Capturing and composing these fragments poses a number of key challenges we need to address to enable our approach.

First, we need to identify meaningful code fragments that can be assembled together. The problem is difficult, because, as we have shown in our study of test case structure, the logical fragments in test cases are mixed and not easily identifiable (Section 3.2). We need to map the logical structure of the test cases to the underlying syntactic structure to enable test case analysis, automatic extraction and manipulation of code fragments. We propose to segment test cases in composable fragments that correspond to instantiation

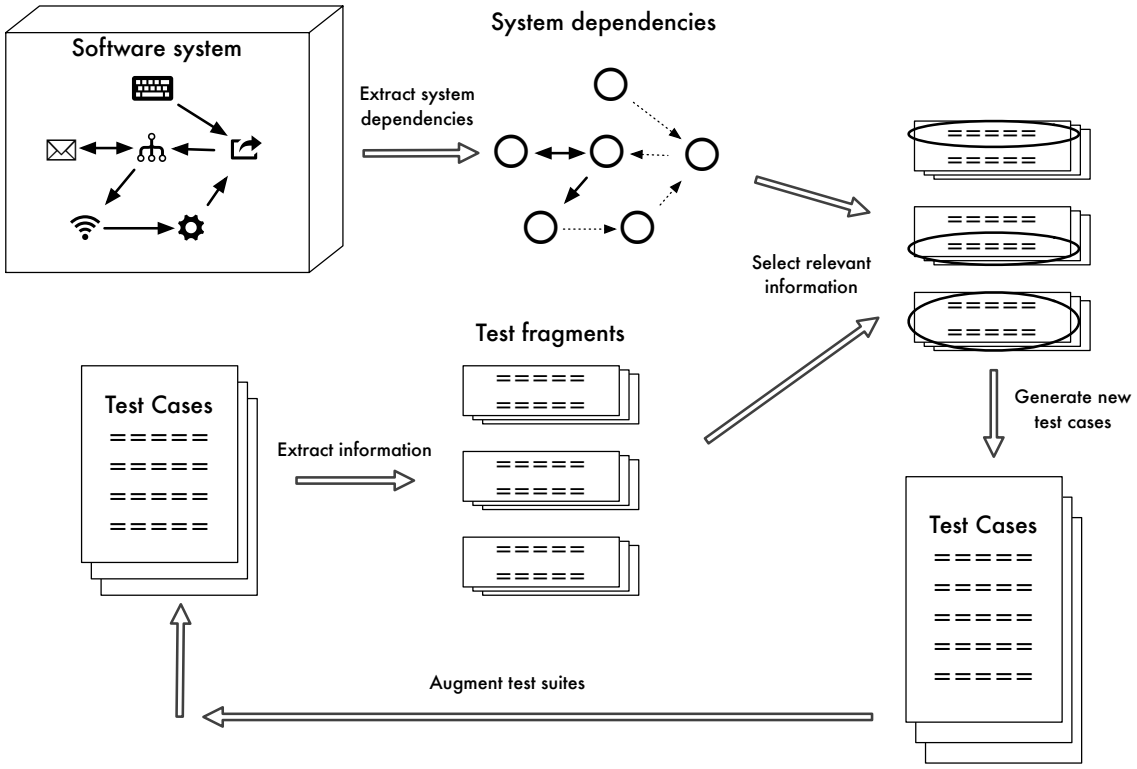

Figure 4.1. Overview of the approach

of system classes and scenarios of class usage, and we solve this problem with data flow analysis of the test cases as illustrated in Section 4.3.

Second, for any non-trivial software system the number of possible combinations of test case fragments is prohibitively large. We need to efficiently combine test case fragments in effective test suites.

Since not all combinations of fragments are meaningful, we propose to explore the combinations for clusters of dependent classes and use the system dependencies to indicate these clusters. In Section 4.2 we describe our approach to identifying and capturing class dependencies.

Finally, we need to combine test case fragments into valid and useful test cases. Combinations need to be valid with respect to the rules of a programming language the system is implemented in, and the underlying constraints imposed by the operating environment, including initialization and setup of the system. Useful combinations should trigger interactions between the components of the system, explore new and untested system states, and detect integration faults.

We address these challenges by developing integration strategy that drives the generation of new test cases using dependence and data flow information. In Section 4.4 we present the details of the integration strategy that we propose and the overall test generation process.

Three main phases

Our approach to generate complex integration test cases from simple unit ones requires the system source code and a set of test cases as input, and works in three main phases: (1) identify class dependencies within the system in the form of an object relation diagram (ORD), (2) compute the data flow information within the input test cases, and use this information to segment the test cases into useful fragments (initialization and execution), and (3) generate new test cases using the class dependence information together with the data flow information to combine the fragments extracted from the simple test cases to build feasible complex test cases.

4.2 Extracting class dependencies

In this phase, we build an object relation diagram of the system under test to identify clusters of dependent classes that shall be tested together. We derive the class dependencies from the system source code.

We distinguish *implicit* and *explicit* class dependencies in our approach. The goal of our approach is to exercise interactions between dependent classes to reveal integration faults. As we show in this section, implicit class dependencies can be tested in the context of a single unit and are less relevant for exercising class interactions. In contrast, explicit class dependencies indicate stronger connections between components and potential class integrations that we exploit and exercise in our approach.

Implicit and explicit class dependencies

A *class dependency* exists when a class uses/accesses an instance of another class. A class A depends on a class B when either the constructor or a method of class A requires one or more instances of class B. The dependency is *explicit* when a class A declares explicitly an argument of type B in the interface of its constructors or methods. The dependency is *implicit* when a class A uses some instances of B, but B is not specified in the interfaces of any method of class A. An example of such implicit dependency is a class that accesses some instances of other classes using the Singleton pattern.

Our categorisation of class dependencies redefines a *"uses"* relation proposed by Parnas [Par78]. Parnas defines the *"uses"* relation for pairs of programs: a program *A uses* program *B* if correct execution of *B* may be necessary for *A* to complete the task described in its specification. In the same way, we distinguish class dependencies

that for pairs of classes indicate if correct execution of one class is required for correct execution of another class.

We give the classification of different kinds of implicit and explicit class dependencies using Java programming language constructs.

1. Implicit field/variable. A class instantiates and uses an object of another class:

```
1 public class ImplicitDependency1 {
2 private Writer writer = new Writer(null); // instantiation of class Writer shows
      implicit dependency to Writer
3
4 public void write() {
5  Format format = new Format("Times"); // instantiation of class Format shows implicit
      dependency to Format
6  writer.compile(format);
7 }
```

2. Implicit instance access. Class accesses existing instance of another class using static method:

```
1 public class ImplicitDependency2 {
2 private Writer writer = Writer.getInstance(); // implicit dependency to Writer (
      instance of Writer may exist already)
3
4 public void write() {
5  writer.compile(Format.TIMES); // static field access shows implicit dependency to
      Format
6 }
```

3. Implicit intermediary class method call. Class accesses (existing) instance of another class through a method invocation on a intermediary class:

```
1 public class ImplicitDependency3 {
2 private WriterFactory writerFactory = new WriterFactory(); // intermediary class
      WriterFactory
3
4 public void write() {
5  Writer writer =  writerFactory.createWriter(new GoodMood()); // return type shows
      implicit dependency to Writer
6  writer.compile("style");
7 }
```

4. Explicit class creation. A class cannot be instantiated without manifested dependency:

```
1 public class ExplicitDependency1 {
2 private Writer writer;
3 public ExplicitDependency1(Writer writer) { // formal parameter shows explicit
      dependency to Writer
4  this.writer = writer;
5 }
6 public void write() {
7  writer.compile("style");
8 }
```

5. Explicit method invocation. Method cannot be used without manifested dependency:

```
1 public class ExplicitDependency2 {
2
3 public void write(Writer writer) { // formal parameter shows explicit dependency to
      Writer
4  writer.compile("style");
5 }
```

Different types of dependencies determine the class *instantiation order* and allow or prohibit *class substitution* during class instantiation. In particular, class substitution is not configurable for classes related through all types of implicit class dependencies. On the contrary, for both types of explicit dependencies class substitution is possible due to polymorphism and is configurable outside of the class. Consequently, explicit dependencies allow to instantiate classes using mock objects or `null` values without using instances of dependent classes.

Explicit class dependencies impose class instantiation and method invocation orders, while implicit dependencies do not impose such orders. For implicit dependencies these orders can be indicated in the class specification. Explicit class creation dependency (case 4) imposes class instantiation order with respect to the order of constructor invocation, while explicit method invocation dependency (case 5) imposes class instantiation order with respect to the order of method invocation (execution order). The following listing shows the explicit class dependencies and syntactically valid constructs with the imposed order of class instantiation:

```
1  public class TestExplicitDependencies {
2      public void testExplicitDep1() {
3          Writer writer = new Writer(); // Writer has to be instantiated before constructor
       call
4          ExplicitDependency1 dep = new ExplicitDependency1(writer);
5          dep.write();
6      }
7      public void testExplicitDep2() {
8          ExplicitDependency2 dep = new ExplicitDependency2();
9          Writer writer = new Writer(); // Writer has to be instantiated before method call
10         dep.write(writer);
11     }
12 }
```

In practice explicit dependencies (cases 4 and 5) are used to integrate collaborating classes and represent situations, when the state of the dependent class may be changed from the outside by other classes. Such dependent classes should be either tested in conjunction or can be partially substituted by the mock objects for testing purposes.

In the study of many software systems we have observed that class integration through explicit dependencies usually follows two patterns. The *first pattern* involves class integration through class constructor parameters where each parameter is instantiated in a complex class instantiation sequence. The *second pattern* involves

class instantiation where classes are instantiated through constructor calls with few or no parameters followed by a sequence of set/add method invocations that integrate dependent classes through method parameters.

Implicitly instantiated classes are less likely to change their state from the outside. Implicit class instantiation is used in practice to encapsulate dependent classes in the scope of a single class without exposing them outside as a part of the state of the class. The class dependency formed this way can be tested in the context of the class that integrates other classes (in the other terms – it can be targeted during unit testing).

There are two rare cases when classes related by implicit dependency can have state changes initiated outside of the class. First case happens for classes integrated through an implicit class dependency using instance access methods (case 2). In this case the instance of a dependent class is not created in the user-class, but is accessed with the static (instance access) method call. Since the dependent class instance is not controlled by the user-class it can have state changes initiated from the outside of the user-class, and unexpected state changes may trigger integration faults.

The second case takes place when a class implicitly instantiates other classes, but does not encapsulate them well and allows their modification from the outside (for instance, class with public fields). This case is rare and represents a poor programming practice that can be prevented with various analysis techniques.

In our approach we model class dependencies of a system in the form of object relation diagram and we use the diagram in the next phases of the approach.

Object Relation Diagram (ORD)

ORD represents the class dependencies extracted from the class relations such as inheritance (I), association (As) and aggregation (Ag), and can be extracted from a system class diagram [BLW03]. ORDs are commonly used to derive integration and test orders from class dependencies. Extended versions of ORD such as *Test Dependency Graph* additionally model dependencies of the method level, although these dependencies do not impose a strict test order of classes.

We capture class dependencies in a modified ORD. Differently from the general ORD that models multiple class relations we model only explicit class dependencies that correspond to the class association (As). These dependencies impose class instantiation and execution orders, and indicate classes that shall be tested in conjunction. We also do not highlight inheritance relation (I) in ORD, and we substitute it with the association (As) between subclasses and other classes directly associated with the corresponding super class.

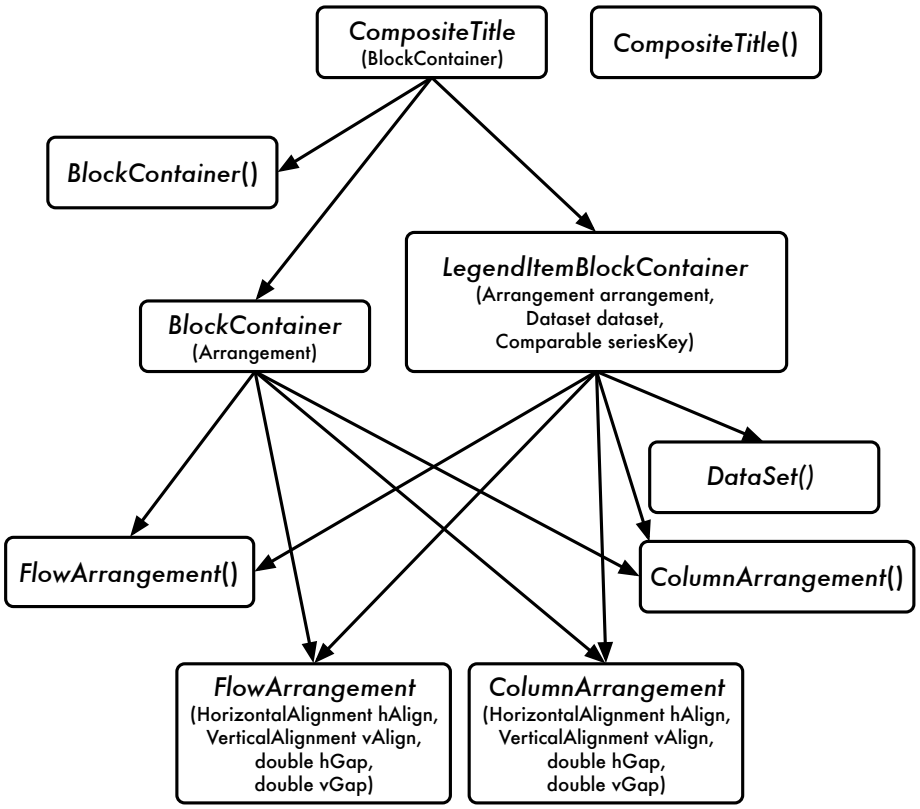

Figure 4.2. Fragment of ORD for classes of *JFreeChart*

Consider, for example, a fragment of ORD for the *JFreeChart* library[1] in Figure 4.2. The ORD is extracted from the constructor signatures shown in Figure 4.3 and the class diagram shown in Figure 4.4.

In the diagram, nodes represent the class constructors and directed arcs represent the dependencies between the constructors. Classes can have several constructors that may manifest dependencies to different classes and each constructor corresponds to a different way class can be instantiated. Thus nodes of the ORD correspond to distinct class constructors and ORD can contain several nodes for a given class with several constructors. ORD also contains disconnected nodes (`CompositeTitle()` constructor node in Figure 4.2) that do not require other class instances and are not required to construct other classes.

