Software Redundancy: What, Where, How

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
Andrea Mattavelli

under the supervision of
Prof. Mauro Pezzè
co-supervised by
Prof. Antonio Carzaniga

October 2016

Dissertation Committee

Prof. Matthias Hauswirth	USI Università della Svizzera italiana, Switzerland
Prof. Cesare Pautasso	USI Università della Svizzera italiana, Switzerland
Prof. Earl Barr	University College London, United Kingdom
Prof. Paolo Tonella	Fondazione Bruno Kessler, Italy

Dissertation accepted on 25 October 2016

Prof. Mauro Pezzè
Research Advisor
USI Università della Svizzera italiana, Switzerland

Prof. Antonio Carzaniga
Research Co-Advisor
USI Università della Svizzera italiana, Switzerland

Prof. Walter Binder
PhD Program Director

Prof. Michael Bronstein
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.

Andrea Mattavelli
Lugano, 25 October 2016

To Katia, my wonderful family, and me

Abstract

Software systems have become pervasive in everyday life and are the core component of many crucial activities. An inadequate level of reliability may determine the commercial failure of a software product. Still, despite the commitment and the rigorous verification processes employed by developers, software is deployed with faults. To increase the reliability of software systems, researchers have investigated the use of various form of redundancy. Informally, a software system is redundant when it performs the same functionality through the execution of different elements. Redundancy has been extensively exploited in many software engineering techniques, for example for fault-tolerance and reliability engineering, and in self-adaptive and self-healing programs. Despite the many uses, though, there is no formalization or study of software redundancy to support a proper and effective design of software.

Our intuition is that a systematic and formal investigation of software redundancy will lead to more, and more effective uses of redundancy. This thesis develops this intuition and proposes a set of ways to characterize qualitatively as well as quantitatively redundancy. We first formalize the intuitive notion of redundancy whereby two code fragments are considered redundant when they perform the same functionality through different executions. On the basis of this abstract and general notion, we then develop a practical method to obtain a measure of software redundancy. We prove the effectiveness of our measure by showing that it distinguishes between shallow differences, where apparently different code fragments reduce to the same underlying code, and deep code differences, where the algorithmic nature of the computations differs. We also demonstrate that our measure is useful for developers, since it is a good predictor of the effectiveness of techniques that exploit redundancy.

Besides formalizing the notion of redundancy, we investigate the pervasiveness of redundancy intrinsically found in modern software systems. Intrinsic redundancy is a form of redundancy that occurs as a by-product of modern design and development practices. We have observed that intrinsic redundancy is indeed present in software systems, and that it can be successfully exploited for good purposes. This thesis proposes a technique to automatically identify equivalent method sequences in software systems to help developers assess the presence of intrinsic redundancy. We demonstrate the effectiveness of the technique by showing that it identifies the majority of equivalent method sequences in a system with good precision and performance.

Contents

Contents v

List of Figures ix

List of Tables xiii

1 Introduction 1
- 1.1 Research Hypothesis and Contributions 4
- 1.2 Structure of the Dissertation 5

2 Redundancy in Software Engineering 7
- 2.1 Software Redundancy 8
- 2.2 Applications of Software Redundancy 15
  - 2.2.1 Deliberate Redundancy 15
  - 2.2.2 Intrinsic Redundancy 21
  - 2.2.3 Open Challenges for Software Redundancy 25
- 2.3 Techniques to Identify Redundancy 27
  - 2.3.1 Code Clones Detection 27
  - 2.3.2 Specification Inference 28
  - 2.3.3 Open Challenges for Automatically Identifying Redundancy . . . 29

3 A Notion of Software Redundancy 33
- 3.1 An Abstract and General Notion of Redundancy 36
  - 3.1.1 Basic Definitions 36
  - 3.1.2 Observational Equivalence 37
  - 3.1.3 Software Redundancy 39
- 3.2 Behavioral Equivalence and Execution Diversity 40
  - 3.2.1 Behavioral Equivalence 40
  - 3.2.2 Execution Diversity 42

4 A Measure of Software Redundancy 45
- 4.1 A Practical Measure of Redundancy 46

4.1.1 A Non-Binary Measure of Redundancy . . . 47
4.1.2 Sampling the State Space . . . 48
4.1.3 Measuring Observational Equivalence . . . 50
4.1.4 Measuring Execution Diversity . . . 50
4.1.5 Aggregating the Measures of Redundancy . . . 52
4.2 Experimental Validation . . . 54
4.2.1 Experimental Setup . . . 54
4.2.2 Internal Consistency (Q1) . . . 55
4.2.3 Significance (Q2) . . . 63
4.3 Limitations and Threats to Validity . . . 69

5 Automatic Identification of Equivalences 71
5.1 Background . . . 72
5.2 Automatic Synthesis of Equivalent Method Sequences . . . 77
5.3 SBES: A Tool to Synthesize Equivalent Method Sequences . . . 79
5.3.1 Initialization: Execution Scenarios . . . 80
5.3.2 First Phase: Candidate Synthesis . . . 81
5.3.3 Second Phase: Candidate Validation . . . 85
5.4 Experimental Validation . . . 87
5.4.1 Experimental Setup . . . 88
5.4.2 Effectiveness (Q1, Q2) . . . 90
5.4.3 Efficiency (Q3) . . . 96
5.4.4 Counterexamples (Q4) . . . 97
5.4.5 Limitations and Threats to Validity . . . 99

6 Conclusion 101

A Measure of Software Redundancy: Experiments 109
A.1 Redundancy Measurements . . . 109
A.1.1 Identity . . . 109
A.1.2 Refactoring . . . 110
A.1.3 Ground-truth Benchmark Results . . . 113
A.2 Benchmark Implementations . . . 141
A.2.1 Binary search . . . 141
A.2.2 Linear search . . . 143
A.2.3 Bubble Sort . . . 145
A.2.4 Insertion Sort . . . 149
A.2.5 Merge Sort . . . 151
A.2.6 Quick Sort . . . 155
A.3 Statistical Test Results . . . 158

Bibliography 161

Figures

2.1 An example of N-version programming implementation of a function that converts an array of 10 ASCII coded digits into a binary number [CA78]. 10
2.2 Methods put and putAll of the AbstractMultimap<K,V> Class from the Google Guava library 11
2.3 Documentation of methods clear and retainAll of the class Stack 30

3.1 Informal visualization of functional equivalence 34
3.2 Informal visualization of execution diversity 34
3.3 Examples of redundant code fragments 35
3.4 Abstract representation of the execution of a generic code fragment C . 37
3.5 Representation of the execution of the redundant code fragments at the top of Figure 3.3 38
3.6 Observational equivalence between two code C_A and C_B 39

4.1 General algorithm to measure redundancy. 48
4.2 Example of random test case generated with Randoop for the ArrayList-Multimap class of Google Guava. The invocation of the method put(key, value) is highlighted at line 295. 49
4.3 Example of a generated probing code executed immediately after one of the code fragments (A) under measurement. The linkage between the code fragment and the probing code is highlighted at line 6. 51
4.4 Stability of the redundancy measure on an implementation of binary search (Benchmark 1). 58
4.5 Stability of the redundancy measure on an implementation of linear search (Benchmark 1). 59
4.6 Comparison among data projections on binary search (Benchmark 1). . 61
4.7 Redundancy between implementations of the same algorithm (*) and between different algorithms (†). 64

5.1 Examples of chromosomes for the Stack class of the Java standard library. 75

5.2 Examples of mutations applied on the chromosomes of Figure 5.1. The mutations are located at line 2 on Chromosome A, and at lines 4–7 on Chromosome B. 75
5.3 Examples of crossover applied on the mutated chromosomes of Figure 5.2. The single crossover point is applied between lines 2 and 3. 75
5.4 High-level representation of our approach to synthesize equivalences. . . 77
5.5 General algorithm to synthesize an equivalence. 78
5.6 Main components of SBES 80
5.8 The synthesis stub generated for the Stack class. 84
5.9 The validation stub generated for the Stack class. 86

A.1 Measurements obtained on two identical execution traces 109
A.2 Measurements after refactoring: binary and linear search 110
A.3 Measurements after refactoring: bubble sort and insertion sort 111
A.4 Measurements after refactoring: merge sort and quick sort 112
A.5 Ground-truth binary algorithm data projections 113
A.6 Ground-truth binary algorithm code projections 114
A.7 Ground-truth linear algorithm data projections 115
A.8 Ground-truth linear algorithm code projections 116
A.9 Ground-truth linear implementation vs every binary implementation, data projections 117
A.10 Ground-truth linear implementation vs every binary implementation, code projections 118
A.11 Ground-truth binary implementation vs every linear implementation, data projections 119
A.12 Ground-truth binary implementation vs every linear implementation, code projections 120
A.13 Ground-truth bubble sort algorithm data projections 121
A.14 Ground-truth bubble sort algorithm data projections (cont.) 122
A.15 Ground-truth bubble sort algorithm code projections 123
A.16 Ground-truth bubble sort algorithm code projections (cont.) 124
A.17 Ground-truth insertion sort algorithm data projections 125
A.18 Ground-truth insertion sort algorithm code projections 126
A.19 Ground-truth merge sort algorithm data projections 127
A.20 Ground-truth merge sort algorithm code projections 128
A.21 Ground-truth quick sort algorithm data projections 129
A.22 Ground-truth quick sort algorithm code projections 130
A.23 Ground-truth bubble sort implementation vs every other sorting algorithm implementation, data projections 131
A.24 Ground-truth bubble sort implementation vs every other sorting algorithm implementation, data projections (cont.) 132