[1] `www.jfree.org`

```java
/**
 * A title that contains multiple titles within a {@link BlockContainer}.
 */
public class CompositeTitle extends Title implements Cloneable, Serializable {
    public CompositeTitle() {}
    /**
     * Creates a new title using the specified container.
     * @param container  the container (<code>null</code> not permitted).
     */
    public CompositeTitle(BlockContainer container) {}
}
/**
* A container for a collection of {@link Block} objects.  The container uses
* an {@link Arrangement} object to handle the position of each block.
*/
public class BlockContainer extends AbstractBlock implements Block, Cloneable,
     PublicCloneable, Serializable {
    public BlockContainer() {}
    /**
     * Creates a new instance with the specified arrangement.
     * @param arrangement  the arrangement manager (<code>null</code> not permitted).
     */
    public BlockContainer(Arrangement arrangement) {}
}
/** A container that holds all the pieces of a single legend item. */
public class LegendItemBlockContainer extends BlockContainer {
    /** Creates a new legend item block. */
    public LegendItemBlockContainer(Arrangement arrangement, Dataset dataset, Comparable
     seriesKey) {}
}
/**
 * An object that is responsible for arranging a collection of {@link Block}s within a {@link
      BlockContainer}.
 */
public interface Arrangement {}
/** Arranges blocks in a flow layout.  This class is immutable. */
public class FlowArrangement implements Arrangement, Serializable {
    public FlowArrangement() {}
    public FlowArrangement(HorizontalAlignment hAlign, VerticalAlignment vAlign, double hGap,
      double vGap) {}
}
/** Arranges blocks in a column layout.  This class is immutable. */
public class ColumnArrangement implements Arrangement, Serializable {
    public ColumnArrangement() {}
    public ColumnArrangement(HorizontalAlignment hAlign, VerticalAlignment vAlign, double
     hGap, double vGap) {}
}
```

Figure 4.3. System interfaces for *JFreeChart* classes

More formally, given a set of classes S, ORD is a tuple $\langle C, E \rangle$, where

C is a set of constructors for the classes in S;
E is a set of edges between constructors;
$E = \{ \langle c_1, c_2 \rangle \in C^2 \mid \exists p \in param(c_1) \quad subtype(class(c_2), p) \}$, where

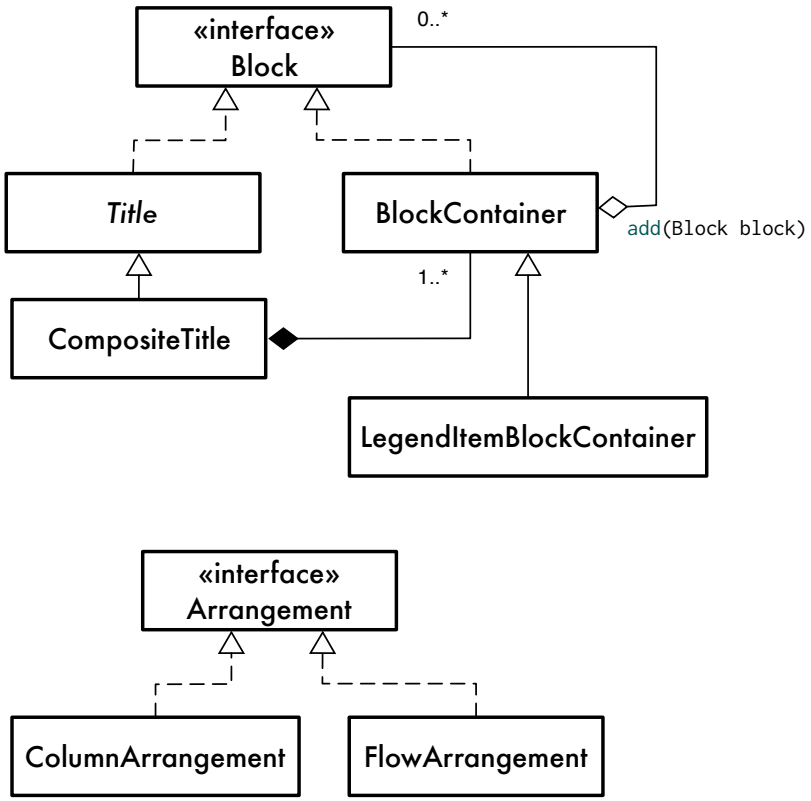

Figure 4.4. Fragment of class diagram for *JFreeChart*

$param : C \rightarrow 2^S$ associates constructors with the classes of formal parameters in constructors,

$class : C \rightarrow S$ associates constructors with the classes they instantiate, and

$subtype : S \times S \rightarrow \{T, F\}$ is a predicate over inheritance relation between the pairs of classes with the property that $subtype(s_1, s_1)$ evaluates to true. $subtype(s_1, s_2)$ evaluated to true denotes "s_1 is a subtype of s_2".

Constructor dependencies define a *preorder* relation between classes and can potentially form cycles in the ORD.

ORD construction We build ORD from class references and class hierarchy information. We identify class references from syntactic entities in a system source code. To identify class references we statically analyze the system source code, identify the public constructors and analyze formal parameters of the class constructors. We extract class hierarchy information analyzing class declarations and we use it to connect the nodes of the object relation diagram. We connect the constructor node of a class X

to the constructor node of a class Y if Y or Y's super class is referenced in the formal parameters of the constructor of class X.

Consider, for example, the ORD shown in Figure 4.2. The corresponding constructor interfaces are shown in Figure 4.3 and a corresponding class diagram is shown in Figure 4.4. In the ORD, a constructor of `BlockContainer` class is connected with the constructors of `ColumnArrangement` and `FlowArrangement` classes.

Both `ColumnArrangement` and `FlowArrangement` classes implement the `Arrangement` interface that is a formal parameter in `BlockContainer`'s constructor, and thus both subtypes of `Arrangement` can be used in the `BlockContainer`'s constructor as indicated in the ORD. Similarly, as one can see from the class diagram in Figure 4.4, `LegendItemBlockContainer` extends the `BlockContainer` class and thus can be used in a constructor of `CompositeTitle` class. This is modelled in the ORD with an edge connecting the `CompositeTitle` and the `LegendItemBlockContainer` constructors.

We use ORD to model only dependencies for the classes of the system under test and we do not analyze external or system dependencies. In the example of the ORD in Figure 4.2, there is no dependency between `LegendItemBlockContainer` and `Comparable`, although `LegendItemBlockContainer` depends on the interface `Comparable` as indicated in line 27 in Figure 4.3. Node `Comparable` and the corresponding dependency are not included in the ORD because this is a dependency to the external Java interface.

4.3 Extracting instantiation and execution sequences

In this phase, we use data flow information on the input test cases to identify test case fragments and dependencies among them. We automatically derive only relevant statements that contribute to particular logical parts of test cases using *dominance data flow analysis*. We aim to identify fragments that correspond to instantiation of classes and fragments that correspond to different scenarios of class usage.

We analyze data flow and control flow information in input test cases to extract homogeneous test case fragments that represent subsets of statements from test cases with a specified order of execution. We extract fragments corresponding to sequences of statements that either instantiate the classes involved in the test (*instantiation sequence*), or execute methods of the instantiated classes for testing a specific class (*execution sequence*). We derive the order of execution in the sequences from the control structure of test case imposed by testing frameworks like JUnit.

There can be several instantiation and execution sequences for each test case, depending on the number of classes involved in the test. We identify instantiation and execution sequences from a given test case for all the system classes that occur in the test cases, both for the class under test and for the system classes used to create auxiliary objects in the test. We capture library and external class instances as an auxiliary data for the sequences of the analyzed system classes.

We associate instantiation and execution sequences with the classes that are instantiated and used by the sequences, and with the classes for which the sequences can be applied due to polymorphism. We associate the sequences and classes in the ORD by augmenting the nodes of the ORD with the corresponding sequences. Since each node in the ORD represents a class constructor, we augment the nodes that are related to a specific class and the nodes that are related to its sub classes through polymorphic relations.

Figure 4.5 shows one example of instantiation and execution sequences that can be derived from the test in the figure. The test class instantiates four system classes and thus contains four instantiation sequences (I1-I4): one sequence for the class under test `CompilationUnitBuilder`, and other sequences for the classes `Qualifier`, `JavaSrcRepository`, and `JavaClassRepository` that are used to create auxiliary objects in the test. The test class contains five execution sequences (E1-E5): each execution sequence is composed of statements that invoke methods on the instantiated classes in the order imposed by the test structure.

```
1  public class JavaClassInfoBuilderTest extends TestCase {
2  I1 I3 I4 ClassRepository parent = new JavaClassRepository();
3  I1 I3 JavaSrcRepository repository = new JavaSrcRepository(parent, null);
4  I1 I2 Qualifier qualifier = new Qualifier();
5  I1 CompilationUnitBuilder builder;
6
7    protected void setUp() {
8  I1 E3 qualifier.setPackage("pkg");
9  I1 builder = new CompilationUnitBuilder(repository, qualifier, "");
10   }
11   public void testClassNameIsConcatinationOfPackageAndType() throws Exception {
12 E1 builder.startType(0, "A", null, new ArrayList<Type>());
13 E1 builder.endType();
14    assertNull(builder.type);
15   E4 assertEquals("pkg.A", repository.getClass("pkg.A").getName());
16   }
17   public void testInnerClass() throws Exception {
18   E3 qualifier.addAlias("B", "pkg.A$B");
19 E2 builder.startType(0, "A", null, new ArrayList<Type>());
20 E2 builder.startType(0, "B", null, new ArrayList<Type>());
21 E2 builder.endType();
22 E2 builder.endType();
23    assertNull(builder.type);
24   E5 assertEquals("pkg.A$B", repository.getClass("pkg.A$B").getName());
25   }
26 }
```

Figure 4.5. Instantiation and execution sequences of a test case

The execution order within test classes is determined by the control flow structure that is generally imposed by the testing framework. The majority of testing frameworks support the three-part logical structure of test cases described in Section 3.2. In

testing frameworks like JUnit the logical structure is mapped to the corresponding `setup()`, `test()` and `assert()` methods. Testing frameworks also distinguish a cleanup (`tearDown()`) test part that follows test execution.

Test frameworks enforce execution order between `setup()`, `tearDown()` and `test()` methods as shown in Figure 4.6. This control structure of the test cases allows to determine the execution order in the instantiation and execution sequences that we extract from test classes.

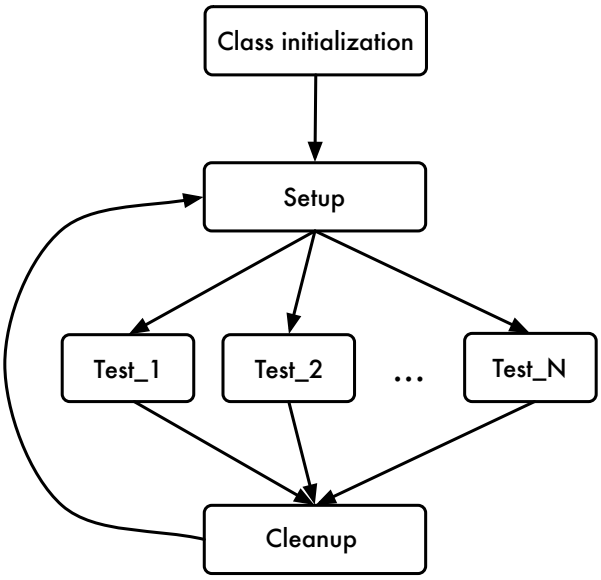

Figure 4.6. Test execution order

An *instantiation sequence* includes all the method invocations and the object instantiations that occur in a test case, and that contribute to instantiate a certain class.

An instantiation sequence includes the statements that directly instantiate the objects required for the test execution, the method invocations of these objects, as well as the statements that are involved in the instantiations indirectly, i.e., for which there is a dependency relation with the direct instantiations. For example a statement that instantiates an object required for testing may need some value produced by another statement, which, as such is not directly involved in the instantiation, but must be included in the sequence. We identify all the statements that belong to the instantiation

sequences by means of dominance data flow analysis on the test cases described in the next section.

In Figure 4.5, the instantiation sequence for the class `CompilationUnitBuilder` is marked as I1. Class `CompilationUnitBuilder` requires instances of classes `JavaSrcRepository` and `Qualifier` to be instantiated. `JavaSrcRepository` class also requires another `ClassRepository` class to be instantiated and its instantiation (line 2) is included in the sequence I1. Moreover, method `setPackage()` (line 8) is invoked on the `Qualifier` class instance before `CompilationUnitBuilder` instantiation, and it is added to the instantiation sequence I1.

> An *execution sequence* includes all the method invocations on the class instance of interest and captures the scenario of usage of this class.

In Figure 4.5, the execution sequences for the class `CompilationUnitBuilder` are marked with E1 and E2. Both sequences correspond to the sets of statements that belong to single test methods. Since the execution order between single test methods is unknown we build execution sequences considering the sets of statements in different test methods separately. The execution sequence E3 for the class `Qualifier` spans two methods: scaffolding method `setup()` and test method `testInnerClass()`. Statements in these methods are considered together to construct execution sequence, because there is a known execution order between `setup()` and `test()` methods.

Similarly to instantiation sequences, execution sequences may include statements not directly involved in method invocations, but required for the invocations. For example, the execution sequence on class `Message` in Figure 4.7 includes method invocations on `ms` object (lines 7, 8, and 12). These method invocations require other data and corresponding statements are underlined in the figure, and belong to the execution sequence E1. We identify these statements with dominance data flow analysis as for the instantiation sequences.

Instantiation and execution sequences contain not only the statements involved in object instantiations and method invocations, but also the context data: statements that produce data values used in method invocations, the referenced inner classes and the static fields. For instance, for the test case in Figure 4.8 we capture inner class `Medium` referenced in `testMediumCost1()` test method as a context data. We use syntactic analysis to extract corresponding data values and declarations, and we associate these data with the corresponding sequences.

```
1  public class MessageTest extends TestCase {
2  E1 private File f = new File(System.getProperty("java.io.tmpdir"), "message.txt");
3
4  public void testPrintStreamDoesNotGetClosed() {
5   Message ms = new Message();
6  E1 Project p = new Project();
7  E1 ms.setProject(p);
8  E1 ms.addText("hi, this is an email");
9  E1 FileOutputStream fis = null;
10   try {
11 E1  fis = new FileOutputStream(f);
12 E1  ms.print(new PrintStream(fis));
13     fis.write(120);
14   } catch (IOException ioe) {
15     fail("we should not have issues writing after having called Message.print");
16   } finally {
17     FileUtils.close(fis);
18   }
19  }
20 public void tearDown() {
21   if (f.exists()) {
22     FileUtils fu = FileUtils.getFileUtils();
23     fu.tryHardToDelete(f);
24   }
25  }
26 }
```

Figure 4.7. Execution sequence for class `Message`

Data flow analysis

We extract instantiation and execution sequences from existing test cases using dominance data flow analysis. Our algorithms for data flow analysis statically identify relevant statements for each system class that is used in test classes and aggregate these statements in corresponding instantiation and execution sequences.

Based on our study of existing test cases (Section 3) we make a number of assumptions on test case structure that enable our data flow analysis and make it lightweight. Our investigation of test cases indicates that the majority of test code is linear (Section 3.3). Test code rarely contains loops or branching statements. This allows us to exclude analysis of control dependence in the test code and only derive control dependence imposed by the testing frameworks.

Our data flow analysis of test cases without control dependence is analogous to a *local analysis* – analysis of single basic blocks of source code. However, the scope of these analyses differs. Single basic blocks in source code are usually short, while data flow analysis of test cases tackles longer control dependence-free blocks of code of a size of test classes.

Our data flow analysis is intra-procedural despite the majority of test code being organized in classes and test methods. This is because the majority of test cases exhibit

```
1  public class MetricComputerTest extends TestCase {
2      private MetricComputerJavaDecorator computer;
3      private final ClassRepository repo = new JavaClassRepository();
4
5      protected void setUp() throws Exception {
6          super.setUp();
7          RegExpWhiteList regExpWhitelist = new RegExpWhiteList("java.");
8          regExpWhitelist.addPackage("javax.");
9          MetricComputer toDecorate = new MetricComputerBuilder().withWhitelist(regExpWhitelist
       )
10         .withClassRepository(repo).build();
11         computer = new MetricComputerJavaDecorator(toDecorate, repo);
12     }
13     public void testMediumCost1() throws Exception {
14         MethodInfo method = repo.getClass(Medium.class.getCanonicalName())
15           .getMethod("int␣statiCost1()");
16         assertFalse(method.canOverride());
17         MethodCost cost = computer.compute(Medium.class, "int␣statiCost1()");
18         assertEquals(1l, cost.getTotalCost().getCyclomaticComplexityCost());
19     }
20     // inner class
21     public static class Medium {
22         public Medium() {
23             statiCost1();
24             cost2();
25         }
26         public static int statiCost1() {
27             int i = 0;
28             return i > 0 ? 1 : 2;
29         }
30         /*...*/
31     }
32 }
```

Figure 4.8. Test case with inner class

neither inter-class, nor inter-procedural dependencies and can be dealt with as separate methods. Test cases that have inter-class and inter-procedural dependencies usually depend on utility methods and external scaffolding. Such external code can often be merged in the test code enabling intra-procedural analysis.

Our analysis suffers from the same limitations as classic data flow analysis. In particular, it does not deal completely with data flow information that may derive from side effects.

Algorithms To extract *instantiation sequences* we developed an algorithm *ExtractInstSeq* shown in Figure 4.9. The algorithm statically identifies all variables *used* in a constructor call instantiating a certain class. It recursively identifies a set of variables necessary for the constructor invocation. For each variable in the set, the algorithm captures sequences of constructor and method invocations that define or use the vari-

able, considering the order of execution between test class fields, `setup()` and `test()` methods imposed by testing framework.

To extract *execution sequences* we developed an algorithm *ExtractExecSeq* shown in Figure 4.11. The algorithm identifies the constructor call instantiating a certain class as the algorithm for extracting instantiation sequences, and identifies definitions and uses of the variable *defined* by the constructor call. It recursively identifies variables and parameters required for the successive method invocations in the same way as the algorithm *ExtractInstSeq* identifies required constructor and method invocations for instantiation sequences.

The interpretation of variable *definitions* and *uses* in the algorithms reflects the goal of our analysis. The primary goal of algorithms *ExtractInstSeq* and *ExtractExecSeq* is to extract sequences of constructor and method invocations relevant to class instantiation and scenarios of class usage. In particular, *ExtractInstSeq* extracts minimal sequences required to instantiate classes, and *ExtractExecSeq* extracts complete sequences of method invocations on class instances.

We label def_{var} an assignment to a variable var or method invocation on an object pointed by variable var. We assume that method invocation of an object can modify its state and is of interest to our analysis. Examples of such definitions are in lines 15 and 30 in Figure 4.10. We label use_{var} if an object pointed by variable var is used as a parameter in constructor or method invocation. Line 32 in Figure 4.10 contains uses of variables `repository` and `qualifier`. We do not capture side effects that may arise from inter-procedural dependencies.

We demonstrate the algorithm *ExtractInstSeq* for extracting instantiation sequences and the algorithm *ExtractExecSeq* for extracting execution sequences on the example in Figure 4.10. Both algorithms are applied on a complete set of statements from a given test class. The statements from original test class are merged in a single test fragment in Figure 4.10 for demonstration purposes. These statements come from `setup()`, `cleanup()` and `test()` methods of a test class and are considered together because of the known order of execution between them.

Algorithm for extracting instantiation sequences The output of the algorithm *ExtractInstSeq* shown in Figure 4.9 is a set of statements that directly or indirectly contribute to the instantiation of a certain class. They include constructor and method calls as well as auxiliary data.

The algorithm relies on several auxiliary procedures: `getStatement(def)` that selects a statement for a variable definition, `getParameterVariables(stmt)` that returns a set of variables used as parameters in a constructor of method invocation, and `getDefLocations(var)` that computes all definition locations for a variable.

The algorithm starts with identifying the constructor invocation location for the class of interest. Given the variable of a class of interest `startVar`, a procedure

Input: $startVar$
Output: OUT ▷ Set of statements that contribute to instantiate $startVar$

```
1: def_startVar ← getFirstDefLocation(startVar)
2: startStatement ← getStatement(def_startVar)
3: OUT ← OUT ∪ startStatement
4: GEN ← getParameterVariables(startStatement)
5: ANALYZED ← {}
6: for all var ∈ GEN do
7:     DFA(var, def_startVar)
8: end for

9: procedure DFA(var, limit)
10:    ANALYZED ← ANALYZED ∪ var
11:    DEF_var ← getDefLocations(var)
12:    for all def ∈ DEF_var & precedence(def, limit) do
13:        statement ← getStatement(def)
14:        OUT ← OUT ∪ statement
15:        GEN ← getParameterVariables(statement) \ ANALYZED
16:        for all var ∈ GEN do
17:            DFA(var, limit)
18:        end for
19:    end for
20: end procedure
```

Figure 4.9. Recursive algorithm *ExtractInstSeq* for extracting statements defining variables used in an instantiation of a given class

`getFirstDefLocation(startVar)` searches for a location with a `new` expression that invokes class constructor and defines the variable `startVar`. Consider the example in Figure 4.10: for `CompilationUnitBuilder` class of interest, its constructor call *defines* variable `builder` at line 32. The location marks the limit of the scope of the data flow analysis, because statements that *define* any variable contributing to the instantiation of the class of interest occur before the constructor invocation. This is controlled by the procedure `precedence(def, limit)` that checks the partial order across definition locations (line 12 of the algorithm in Figure 4.9).

Variables *used* as parameters to the constructor call constitute the initial set of objects (GEN) that are needed to instantiate the class of interest. In the example, variables `repository` and `qualifier` belong to this set. For each variable in the GEN set the algorithm invokes the `DFA()` procedure that captures all *definitions* of this variable and records in the OUT set corresponding statements where *definitions* occur. For instance,

```
1  /**
2   * A fragment of a test case as seen in the data-flow analysis: test class instantiation,
3   * setup and test methods are merged in one procedure.
4   *
5   * Class of interest: CompilationUnitBuilder;
6   * startVar = builder (line 32);
7   * GEN={parent, repository, qualifier}
8   * OUT={15,17,19,28,30} // line numbers of statements in the OUT set.
9   * Statements on lines 34-38 do not belong to the OUT set, because they appear after first
10  * definition of startVar (line 32).
11  * Statements on lines 23-24, 26 are not in the OUT set, although they use variable parent,
12  * but they do not contribute to instantiate CompilationUnitBuilder - the variables
13  * they define are not used in the statements of interest.
14  */
15   ClassRepository parent = new JavaClassRepository();
16     // DEF={parent}     USE={}
17   JavaSrcRepository repository = new JavaSrcRepository(parent, null);
18     // DEF={repository} USE={parent}
19   Qualifier qualifier = new Qualifier();
20     // DEF={qualifier}  USE={}
21   CompilationUnitBuilder builder;
22   MetricComputerDecorator decoratedComputer;
23   MetricComputer toDecorate = new MetricComputerBuilder().
24         withClassRepository(parent).build();
25     // DEF={toDecorate} USE={parent}
26   decoratedComputer = new MetricComputerDecorator(toDecorate, parent);
27     // DEF={decoratedComputer} USE={toDecorate, parent}
28   ClassInfo myInfo = parent.getClass(My.class.getName())
29     // DEF={myInfo, parent} USE={}
30   qualifier.setPackage("pkg");
31     // DEF={qualifier}  USE={}
32   builder = new CompilationUnitBuilder(repository, qualifier, "");
33     // DEF={builder}    USE={repository, quialifier}
34   qualifier.addAlias("B", "pkg.A$B");
35     // DEF={qualifier}  USE={}
36   Type type = JavaType.fromJava(My.class);
37     // DEF={type}       USE={}
38   builder.startType(0, "A", type, new ArrayList<Type>());
39     // DEF={builder}    USE={}
```

Figure 4.10. Intermetiate data and statements captured with the data flow analysis for instantiation of class `CompilationUnitBuilder`

for the `qualifier` variable `DFA()` identifies the statements that *define* the variable in lines 19 and 30, and for the `repository` variable `DFA()` identifies statement that *defines* the variable in line 17. The `DFA()` procedure also captures all the variables *used* as parameters in the identified statements. These variables are added to the *GEN* set and are recursively analyzed in the corresponding `DFA()` execution. In other words, `DFA()` is invoked *recursively* for each variable *used* in method or constructor calls that *define* the variables required to instantiate the class of interest. For instance, variable `repository` is *defined* in line 17 and variable `parent` is *used* in this statement. Thus, the algorithm proceeds recursively by adding variable `parent` in the *GEN* set and analyzing it in the

Input: $startVar$
Output: OUT
1: $def_{startVar} \leftarrow getDefLocation(startVar)$
2: $OUT \leftarrow \{\}$
3: $ANALYZED \leftarrow \{\}$
4: DFA-EXEC$(startVar, def_{startVar})$

5: **procedure** DFA-EXEC$(var, lowerLimit)$
6: $DEF_{var} \leftarrow getDefLocations(var)$
7: **for all** $def \in DEF_{var}$ & $precedence(def, lowerLimit)$ **do**
8: $statement \leftarrow getStatement(def)$
9: $OUT \leftarrow OUT \cup statement$
10: $GEN \leftarrow getParameterVariables(statement) \setminus ANALYZED$
11: **for all** $var \in GEN$ **do**
12: $def_{var} \leftarrow getDefLocation(var)$
13: DFA(var, def_{var}) ▷ `DFA()` procedure in *ExtractInstSeq* in Figure 4.9
14: **end for**
15: **end for**
16: **end procedure**

Figure 4.11. Recursive algorithm *ExtractExecSeq* for extracting method invocation statements on a given class

`DFA()` procedure. As a result, `DFA()` captures the statements in lines 15 and 28 for variable `parent`.

The *ExtractInstSeq* algorithm applied to the example in Figure 4.10 for a class `CompilationUnitBuilder` produces the output represented by a sequence of statements $OUT = \{15, 17, 19, 28, 30\}$ where each statement contributes to instantiate the `builder` object.

Algorithm for extracting execution sequences The algorithm *ExtractExecSeq* shown in Figure 4.11 extracts execution sequences for a given class. *ExtractExecSeq* captures all the statements that *follow* the instantiation of a class of interest and *define* the instance of the class. For each statement that *defines* the class instance the algorithm also captures all the statements that contribute to instantiate and *define* their parameters using the recursive `DFA()` procedure of *ExtractInstSeq* algorithm. The algorithm relies on the same auxiliary procedures as *ExtractInstSeq* algorithm.

Consider for example class `CompilationUnitBuilder` in Figure 4.10. It is instantiated with the constructor call at line 32. The statement at line 38 follows `CompilationUnitBuilder` instantiation and *defines* the instance of the class with the `startType()`

method call. In its turn method `startType()` *uses* `type` object among its parameters. The algorithm adds `type` variable to the *GEN* set and invokes the `DFA()` procedure on it. This results in the statement on line 36 added to the resulting execution sequence $OUT = \{ 36, 38\}$.

The complexity of the algorithms presented in this section is of the same order, $O(n \cdot log(n))$ in the worst case.

As a result of this phase, for each class of the system we capture instantiation and execution sequences. We augment the system ORD with these sequences and with the corresponding context data.

4.4 Generating test cases

In the last phase of the approach we identify sets of dependent classes from the ORD generated in the first phase, and for each set of classes we suitably combine the sequences identified in the second phase to generate new test cases. This is an iterative process articulated in two steps.

Step 1

We first identify dependent classes we want to integrate using ORD. Each node in ORD that has outgoing arcs represents a class that directly depends on other classes for being constructed (a corresponding class constructor manifests its dependencies through constructor parameters).

Using ORD we can identify directly related classes and classes related through the intermediary classes, when an intersection of pairs of transitively dependent sets of classes is non-empty. For instance, in ORD in Figure 4.12 classes A and B are directly related, while classes $\langle A, E\rangle$ are related through class $subB$, and classes $\langle B, E\rangle$, $\langle A, E\rangle$, and $\langle subB, E\rangle$ are related through class C. Both direct and indirect class relations shall be tested to stress possible interactions between dependent classes.

In our study of existing test cases (Section 3) we have observed that unit test cases mainly exercise direct class relations. They test class interactions, where one class changes its state, while other dependent classes represent test scaffolding and rarely change their state. At the same time, unit test cases do not test many potential class interactions involving multiple state transitions for the sets of directly dependent classes, as well as for sets of classes related through the intermediary ones. Our approach focusses on exercising direct class dependencies, and can be easily extended to deal with classes related through the intermediary ones.

To select directly related classes we traverse the ORD and select incident pairs of nodes. Consider, for example, the ORD in Figure 4.12. Starting from the class *A*, the first

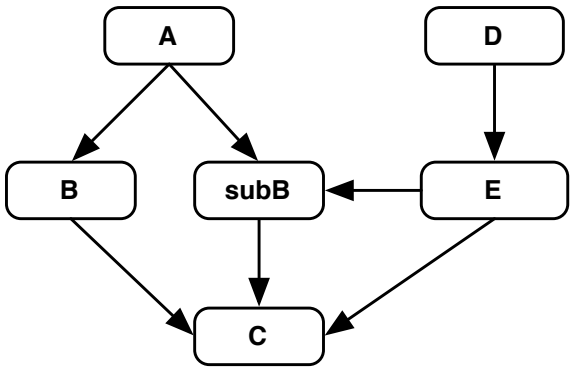

Figure 4.12. An example of ORD

pair of dependent classes is $\langle A, B \rangle$, where A requires and instance of B to be constructed (alternatively, A requires an instance of $subB$, where $subB$ is a subclass of B).

We then integrate directly related classes by suitably combining their instantiation sequences and generating a new instantiation sequence. We start with the instantiation sequences of a class at the top of the dependency hierarchy in the selected subset of dependent classes, and we incrementally add instantiation sequences for the dependent classes in the order of occurrence in the original test cases.