A.25 Ground-truth bubble sort implementation vs every other sorting algorithm implementation, code projections 133
A.26 Ground-truth bubble sort implementation vs every other sorting algorithm implementation, code projections (cont.) 134
A.27 Ground-truth insertion sort implementation vs every other sorting algorithm implementation, data projections 135
A.28 Ground-truth insertion sort implementation vs every other sorting algorithm implementation, code projections 136
A.29 Ground-truth merge sort implementation vs every other sorting algorithm implementation, data projections 137
A.30 Ground-truth merge sort implementation vs every other sorting algorithm implementation, code projections 138
A.31 Ground-truth quick sort implementation vs every other sorting algorithm implementation, data projections 139
A.32 Ground-truth quick sort implementation vs every other sorting algorithm implementation, code projections 140
A.33 Implementation 1 of the binary search algorithm 141
A.34 Implementation 2 of the binary search algorithm 141
A.35 Implementation 3 of the binary search algorithm 142
A.36 Implementation 4 of the binary search algorithm 142
A.37 Implementation 1 of the linear search algorithm 143
A.38 Implementation 2 of the linear search algorithm 143
A.39 Implementation 3 of the linear search algorithm 144
A.40 Implementation 4 of the linear search algorithm 144
A.41 Implementation 1 of the bubble sort algorithm 145
A.42 Implementation 2 of the bubble sort algorithm 145
A.43 Implementation 3 of the bubble sort algorithm 146
A.44 Implementation 4 of the bubble sort algorithm 146
A.45 Implementation 5 of the bubble sort algorithm 147
A.46 Implementation 6 of the bubble sort algorithm 147
A.47 Implementation 7 of the bubble sort algorithm 148
A.48 Implementation 1 of the insertion sort algorithm 149
A.49 Implementation 2 of the insertion sort algorithm 149
A.50 Implementation 3 of the insertion sort algorithm 150
A.51 Implementation 1 of the merge sort algorithm 151
A.52 Implementation 2 of the merge sort algorithm 152
A.53 Implementation 3 of the merge sort algorithm 153
A.54 Implementation 4 of the merge sort algorithm 154
A.55 Implementation 1 of the quick sort algorithm 155
A.56 Implementation 2 of the quick sort algorithm 156
A.57 Implementation 3 of the quick sort algorithm 157

Tables

2.1 Rewriting rules found for representative Java libraries and applications. 12

4.1 Projections used to derive action logs 53
4.2 Similarity measures applied to execution logs. The abbreviations on the right side identify the measures in the experimental evaluation in Section 4.2. 54
4.3 Benchmark 1: Different implementations of search and sorting algorithms 56
4.4 Benchmark 2: Guava Classes, number of considered methods and equivalent implementations considered 56
4.5 Observational equivalence for the methods of the class ArrayListMultimap from the Google Guava library (Benchmark 2). 62
4.6 Equivalence measures of methods *keys()* and *keySet()* of class ArrayList-Multimap (Benchmark 2). 62
4.7 Statistical significance of the results in Figure 4.7. We omit from the results the *DamLev* similarity metric since it is identical to *Lev*. 66
4.8 Correlation between redundancy measure and the effectiveness of Automatic Workarounds. 68

5.1 Case studies analyzed in our SBES experiments. 89
5.2 Sample sequences synthesized with SBES. 91
5.3 Q1, Q2: Effectiveness of SBES. 92
5.4 Contribution of the generic-to-concrete transformation on the effectiveness of SBES. 94
5.5 Comparison between Genetic Algorithms (GA) and Memetic Algorithms (MA) on the effectiveness of SBES. 95
5.6 Q4: Efficiency of the approach 97
5.7 Q5: Effectiveness of counterexamples 98

A.1 Statistical tests - Search benchmark 158
A.2 Statistical tests - Sorting benchmark 159
A.3 Statistical tests - Sorting benchmark (cont.) 160

Chapter 1

Introduction

Redundancy is an essential mechanism in engineering. Different forms of redundant design are the core technology of well-established reliability and fault tolerant mechanisms in traditional as well as software engineering. Despite its widespread use and importance, software redundancy has never been formalized and investigated by and of itself. This dissertation presents the first systematic investigation of software redundancy, and in particular it proposes a formalization of the general notion of redundancy, a practical measure of redundancy, and an approach to automatically identify functionally equivalent method sequences.

A software system is redundant when it contains different elements that perform the same functionality. A software system may include various versions of the same functionally equivalent algorithm. For example, a container library may expose two or more sorting algorithms: one optimized for small sequences of data, such as insertion sort, and a more complex but asymptotically faster algorithm like Timsort for larger sequences. Even algorithmically equivalent operations could be performed in different ways. For instance, a program may generate a matrix by filling the data structure by row, or by adding the values by column. A system may even contain replicas of exactly the same functionality and sometimes that might be the result of good design principles. For example, the API to access the file system of a utility library may have been refactored to improve their usage experience, but the library may also retain the old versions for backward compatibility. In contrast, copy-and-paste code clones cannot be deemed as redundant. Code clones perform the same functionality, but there is no difference in their execution since they are often exact copies of the same code.

In the aforementioned cases—whenever a system is capable of performing the same equivalent functionality by executing different code elements—we say that the software system is *redundant*. Redundant elements can be present in different forms and at different level of abstraction, from simple code fragments, such as different algorithm implementations, to components and even entire software systems, as in the case of web services, where we can retrieve several implementations of the same service.

Redundancy is sometimes introduced systematically, by design, into software systems, while in other cases it arises naturally as a by-product of other design and development disciplines.

Redundancy has been used as the key ingredient of the majority of the runtime mechanisms that prevent or mask failures. Researchers have proposed approaches based on the *deliberate* introduction of redundancy through the independent design and implementation of multiple versions of the same components [Avi85, LBK90, Ran75]. More recently, researchers have proposed self-healing approaches to detect and react "autonomically" to functional failures in order to avoid or at least alleviate those failures. Several proposed techniques exploit a form of redundancy that is *intrinsically* found in software systems at various levels of abstraction, from method calls [CGM+13, CGPP15], to entire software components [BGP07, Dob06, STN+08].

The use of software redundancy has been recently extended to other research areas of software engineering, such as software testing and automatic program repair. In software testing, the availability of redundant alternatives—either deliberately introduced or intrinsically present in the system—can be used as test oracles: each test is executed by all different alternatives, and if all executions produce the same results and lead to the same states, then the test is considered passed, while any discrepancy in the behavior of a different implementation is considered a failed test [CGG+14, DF94, MFC+09, Wey82]. In automatic program repair, intrinsic redundancy is the main ingredient to overcome failures and fix faults: when a failure is detected, the code is "evolved" by replacing code fragments with alternative (redundant) code snippets towards the correct functionality, as defined by an acceptance test suite [AY08, WNGF09].

Regardless of the source of software redundancy (deliberate or intrinsic), its level of abstraction (method, component, system, etc.), and its potential use (fault tolerance, testing, automatic repair, etc.) there is no definition of *what is software redundancy*, that is there is no formalization of the intuitive concept of redundancy. In addition, a common limitation of all the techniques that use redundancy is that, while they seek to exploit redundancy, they provide no mechanisms to assess *how much redundancy is actually there* to be exploited. For example, different teams may work on independent versions of the same component for a safety critical system, but the resulting versions may turn out to be based on the same underlying algorithm, and therefore they might be susceptible to correlated failures [BKL90, Hat97, KL86]. In essence, the developers are not able to assess the level of independence of the several versions, and thus the effectiveness of the fault tolerance capability of the system.

In the context of intrinsic redundancy, the issue of effectively exploiting redundancy is exacerbated by the unavailability, at design time, of the set of the alternative implementations. In fact, due to the nature of this form of redundancy, the various alternatives stem spontaneously in the system and there is no formal documentation on such redundant functionalities, as opposed to N-version programming. As a result, the developers have to manually identify *where are the alternative implementations*. This

is a tedious and error-prone operation, and it is a showstopper for the extensive and profitable use of intrinsic redundancy.

This thesis proposes to investigate software redundancy, defining approaches to characterize quantitatively as well as qualitatively redundancy. The intuition is that a systematic and formal investigation of software redundancy will lead to more, and more effective uses of redundancy. The first major contribution of the thesis is the formulation of a *notion* of software redundancy. We are interested in the redundancy at the code level. More specifically, we define redundancy as a relation between fragments of code. In essence, two code fragments are redundant if they perform the same equivalent functionality and at the same time their executions differ. We say that two code fragments are functionally equivalent when their execution produces the same result and does not cause any difference in the behavior of the system. In other words, the two fragments produce the same result and equivalent state changes. Two executions are different when the code fragments perform dissimilar sequences of instructions. Specifically, two execution traces may differ in the set of executed instructions, in the order at which the instructions are executed, or both. For example, in a concurrent system the interleaving leads to execution traces that contain the same instructions but in a different order, even if the code being executed is identical.

Our notion of redundancy is general and abstract, but it is ultimately undecidable and thus not directly usable in practice. In fact, equivalence alone is undecidable in the general case, and the dissimilarity between executions should be computed over an infinite set of possible traces. We thus propose a *practical* measure of redundancy based on our notion, but that is feasible and efficient. To measure functional equivalence between code fragments, our measure limits the space of possible executions to a finite set of executions that we obtained from both automatically generated and developers' hand-written tests. To compute diversity between executions, we use a particular form of execution traces in which we log the read and write operations performed during the execution that resulted in changes to the application's state, and a specific form of edit distance between such traces.

We show through an extensive empirical evaluation that our measure, and therefore our notion of redundancy, is both consistent and useful. First, we validate our measure through micro-benchmarks to confirm the correctness and effectiveness of our notion of redundancy. We then use the method to quantify the level of redundancy in a number of case studies to assess the usefulness and significance of the proposed measure itself. The results obtained show that our measure is a good indicator of redundancy. In particular, our measure distinguishes shallow redundancy—where two apparently different code fragments reduce the same code being executed underneath—from deeper redundant code, from algorithmic redundancy, where not only the code being executed is different but also the algorithmic nature of the computation differs considerably. Furthermore, we also demonstrate that our measure is a good predictor of the effectiveness of techniques that exploit redundancy, for example the Automatic Workarounds technique [CGM+13,

CGPP15].