For example, to integrate classes A and B we first select any of their instantiation sequences excerpted from test cases. We do not use A's instantiation sequence directly but we modify it and provide it with the new instance of B instantiated from the B's selected sequence. To that end we substitute the original instance of class B in the A's sequence invoking A's constructor and providing it with the new instance of B. Instantiating dependent class from the existing instantiation sequence allows us to obtain a correctly instantiated object that can instantiate auxiliary classes. As seen from the ORD, B depends on C and B's instantiation sequence should contain instantiation of class C.

Simple unit test cases tend to use minimal initialization of classes, often with *null* references instead of objects, and may not fully instantiate dependent classes. We adapt instantiation sequences and substitute class instances used in original sequences to the instances of the dependent classes that have a compatible type. This way we create a new instantiation sequence that instantiates and wires selected set of dependent classes, and we explore new, possibly untested combinations.

Step 2

In the first step we integrate clusters of classes by merging instantiation sequences. In this step we complete the test cases by adding execution sequences. We iteratively combine the execution sequences of the considered classes to trigger different interactions between collaborating classes. Class interactions can either follow the initialization of collaborating classes or take place during the initialization. Thus, we merge the execution sequences of the classes involved in the interactions both after the instantiation sequence and interleaved with it, following the data precedence relation among sequences that we infer in the former phases.

Precedence relations Sequences can be combined differently depending on the possible orders among sequences. We identify three types of arrangements of sequences: (1) based on the instantiation order of dependent classes; (2) based on the order between sequences of a certain class; (3) based on the order across sets of sequences appended previously.

1. Arranging execution sequences following the class instantiation order, while sequences may interleave instantiation sequences:

```
// Legend:
// A - instantiation sequence for class A
// a1 - execution sequence for class A

ABC a1 b1 c1 // follow instantiation order
A a1 B b1 C c1  // follow instantiation order
A a1 a2 B b1 b2 C c1 c2 ... // follow instantiation order; sequences between
     instantiations
A a1 B b1 C c1 a2 b2 c2 ... // follow instantiation order; sequences after all
     instantiations
ABC c1 b1 a1 // opposite to instantiation order
```

   We observed in existing test cases that the most common order of method invocations corresponds to and follows the order of class instantiation. It represents the case when the class instantiated last depends on the states of the objects used for its instantiation, moreover it requires certain states of these objects for method invocation. In this case, the classes instantiated first are also subjects to method invocations in the order of class instantiation.

2. Arranging sequences based on the order between sequences of a certain class:

```
ABC a1 a2 b1 b2 c1 c2 ... // for each class a new sequence is after the previous one
ABC c1 c2 b1 b2 a1 a2 ... // for each class a new sequence is after the previous one
ABC a2 a1 b2 b1 c2 c1 ... // for each class a new sequence is before the previous one
ABC c2 c1 b2 b1 a2 a1 ... // for each class a new sequence is before the previous one
```

   This arrangement aims to maximize the length of sequences for each dependent class, when sequences can be iteratively appended extending sequences of a given class already in the test.

3. Arranging sequences based on the order between sets of sequences:

```
ABC a1 b1 c1 a2 b2 c2 ... // set '2' is after the previously added set '1'
ABC c1 b1 a1 c2 b2 a2 ... // set '2' is after the previously added set '1'
ABC a2 b2 c2 a1 b1 c1 ... // set '2' is before the previously added set '1'
ABC c2 b2 a2 c1 b1 a1 ... // set '2' is before the previously added set '1'
```

   In this case the order is determined between sets of sequences and not between individual sequences. This arrangement aims to alternate groups of sequences, where each group corresponds to a certain combination of possible class interactions for a set of dependent classes.

Different arrangements of executions sequences allow to construct interleaving combinations of states for dependent classes that aim to highlight interaction faults.

Data relations Additionally to the integration of classes through instantiation sequences described in Step 1, we aid further class integration and interactions by exploiting data relations across execution sequences: we reuse auxiliary objects and data across sequences in method and constructor parameters.

Initially, the method parameters in the sequences in the generated test cases use objects and data observed in the corresponding original test cases. In our approach we use parameter objects and data of matching types to integrate new classes in the generated test case or to integrate parts of the generated test case. To that end we substitute actual parameters of method invocations for class instances of compatible type observed in the preceding method invocations of previously appended sequences.

Integration strategy

We develop an *integration strategy* that generates test cases with different order of execution sequences to explore the maximal number of the possible combinations of class states and transitions for dependent classes. We maximize the combination of class states, because integration faults may be hidden by the state-dependent behavior. New combinations of classes aim to discover *dynamic mismatches*, when polymorphic calls may be dynamically bound to incompatible methods causing integration faults. We maximize the length of the execution sequences, because long execution sequences were shown to be more effective in revealing integration faults [MPP07].

Our definition of integration strategy differs from the integration test order strategies that are commonly used in integration testing [BLW03]. These strategies generate test order trying to minimize stubbing of the untested components. In contrast, our integration strategy aims to explore possible class combinations and interactions through different combinations of sequences.

In a nutshell, our integration strategy works as follows. For each pair of classes c_j and c_k such that c_k depends on c_j and is instantiated after c_j, the execution sequence of e_{c_j} for the class c_j can be appended after the instantiation of c_j and before the

instantiation of c_k (seeking class interactions during the instantiation of the class c_k), or after the instantiation of c_k and before or after the e_{c_k} execution sequence (seeking class interactions for the two combinations of states for instantiated classes c_j and c_k).

We have developed and verified integration strategy that represents our experience and intuition towards the causes and effects of class interactions for generating complex test cases.

The intuition for our integration strategy comes from the observation of many existing unit test cases. We have observed that classes instantiated last depend on the state of the classes they use for instantiation. Thus, in this strategy we append sequences in the order of class instantiation:

```
ABC a1 b1 c1 ...
```

The execution sequences observed in unit test cases tend to be quite short, and unit test cases usually test simple behaviors. Thus, we assume that each execution sequence corresponds to a single state transition in a class. To better test the different combinations of class state transitions we conjecture that state changes can be differently distributed between interacting classes when sets of sequences are appended one after another or when sequences intersect. In the integration strategy we append set of sequences after the previously appended set on each iteration up to the predefined limit:

```
ABC a1 b1 c1 a2 b2 c2 ...
```

We exploit data relations within the test case and we integrate new objects in the sequences by substituting system classes passed as method parameters in the execution sequences. We select and add instantiation sequences for classes of types matching the parameters in method interfaces and substitute actual method parameters with the new class instances. We exploit data relations and we reuse auxiliary objects and data across sequences.

We extend sequences only for the initial set of integrated classes thus focussing on their interactions. We do not append nor extend sequences on the class instances used as parameters for the method calls on the integrated classes. We iteratively select and combine execution sequences, and for each iteration we take into account the constrains that derive from the sequences already appended and their relative positions. Once we produce new longer test cases, we can use these test cases as input to the process to produce even more complex test cases.

Our approach automates the combination of instantiation and execution sequences, but does not deal with assertions yet. Assertions in unit test cases are usually partial comparison-based oracles. Only a small fraction of these assertions can be reused in generated test cases without adaptation. Additional heuristics are necessary to enable the adaptation and import of assertions taking into account state-dependent behavior

of classes. In our experiments to reveal failures, we rely on exceptions thrown during the executions.

4.5 Example

We present the core idea of our approach through a working example, a simple integration fault in *JFreeChart*[2], a professional chart generation library. We show that the fault depends on the interaction between several modules, and as such can be hardly detected with simple test cases that check the behavior of single units. In fact, non of the unit test cases provided with *JFreeChart* exercises the faulty behavior nor detects the fault, despite mature test suite and continous project support and testing. We illustrate the relations between an integration test case that reveals the fault and the unit test cases for the involved modules, and use the example to illustrate different phases of the approach.

The *JFreeChart* library provides functionality to create charts with multiple composite titles that can be placed at the top, bottom, left or right of the chart. Composite title structure follows *Composite* design pattern as shown in a fragment of *JFreeChart* class diagram in Figure 4.4. Titles can be composed of a number of blocks that represent arbitrary items that can be drawn. To hold a collection of blocks `CompositeTitle` class uses an instance of a container class `BlockContainer` that aggregates `Block` objects. To correctly position each `Block` object within a composite title `BlockContainer` uses different `Arrangement` classes.

The described functionality of *JFreeChart* conceals integration problem that is related to the arrangement of the elements in the composite chart title. Positioning of the blocks of `CompositeTitle` using `ColumnArrangement` triggers exceptional condition: invocation of an `arrange()` method of class `CompositeTitle` fails with an uncaught exception if `CompositeTitle` contains `BlockContainer` with `null` objects of class `Block`. This behavior contradicts the specification for the class `BlockContainer` that permits `null` objects of class `Block`.

The integration fault originates in the `ColumnArrangement` class that fails to handle `null` objects, while such objects are permitted in the classes `CompositeTitle` and `BlockContainer` that interact with the `ColumnArrangement`.

Simple test cases Figures 4.13 and 4.14 show the excerpt of some unit test cases for the `BlockContainer` and `CompositeTitle` components. These test cases exercise short sequences of method invocations on individual classes aiming to check the behavior of the single classes. They do not detect the described integration fault, because they do not test class interactions.

[2]http://jfree.org/

```
1  public void testEquals() {
2      BlockContainer c1 = new BlockContainer(new FlowArrangement());
3      BlockContainer c2 = new BlockContainer(new FlowArrangement());
4      assertTrue(c1.equals(c2));
5      assertTrue(c2.equals(c2));
6      c1.setArrangement(new ColumnArrangement());
7      assertFalse(c1.equals(c2));
8      c2.setArrangement(new ColumnArrangement());
9      assertTrue(c1.equals(c2));
10     c1.add(new EmptyBlock(1.2, 3.4));
11     assertFalse(c1.equals(c2));
12     c2.add(new EmptyBlock(1.2, 3.4));
13     assertTrue(c1.equals(c2));
14 }
```

```
1  public void testCloningBlockContainer() {
2      BlockContainer c1 = new BlockContainer(new FlowArrangement());
3      c1.add(null);
4      BlockContainer c2 = null;
5      try {
6          c2 = (BlockContainer) c1.clone();
7      } catch (CloneNotSupportedException e) {
8          fail("Failed to clone.");
9      }
10     assertTrue(c1 != c2);
11     assertTrue(c1.getClass() == c2.getClass());
12     assertTrue(c1.equals(c2));
13 }
```

Figure 4.13. Unit test cases for class `BlockContainer`

In particular, the two test cases in Figure 4.13 exercise the `BlockContainer` functionality to set arrangement and add new blocks to the container and check the equality of objects (the code corresponding to these method invocations is underlined in the figures), but they do not check the interactions with the different `Arrangement` classes. These unit test cases focus on the effects of the method invocations in the scope of the class `BlockContainer`.

Similarly, the unit test cases for the `CompositeTitle` class shown in Figure 4.14 do not check the interactions between the `CompositeTitle`, `BlockContainer` and `Arrangement` classes, although they instantiate and integrate them. None of the 2219 unit test cases distributed with *JFreeChart* reveals the integration fault illustrated at the beginning of the section.

A complex test case Figure 4.15 shows a more complex test case generated with our approach for *JFreeChart*. This test case exercises interactions between the classes `CompositeTitle`, `BlockContainer` and `ColumnArrangement`, and reveals the fault. The test case integrates `CompositeTitle` and `BlockContainer` classes. It then adds different `Block` objects to the container and sets the `ColumnArrangement`. The incorrect

```
1  public void testHashcode() {
2      CompositeTitle t1 = new CompositeTitle(new BlockContainer());
3      t1.getContainer().add(new TextTitle("T1"));
4      t1.setBackgroundPaint(new GradientPaint(1.0f, 2.0f, Color.red, 3.0f, 4.0f,
5                            Color.yellow));
6      CompositeTitle t2 = new CompositeTitle(new BlockContainer());
7      t2.getContainer().add(new TextTitle("T1"));
8      assertTrue(t1.equals(t2));
9      int h1 = t1.hashCode();
10     int h2 = t2.hashCode();
11     assertEquals(h1, h2);
12 }
```

```
1  public void testArrange(){
2      BlockContainer container = new BlockContainer(new GridArrangement(2, 3));
3      CompositeTitle title = new CompositeTitle(container);
4      Size2D s = title.arrange(null, new RectangleConstraint(200, 100));
5      assertEquals(200.0, s.getWidth(), EPSILON);
6      assertEquals(100.0, s.getHeight(), EPSILON);
7  }
```

Figure 4.14. Unit test cases for class `CompositeTitle`

implementation of `ColumnArrangement` is revealed through the interactions between the classes involved in the test. `CompositeTitle` invokes `arrange()` method. In this method an instance of the `ColumnArrangement` class accesses the blocks stored in `BlockContainer` and fails while handling `null` `Block` object.

Procedure

We now illustrate our approach with the *JFreeChart* working example introduced in the previous section, by presenting the results of the different phases when the approach is applied to the classes `BlockContainer` and `CompositeTitle`, and to the simple test cases presented in Figures 4.13 and 4.14.

Phase 1. Identifying dependencies between collaborating classes: Figure 4.2 shows an excerpt of the ORD for *JFreeChart*. The excerpt is limited to the relations that we need for the example.

The classes of *JFreeChart* are integrated through dependency injection, and manifest their dependencies in the constructor signatures. The class `CompositeTitle` depends on the class `BlockContainer`. The class `BlockContainer` directly depends on the classes implementing the `Arrangement` interface.