The second major contribution of the thesis is the definition of a technique to automatically *identify* redundancy in software systems. More specifically, we aim to automatically identify the alternative implementations already available in a software system. Such a technique can help to both spread the use of those techniques based on intrinsic redundancy, and help studying the pervasiveness of such form of redundancy. Our approach automatically extracts methods or sequences of methods that are functionally equivalent to a target input method. For example, given a method that adds values on the top of a stack, we are interested in identifying the complete set of methods or combination of methods that perform the same equivalent functionality.

We define an approach that can efficiently explore the search space to effectively identify equivalent method sequences. As for our measure of redundancy, our technique synthesizes equivalent code fragments by restricting the number of possible executions to a finite set, that we call execution scenarios. First, we generate a candidate solution that computes the same result and exhibits equivalent side-effects on the application's state when executed on the chosen execution scenarios. We then validate the candidate solution through a second step in which we explore new, additional execution scenarios. If the approach identifies an execution that invalidates the equivalence, we prune the search space from such a spurious result. Otherwise, we deem the synthesized code fragment as likely-equivalent, to indicate that they may behave differently for executions not considered in the process. We demonstrate through an exhaustive empirical validation that our approach is both effective and efficient in identifying equivalent methods or combination of methods. In particular, we show that such technique correctly identifies a large amount of the possible equivalences where redundancy was known to exist.

1.1 Research Hypothesis and Contributions

The main research hypothesis of this thesis is that:

> *Software systems are redundant, in the sense that they can provide the same functionality though different executions, that is they execute a different sequence of actions that leads to indistinguishable states and results. Redundancy can be automatically identified, measured, and exploited in different ways.*

The first part of the hypothesis describes the main motivation for this thesis, which is that software systems contain various forms of redundancy, and at different levels of abstraction. Although numerous studies have directly or indirectly discussed or used redundancy, nobody has ever formalized or investigated redundancy as a notion interesting by and of itself.

The second part of the research hypothesis concerns how to effectively support the use of software redundancy. There is currently no comprehensive study on how to

qualitatively and quantitatively assess redundancy. Furthermore, there are only a few studies on how to assess the semantic similarity of code fragments within a system [HK14, JS09]. Our idea is that a comprehensive investigation on how to automatically identify and measure software redundancy will lead to more, and more effective approaches that exploit redundancy. This intuition comes from some studies on the effectiveness of N-version programming [KL86, Hat97], and is built upon our experience in the design and implementation of techniques that exploit intrinsic redundancy to automatically generate and deploy test oracles [CGG+14], and to automatically avoid failures at runtime [CGPP10, CGM+13].

This thesis makes two major contributions:

A notion and measure of redundancy: The first contribution is the formalization of the notion of redundancy in software systems. More precisely, we propose an abstract and general notion of software redundancy at the code level. Two code fragments are redundant when they perform the same equivalent functionality, and at the same time their executions are different. Then, on the basis of such formalization, we propose a practical and efficient measure of software redundancy. Finally, we demonstrate the significance and correctness of the redundancy measure by correlating the measurements with the success of techniques that exploit redundancy, such as the Automatic Workarounds technique.

Automatic identification of redundancy: The second contribution of this thesis is a technique to automatically identify equivalent functionalities. In particular, we define an approach to identify methods or combination of methods that are functionally equivalent to a given target method. We propose a general approach, and a concrete implementation for Java applications and libraries. We then evaluate the effectiveness of the technique on a set of case studies where redundancy was known to exist. For each case study, we provide the list of equivalences identified and a comparison with the manual identification.

1.2 Structure of the Dissertation

The remainder of this dissertation is structured as follows:

- Chapter 2 provides an *overview* of the several techniques related to software redundancy. We present the techniques that identify redundancy in software, and techniques that exploit some form of redundancy to test programs, handle failures at runtime, or automatically repair programs.

- Chapter 3 presents the *notion* of software redundancy. It details our formalization of the intuitive concepts of redundancy as fragments of code being functionally equivalent and at the same time performing different executions.

- Chapter 4 presents our practical *measure* of software redundancy. It describes the issues of the general and abstract notion of redundancy, and provides a detailed description of the trade-offs applied to develop a practical and efficient measure. Furthermore, it presents our empirical validation on the significance and usefulness of the measure.

- Chapter 5 describes our technique to automatically *identify* equivalent methods or combination of methods, presents both the prototype implementation and the empirical results of the experiments we performed on Java Stack, Graphstream, and Google Guava to show the effectiveness and efficiency of the technique.

- Chapter 6 summarizes the contributions of this dissertation, and discusses future directions.

Chapter 2

Redundancy in Software Engineering

Software redundancy is the key ingredient of many techniques in several areas of software engineering, especially fault tolerance, software testing, and automatic program repair. This chapter overviews the main software engineering techniques that either rely on or identify redundancy. We start with a categorization of the forms of redundancy in a software system, and then describe the techniques that either identify or exploit redundancy, highlighting their limitations and motivating the main contributions of this dissertation.

In engineering, redundancy is a well-known design paradigm when safety and reliability are key requirements. In hardware and software engineering, redundancy is achieved through the replication of hardware components and software functionalities. Redundant hardware has been developed since the sixties to tolerate physical faults in circuits [LV62, PGK88]. The rationale is to design hardware that contains multiple replicas of the same components. RAID, for example, is a very successful approach that overcomes faults by replicating and allocating data on an array of disks. When a fault compromises one component, its counterpart can continue operating, and the functionality of the overall system is preserved.

The successful results obtained by using redundancy to increase hardware reliability led researchers to apply the same principles to software, in particular to increase reliability. A software system is redundant when it can perform the same functionality through different executions. The presence of alternative execution paths or execution environments is the primary ingredient of all those techniques that exploit redundancy. Differently from hardware systems, the simple replication of software functionalities cannot deal with many failures that derive from development and integration problems that often occur in software systems. For this reason, researchers have proposed several approaches based on the independent design and implementation of multiple versions of the same components [Avi85, LBK90, Ran75]. These techniques are based on the assumption that programming faults are not correlated, and therefore several independently designed and implemented versions are unlikely to fail on the same input

and system's state. These multiple versions are then executed independently, and the output of the system is determined by a majority voting algorithm from the output of the various versions.

Software redundancy has been also exploited to generate test oracles, and to automatically fix faults. For example, Weyuker proposes to generate test oracles for complex, "non-testable" programs by providing an independently developed version that fulfills the same specifications [Wey82]. The testers then run both programs on identical input data and compare the final outcomes. Yet another use of redundancy was proposed by Carzaniga et al. with the Automatic Workarounds approach to automatically recover from runtime failures. The technique avoids failures at runtime by substituting a failing method with a redundant alternative implementation that is intrinsically present in the system.

The techniques presented so far exploit a form of redundancy present in the *code*. Other approaches exploit redundancy present in different forms and at different abstraction levels, such as redundancy present in the *environment* or in the *data*.

Rejuvenation techniques exploit a form of redundancy rooted in the execution environment. Rejuvenation techniques periodically re-execute some of the initialization procedures of the application to produce a fresh execution environment, to increase the system reliability and prevent potential failures [GT07, HKKF95].

Yet other approaches rely on redundancy in the data used in the computation, and not in the computation itself. For example, data diversity techniques apply deliberate data redundancy to cope with failures that depend on the input conditions [AK88]. The techniques rely on some user-defined functions that given an input can generate logically equivalent data sets.

In this chapter, we survey the existing software engineering techniques that exploit software redundancy for various purposes. Among the various levels of redundancy that researchers have investigated—either code, environment, or input data—we focus at the code level, since it is the main concern of this dissertation. We introduce the concepts of deliberate and intrinsic redundancy in Section 2.1. We analyze the main techniques that exploit such forms of redundancy for various purposes in Section 2.2. We discuss the techniques that can be used to automatically identify redundancy in a software system in Section 2.3.

2.1 Software Redundancy

Redundancy can be either *deliberately* introduced in the design or *intrinsically* present in the system. The main difference between deliberate and intrinsic redundancy is in the developer's *intention*. In the case of deliberate redundancy, a developer intentionally introduces redundancy in the system for a specific ultimate purpose. In contrast, intrinsic redundancy naturally stems as consequence of the combination of several aspects of the design and development process, and can be exploited for other ultimate purposes.

Different design decisions and purposes may lead to different redundancy categorizations. In this thesis, we focus on the reliability of software systems, and we thus consider redundancy as deliberate only if it was explicitly added to improve the reliability of the system.

Deliberate redundancy. Some techniques deliberately add redundancy to the system. This is the case of all those techniques that replicate the design process to produce redundant functionalities. N-version programming, for example, proposes to deliberately design the system with *N* replicas of the same component as part of the fault-tolerance infrastructure that enables the system to operate and provide correct outputs despite the presence of faults [Avi85]. Each version in an N-version system is a complete implementation of the same specification designed and developed independently from the *N-1* other versions.

Figure 2.1 presents an example of N-version programming by Chen and Avizienis [CA78]. The code presents a function that converts an array of 10 ASCII coded digits into a binary number. The conversion function is replicated in three different procedures such that the overall system can tolerate one, and only one, failure. Each independent version contains supplementary boilerplate code that wraps the main functionality and calculates the final result through majority voting. In particular, lines 12-14 are the core fragment of the function and are designed in the replicas to use different algorithms to convert the array of digits. By developing several versions separately, it is assumed that they will be based on designs that are different, and thus not susceptible to correlated failures. Deliberately introducing redundancy into software system for reliability purposes is generally expensive. In particular, separate teams must work on independent code bases, and all the artifacts must pass through the entire development life cycle. This is the main reason why techniques based on the deliberate introduction of redundancy in the system have been applied primarily in those industrial settings where the cost of failure is not acceptable, such as the aerospace industry.