Phase 2. Identifying instantiation and execution sequences in existing test cases: The instantiation sequences of *JFreeChart* for `BlockContainer` include $i_{BlockContainer_1}-$

```
1  // Automatically generated test case
2  public void testCompositeTitle837() throws Exception {
3      // integrate dependent classes CompositeTitle and BlockContainer
4      BlockContainer var1739 =  BlockContainer(new GridArrangement(2, 3));
5      CompositeTitle var1740 = new CompositeTitle(var1739);
6
7      // first sequence for BlockContainer
8      BlockContainer var1731 = new BlockContainer(new FlowArrangement());
9      var1739.equals(var1731);
10     var1739.setArrangement(new ColumnArrangement()); // ColumnArrangement is faulty and
        cannot deal with nulls
11     var1739.add(new EmptyBlock(1.2, 3.4));
12     var1739.equals(var1731);
13
14     // first sequence for CompositeTitle
15     var1740.getContainer();
16     var1740.hashCode();
17
18     // second sequence for BlockContainer
19     var1739.add(null); //null Block is permitted in the specification of BlockContainer.add()
20     var1739.clone();
21     var1739.getClass();
22     var1739.equals(var1731); // substitution across sequences: used object from the first
        sequence
23
24     // second sequence for CompositeTitle
25     var1740.arrange(null, new RectangleConstraint(200, 100)); // NullPointerException in
        ColumnArrangement
26     // failure in CompositeTitle.arrange() is caused by null Block that was added to
        BlockContainer in line 19.
27     // fault is located in ColumnArrangement class
28 }
```

Figure 4.15. Integration test case exposing integration fault in class `ColumnArrangement` through interaction of classes `BlockContainer` and `CompositeTitle`

$i_{BlockContainer_3}$ and correspond to constructor invocations observed in test cases in Figures 4.13 and 4.14:

```
// i_BlockContainer_1
BlockContainer c1 = new BlockContainer(new FlowArrangement());
```

```
// i_BlockContainer_2
BlockContainer container = new BlockContainer(new GridArrangement(2, 3));
```

```
// i_BlockContainer_3
new BlockContainer()
```

Instantiation sequences for the class `CompositeTitle` include $i_{CompositeTitle_1}$ and $i_{CompositeTitle_2}$:

```
//i_CompositeTitle_1
CompositeTitle t1 = new CompositeTitle(new BlockContainer());
```

```
//i_CompositeTitle_2
BlockContainer container = new BlockContainer(new GridArrangement(2, 3));
CompositeTitle title = new CompositeTitle(container);
```

The execution sequences for `BlockContainer` correspond to the sequences of method invocations on `c1` and `c2` objects observed in the test cases shown in Figure 4.13. We can identify four execution sequences, $e_{BlockContainer_1} - e_{BlockContainer_4}$:

```
// e_BlockContainer_1
BlockContainer c2 = new BlockContainer(new FlowArrangement());
c1.equals(c2);
c1.setArrangement(new ColumnArrangement());
c1.equals(c2);
c1.equals(c2);
c1.add(new EmptyBlock(1.2, 3.4));
c1.equals(c2);
c1.equals(c2);
```

```
// e_BlockContainer_2
c2.equals(c2);
c2.setArrangement(new ColumnArrangement());
c2.add(new EmptyBlock(1.2, 3.4));
```

```
// e_BlockContainer_3
BlockContainer c2 = null;
c1.add(null);
c1.clone();
c1.getClass();
c1.equals(c2);
```

```
// e_BlockContainer_4
BlockContainer c2 = null;
c2.getClass();
```

The execution sequences for the class `CompositeTitle` correspond to the sequence of method invocations on the `title`, `t1`, `t2` objects observed in test cases shown in Figures 4.14:

```
//  e_CompositeTitle_1
title.arrange(null, new RectangleConstraint(200, 100));
```

```
//  e_CompositeTitle_2
t1.getContainer();
t1.setBackgroundPaint(new GradientPaint(1.0f, 2.0f, Color.red, 3.0f, 4.0f, Color.yellow));
CompositeTitle t2 = new CompositeTitle(new BlockContainer());
t2.getContainer().add(new TextTitle("T1"));
t1.equals(t2);
t1.hashCode();
```

```
//  e_CompositeTitle_3
t2.getContainer();
t2.hashCode();
```

We also identify data values, inner classes, static field information, and auxiliary objects used in instantiation and execution sequences:

```
// context data
new FlowArrangement()
new ColumnArrangement()
new EmptyBlock(1.2, 3.4))
new GridArrangement(2, 3)
new RectangleConstraint(200, 100)
EPSILON
new TextTitle("T1")
new GradientPaint(1.0f, 2.0f, Color.red, 3.0f, 4.0f, Color.yellow)
```

Phase 3. Combining the execution sequences: The integration strategy works in two steps. We first integrate the classes involved in the new test case by merging the corresponding instantiation sequences (step 1), and then incrementally add execution sequences to test complex object interactions (step 2). Here we illustrate the approach with two iterations of step 2.

Step 1: Integrate dependent classes.

We integrate the classes `BlockContainer` and `CompositeTitle` from the fragment of dependencies identified in *Phase 1*.

The dependency information computed in the first phase (class `CompositeTitle` depends on class `BlockContainer`) dictates the order of integration of the initialization sequences: $\langle i_{BlockContainer_2}, i_{CompositeTitle_2} \rangle$. The resulting instantiation sequence is:

```
BlockContainer var1739 = new BlockContainer(new GridArrangement(2, 3))
CompositeTitle var1740 = new CompositeTitle(var1739);
```

Step 2, Iteration 1: Incrementally extend the test case by integrating execution sequences for the involved classes.

We randomly select an execution sequence for each class involved in the test case, and we integrate them according to the dependency between the classes. In this first step, we select $e_{BlockContainer_1}$ and $e_{CompositeTitle_3}$. Since class `CompositeTitle` requires class `BlockContainer`, we first append the sequence $e_{BlockContainer_1}$ to change the state of the `BlockContainer` and possibly affect the state of the `CompositeTitle`, and then we append the sequence $e_{CompositeTitle_3}$ to invoke the methods of `CompositeTitle` in the new state.

The invocations of methods in both sequences require objects as parameters. While extending the test case with the new sequences, we need to add also the auxiliary objects required by the sequences.

```
// test case extended with e_BlockContainer_1
BlockContainer var1731 = new BlockContainer(new FlowArrangement());
var1739.equals(var1731);
var1739.setArrangement(new ColumnArrangement());
var1739.equals(var1731);
// test case extended with e_CompositeTitle_3
var1740.getContainer();
var1740.hashCode();
```

Step 2, Iteration 2: Incrementally extend the test case by integrating additional execution sequences.

We can further extend the test case by appending additional execution sequences. We obtain the following test case by incrementally appending the sequences $e_{BlockContainer_3}$ and $e_{CompositeTitle_1}$:

We add the $e_{CompositeTitle_1}$ sequence after $e_{BlockContainer_3}$, and after previously appended sequences, and thus we can reuse the objects with matching types across sequences. In $e_{BlockContainer_3}$, the object of type `BlockContainer` is used as a parameter for method `equals()` and is substituted for the one from the sequence $e_{BlockContainer_1}$.

```
// final test case incrementally extended with e_BlockContainer_3 and e_CompositeTitle_1
public void testGenerated(){
    BlockContainer var1739 = new BlockContainer(new GridArrangement(2, 3))
    CompositeTitle var1740 = new CompositeTitle(var1739);
    BlockContainer var1731 = new BlockContainer(new FlowArrangement());
    var1739.equals(var1731);
    var1739.setArrangement(new ColumnArrangement());
    var1739.equals(var1731);
    var1740.getContainer();
    var1740.hashCode();
    var1739.add(null);
    var1739.getClass();
    var1739.equals(var1731);
    var1740.arrange(null, new RectangleConstraint(200, 100));
}
```

The test case that we generate automatically reveals the fault presented in Section 4.5 and illustrated in Figure 4.15.

This example demonstrates how information captured in existing simple test cases can be reused and combined to create new test cases. Applying proposed approach one can generate complex test cases, that can potentially reveal integration faults in interacting sets of classes.

Chapter 5

Evaluation

> *To evaluate our approach we developed a prototype* Fusion*. We applied* Fusion *on open-source projects to evaluate the feasibility, the usefulness and the effectiveness of the approach. In this chapter we describe the prototype, report the results of our experiments and of the comparison of our approach with state of the art test case generation techniques. The results show that our approach generates new complex test cases and detects interaction faults differently from the state of the art techniques.*

The research hypothesis of this thesis stated in Chapter 1 is that *"test cases contain important and meaningful information; this information can be automatically captured and exploited to construct new more complex test cases."*

In Chapter 3 we have provided evidence that test cases contain important information and we have shown that this information can be captured and exploited. In this chapter we validate the last part of the research hypothesis that important information in existing test cases can be effectively reused to construct more complex and useful test cases automatically.

In our evaluation we address the following research questions:

Research Question 1: Can our approach generate new test cases from the information available in the code and the existing test cases?

Research Question 2: How useful are the generated test cases, i.e., can they reveal faults that are not detected by the original test cases, and, if so, what kinds of faults can be detected?

Research Question 3: Is our approach more effective than existing approaches to automatically generate test cases, i.e., can the generated test cases detect faults not revealed by the test cases generated with other automated techniques?

The first research question (*RQ1*) explores the feasibility of generating new test cases from the information in the source code and the existing test cases. *RQ1* evaluates the

ability of the approach presented in Chapter 4 to generate valid test cases automatically. *RQ1* also explores the applicability of the approach by indirectly answering a question whether there is sufficient information in existing test cases to generate new and valid test cases. To answer *RQ1* we quantitatively assess the amount of test cases that can be generated exploiting the information in the source code and original test cases.

The second research question (*RQ2*) explores the usefulness of generated test cases. The ultimate usefulness of a test case is its ability to reveal faults. We directly measure the fault-detection capability of test cases by executing them and classifying the detected failures and corresponding faults. We use original test cases as a baseline to measure whether our approach detects known or *new* faults.

In the third research question (*RQ3*) we assess the effectiveness of the approach. Given that our approach generates useful test cases (*RQ2*), we show its effectiveness by comparing generated test cases and corresponding faults with the test cases produced with the state of the art test generation techniques. Our approach is effective if it can detect new faults not detected by the state of the art approaches.

To address the stated research questions we evaluated our approach experimentally by means of a prototype implementation *Fusion* and we applied it on a number of open source projects with available test suites to generate new test cases.

In the next section we describe implementation details of the prototype. In Section 5.2 we present experimental setup and experimental procedure followed by discussion of the experiments and experimental results.

5.1 Prototype implementation *Fusion*

The prototype *Fusion* implements the approach to test case generation described in Chapter 4. *Fusion* works on software projects in Java and test cases in JUnit format[1]. The generation process is fully automated. *Fusion* automatically analyzes the software project to extract class dependencies. For all the classes in the project *Fusion* analyzes the corresponding test cases to extract the relevant test case fragments. It then combines these fragments according to the integration strategy, and produces valid combinations as new executable test cases in JUnit format.

Fusion is an Eclipse Plug-in developed in Java using *Eclipse JDT*[2]. Figure 5.1 shows the high-level architecture of the prototype. The prototype relies on Eclipse JDT services that require software projects to be configured without compilation errors and missing external dependencies. Software projects shall be manually configured by the user to enable analysis and test case generation.

The *Eclipse UI* component of *Fusion* handles interactions with the user. *Fusion* extends the Eclipse interface by adding a main menu item in Eclipse IDE. From this

[1] *Fusion* supports the versions JUnit3 and JUnit4

[2] `www.eclipse.org/jdt/` - Eclipse Java Development Tools

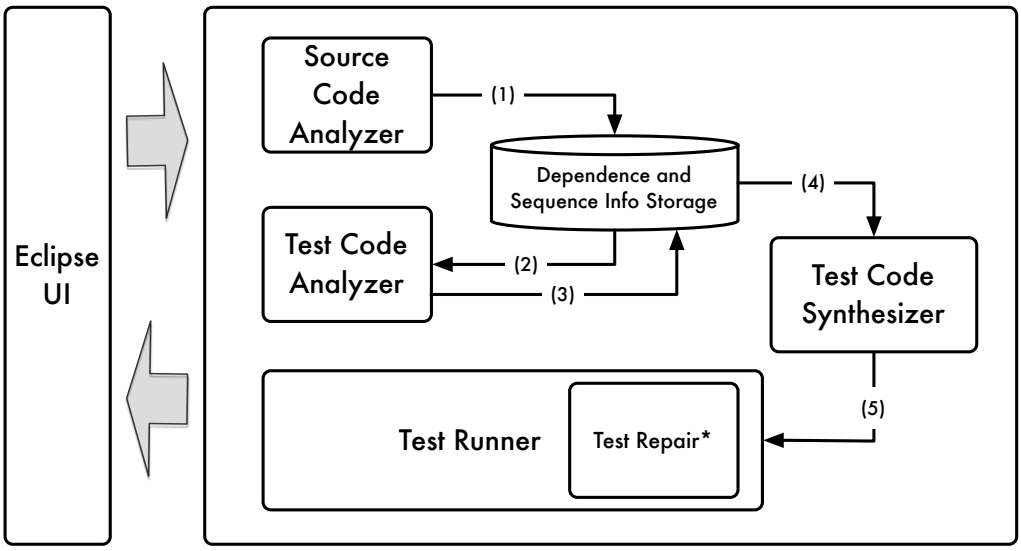

Figure 5.1. High-level architecture of *Fusion*

menu one can activate project analysis and test case generation processes. Upon completion *Eclipse UI* delivers test generation and test execution reports to the user, while Eclipse project explorer indicates the folders with generated test cases.

The *Source Code Analyzer* component analyzes system source code to capture class hierarchy information and dependencies between the classes of the system. The *Source Code Analyzer* relies on the JDT interfaces `org.eclipse.jdt.core.dom.ASTParser` and `org.eclipse.jdt.core.IJavaProject` to extract this information. It statically analyzes class constructor declarations, extracts class references from their formal parameters by parsing Abstract Syntax Trees (AST) of the system classes.

The *Source Code Analyzer* aggregates the results of the analysis in a directed dependence graph that it stores in a *Dependence and Sequence Info Storage* component. Graph storage and traversal mechanisms use the *JGraphT* graph manipulation library[3]. Dependence graph represents Object Relation Diagram (ORD) of the system introduced in Section 4.2. The dependence graph records references to the public constructors of the system classes as its nodes, and dependences between these constructors as its directed edges.

The *Test Code Analyzer* component identifies public constructor invocations in test cases using the Eclipse Search Engine (`org.eclipse.jdt.core.search.SearchEngine` interface). The *Test Code Analyzer* searches for constructor references using the constructor signatures stored in the *Dependence and Sequence Info Storage* component.

The *Test Code Analyzer* implements the data flow analysis algorithms introduced in Section 4.3. For each JUnit test class with detected system class constructors the

[3] `www.jgrapht.org`

component applies data flow analysis to identify the test case fragments relevant to class instantiation or class usage. Each fragment is then associated with the corresponding node of the ORD and stored in the *Dependence and Sequence Info Storage*.

The data flow analysis for extracting relevant test case fragments is implemented on top of the Eclipse interfaces for AST representation with Bindings (`org.eclipse.jdt.core.dom.AST` and `org.eclipse.jdt.core.dom.IBinding` interfaces). Bindings provide contextual information and alias resolution. Bindings contain resolved information about type, method or variable references that allows to identify these references uniquely.

Data flow analysis is based on a recursive traversal of test class AST. To collect an instantiation sequence for a given constructor invocation location, the *Test Code Analyzer* statically traverses AST and collects statements that operate on variables that contribute directly or transitively to the actual arguments in constructor invocation. To collect an execution sequence *Test Code Analyzer* records sequences of statements that operate on variable instantiated in a given constructor call. For method invocations on the variable of interest *Test Code Analyzer* records their actual arguments and captures statements that operate on variables that contribute to the arguments directly or transitively.

Each set of statements in instantiation and execution sequences is captured as an AST subtree that can be replicated and manipulated. The *Test Code Analyzer* augments the nodes of the ORD in the *Dependence and Sequence Info Storage* with references to the corresponding instantiation and execution sequences, that is, to the corresponding AST subtrees.