Intrinsic redundancy. Redundancy can also be intrinsically present in a system. Consider, for example, the methods put(K key, V value) and putAll(K key, Iterable<? extends V> values) of the AbstractMultimap<K,V> class of the popular Google Guava library.[1] The put and putAll methods associate a given key with a value and collection of values, respectively. This suggests that put(k,v) would be equivalent to putAll(k,c) with the same key k and a collection c containing a single value v. Figure 2.2 reproduces the code of the two methods with cosmetic code formatting changes. The structure of the two methods is dissimilar, however the execution of the two methods may in turn invoke the same underlying code. Consider for example line 12 of put and line 13 of putAll where, depending on the dynamic binding, both methods may invoke the same implementation of add(value) for Java collections. Another similar example regards line 4 of put and line

[1] https://github.com/google/guava

```
1  CONVERSION1: PROCEDURE OPTIONS (TASK);
2   DCL DIGITS(10) BINARY FIXED(6) EXTERNAL;
3   DCL NUMBER1 BINARY FIXED(31) EXTERNAL;
4   DCL (SERVICE1,COMPLETE1) EVENT EXTERNAL;
5   DCL (DISAGREE1,GOODBYE) BIT(1) EXTERNAL;
6   DCL FINIS BIT(1) INIT('0'B);
7   DO WHILE (¬FINIS);
8    WAIT (SERVICE1) ;
9    COMPLETION (SERVICE1) = 'O'B;
10   NUMBER1 = 0;
11   IF ¬GOODBYE & ¬DISAGREE1
12   THEN DO I = 1 TO 10;
13    NUMBER1=NUMBER1*10+DIGITS(I)-60;
14    END;
15   ELSE FINIS = '1'B;
16   COMPLETION (COMPLETE1) = '1'B;
17  END;
18  END CONVERSION1;
```

```
1  CONVERSION2: PROCEDURE OPTIONS (TASK);
2   DCL DIGITS(10) BINARY FIXED(6) EXTERNAL;
3   DCL NUMBER2 BINARY FIXED(31) EXTERNAL;
4   DCL (DISAGREE2,GOODBYE) BIT(1) EXTERNAL;
5   DCL (SERVICE2,COMPLETE2) EVENT EXTERNAL;
6   DCL FINIS BIT(1) INIT('0'B);
7   DO WHILE (¬FINIS);
8    WAIT (SERVICE2) ;
9    COMPLETION (SERVICE2) = 'O'B;
10   NUMBER2 = 0;
11   IF ¬GOODBYE & ¬DISAGREE2
12   THEN DO I = 1 TO 10;
13    NUMBER2=NUMBER2*10+MOD(DIGITS(I),60);
14    END;
15   ELSE FINIS = '1'B;
16   COMPLETION (COMPLETE2) = '1'B;
17  END;
18  END CONVERSION1;
```

```
1  CONVERSION3: PROCEDURE OPTIONS (TASK);
2   DCL DIGITS(10) BINARY FIXED(6) EXTERNAL;
3   DCL NUMBER3 BINARY FIXED(31) EXTERNAL;
4   DCL (DISAGREE3,GOODBYE) BIT(1) EXTERNAL;
5   DCL (SERVICE3,COMPLETE3) EVENT EXTERNAL;
6   DCL FINIS BIT(1) INIT('0'B);
7   DO WHILE (¬FINIS);
8    WAIT (SERVICE3) ;
9    COMPLETION (SERVICE3) = 'O'B;
10   NUMBER3 = 0;
11   IF ¬GOODBYE & ¬DISAGREE3
12   THEN DO I = 1 TO 10;
13    NUMBER3 = NUMBER3+(DIGITS(I)-60)*10**(10-I);
14    END;
15   ELSE FINIS = '1'B;
16   COMPLETION (COMPLETE3) = '1'B;
17  END;
18  END CONVERSION3;
```

Figure 2.1. An example of N-version programming implementation of a function that converts an array of 10 ASCII coded digits into a binary number [CA78].

```
1  public boolean put(K key, V value) {
2   Collection collection = map.get(key);
3   if (collection == null) {
4     collection = createCollection(key);
5     if (collection.add(value)) {
6       totalSize++;
7       map.put(key, collection);
8       return true;
9     } else {
10      throw new Exception();
11    }
12  } else if (collection.add(value)) {
13    totalSize++;
14    return true;
15  } else {
16    return false;
17  }
18 }
```

```
1  public boolean putAll(K key,Iterable values){
2   if (!values.iterator().hasNext()) {
3     return false;
4   }
5   Collection c = getOrCreateCollection(key);
6   int oldSize = c.size();
7   boolean changed = false;
8   if (values instanceof Collection) {
9     Collection c2 = Collections2.cast(values);
10    changed = c.addAll(c2);
11  } else {
12    for (V value : values) {
13      changed |= c.add(value);
14    }
15  }
16  totalSize += (c.size() - oldSize);
17  return changed;
18 }
```

Figure 2.2. Methods put and putAll of the AbstractMultimap<K,V> Class from the Google Guava library

5 of putAll where, depending on the implementation of the method getOrCreateCollection(key), both methods may invoke method createCollection(key). Still, despite these similarities, the put and putAll methods are implemented with substantially different code, and more importantly, different algorithms.

Compared to deliberate redundancy, the main advantage of using intrinsic redundancy is that it does not incur serious additional costs when it is exploited for reliability purposes. However, it should be clear that the effectiveness of approaches that exploit intrinsic redundancy still rely on code redundancy, and is bound to the existence of such type of redundancy. Therefore, the first main question is whether intrinsic redundancy actually exists in modern software systems. Some studies show that the presence of semantically equivalent code at the level of code fragments in software is quite high [GJS08, JS09]. In essence, these studies implicitly suggest that intrinsic redundancy does exist in real projects.

More recently, Carzaniga et al. performed an empirical study to validate the presence of intrinsic redundancy at the method level in several Javascript libraries [CGPP10, CGPP15]. For each case study, they created a collection of code rewriting rules, which encode pairs of code fragments with the same observable behavior. In particular, two code fragments have the same observable behavior when they produce the same expected effect—from the perspective of an external observer—for all relevant inputs. A code rewriting rule is a syntax-based transformation rule that substitutes a code fragment with a different code fragment. The authors manually inspected the documentation of several well-known libraries, such as Google Maps, YouTube, and JQuery, and created a database of more than 350 rules.

Assessing quantitatively as well as qualitatively the pervasiveness of intrinsic redun-

Case study	Classes considered	Rewriting rules found	Avg per class
Apache Ant	213	804	3.80
Apache Lang3	5	45	9.00
Apache Lucene	160	205	1.28
Apache Primitives	16	216	13.50
Canova	95	345	3.63
CERN Colt	27	380	14.07
Eclipse SWT	252	1494	5.93
Google Guava	116	1715	14.78
GraphStream	9	132	14.67
Oracle JDK	2	85	42.50
Joda-Time	12	135	11.25
Trove4J	54	257	4.76
Total	961	5813	6.04

Table 2.1. Rewriting rules found for representative Java libraries and applications.

dancy in software systems is an important aspect for effectively studying and exploiting intrinsic redundancy. We have extended the empirical study by Carzaniga et al. in two directions.

First, we have extended the investigation from dynamically typed languages, such as Javascript, to a statically typed language, such as Java. Java is a statically typed and object-oriented language, by extending the study to other important classes of programming languages, we have also assessed the relation of intrinsic redundancy with some features of the programming language used.

Second, we have investigated the pervasiveness of intrinsic redundancy in both libraries and applications. Software libraries are amenable to contain more redundancy simply because they are designed to offer the same functionality in different forms and through different interfaces, and that diversity of forms and interfaces leads quite naturally to redundancy.

In Table 2.1 we report the results of our investigation where we studied 12 open source projects. We analyzed three applications, a build process manager (Apache Ant), a high-performance search engine (Apache Lucene), and a machine-learning system (Canova). The libraries considered in the investigation provide support for additional language features (Apache Lang3), supplementary data structures (Apache Primitives, CERN Colt, Google Guava, Trove4J), additional GUI features (Eclipse SWT), dynamic graph support (Graphstream), and increased time and date support (Joda-Time). We also analyzed some classes from the Java Standard library itself (Oracle JDK). In summary, we analyzed 961 classes taken from the 12 projects and manually extracted a set of rewriting rules to capture redundancy.

We found 5813 rewriting rules in the considered 961 classes, that is an average

of more than six rewriting rules for each class. This is a very interesting result since it shows that there is a good amount of redundancy intrinsically present in software. Moreover, it is an important outcome for those technique that use redundancy, since it shows that intrinsic redundancy is not a rare phenomenon.

Despite the fact that we found redundant functionalities in *all* software systems, the amount of redundancy varies across projects, and between libraries and application. For example, Apache Primitives and Trove4J are similar libraries that provide developers with additional collections optimized for primitive values. However, the rewriting rules extracted from Apache Primitives per class are on average three times more than those extracted from Trove4J. Similarly, we can observe that libraries present more redundancy than programs. In our evaluation, we extracted an average number of 9.04 rewriting rules per class from the libraries under analysis, while we extracted on average only 2.89 rewriting rules from the applications.

The results of our investigation show that there are plausible and general reasons to assume that modern software systems, especially modular components, are intrinsically redundant: design for reusability, performance optimization, backward compatibility, and lack of software reuse. We now briefly analyze each reason.

Modern development best practices naturally induce developers to design their libraries with high flexibility to improve reusability. Such *design for reusability* is a source of intrinsic redundancy. In fact, the required flexibility of well designed reusable components leads to interfaces that are intended to offer different ways of executing the same functionalities. As an example, consider JQuery, a popular and actively developed library for Web applications.[2] JQuery offers many alternatives to display elements in a Web page: show(), animate(), fadeTo(), fadeIn(), etc. Although these functions differ in some details, they are essentially equivalent in terms of the end result. Similarly, many Graphical User Interface libraries provide different methods for organizing and drawing graphic objects. For instance, the Eclipse SWT library[3] provides developers with the ability to draw a rectangle and a line with methods drawRectangle(Rectangle rect) and drawLine(int x1, int y1, int x2, int y2) respectively. The SWT library provides also a method drawPolygon(int[] pointArray), to draw a rectangle, a line or a generic polygon, depending on the number of points given as argument. Thus, the method drawPolygon(int[] pointArray) is redundant with respect to methods drawRectangle(Rectangle rect) and drawLine(int x1, int y1, int x2, int y2), since it provides equivalent functionality. Furthermore, methods drawRectangle, drawLine, and drawPolygon execute at runtime substantially different code. These methods have been designed and documented to allow developers to fine-tune the drawing process.