Test Code Synthesizer implements the integration strategy with the incremental pairwise combination of classes described in Section 4.4 and combines captured test case fragments into new JUnit test cases. Test cases are composed into individual JUnit test methods in JUnit test classes for each system class. The *Test Code Synthesizer* uses Eclipse AST manipulation mechanism to prepare the test case skeletons and fill them by creating new code entities (for instance, using method `AST.newMethodDeclaration()`) or by copying existing subtrees with `ASTNode.copySubtree()` methods.

The *Test Code Synthesizer* generates new test cases for all the system classes for which the *Source Code Analyzer* allocated the corresponding nodes in the ORD. The *Test Code Synthesizer* traverses the ORD and, for each node, identifies pairs of directly dependent classes and corresponding instantiation and execution sequences. Sequences are selected pseudo-randomly from all the sequences in ORD related to the constructors of matching type.

Based on the direction of dependency between edges in the ORD, the *Test Code Synthesizer* integrates instantiation sequences of the dependent classes. The classes are integrated by substituting instances in the original code fragments with instances of the dependent classes. Then the *Test Code Synthesizer* adds selected execution sequences of dependent classes in a number of iterations with practically determined limit. Execution sequences of subordinate classes are appended before execution

sequences of a predominant class. Auxiliary class instances of matching types used in method invocation statements are reused across sequences through object substitution. The *Test Code Synthesizer* automatically resolves external class dependencies, naming conflicts and import declaration conflicts.

The *Test Runner* component compiles the generated test cases, optionally resolves compilation errors relying on the *Test Repair* component, executes the test cases, and produces reports. *Test Runner* uses native Eclipse JUnitTestRunner (`org.eclipse.jdt.junit.JUnitCore` interface) to produce test execution report and classify passing and failing test cases. For test repair it uses the special Eclipse compilation error marker mechanism (interface `org.eclipse.jdt.core.IJavaModelMarker`) to relate compilation errors to source code locations. Should compilation errors occur, *Test Runner* filters out non-compiling test cases, and automatically repairs errors related to conflicting class imports in the *Test Repair* component.

5.2 Experimental setup

Case study subjects

To evaluate our approach we selected four case studies from distinct program domains with available test suites. All subjects are open-source projects with publicly available version control and bug tracking repositories. All projects contain manually written test suites and are actively supported by software developers.

We selected four subject programs developed in Java: *TestabilityExplorer*[4], a source code analyzer, *JGraphT*[5], a library that supports graph theory objects and algorithms, *Apache Ant*[6], a Java library and a state of the art build tool, and *JFreeChart*[7], a professional chart generation library.

Program (version)	*LOC*	*unit test cases, LOC*	*test coverage (line - branch)*
TestabilityExplorer (1.3.2)	8214	5596	81% - 66%
JGraphT (0.8.3)	12207	5637	70% - 63%
Apache Ant (1.8.4)	104307	24384	48% - 42%
JFreeChart (1.0.14)	93460	49644	56% - 46%

Table 5.1. Subject programs with unit test cases

Table 5.1 presents some details about the complexity of the case studies: number of lines of source code (LOC), number of lines of test source code (LOC), statement

[4] `http://code.google.com/p/testability-explorer/`
[5] `http://jgrapht.org/`
[6] `http://ant.apache.org/`
[7] `http://jfree.org/`

and branch coverage metrics for the available test suites. These data are collected with *Google CodePro AnalytiX* and *Cobertura*[8] Eclipse plugins.

Case studies are small to medium size applications from 8 KLOC to 105 KLOC. All case studies come with mature test suites of medium to high test coverage. Most of the subjects have been used in a number of research papers as case studies.

Evaluation procedure

We applied our approach on each case study to generate new test cases from the test cases available with the distribution of the applications. We executed both the original and the new test cases. Then we inspected the generated test cases and the revealed faults both to check whether the generated test cases find new faults and to classify the type of faults revealed with the new test cases.

We compared the test cases generated with *Fusion* with the test cases generated with state of the art test generation tools that work with Java programs. In particular we compared our results with the results obtained with *Palus* [ZSBE11] and *Randoop* [PLEB07] on the same four case studies.

We executed the test cases generated with the different approaches and inspected the corresponding faults to measure the effectiveness of our approach compared with the selected approaches. We did not compare *Fusion* with other approaches, and in particular with *MSeqGen* [TXT+09b] and *Seeker* [TXT+11], because they work only with .NET programs.

We performed our experiments on two configurations: a *Desktop* configuration consisting of a 2.53 GHz Intel Core i5 Processor with 4GB of RAM running Mac OS X Lion 10.7.4 using Java 6, and a *Server* configuration consisting of a 16 Core machine with 2.53GHz Intel Xeon Processors with 16GB of RAM running Ubuntu 12.04.1 LTS. We used *Desktop* configuration to run all the selected tools. In case of *Palus* we used *Server* configuration to run resource-intensive analysis.

5.3 Applicability and feasibility

To answer *RQ1*, we ran our prototype on the subject programs. Table 5.2 shows the number of test cases that we generated with *Fusion* and the execution time of the prototype on the *Desktop* configuration. We can see that the *Fusion* execution time is quite small with an average of about 8 minutes to analyze an application, and that it generates many new test cases, with an average of 671 test cases per program. Figure 4.15 in Section 4.5 shows an example of a test case automatically generated with *Fusion* for *JFreeChart*.

[8] `http://cobertura.sourceforge.net/`

Program	*No. of Generated test cases*	*Execution time (min.)*
TestabilityExplorer	420	1
JGraphT	375	1
Apache Ant	540	6
JFreeChart	1350	24

Table 5.2. Test cases generated with *Fusion* and execution time on the *Desktop* configuration

About 60% of the generated test cases compile and execute immediately, while the rest must be slightly modified to become executable. The execution problems of the generated test cases do not depend on the approach, but are due to the limitations of the current prototype. Most of the modifications required to fix the test cases can be avoided with minor modifications of the prototype currently under implementation.

We can then answer positively to the research question *RQ1*: our approach generates many new executable test cases from the information available in the code and the existing test cases.

5.4 Usefulness

To answer the second research question *RQ2*, we executed both the original and the generated test cases, and we inspected the revealed faults. Not surprisingly, the original test suites do not reveal faults in the programs, since the case studies are stable programs. The test cases generated with our approach detected *10 new faults*, out of which 6 faults are integration faults that involve from 2 to 4 classes. Strictly speaking, the other 4 faults are unit faults, since they involve only one class, but represent corner cases hard to detect even with good unit test cases and that were not detected by the original test suites.

We have contacted the application developers, who confirmed that we found new faults. The last column of Table 5.3 shows the faults revealed by the generated test cases and the false positives, i.e., test cases that raise exceptions that are considered correct by the developers.

Most of the test cases that resulted in false positives were valid sequences of method invocations leading to the situations anticipated by developers that raise checked exceptions.

Figure 5.2 shows an example of a test case generated with *Fusion* that detects unit fault. The test case detected a corner case unit fault in *JGraphT*: A second call to a public method `generateGraph()` triggers runtime exception. The fault is located in the scope of one class `GraphReader`. Its method `generateGraph()` invokes a private method `readNodeCount()` to allocate memory for the vertices for the graph. The internal procedure in the method `readNodeCount()` parses input with a graph description to

```
1  public void testGraphReader205() throws Exception {
2    String var761 = "p␣3\ne␣1␣2␣.5\ne␣1␣3␣7\n";
3    GraphReader<Integer, DefaultWeightedEdge> var762 = new GraphReader<Integer,
        DefaultWeightedEdge>(
4        new StringReader(var761), 1);
5    Graph<Integer, DefaultWeightedEdge> var763 = new SimpleWeightedGraph<Integer,
        DefaultWeightedEdge>(
6        DefaultWeightedEdge.class);
7    VertexFactory<Integer> var764 = new IntVertexFactory();
8    var762.generateGraph(var763, var764, null);
9    var762.generateGraph(var763, var764, null); // fails with runtime exception
10 }
```

Figure 5.2. Test case generated with *Fusion* for *JGraphT* that detects corner case unit fault

allocate the memory. This procedure accesses a memory location with `null` reference and fails with runtime exception. The fault is related to the fact that `readNodeCount()` method does not check memory allocation and fails due to the state-dependent behavior of the underlying input reader.

Most of the *integration faults* were found by the test cases of two categories. The *first category* of test cases are cases that contain sequences of method invocations longer than average sequences in original test cases. These test cases detected untested class interactions resulting from specific combinations of method invocations.

An example of such test case for *JGraphT* is shown in Figure 5.3. Detected integration fault affects general graph behavior and depends on integration of two classes `ListenableDirectedGraph` and `DirectedNeighborIndex`. In the test case a graph `ListenableDirectedGraph` is associated with a listener `DirectedNeighborIndex` that is associated with another graph. The failure happens when test case adds a new edge in the `ListenableDirectedGraph`. Method `addEdge()` raises unexpected exception `IllegalArgumentException` “no such vertex in graph”, although the vertex exists. In this situation the graph behavior should not change according to its specification. The fault depends on the fact that method `addGraphListener()` of the class `ListenableDirectedGraph` does not check whether added listener is already associated with another graph. Specific combination of class instantiations and method invocations in this test case allows to reveal the fault.

The *second category* of test cases contains short sequences of method invocations (2-4 invocations) with specific combinations of class instantiation and method parameter values. These test cases detected untested combinations of class aggregations that exposed failures with few method invocations.

Figure 5.4 shows an example of such a short test case generated with *Fusion* for *TestabilityExplorer*. The integration problem involves three classes. Class `LocalField`

```
1  public void testListenableDirectedGraph85() throws Exception {
2    ListenableDirectedGraph<String, DefaultEdge> var310 = new ListenableDirectedGraph<String,
       DefaultEdge>(DefaultEdge.class);
3    String var311 = "v1";
4    var310.addVertex(var311);
5    String var313 = "v2";
6    var310.addVertex(var313);
7    var310.addEdge(var311, var313);
8    ListenableDirectedGraph<String, DefaultEdge> var312 = new ListenableDirectedGraph<String,
       DefaultEdge>(DefaultEdge.class);
9    var312.addVertex(var311);
10   var312.addVertex(var313);
11   var312.addEdge(var311, var313);
12   DirectedNeighborIndex<String, DefaultEdge> var314 = new DirectedNeighborIndex<String,
       DefaultEdge>(var312);
13   var310.addGraphListener(var314);
14   String var315 = "v3";
15   var310.addVertex(var315);
16   var310.addEdge(var315, var311);   // fails with runtime exception
17   var310.removeEdge(var315, var311);
18   var310.removeVertex(var313);
19 }
```

Figure 5.3. Test case generated with *Fusion* for *JGraphT* that detects integration fault

```
1  public void testLocalField54() throws Exception {
2     FieldInfo var111 = new FieldInfo(null, "field", null, true, true, false);
3     LocalField var112 = new LocalField(null, var111);
4     var112.getDescription();  // fails with runtime exception
5     var112.computeHashCode();
6  }
```

Figure 5.4. Test case generated with *Fusion* for *TestabilityExplorer* that detects integration fault

integrates classes `Variable` and `FieldInfo`. `LocalField` also extends `Variable` and inherits its methods including method `getType()`. `LocalField`'s constructor invokes method `getType()` of class `FieldInfo` for initialization. Neither `LocalField` nor `FieldInfo` check for `null` type parameter and upon `getDescription()` method invocation runtime exception occurs. The test case detected the fault for a new combination of constructor-method invocations with valid inputs that is not present in original test cases.

We can then answer the second research question *RQ2*: the generated test cases find *new* faults that are not detected with the unit test cases available for the applications, since they represent either interaction faults or rare corner cases.

5.5 Effectiveness

To answer *RQ3*, we generated test cases with *Palus* and *Randoop* for the same case studies, and compared the faults revealed with the different suites. Table 5.3 summarizes the results.

Program	*Randoop*			*Palus*			***Fusion***		
	r.f.	*c.v.*	*f.p.*	*r.f.*	*c.v.*	*f.p.*	*r.f.*	*c.v.*	*f.p.*
TestabilityExplorer	1	1	2	0	0	80	3	1	1
JGraphT	2	3	9	1	1	1	3	0	4
Apache Ant	3	6	128	0	0	94	1	0	7
JFreeChart	4	19	17	0	2	34	3	2	46
Total	10	29	156	1	3	209	10	3	58

Table 5.3. Faults found with Fusion, Randoop and Palus (*r.f.: real faults*; *c.v.: implicit contract violations*; *f.p.: false positives*)

Both *Palus* and *Randoop* generate unit test cases with new execution sequences. *Randoop* implements a feedback-directed random test generation loop, and generates new test cases from the code of the system under test. *Randoop* generates class instantiation and method call sequences randomly, and selects valid sequences after executing and filtering them. Although *Randoop* is primary considered as a unit testing tool, it is not limited to generation of simple test cases. Some of the sequences produced by *Randoop* can be rather long (more than 20 method invocations) and may integrate several classes.

Palus builds on *Randoop* and guides random test generation using both a call sequence model inferred from sample executions of the system, and a method dependence information that derive from accesses to the common class fields. *Palus* combines static and dynamic automated test generation approach. It uses dynamic analysis to infer a call sequence model from a system execution, it then uses static analysis to identify intra-class method dependences that derive from accesses to common fields. *Palus* extends captured sequences with new calls to methods whose invocation depends on methods already in the traces, and generates test cases from the extended sequences by means of directed random test generation.

Both *Randoop* and *Palus* detect violations of implicit programming rules (for instance, the symmetry property of equality: `o.equals(o) == true`) and signal exceptions. In our comparison we consider these classes of notifications separately. Implicit contract violations represent minor faults that developers commonly overlook, but the faults are easily preventable. On the other hand, exceptional conditions can be related both to major faults caused by unexpected system behavior and to minor faults. Such minor faults arise from violations of implicit class contracts, for instance, when methods access uninitialized class fields and raise exceptions.

In our comparison we distinguish three types of faults. We consider both types of minor faults as *implicit contract violations*. We consider exceptional conditions not related to implicit class contracts either as *real faults* or as *false positives*, if the exceptional conditions are anticipated by developers.

We ran *Randoop* on each subject multiple times with a time limit of 100 seconds per subject on the *Desktop* configuration and we averaged the results. *Randoop* found more faults than *Fusion*, but most of the faults found by *Randoop* are unit faults that concern with class methods accessing uninitialized class fields and violation of the `equals()` method contract, while the faults found by *Fusion* are integration faults or subtle corner cases. *Randoop* did not find any of the faults detected by *Fusion*. Moreover, 2 faults found by *Fusion* were captured in the regression assertions of *Randoop* test cases as valid behaviors.