Many other examples are easy to find in container libraries, such as Google's Guava library, or the util package of the Java Class Library. Containers typically offer methods to add single elements (for example, add(Object e)) and methods to add a set of

[2] http://jquery.com
[3] https://www.eclipse.org/swt

elements elements from another container (for example, addAll(Collection c)), which are redundant when the latter is used with a container that contains a single element. Conversely, add is redundant with respect to addAll when it is invoked to insert each element of the collection c, one at a time.

Optimizations of non-functional requirements are another source of intrinsic redundancy. For example, the function StringUtils.endsWith(String s, String suffix) of the Apache Ant[4] application reimplements the method endsWith(String suffix) of the Java String class more efficiently, the method CollectionUtils.frequency() reimplements Collection.frequency() of the Java Standard library, the method tokenizePathAsArray() of the SelectorUtils class reimplements tokenizePath(), etc. Similarly, the log4J[5] library implements many of the functionalities offered by the standard Java class java.util.Logging with improved performance.

In other cases, the same library might offer the same service in multiple variants, each one optimized for different cases. An excellent example is the sorting functionality: good implementations of a sorting functionality use different algorithms that implement the same interface function. For example, the GNU Standard C++ Library basic sorting function implements the insertion-sort algorithm for small sequences, and merge-sort for the general case. Similarly, different functions may implement the same functionality with different optimization targets, for instance to minimize the memory footprint or to compute the result in less time.

Redundancy may stem from the need to guarantee *backward compatibility*. A library might maintain different versions of the same components to ensure compatibility with previous versions. For example, the Java 7 Class Library contains at least 45 classes and 365 methods that are deprecated and that overlap with the functionality of newer classes and methods.[6]

Software redundancy may also be unintentionally included in software due to *lack of software reuse*. Sometimes, developers are simply not aware that a functionality is already available in the system, and may implement the same functionality multiple times [BEHK14, BVJ14, KR09]. For example, in the same project a developer may rely on a third-party library for some container implementations, while another developer may implement the data structure from scratch ignoring the availability of the third-party library. Similarly, several developers—or even the same developer—might implement the same equivalent function several times and in different ways for the benefit of simplicity. Such duplication of logically similar components might be a natural consequence of stringent time schedules. The poor quality—or the lack—of documentation exacerbates this problem. Environmental and personal issues among developers, such as lack of communication and trust, are yet another cause of scarce software reuse, which

[4]http://ant.apache.org
[5]http://logging.apache.org/log4j
[6]http://docs.oracle.com/javase/8/docs/api/deprecated-list.html

consequently leads to the introduction of redundant functionalities into software systems.

Regardless of the sources of redundancy, software systems *are* intrinsically redundant, and software modules expose several operations that are functionally equivalent, or that can become functionally equivalent once suitably combined. The presence of implicit redundant elements in software systems has been exploited in several ways that we briefly survey in the reminder of this section.

2.2 Applications of Software Redundancy

In this section, we survey the main approaches the exploit software redundancy, focusing on both deliberate and intrinsic redundancy. Deliberate redundancy is mainly used for guaranteeing high reliability in critical applications by preventing or avoiding failures at runtime. Intrinsic redundancy has been explored in some different areas, from fault tolerance to software testing and automatic program repair.

In Section 2.2.1, we survey the main techniques that deliberately introduce redundancy into software systems for various purposes. In Section 2.2.2 we survey the main techniques that exploit the redundancy intrinsically present into systems for self-healing, test oracle generation and automatic program repair. In Section 2.2.3 we discuss the limitations of the techniques that exploit deliberate and intrinsic software redundancy.

2.2.1 Deliberate Redundancy

Deliberate software redundancy is widely exploited at the code level. Classic techniques as N-version programming and recovery-blocks explore software redundancy to tolerate software failures. Pseudo-oracles rely on redundancy to reveal faults. Other techniques exploit redundancy for producing self-checking and self-optimizing code to overcome either a wide variety of faults, or performance problems. Wrappers represent yet another form of deliberate redundancy present in various contexts. Recently, deliberate redundancy has been also explored to automatically repair corrupted data structures.

N-version programming. N-version programming has been originally proposed by Avizienis et al. and is one of the classic techniques to design fault tolerant systems [Avi85]. N-version programming relies on several versions of the same component that are executed in parallel. The different versions of the same component shall be designed and developed independently by different teams to reduce the dependency among faults in the different versions. The results of the computations are compared, and the overall result is decided through a voting mechanism. The voting algorithm compares the results, and selects the final output based on the output of the majority of the versions. Since the final output needs a majority, the number of redundant alternatives determines the number of tolerable failures. To tolerate k failures, a system must consist

of $2k+1$ versions. For instance, to tolerate two faulty results a system must consist of five independent versions, and so on.

The original N-version programming mechanism has been extended to different domains, for example to increase the reliability of Web applications and service-oriented computing. Service-oriented computing foster the implementation of various versions of the same service. These implementations are designed and executed independently, possibly offering different level of quality of service, but all of them complying with a common interface. The availability of multiple independently developed versions of the same or similar services has been successfully exploited by researches to improve the reliability of service-oriented applications.

Looker et al. propose a mechanism to increase the reliability of service-based application. The authors define a framework to run several independently-designed Web services in parallel. The results are then validated on the basis of a majority voting algorithm [LMX05]. Dobson proposes to implement N-version programming as an extension of WS-BPEL. The services are run in parallel, and the obtained responses are validated by a voting algorithm [Dob06]. These techniques are effective in case of failures caused by malfunctioning services or by unanticipated changes in the functionalities offered by the current implementation.

Gashi et al. describe and evaluate another application of N-version programming to SQL servers [GPSS04]. In a nutshell, the authors studied four off-the-shelf SQL servers to understand whether their simultaneous usage could be exploited for fault tolerance. In their study, the authors selected faults from each of the servers, and tried to investigate whether the other servers were affected by the same exact fault. They observed that only four faults were replicable on two different SQL servers, and that no fault was found in more than two servers. On the basis of this investigation, Gashi et al. propose an architecture for a fault-tolerant database management system [GPS07]. However, comparing the output and the state of multiple, heterogeneous SQL servers may not be trivial, due to concurrent scheduling and other sources of non-determinism.

N-version programming has been designed and investigated to overcome development faults. However, if N-version programming is combined with distinct hardware where to run each replica, it can tolerate also some classes of physical faults. This possibility makes N-version programming particularly attractive in the context of fault tolerance for service-oriented computing, where redundant services can be executed on different, geographically separated servers, thus overcoming service unavailability due to server or network problems.

Recovery-blocks. Recovery-blocks were originally proposed by Randell, and rely on the independent design and implementation of multiple versions of the same component, but differently from N-version programming, the various versions are executed sequentially instead of in parallel [Ran75]. Recovery-blocks execute an alternative version when the running component fails. If the alternative version fails as well, the technique selects a

new alternative, and this process continues as long as either the system continues to fail, or further alternative components are available. The recovery-blocks mechanism detects failures using acceptance tests, and relies on a checkpoint-recovery mechanism to bring the state of the application to a consistent state before trying the execution of an alternative version.

The recovery-blocks core ideas have been extended to Web applications and service-oriented computing [Dob06, GH04, KPR04]. Dobson applies recovery-blocks to web applications by exploiting the *retry* command of the BPEL language to execute alternative services when the current one fails [Dob06]. Gutiérrez et al. implement recovery-blocks exploiting agent-oriented programming (AOP), a programming paradigm where the construction of the software is centered on the concept of software agents. An agent provides abstractions similar to objects in object-oriented programming, with interfaces and messaging capabilities at its core. Gutiérrez et al. identify both critical points in a process and redundant Web services. The developers are responsible to implement both the invocations of the Web service, as the main functionality of the agent, and the evaluation of the correctness of the result. At runtime, if an agent fails, the others can continue operating without affecting the overall functionality of the system. As in the classic recovery-blocks technique, all these approaches exploit alternative services that are provided by developers at design time.

Similarly to N-version programming, the recovery-blocks technique targets development faults, but differently from N-version programming it executes the redundant alternatives in sequence and not in parallel.

Self-checking programming. Yau et al. first, and Laprie et al. later, further extend and enhance the ideas of N-version programming and recovery-blocks by introducing self-checking programming [LBK90, YC75]. In a nutshell, self-checking programming augments programs with code that checks the system's dynamic behavior at runtime.

The general and abstract concept of adding self-checking capabilities to programs has been implemented in several ways. A self-checking component can be implemented either as a software component with a built-in acceptance test suite, or as some independently designed components that are compared for correctness (similar to N-version programming). Each functionality is implemented by at least two self-checking components that are both designed and developed independently, and then executed in parallel. If the primary self-checking component fails, the program automatically checks the results produced by the alternative versions of the component to produce the correct result. During the execution, the components are classified as either "acting" components, which are responsible for the computation, or "hot spare" components, which are executed in parallel as backup of acting components to ultimately tolerate failures. An acting component that fails a self-check is discarded and replaced by one of its hot spares. By implementing this replacement strategy, self-checking programming does not require any checkpoint and recovery mechanism, which is essential for recovery-blocks,

but potentially introduces a significant overhead.

As in the case of N-version programming and recovery-block, Dobson implements self-checking programming for Web application and service-oriented computing. Dobson implements self-checking programs by calling the several redundant services in parallel and considering the results produced by the hot spare services only in case of failures of the main acting service [Dob06].

Similarly to N-version programming and recovery-blocks, self-checking programming targets development faults.

Wrappers. Deliberate redundancy has been exploited in several different contexts in the form of *wrappers*. A wrapper is defined as an element that mediate the interactions between components to solve integration issues. Using wrappers for fault tolerance was originally proposed by Voas [Voa98], who proposed the development of protectors (wrappers) to improve the overall system dependability. The goal is to protect both the system against erroneous behavior of a Commercial Off-The-Shelf (COTS) component, and the COTS component against misuses of the system.

Thanks to the generality and flexibility of the idea, the use of wrappers to mediate the communication has been extended to various contexts. Salles et al. exported the original idea in the context of operating systems [SRFA99]. They propose wrappers to increase the dependability of OS microkernels that communicate with COTS components, with different dependability levels. A wrapper verifies the consistency of constraints against to executable assertions that model the expected behavior of the functionality. The executable assertions are specified at design time by developers, requiring additional effort as in the case of N-version programming and recovery-blocks. Salles et al. apply wrappers to microkernels, as they are composed of few functional classes (for example, synchronization, scheduling, and memory management), whose specifications are simpler to understand and thus amenable to modeling their expected behavior.

Popov et al. propose wrappers to avoid failures that derive from incomplete specifications, misuse, or failures in the external system in which the COTS are integrated [PRRS01]. Incompletely specified COTS components may be used incorrectly or in conditions that are not the ones they have been designed for. The wrappers defined by Popov et al. detect classic mismatches, for example whether the input is outside the acceptable value domain, and triggers the correspondent appropriate recovery action, for example the switch to a redundant alternative as in the recovery-blocks technique. Chang et al. require developers to release components together with a set of predefined recovery actions that can deal with failures caused by common misuses of the components [CMP08, CMP09]. Similarly, Denaro et al. applies developer-written adapters to avoid integration problems among Web services [DPT07, DPT13].

Fuad et al. present a technique that introduces self-healing capabilities in distributed systems by means of wrappers that deal with runtime failures [FDO06, FO07]. Their wrapper detects and reacts to failures by applying some user-defined actions to overcome

the failure and re-initialize the computation. The failures that cannot be handled at runtime cause the termination of the application, and produce a log that can help developers implement an ad-hoc wrapper to handle the future occurrences of such failures.

Deliberate redundancy has been exploited to increase the security of software systems too. Baratloo et al. present two different approaches to prevent buffer overflows on the stack [BST00]. The first method intercepts all the calls to library functions that are known to be vulnerable to buffer overflows. A redundant alternative version of the corresponding function implements the original functionality but ensures that any memory read and write are contained within the current stack frame. The second method instruments the binary of the application to force the verification of critical accesses to the stack before their use.

Fetzer et al. introduce the concept of "healers" to prevent some classes of malicious buffer overflows on the heap [FX01]. A healer is a wrapper that intercepts all the function calls to the C library responsible for writing on the heap. The healer performs suitable checks on the memory boundaries accessed to prevent buffer overflows.

Wrappers deliberately introduce redundancy in software systems to prevent both development faults, and malicious security attacks.

Exception handling. Exception handling is the most common software mechanism for catching predefined classes of errors and activating recovery routines [Cri82, Goo75]. The basic mechanism of exception handling has been explored and extended in different application areas.

Applications of exception handling to service-oriented systems extend the classic mechanism with a registry of rule-based recovery actions. The registry contains a list of predefined failures to which the system should react, and a set of rules created by developers at design time as recovery actions. Baresi et al. and Modafferi et al. propose registry-based techniques for service-oriented BPEL processes. Both techniques detect failures at runtime by observing the violation of some predetermined safety conditions. The approaches differ in the way they define and execute rules and recovery actions. Modafferi et al. extend a standard BPEL engine with an additional manager that intercepts the message exchange between the application and Web services and, transparently to the BPEL engine, influence and replace the Web services based on the rules [MMP06]. Baresi et al. define the recovery strategies as compositions of atomic actions called strategy steps that can be combined and executed alternatively or together [BGP07].

Cabral et al. propose an approach to improve the classic exception handling mechanism by automating some basic recovery actions [Cab09, CM11]. Their idea is based on a field study on the exception handling practices in Java and C# [CM07]. In such study, the authors observe that many exceptions are handled with application independent recovery actions that can be executed automatically. For instance, when a

`DiskFullException` occurs, instead of directly throwing an exception—for which the programmer has to explicitly provide an exception handling block—the runtime system tries to remove temporary files to reduce the disk usage. Relying on general predefined recovery actions can ease the developer's effort of dealing with application-agnostic exceptions.

Mechanisms that rely on exception handlers and rule-based recoveries deliberately add redundant code to address development faults.

Data structure repair. Several techniques exploit redundancy—expressed in form of specifications—to guarantee the consistency of data structures.

The original idea of data structure repair was introduced by Demsky and Rinard [DR03, DR05], who propose a framework that use consistency constraints for data structures specified by developers, to automatically detect and repair constraint violations at runtime. The approach relies on two different views of the data structure: a concrete view at the memory level and an abstract view at the level of relations between objects. The abstract view eases both the specification of high-level constraints and the algorithms and knowledge required to repair any inconsistencies. Each specification contains a set of models of the high level data structure and a set of consistency constraints. Given these rules and constraints, the technique first automatically generates the model, then inspects the model and the data structures to find violations of the constraints, and finally repairs any violations. Demsky and Rinard later improved their work by integrating their framework with Daikon to automatically infer likely consistency constraints [DEG+06].

Elkarablieh et al. presented a similar technique to repair data structures, and a tool called Juzi [EGSK07, EK08]. The technique relies on program assertions to automatically detect inconsistencies in complex data structures. The repair procedure invokes symbolic execution to systematically explore all possible mutations of the data structure. The repair algorithm uses on-demand dynamic symbolic execution (DSE) that treats corrupt fields symbolically, and computes path conditions for the corrupted fields. The algorithm solves the path condition with a theorem prover to determine the correct value for repairing the field. Similarly, Hussain and Csallner exploit DSE to repair complex data structures, by mutating the data structure in such a way that the repaired data structure takes a predetermined (non-failing) execution path [HC10]. The main difference between the two approaches resides in the way they exploit DSE. Elkarablieh et al. use DSE to systematically explore all possible mutation aiming to repair the data structure, whereas Hussain and Csallner use DSE to obtain the faulty path condition. They obtain the correct repair action by suitably inverting the conjunctions in the faulty path condition and solving the resulting constraints. In this way, the technique can both handle generic repairs, and scale to larger and more complex data structures.

Data structure repair techniques verify data structure consistency relying on constraints, which are redundant specifications that have been deliberately added to code. All these techniques focus on identify and fix deterministic development faults.

Oracles. Deliberate redundancy has been exploited in the context of software testing to deal with the oracle problem. Weyuker exploits software redundancy for "non-testable" programs that she defines as programs for which either an oracle does not exist or is impractical, since it is too complex to determine the correctness of the outcome [Wey82]. Weyuker addresses the problem by introducing *pseudo-oracles* that amount to independently written programs that satisfy the same functional specifications of the program under analysis. By running both programs on the same input data, it is possible to reveal the presence of faults if the independently computed results are not identical. One redundant alternative is sufficient for generating a pseudo-oracle. If however the goal is to identify the faulty program, the approach requires at least two alternative versions, as in the case of N-version programming.

Pseudo-oracles share the same principles and intuitions of N-version programming, and they are thus meant to detect any development fault.

Self-optimizing code. Software redundancy has been exploited to overcome non-functional failures [DMM04, NG07]. The term self-optimization is commonly used in the research area of autonomous systems, and refers to an automatic reaction of a system aimed to either compensate or recover from performance issues. The main approaches to self-optimization rely on deliberate redundancy. Diaconescu et al. propose a self-optimization approach based on the development of multiple versions of the same functionalities, each version optimized for different runtime conditions [DMM04]. The application performance is collected at runtime to detect potential performance problems, and the application adapted to emerging performance issues by selecting and activating the suitable optimal alternative based on the current running context.

Naccache et al. exploit a similar idea in the domain of service-oriented computing [NG07]. The approach enhances Web services applications with mechanisms that switch among several implementations of the same service depending on the chosen quality of the service. Given the required performance characteristics of the Web service, the framework automatically selects the most suitable implementation among the available services.

Self-optimizing techniques are meant to tolerate development faults at runtime, and in particular non-functional faults.

2.2.2 Intrinsic Redundancy

While deliberate software redundancy has been used since the seventies in a variety of different contexts, intrinsic redundancy has been explored only recently with very promising results. Intrinsic redundancy at the code level can stem from good design principles, such as design for reusability, or simply from the modularity and complexity of modern software systems. Intrinsic redundancy has been mainly exploited to increase the reliability of both service-oriented and general-purpose applications. Recently, the use

of intrinsic redundancy has been extended to improve the reliability of service-oriented applications, generate program patches and automatically generate test oracles.

Dynamic service substitution. Service-oriented computing is an emerging paradigm for distributed computing. At the core of service-oriented computing are services that provide independent computational elements that can be published, discovered, orchestrated, and invoked to build distributed and collaborating applications. The vision of service-oriented computing is to foster the implementation of alternative, functionally equivalent services. In recent years, popular services have often become available in multiple implementations. Each implementation is designed and deployed independently, offer various levels of quality of service, but ultimately complies with a standard common interface. Some researches have proposed to take advantage of this form of intrinsic redundancy by exploiting the availability of the independent implementations of the same service to increase the reliability of service-oriented applications. The most relevant research work has focused on those failures that may be caused either by malfunctioning services or by unforeseen changes in the functionalities offered by the current reference implementation.

Sadjadi et al. propose to increase the reliability of Web services by augmenting applications with self-managing capabilities. They propose to automatically substitute similar service implementations by weaving alternative services invocation at runtime, thus avoiding manual modification of the original code [SM05].

Taher et al. enhance runtime service substitution to find at runtime services by both implementing similar interfaces and introducing suitable converters that, although different, are sufficiently similar to admit to a simple adaptation [TBFM06].

Subramanian et al. enhance BPEL with new constructs to identify alternative service implementations of the same identical interfaces in order to deal with unexpected responses or unavailability issues [STN+08].

Mosincat and Binder propose a framework for runtime service adaptation that supports both stateless and stateful Web services [MB08].

In summary, dynamic service substitution amounts to take advantage of the redundancy at the code level intrinsically present in service-oriented computing to tolerate both development and physical faults.