Palus requires sample executions of the system to infer its call sequence model. For the subject libraries (*JGraphT* and *JFreeChart*) we used their test suites as sample executions. For the subject applications (*TestabilityExplorer* and *Apache Ant*) we executed them invoking their main functionality. We ran the complete code analysis and reporting functionality of *TestabilityExplorer* and a complete application build with test execution of *Apache Ant*.

Palus does not scale well when analyzing sample executions for large number of classes and class instances, as well as for processing large collected models (>1GB). *Palus* was inconclusive both on the *Desktop* and on the *Server* configurations for all programs: collecting a model for 200 classes using unit test cases as sample executions fails due to insufficient JVM heap space after many execution hours.

To overcome scalability problems in the experiments with *Palus* we produced the call sequence models for reduced sets of classes that include classes for which our prototype detected faults.

Palus found two faults in *JGraphT*, one of which was also reported by Randoop while the other is a new relevant fault. *Palus* found two faults in *JFreeChart*, both of which are `equals()` contract violations. The two faults caused 10,392 executions to fail. *Palus* did not find any fault for *TestabilityExplorer* and *Apache Ant*. None of the test cases generated by *Palus* detected any fault revealed by *Fusion*.

All test suites generated some false positives. The total amount of false positives for the different test suites is comparable.

We can then answer the third research question *RQ3*: *Fusion* finds different faults than stat of the art random-based testing approaches and produces a comparable number of false positives.

5.6 Discussion

Differently from other approaches, our approach aims to generate integration test cases that exercise complex class interactions. This is why we can find different faults than the ones found with *Randoop* and *Palus*.

It is difficult for *Randoop* to create valid sequences of class instantiations using its randomized algorithm. For this reason *Randoop* is more likely to produce simple class instantiations, and, for classes that require complex instantiation, *Randoop* triggers error-handling procedures in class instantiations as happened for *Apache Ant*. In contrast, our approach reuses and combines existing valid instantiation sequences and is more likely to generate valid class instantiations.

We also noticed that the scope and applicability of our approach and *Palus* are different. *Palus* is more likely to generate valid class instantiations from the information obtained in sample executions, but for relatively big models, it is likely to miss the sequences of method invocations that only appear in a very small portion of the model, due to the randomized selection of methods. Our approach uses all observed method invocations, and thus better explores complex and unusual combinations.

Limitations and threats to validity

Although this empirical evaluation provides evidence of the usefulness and effectiveness of the approach developed in this research, there are several limitations and threats to the validity of the empirical results that should be considered in their interpretation.

Fusion generates test cases by reusing existing test cases. This is why availability and quality of original test cases directly affects the effectiveness of the approach. For instance, our approach may integrate several classes. If original test cases do not exercise methods of some class, then no execution sequences will be captured for that class and, consequently, generated test cases will not be able to change the state of this class by calling its methods. Despite integration of classes, there may be no mutual state transitions for the integrated classes and problematic interactions may not be triggered.

Figure 5.5 shows an example of such situation. Test case in the figure integrates four classes, however, there are no execution sequences for these classes in original test suites. Therefore, *Fusion* adds no sequences to the generated test case that, consequently, does not check possible class interactions, only the integration of classes.

Our approach can not guarantee to report all integration faults available in software. It is an optimistic technique and no finite number of tests can guarantee correctness of software [PY07]. Although our approach cannot report all possible integration faults, it can reveal faults different from other approaches. Our evaluation results show that our approach detects faults that are missed by good original test suites and by test suites generated with the state of the art test generation tools.

```
1 public void testDateTickUnit83() throws Exception {
2   SimpleDateFormat var327 = new SimpleDateFormat("d-MMM-yyyy", Locale.UK);
3   TimeZone var328 = TimeZone.getTimeZone("GMT");
4   GregorianCalendar var329 = new GregorianCalendar(var328, Locale.UK);
5   var327.setCalendar(var329);
6   DateTickUnit var330 = new DateTickUnit(DateTickUnit.MONTH, 1, var327);
7 }
```

Figure 5.5. Test case generated with *Fusion* for *JFreeChart*

We evaluate the effectiveness of the approach by comparing the types and the number of faults found by test cases generated with our approach and state of the art techniques. The quantitative measure based on numbers of faults alone represents a threat to construct validity as it does not account for the severity and importance of the discovered faults. Moreover, this measure does not allow to draw statistically significant conclusions from the obtained experimental data. We draw our conclusions about the effectiveness of the approach from the qualitative measure that takes into account the types of the detected faults, including implicit contract violations, unit faults, corner cases of unit faults, and integration faults.

Our results agree with the state of the practice in fault detection of integration faults. On average, fault detection rate for integration faults is an order of magnitude lower than that of unit test cases. We manually analysed fault repositories for the projects used in our evaluation (Table 5.1). Our inspection indicates an average fault detection rate of one integration fault per forty unit faults. Taking into account the size and the complexity of the case studies, together with the amount and severity of the integration faults found by our approach (Table 5.3), we conclude that our approach is effective.

To improve the statistical significance of the experimental results one may turn to fault seeding techniques and mutation testing in particular. These techniques allow to evaluate the effectiveness of the approach with the following assumption: effective test suites that detect simple syntactic faults can detect more complex real faults [Mor90]. For the technique to be effective, a large number of mutants must be automatically derived in a systematic way.

Application of mutation testing for evaluation of our approach requires specification of fault models of unit and integration faults. Models of unit faults exist and include mutation operators used to implement the fault model, mutant generation techniques, equivalent-mutant detection, and mutant-killing determination approaches [FZ12]. To the best of our knowledge, there are no models of integration faults. Such models are to be developed and evaluated by future research.

The approach generates some false positives because it explores new combinations of method invocation that may trigger not only real faults, but also false positives. Our technique partially mitigates this problem because it reuses existing valid instantiation and execution sequences. This way it avoids generation of a large amount of false

positives related to violations of class instantiation protocols. Such violations are a common problem for automated test generation techniques that our technique mitigates. Additional sources of information may improve the soundness of the technique by providing information for construction of test oracles for class integration and potentially reduce the amount of false positives.

The cost of the approach for automatic test case generation includes machine time and computational resources for test case generation and execution, and human time for inspection of test results. Fusion runtime data reported in Section 5.3 is not sufficient to claim the efficiency and cost-effectiveness of the approach. It supports the feasibility and the applicability claims. However, nowadays computational resources can be obtained at a reasonable cost and do not impede the adoption of test case generation approach, provided the practical benefits of the approach are delivered.

Inspection of test results impacts on the cost-effectiveness of the approach. Failures reported by the prototype need to be inspected and confirmed by developers. Developers need to analyze the test report and trace failing test execution from the manifestation of the fault to its root cause. Manual examination of the false positives is a difficult task that involves deep understanding of automatically generated test cases and the system under test. A cost of manual assessment of false positives is thus linear to the size of the generated test suite. This cost can be contrasted with the cost of manually generating complex test cases with test oracles. This cost is a function of a complexity and size of a system under test. The larger and more complex the system is, the larger integration test suites are required for testing that increases the overall testing cost.

Given that our technique produces a moderate amount of false positives we think that the inspection effort does not diminish the usefulness of the technique. The evaluation of the usefulness of our approach does not take into account a developer effort for inspection of test execution results. This issue must be explored with human studies.

A threat to construct validity concerns the prototype implementation. To verify that the prototype generates test cases that corroborate the expected output of the approach, we have manually generated some test cases with our approach and we compared them with the samples of test code generated with *Fusion*. As we mentioned in Section 5.3, some generated test cases shall be slightly modified to become executable. The execution problems are related to the initial assumptions on test case structure implemented in the prototype. The prototype has limited functionality for processing test cases for nested classes, as well as for processing testing classes organized hierarchically. The future versions of the prototype will incorporate code transformations and refactorings for processing complex test class structures.

One threat to external validity relates to the degree to which the subject applications used in our experiment are representative of the state of the practice. The selected case studies belong to distinct program domains, contain mature manually written test suites, and are actively supported by software developers.

We cannot claim that results generalize to other programs or to systems from domains other than those covered in the study. In particular, no generalization can be made as to the effectiveness of our approach. The results of experimental evaluation for object-oriented open source software may not be generalizable for industrial systems that employ different testing standards and procedures. Likewise, results may not be generalizable to systems using special integration frameworks. However, a variety of faults were discovered in this research and thus, the selected subject applications are useful for exploring and verifying the presented approach.

A threat to validity is related to the comparison with other approaches. We compared our technique with two state of the art test generation tools. To the best of our knowledge, *Randoop* and *Palus* are the best publicly available test generation tools for Java.

Chapter 6

Conclusions

Test case generation and maintenance are laborious and expensive activities, and software projects produce large amounts of test cases both manually and automatically. To improve software quality, developers design test cases of different kind and granularity aiming to prevent, detect, and remove different types of faults. The development effort and cost grow with the complexity of the test cases. Complex integration and system test cases are usually more expensive to develop, and harder to generate and maintain than simple unit test cases.

This thesis tackles the problem of generating complex test cases by introducing an approach that exploits the information in unit test cases to generate integration test cases automatically. The idea behind the approach stems from the empirical investigation of many test cases from different software systems. We studied the structure and the complexity of test cases of different granularity. We identified and described the phenomenon of implicit reuse of test cases, where complex test cases share code fragments with simple test cases. The investigation results inspired our approach to test case generation that leverages test cases produced by software projects to automatically generate new test cases.

We designed and implemented the approach focussing on unit and integration test cases. The approach requires only the source code and some unit test cases, and uses classic analysis techniques to extract the information required to build integration test cases. It analyzes system dependencies and identifies fragments of meaningful information in test cases. By aggregating the identified test case fragments for clusters of dependent classes, the approach automatically generates integration test cases that aim to reveal interaction faults.

We implemented the approach in a prototype *Fusion* and demonstrated the effectiveness of the approach on a number of popular open-source projects. We evaluated the applicability and the usefulness of our approach by examining the amount of test cases that our approach can generate and inspecting the types of revealed faults. We evaluated the effectiveness of the approach by comparing the fault-detection capabilities

of the test suites generated with our approach with the test suites generated with the state of the art test generation techniques *Palus* and *Randoop*. The experimental data show that our approach generates test cases that can find faults that depend both on subtle interaction and corner cases of unit faults even in well tested applications, differently from the state of the art approaches that tend to find different and usually less relevant faults.

6.1 Contributions

The first contribution of this dissertation is the study of the test case structure and of the relations between test cases of different granularity. This dissertation empirically investigates the phenomenon of testing reuse and proposes to exploit the interrelation between test cases of different granularity for generating test cases.

The second contribution of this dissertation is the definition of an approach for automatically generating complex test cases by identifying and composing reusable fragments of simpler test cases.

In the following we summarize some aspects of the two major contributions:

Analysis and classification of test cases of different granularity We empirically investigated test suites from a number of open-source projects, and analyzed the test case structure and complexity. We show that current taxonomies of test cases do not aid automatically differentiating test cases. As a result of the investigation we defined a set of complexity indicators that allow to automatically differentiate test cases of different granularity.

Investigation of testing reuse We empirically investigated the phenomenon of reuse and information sharing in real-world test suites. We measured the amount of reuse between simple and complex test cases, and analyzed the kinds of information shared among test cases. We found that informal testing reuse is a common practice of software developers who consult and reuse information from test cases to build new ones.

Analysis of class dependencies We developed a technique for analyzing and representing system dependencies to indicate clusters of dependent classes to be integrated and tested together. We extract system dependencies based on the information from the system source code. We record the dependencies in the form of a modified object relation diagram that indicates class dependencies and class instantiation orders, and takes into account polymorphic class relations.

Analysis of test case fragments We developed a technique based on data flow analysis of test cases to extract relevant fragments of test case. We proposed a technique to extract test case fragments related to the instantiation of classes (instantiation

sequences) and test case fragments related to the different scenarios of class usage (execution sequences). The technique extracts meaningful and reusable test case fragments.

Integration strategy We developed an integration strategy that explores combinations of test case fragments to generate new test cases. The strategy defines the order of assembly of test case fragments and class integration through data relations. For clusters of dependent classes, the strategy enables the incremental generation of test cases of increasing complexity.

Prototype implementation We implemented our approach in a prototype *Fusion*. The prototype implements the techniques for extracting class dependencies and test case fragments, and proposes some integration strategies to generate new integration test cases. We used the prototype to experimentally evaluate the effectiveness of our approach on real-world software projects.

Evaluation of the approach We evaluated our approach on four open-source projects from different program domains. Our prototype generated new test cases for all the case studies. It detected previously unknown faults of two categories: integration faults and unit faults that depend on corner cases. The comparison of our approach with state of the art test generation techniques showed that our approach is effective: It detects relevant integration faults and is complementary to the other techniques that detect less relevant unit faults and violations of implicit contracts.

6.2 Future directions

The work presented in this dissertation opens a number of research directions. Our future research plans include:

Automated generation of oracles Test cases of different granularity have common structure and are composed of three main parts: test initialization, test execution and oracle. In this dissertation we focused on test initialization and test execution, leaving open the problem of automatically generating effective oracles from the information available in the original test cases. We believe that we can augment generated test cases with test oracles to improve the effectiveness of our approach and aid precise fault detection while reducing the amount of false positives.

This issue is opened by our research results. Our investigation of test case structure discovered some relations between oracles in simple and complex test cases. Some oracles are equivalent, while others require adaptation for the context of new test cases. In our future work we aim to define patterns of reuse of oracles and develop a technique for automatic import and adaptation of oracles for test case generation.

Application to test cases of other granularity This thesis proposes a technique to generate complex test cases from simple ones, and shows experimentally that the approach works for generating integration test cases from unit ones. However, our approach is general and incremental, and it is not limited to single iterations of test case generation process.

We plan to experiment with incremental generation of test cases of increasing complexity from the test cases generated with our approach. This experimentation will verify our intuition that providing input test cases generated on previous iterations can yield more complex and useful test cases.

Prototype extension The prototype implementation *Fusion* helped us to evaluate our approach on open source software systems. It enables the automatic analysis of software projects and the automatic generation of new test cases. The prototype has many limitations, and we are working on a new more robust version that will help us to generate more complex test cases and will be extended for the analysis of oracle information.

Bibliography

[ABC+13] Saswat Anand, Edmund Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey on automated software test case generation. *Journal of Systems and Software*, 2013. to appear.

[ADTP10] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. Directed test generation for effective fault localization. In *Proc. 19th Int. Symp. on SW testing and analysis*, pages 49–60. ACM, 2010.

[AEK+06] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)*, October 23, 2006.

[AH11] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 3–12, 2011.

[AML11] J.H. Andrews, T. Menzies, and F.C.H. Li. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering*, 37(1):80–94, 2011.

[BBDP11] Mauro Baluda, Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Enhancing structural software coverage by incrementally computing branch executability. *Software Quality Control*, 19:725–751, December 2011.

[BDP13] Pietro Braione, Giovanni Denaro, and Mauro Pezzè. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. In *Proc. 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2013.