Automatic program repair. Automatic program repair has recently gained considerable attention in the software engineering and programming languages communities. The expression *automatic program repair* refers to techniques that aim to automatically generate a fix for a fault. Researchers have explored several solutions to automatically repair programs: symbolic execution [NQRC13, TR15, MYR15], SMT solvers [DXLBM14, LR15, MKIE15], source code transformations [CH13, KNSK13, QML+14, Sha14, OPM14], machine learning [LR16] and intrinsic redundancy [AY08, CGM+13, CGPP15, DW10, LDVFW12, LNFW12, SDLLR15, WNGF09].

Automatic program repair is a broad research area that can be partitioned in two main categories: code repair and runtime repair. Automatic *code* repair techniques aim to generate patches for the faulty code under analysis. In essence, given the source code of the application and at least one failing execution, code repair techniques generate a patch that allows the application to successfully compute the desired functionality also in the failing executions.

The seminal work on automatic code repair was proposed by Weimer et al. and Arcuri et al. who investigated genetic programming as a way of automatically fixing software faults [AY08, WNGF09, LDVFW12, LNFW12]. Both approaches assume the availability of a test suite that is used to approximate the correct behavior, with at least one failing execution. The goal of these approaches is to modify the code of the application such that all test cases pass by exploiting genetic algorithms. A genetic algorithm works by first generating a set of variants of the application's source code. The algorithm then evolves the variants by exploiting some repair operators. The search terminates as soon as a new "correct" version of the program is generated.

The techniques defined by Weimer et al. and Arcuri et al. apply mutations to the faulty statements by copying similar statements defined elsewhere in the code as repair operators. The underlying assumption of these approaches is that the code fragment that implements the correct behavior is already present in the system. Barr et al. presented an empirical validation of such assumption, showing that a large quantity of the code fragments to patch the fault can be intrinsically found in the same version of the system [BBD+14]. Sidiroglou-Douskos et al. have recently proposed an approach for automatic code repair that collects code fragments for code repair from external "donor" applications [SDLLR15]. Debroy et al. defined a technique for automatic code repair which relies on a set of predefined source code mutations as repair operators [DW10].

While automatic code repair focuses on the code, automatic *runtime* repair techniques aim to avoid or, at least, to minimize the effect of failures on the current execution. This category of automatic program repair techniques is inspired by the research work on fault tolerance and self-healing systems. Several techniques for automatic runtime repair rely on either specifications or inferred behavioral models to fix faults dynamically. Dallmeier et al. developed an approach to automatically build behavioral models for Java classes by running a set of passing and failing test cases [DZM09]. Their tool, called Pachicka, compares the behavioral model inferred on the passing test cases to the model of the failing runs. Pachika then tries to produces possible fixes that amount to modifications of the model of the failing runs to make it as similar as possible with the model of the successful runs. The supported modifications on the model are limited to inserting and removing transitions, and thus can only fix a limited set of fault type.

Wei et al. propose a similar approach called Autotest that automatically detects failures in Eiffel classes, and suggests potential fixes by comparing the outcome of the passing runs to the outcome of the failing run [MFC+09, WPF+10]. Autotest relies on class specifications expressed in the form of Eiffel contracts, that consist of precondi-

tions, post-conditions, assertions, and invariants. The tool builds finite state machines representing the behavior inferred during the executing of either successful or failing test cases, and then compares the models to propose fixes. Autotest supports complex model modifications to the faulty classes, and thus they can successfully deal with more types of faults than Pachika.

Perkins et al. proposed a tool called ClearView that uses dynamic analysis to detect failures and fix faults at runtime. ClearView relies on dynamically inferred invariants computed with Daikon to automatically detect buffer overflows and illegal control flows caused by malicious security attacks [PKL+09]. ClearView builds the invariants of the correct behavior during a training session in which benign inputs and interactions where used. When deployed, ClearView re-writes the value of registers whenever the current execution violates any of the invariants.

Carzaniga et al. propose Automatic Workarounds, a technique that recover from runtime failures by automatically replacing the failing method with a redundant version [CGPP10, CGPP15, CGM+13]. Automatic Workarounds exploit the intrinsic redundancy of software systems, focusing on redundancy at the method level, which relies on the presence of redundant methods or method sequences. The authors propose to express those alternative methods as rewriting rules that capture the redundancy of a pair of method sequences. Therefore, if the execution of a method fails, the Automatic Workarounds technique first restores the state of the system prior to the failure, then replaces the failing method with an alternative implementation, and finally re-executes the code. If the execution fails again, the technique tries with a different equivalent sequence, otherwise the application continues to execute without any further intervention.

Automatic program repair techniques largely exploit intrinsic redundancy. Automatic code repair approaches exploit intrinsic redundancy at the statement level to generate patches for development faults located in code under analysis. Whereas automatic runtime repair approaches exploit intrinsic redundancy at various levels. Pachika, Autotest and ClearView exploit code redundancy at the statement level, because they are changing the code in the application to reproduce a behavior that has been already observed in previous correct executions. The Automatic Workarounds approach exploit intrinsic redundancy at the level of method calls, expressed as equivalent sequences. Automatic program repair techniques can deal with all the types of faults that span from development to interaction.

Automatic oracle generation. As in the case of deliberate redundancy, intrinsic redundancy has been explored to automatically generate oracles for test cases. Doong and Frankl first, and Gotlieb later, have exploited the symmetry intrinsically present in programs to check the correctness of test case execution.

Doong and Frankl proposed ASTOOT, a technique that relies on algebraic specifications of the component under analysis to derive self-checking test cases [DF94]. Each

test case consists of a pair of sequences of operations on the same component, along with a tag indicating whether these sequences are expected to have the same side-effects on the internal state of the component. ASTOOT automatically generates a test driver, which in turn executes the tests by first invoking the sequences of operations, and by later running an equivalence check provided by the developers to compare the resulting states. Any inconsistency between the expected result (expressed by the tag) and the equivalence check reveals a fault.

Gotlieb exploited symmetries of the program present in the form of permutations of operations to automatically check the results of the test case executions [Got03]. An example of program symmetry is a function that multiplies two input values `a` and `b` that shall produce the same output on a permutation of its input parameters. The approach generates test inputs, and exploits the symmetries identified by the developers to identify incorrect results.

Carzaniga et al. proposed a similar method that is also rooted in the idea of a pseudo-oracle [CGG+14]. They generate what they call cross-checking oracles by exploiting the intrinsic redundancy of software systems. The technique transforms redundant method pairs that developers encode in the form of rewriting rules into oracles that can be automatically deployed within a test suite. For example, the technique uses redundant alternatives to instrument the code of a test case by pairing the call to a method with a call to its redundant implementation and cross-checking that both executions are indeed equivalent. The equivalence check inspects both the outcome and the state to identify any discrepancy between the two executions.

Techniques that exploit intrinsic redundancy for testing purposes are suitable for discovering development faults.

2.2.3 Open Challenges for Software Redundancy

Deliberate redundancy has been extensively exploited in software engineering in both hardware and software. Deliberately introducing redundancy into software systems may raise costs without increasing reliability. Deliberate redundancy introduces additional development costs, since independent teams shall design and develop multiple version of the same components, and each version must pass through the whole development life cycle. The reliability improvements depend on the independence of the failures in the different versions. Faults may exhibit some level of correlation even across components that are developed completely in isolation, since the independent development teams may rely on the same algorithms, and therefore the various versions may be susceptible to correlated failures. For instance, three independent teams may work on three different versions of a component, but the resulting versions may all use merge-sort to order the internal elements, and be thus affected by similar errors.

Several studies indicate that the assumption that faults occur independently in different versions is not realistic [BKL90, Hat97, KL86]. Kelly et al. address this problem by extending the original definition of N-version programming [KMY91]. They propose to

distinguish between two different forms of diversity introduced through the independent design of several versions: random and enforced diversity. *Random diversity* refers to the independent design and implementation process as described by Avizienis et al. in their seminal work. The underlying assumption is that different programmers and designers will independently choose distinct approaches to solve a problem. Conversely, in *enforced diversity* there is an explicit effort by programmers and designers to design and implement multiple, diverse algorithms and data structures. In this case, the main assumption is that diverse data structures and algorithms are less likely to fail on the same inputs and environment conditions.

Quantifying the redundancy level between the various implementations, for example in an N-version system, still remains an open challenge. This problem arises also for intrinsic redundancy, for example a system may try to avoid a failure by applying an automatic workaround by replacing a failing method with a supposedly equivalent but hopefully non-failing alternative. In all these cases the fundamental question is: How redundant is the chosen alternative? And thus, how likely is it that the chosen alternative would avoid the failure? More in detail, is the alternative version executing substantially different code, or is it a mere different interface for the same underlying code? And even if the code is different, is the alternative radically diverse, or is it using essentially the same algorithm?

In this dissertation, we address all these questions. In Chapter 3 we provide a general and abstract notion of redundancy, and in Chapter 4 we define and implement a practical measure of redundancy based on such abstract notion.

Although not fully explored and exploited yet, intrinsic redundancy is attracting a growing interest in software engineering. As already discussed, while deliberate redundancy requires significant development effort to increase the reliability of a system, intrinsic redundancy can be exploited for the same purposes with negligible cost, once suitably identified and harnessed. To fully exploit intrinsic software redundancy, we need to address few fundamental issues about its presence, relevance, and identification.

First, although implicitly present into systems, the effectiveness of techniques that exploit intrinsic redundancy is bound to the existence of such form of redundancy. Intrinsic redundancy is likely to be incomplete, that is, it is not plausible that even the majority of code fragments in a general-purpose software system have a redundant alternative without an explicit intent of the developers. In essence, since there is no control or deliberate decisions to introduce such form of redundancy, all those techniques that want to exploit intrinsic redundancy must be opportunistic by design.

Second, as already said, it is still indispensable to quantify if and to what extent two code fragments are intrinsically redundant.