[Bec00] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, 2000.

[Bec02] Kent Beck. *Test-Driven Development By Example*. Addison Wesley, 2002.

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of SW Eng.*, pages 85–103, 2007.

[BHH+11] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos. Symbolic search-based testing. In *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 53–62, 2011.

[BJK+05] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. *Model-Based Testing of Reactive Systems: Advanced Lectures*. Springer, 2005.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *Proc. of the Int. Symp. on SW Testing and Analysis*, pages 123–133, 2002.

[BLW03] Lionel C. Briand, Yvan Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. *IEEE Trans. SW Eng.*, 29(7):594–607, 2003.

[BMZ+05] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change. *J. Softw. Maint. Evol.*, 17(5):309–332, September 2005.

[BPdM09] C. Bertolini, G. Peres, M. d'Amorim, and A. Mota. An empirical evaluation of automated black box testing techniques for crashing guis. In *Int. Conf. SW Testing Verification and Validation*, pages 21–30, 2009.

[CCR10] Isis Cabral, Myra B. Cohen, and Gregg Rothermel. Improving the testing and testability of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 241–255, Berlin, Heidelberg, 2010. Springer-Verlag.

[CGS13] Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *Proc. of 6th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST)*, 2013.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. Adaptive random testing: The art of test case diversity. *J. Syst. Softw.*, 83(1):60–66, January 2010.

[CLOM07] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental assessment of random testing for object-oriented software. In *Proc. Int. Symp. on Software Testing and Analysis 2007 (ISSTA'07)*, pages 84–94. ACM, 2007.

[CLOM08] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Proc. 30th Int. Conf. on Software engineering*, pages 71–80. ACM, 2008.

[CMWE04] T.Y. Chen, R. Merkel, P. K. Wong, and G. Eddy. Adaptive random testing through dynamic partitioning. In *In Proc. 4th International Conference on Quality Software*, pages 79–86, 2004.

[CPDGP01] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. In *Proc. 8th European SW Eng. Conf. held jointly with 9th ACM SIGSOFT Int. Symp. on Foundations of SW Eng.*, ESEC/FSE-9, pages 142–151, 2001.

[CS04] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, September 2004.

[DF94] Roong-Ko Doong and Phyllis G. Frankl. The astoot approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3:101–130, April 1994.

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

[DK06] Christian Denger and Ronny Kolb. Testing and inspecting reusable product line components: first empirical results. In *Proc. ACM/IEEE Int. Symp. on Empirical SW Eng.*, pages 184–193, 2006.

[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. *Int. Conf. on Automated Software Engineering*, 0:59–68, 2006.

[ECDD06] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *SIGSOFT '06/FSE-14: Proc. 14th ACM SIGSOFT Int. Symp. on Foundations of software engineering*, pages 253–264. ACM, 2006.

[ECDJ09] S. Elbaum, Hui Nee Chin, M.B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Trans. SW Eng.*, 35(1):29–45, 2009.

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proc. of the 19th ACM SIGSOFT Symp.*

and the 13th European Conf. on Foundations of SW Eng., pages 416–419, 2011.

[FA12] Gordon Fraser and Andrea Arcuri. The seed is strong: Seeding strategies in search-based software testing. *Proc. of the IEEE 5th Int. Conf. on SW Testing, Verification and Validation*, 0:121–130, 2012.

[FZ11a] G. Fraser and A. Zeller. Exploiting common object usage in test case generation. In *In Proc. of the IEEE 4th Int. Conf. on SW Testing, Verification and Validation*, pages 80–89, 2011.

[FZ11b] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *Proc. Int. Symp. on Software Testing and Analysis*, pages 364–374, 2011.

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

[GGJ+10] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *Proc. 32nd Int. Conf. on Software Engineering*, pages 225–234. ACM, 2010.

[GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *Proc. 2002 ACM SIGSOFT Int. Symp. on Software testing and analysis*, pages 112–122. ACM, 2002.

[GHK+01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, April 2001.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proc. 2005 ACM SIGPLAN Conf. on Programming language design and implementation*, pages 213–223. ACM, 2005.

[GOC06] Leonard Gallagher, Jeff Offutt, and Anthony Cincotta. Integration testing of object-oriented components using finite state machines. *Softw. Test. Verif. Reliab.*, 16(4):215–266, December 2006.

[GvDS13] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *Proc. of 6th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST)*, 2013.

[Har00] Mary Jean Harrold. Testing: a roadmap. In *Proc. of the Conf. on The Future of SW Engineering*, pages 61–72, 2000.

[Har07] Mark Harman. The current state and future of search based software engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 342–357. IEEE Computer Society, 2007.

[Has08] A.E. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance*, pages 48–57, 2008.

[HBB+09] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76, February 2009.

[HKU02] Robert M. Hierons, T.-H. Kim, and Hasan Ural. Expanding an extended finite state machine to aid testability. In *COMPSAC '02: Proc. 26th Int. Computer Software and Applications Conf. on Prolonging Software Life: Development and Redevelopment*, pages 334–342. IEEE Computer Society, 2002.

[HM10] M. Harman and P. 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.

[HMZ12] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.

[HO08] M.J. Harrold and A. Orso. Retesting software during development and maintenance. In *Frontiers of Software Maintenance*, pages 99–108, 2008.

[HX10] Ahmed E. Hassan and Tao Xie. Software intelligence: the future of mining software engineering data. In *Proc. FSE/SDP WS on Future of SW Eng. research*, pages 161–166, 2010.

[JED08] Matthew Jorde, Sebastian G. Elbaum, and Matthew B. Dwyer. Increasing test granularity by aggregating unit tests. In *Proc. of the 23rd IEEE/ACM Int. Conf. on Automated SW Eng.*, pages 9–18, 2008.

[KCM07] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, March 2007.

[KHS11] Abdul Salam Kalaji, Robert Mark Hierons, and Stephen Swift. An integrated search-based approach for automatic testing from extended finite state

machine (efsm) models. *Inf. Softw. Technol.*, 53(12):1297–1318, December 2011.

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

[KKR12] Moonzoo Kim, Yunho Kim, and G. Rothermel. A scalable distributed concolic testing approach: An empirical evaluation. In *Proc. 5th International Conference on Software Testing, Verification and Validation*, pages 340–349, 2012.

[KM06] Ronny Kolb and Dirk Muthig. Making testing product lines more efficient by improving the testability of product line architectures. In *ROSATEA '06: Proc. ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 22–27. ACM, 2006.

[KR11] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proc. 33rd ACM/IEEE Int. Conf. on SW Eng.*, pages 351–360, 2011.

[KRH+08] Adrian Kuhn, Bart Rompaey, Lea Haensenberger, Oscar Nierstrasz, Serge Demeyer, Markus Gaelli, and Koenraad Leemput. Jexample: Exploiting dependencies between tests to improve defect localization. In *Agile Processes in Software Engineering and Extreme Programming*, volume 9 of *Lecture Notes in Business Information Processing*, pages 73–82. Springer Berlin Heidelberg, 2008.

[Kru92] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

[LHG13] Kiran Lakhotia, Mark Harman, and Hamilton Gross. Austin: An open source tool for search based software testing of c programs. *Inf. Softw. Technol.*, 55(1):112–125, January 2013.

[LW90] H.K.N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proc. Conf. on Software Maintenance*, pages 290–301, 1990.

[McC76] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[McM04] Phil McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[MHBT06] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to searchbased test data generation. In *Proc. Int. Symp. on Software Testing and Analysis*, ISSTA '06, pages 13–24, 2006.

[MMS01] C.C. Michael, G. Mcgraw, and M.A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[MOP02] Vincenzo Martena, Alessandro Orso, and Mauro Pezzè. Interclass testing of object oriented software. *IEEE Int. Conf. on Engineering of Complex Computer Systems*, 0:135, 2002.

[Mor90] L.J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, 1990.

[MPP07] Leonardo Mariani, Sofia Papagiannakis, and Mauro Pezzè. Compatibility and regression testing of COTS-component-based software. In *Proc. of the 29th Int. Conf. on SW Eng.*, pages 85–95, 2007.

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

[MPRS11] L. Mariani, M. Pezzè, O. Riganelli, and M. Santoro. Autoblacktest: a tool for automatic black-box testing. In *Proc. 33rd International Conference on Software Engineering*, pages 1013–1015, 2011.

[MPRS12] Leonardo Mariani, Mauro Pezze, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: Automatic black-box testing of interactive applications. In *Proc. International Conference on Software Testing, Verification, and Validation*, volume 0, pages 81–90. IEEE Computer Society, 2012.

[MSK09] C. Murphy, K. Shen, and G. Kaiser. Automatic system testing of programs without test oracles. In *Proc. of the Int. Symp. on SW Testing and Analysis*, pages 189–200, 2009.

[Par78] David L. Parnas. Designing software for ease of extension and contraction. In *Proce. 3rd Int. Conf. on SW Eng.*, ICSE '78, pages 264–277. IEEE Press, 1978.

[PG12] Michael Pradel and Thomas R. Gross. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proc. of the 35th Int. Conf. on SW Eng.*, pages 288–298, 2012.

[PLB08] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .net with feedback-directed random testing. In *Proc. Int. Symp. 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 *Proc. of the 29th Int. Conf. on SW Eng.*, pages 75–84, 2007.

[PM06] Klaus Pohl and Andreas Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, December 2006.

[PV09] Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, October 2009.

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

[QODL10] A. Qusef, R. Oliveto, and A. De Lucia. Recovering traceability links between unit tests and classes under test: An improved method. In *IEEE Int. Conf. on SW Maintenance*, pages 1–10, 2010.

[QRL10] Dawei Qi, Abhik Roychoudhury, and Zhenkai Liang. Test generation to expose changes in evolving programs. In *Proc. of the IEEE/ACM Int. Conf. on Automated SW Eng.*, pages 397–406, 2010.

[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 *Proc. 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 23–32, 2011.

[RH96] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Softw. Eng.*, 22(8):529–551, August 1996.

[RH97] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, April 1997.

[RMP07] Sacha Reis, Andreas Metzger, and Klaus Pohl. Integration testing in software product line engineering: A model-based technique. In Matthew Dwyer and Antónia Lopes, editors, *Fundamental Approaches to Software Engineering*, volume 4422, pages 321–335. Springer Berlin / Heidelberg, 2007.

[RST+04] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *Proc. 19th ACM SIGPLAN Conf. on Object-oriented programming, systems, languages, and applications*, pages 432–448, 2004.

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

[SC96] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. on Software Engineering*, 22:777–793, 1996.

[SHO10] R. Santelices, M.J. Harrold, and A. Orso. Precisely detecting runtime change interactions for evolving software. In *Proc. 3rd Int. Conf. on SW Testing, Verification and Validation*, pages 429 –438, 2010.

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proc. 10th European SW Eng. conference held jointly with 13th ACM SIGSOFT Int. Symp. on Foundations of SW Eng.*, ESEC/FSE-13, pages 263–272, 2005.

[TDH08] Nikolai Tillmann and Jonathan De Halleux. Pex: white box test generation for .net. In *Proc. of the 2nd Int. Conf. on Tests and proofs*, pages 134–153. 2008.

[TEL11] C. Torens, L. Ebrecht, and K. Lemmer. Starting model-based testing based on existing test cases used for model creation. In *2011 IEEE 11th Int. Conf. on Computer and Information Technology (CIT)*, pages 320–327, 2011.

[TG13] Rajeev Tiwari and Noopur Goel. Reuse: reducing test effort. *SIGSOFT Softw. Eng. Notes*, 38(2):1–11, March 2013.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *Proc. of the Int. Symp. on SW Testing and Analysis*, pages 119–128, 2004.

[TXT+09a] K. Taneja, Tao Xie, N. Tillmann, J. de Halleux, and W. Schulte. Guided path exploration for regression test generation. In *Proc. 31st Int. Conf. on SW Eng.*, pages 311 –314, 2009.

[TXT+09b] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. MSeqGen: object-oriented unit-test generation via mining source code. In *Proc. of the 7th joint meeting of the European SW Eng. Conf. and the ACM SIGSOFT Symp. on the Foundations of SW Eng.*, 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. In *Proc. of the ACM Int. Conf. on Object oriented programming systems languages and applications*, pages 189–206, 2011.

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In *Proc. ACM SIGSOFT Int. Symp. on SW testing and analysis*, pages 97–107, 2004.

[WGMO10] Yi Wei, Serge Gebhardt, Bertrand Meyer, and Manuel Oriol. Satisfying test preconditions through guided object selection. *Software Testing, Verification, and Validation, 2008 Int. Conf. on*, 0:303–312, 2010.

[XCR10] Zhihong Xu, Myra B. Cohen, and Gregg Rothermel. Factors affecting the use of genetic algorithms in test suite augmentation. In *Proc. 12th Conf. on Genetic and evolutionary computation*, GECCO '10, pages 1365–1372, 2010.

[Xie09] Tao Xie. Improving automation in developer testing: State of the practice. Technical Report TR-2009-6, North Carolina State University Department of Computer Science, February 2009.

[XKK+10] Zhihong Xu, Yunho Kim, Moonzoo Kim, Gregg Rothermel, and Myra B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proc. of the eighteenth ACM SIGSOFT Int. Symp. on Foundations of SW engineering*, pages 257–266, 2010.

[XKKR11] Zhihong Xu, Yunho Kim, Moonzoo Kim, and G. Rothermel. A hybrid directed test suite augmentation technique. In *22nd International Symposium on Software Reliability Engineering*, pages 150–159, 2011.

[XMSN05] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: a framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, Berlin, Heidelberg, 2005. Springer-Verlag.

[XR09] Zhihong Xu and Gregg Rothermel. Directed test suite augmentation. In *Proc. of the 16th Asia-Pacific SW Eng. Conf.*, APSEC '09, pages 406–413, 2009.

[XTLL09] Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. Data mining for software engineering. *IEEE Computer*, 42(8):35–42, August 2009.

[YH10] Shin Yoo and Mark Harman. Test data regeneration: Generating new test data from existing test data. *Journal of Software Testing, Verification and Reliability*, 22(3):171–201, 2010.

[YX06] Hai Yuan and Tao Xie. Substra: A framework for automatic generation of integration tests. In *WS on Automation of SW Test*, pages 64–70, 2006.

[Zel99] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Proc. 7th European SW Eng. Conf. jointly with 7th ACM SIGSOFT Int. Symp. on Foundations of SW Eng.*, pages 253–267, 1999.

[ZSBE11] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. In *Proc. of the Int. Symp. on SW Testing and Analysis*, pages 353–363, 2011.

[ZZLX10] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. Random unit-test generation with MUT-aware sequence recommendation. In *Proc. of the IEEE/ACM Int. Conf. on Automated SW Eng.*, pages 293–296, 2010.

[ZZWD05] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl. Mining version histories to guide software changes. *IEEE Trans. SW Eng.*, 31(6):429 – 445, june 2005.