Finally, the main prerequisite for effectively use intrinsic redundancy is its identification. In fact, differently from deliberate redundancy—that is introduced consciously and by design into the system—intrinsic redundancy is the result of the combination of several factors in the development process. Redundant functionalities are thus present

in the system but they are not documented as such. It is thus necessary to first identify *where is* such form of redundancy in the system. Manual approaches based on artifacts (such as specifications) or code are time consuming, tedious, and more importantly error-prone. Automating the identification of redundancy can thus lead to both a wide-spread adoption of intrinsic redundancy, and a more effective use. In the next section, we survey the existing approaches to automatically identify redundancy in software.

2.3 Techniques to Identify Redundancy

Despite the increasing use of redundancy in different approaches, the problem of identifying redundancy in software systems has not been satisfactorily addressed yet. So far, there are no available approaches to automatically identify functionally equivalent code fragments whose executions are diverse. More specifically, the problem of identifying redundancy has not been only partially explored. In fact, most of the research work focuses on the identification of *syntactically identical* (or almost identical) code fragments, known as code clones, while the identification of *semantically equivalent* code has been investigated less. Besides techniques to identify semantic code clones, several types of specifications can be exploited to derive semantically equivalent code fragments, such as finite state machines, program invariants, and other formal specifications. Therefore, any technique that can infer these specifications can be used in a technique to identify software redundancy.

In Section 2.3.1 we survey semantic code clone detection techniques, while in Section 2.3.2 we survey some techniques that can infer finite state machines, program invariants, and algebraic specification through dynamic analysis.

2.3.1 Code Clones Detection

The research literature contains various studies on the occurrence of redundancy in software. The largest body of work in this area is focused on the identification and study of code clones. A code clone is defined as a fragment of program code that is syntactically identical, or very similar, to other code fragments either in the same project, or even in other software systems. Most of such code duplications can often be attributed to poor programming practices, caused by developers that copy-paste code to quickly develop identical or similar functionalities.

Although most of the literature work focuses on syntactic code clone [Bak95, Bak97, BPM04, CYC$^+$01, DER10, GJS08, JDHW09, KKI02, KSNM05, KDM$^+$96, LLMZ04, PTK13, RBD12], some researchers have proposed ideas and concrete techniques to assess the *semantic similarity* of code fragments within a system [GJS08, KH01, MM01]. The most relevant work is that of Jiang and Su, who consider the problem of identifying semantic clones [JS09]. Their definition of clones is based on a notion of equivalent code fragments. In essence, Jiang and Su consider two code fragments as equivalent

when they have identical effects, based on the comparison of their output values. More in detail, the authors extract code snippets from the program under analysis. Each snippet is compared to all those code fragments that have compatible input values, that is their parameter types are a permutation of the types of the selected snippet. Then, they use random input generation to execute the code snippets. Two snippets are considered identical if they compute the same output values given the same inputs. These techniques to identify semantic code clones can naturally be used to identify equivalent operations.

2.3.2 Specification Inference

Several techniques to infer specifications from dynamic analysis can be used to identify semantic equivalence in software systems.

Specification inference finds its roots in the pioneering work of Ernst et al. who proposed Daikon to infer likely program invariants from a finite set of executions [ECGN01]. Daikon inspects a program's variable during execution and generalizes the observed behavior to preconditions, post-conditions, method and class invariants. To generate such specifications, Daikon executes an instrumented version of program and analyzes the produced execution traces. On the generated traces, Daikon checks the program states collected in the traces to validate a set of invariants to see which ones hold and compute candidate program invariants. By comparing invariants, preconditions, and post-conditions collected for different methods, it is possible to identify semantically equivalent methods, although the correctness of the inferred specifications is highly dependent on the quality of the test suites used.

Csallner et al. propose DySy, a technique to increase the quality of the inferred invariants by exploiting dynamic symbolic execution [CTS08]. DySy builds the invariants through the symbolic path conditions explored while executing a user-supplied test suite. DySy then summarizes the generated path conditions as invariants. A key difference between Daikon and DySy is that techniques starting with predefined set of candidate invariants, such as Daikon, may over-approximate the inferred invariants if the set of test traces does not provide enough information to refute the generated candidates. Whereas techniques that build invariants from the observed executions, such as DySy, can generate a more precise set of invariants. However, DySy generates invariants that are more similar to symbolic summaries. Similarly, Xie et al. and Zhang et al. propose a feedback loop framework that refines the likely invariants inferred with Daikon to improve their correctness [XN03, ZYR+14]. The intuition is to either use a test case generator, or symbolic execution, to create new, addition test cases. Such test cases can then be executed to help refining the invariants generated by Daikon.

Other specification mining techniques infer finite state machine models from various types of execution traces. Dallmeier et al. propose ADABU to infer finite state machines for entire Java components, and that represent how the component changes the concrete program state after the execution of each method [DLWZ06]. The authors also propose

to exploit test case generation to explore unobserved behavior [DKM+12]. ADABU generates behavioral models that can be used to identify likely semantically equivalent methods in a software component, although the high level of abstraction they use may impact on the precision of the final result. Lorenzoli et al. propose the GK-tail technique to infer finite state machine models that represent the protocol of software components [LMP08]. In particular, the GK-tail algorithm produces FSAs annotated with data constraints. Beschastnikh et al. proposed a framework to specify model inference algorithms declaratively [BBA+13]. Although finite state machines can express equivalence among different operations, they typically abstract from the concrete events observed in the program execution. Consequently, the equivalent event sequence that can be obtained from the inferred models hold for the abstract state, but not necessarily for the complete, concrete one.

Algebraic specifications formalize component behavior in terms of functions on objects and axioms that describe the mutual relationship among such functions. Henkel et al. use reflection to get the list of methods in a Java class, and then they generate executions to infer algebraic specifications for such class [HRD08]. The axioms that they generate to describe the behavior of the class include information that can be used to infer the equivalence of method sequences. Ghezzi et al. generalize finite-state models by means of graph-transformation systems, whose rules correspond to equations of algebraic specifications [GMM09]. The main difference between the two approaches is that Ghezzi et al. produce models with a different abstraction that makes the semantic equivalence representation more precise.

2.3.3 Open Challenges for Automatically Identifying Redundancy

Software redundancy can be identified either automatically or manually. Equivalent functionalities can be derived automatically from formal specifications that—when properly extracted, expressed, and verified—offer a sound and complete description of the semantic of the software systems. Formal specifications can be provided at development time, especially for critical applications, or can be inferred from system executions.

Developers can write sets of redundant functionalities manually as an additional form of specification for their software, or identify equivalent functions from the documentation written in natural language. Such additional activity potentially impacts less on developers who are familiar with the software, since according to our experience they can write a set of correct equivalent functionalities in few hours. In contrast, less experienced developers and users may require more time and effort to effectively and efficiently write such form of specifications. Regardless of the familiarity of developers with the system, deriving redundancy specifications is an error-prone activity, especially when the semantic relation among equivalent code fragments is not trivial or even counter-intuitive. For instance, consider methods clear() and retainAll(Collection<?>

```
816 /**
817  * Removes all of the elements from this Vector. The Vector will
818  * be empty after this call returns (unless it throws an exception).
819  */
820  public void clear() {...}

883 /**
884  * Retains only the elements in this Vector that are contained in the
885  * specified Collection. In other words, removes from this Vector all
886  * of its elements that are not contained in the specified Collection.
887  *
888  * @param c a collection of elements to be retained in this Vector
889  *        (all other elements are removed)
890  * @return true if this Vector changed as a result of the call
891  */
892 public boolean retainAll(Collection<?> c) {...}
```

Figure 2.3. Documentation of methods clear and retainAll of the class Stack

c)) of the class Vector of the Java Class Library.[7] The clear() method removes all the elements contained in the container, while retainAll(c) retains all the elements of the container that are present in the collection c and removes all the others, as described by the Javadoc documentation in Figure 2.3. Although methods clear and retainAll provide almost complementary functionalities, they are functionally equivalent when the method retainAll is invoked with an empty collection, causing the method to retain no elements. Such peculiar equivalence relation is likely to be missed even by experienced developers, since it is counter-intuitive with respect to the functionality provided by the retainAll method.

To alleviate the error-proneness of manual approaches, we presented several techniques that can be used to automatically identify redundancy in a software system. Most of the research work on automation focuses on the identification of code clones. Techniques to detect semantic code clones can detect only a limited amount of equivalent functionalities in software systems. In fact, they rely either on abstractions of the source code being analyzed, or they focus on simplistic relations between code fragments, such as input/output equivalence. As a result, only a limited amount of equivalent functionalities can be identified by current approaches, and in addition such approaches are not sound, that is, some code fragments categorized as equivalent might exhibit different non-equivalent behaviors when executed in previously unseen environments or execution conditions.

When formal specifications are not provided, several techniques propose to infer them from the execution of the software system. However, in general models that are inferred from program execution are neither complete, as they build the information on the basis of the observed behavior only, nor sound, as they may either represent also failing behaviors that have been observed, or abstract and generalize over observed

[7]http://docs.oracle.com/javase/8/docs/api/java/util/Vector.html

values. Therefore, inferred models can be used to identify equivalent functionalities, but they may lead to incorrect results and are thus subject to the approval of developers, impacting on the efficiency of the overall approach.

In this chapter, we have surveyed how various forms of redundancy can be exploited for different purposes, from fault tolerance to software testing. Although redundancy has been exploited successfully in many areas and for various purposes, the notion of redundancy by and of itself has not been explored and studied. In particular, there is no formal definition of the concept of redundancy. In Chapter 3 we provide an abstract and general notion of software redundancy. We then provide a meaningful measure of software redundancy based on such notion in Chapter 4, as well as an effective approach to identify equivalent functionalities in Chapter 5.

Chapter 3

A Notion of Software Redundancy

Software redundancy has been extensively exploited in software engineering for various purposes, from fault tolerance to testing, optimization, and automatic repair. Despite the considerable number of techniques that make use of redundancy, a formal notion of redundancy is still missing. Intuitively, two code fragments are redundant when they perform the same functionality and yet their execution is different. In this chapter we present a formal notion of redundancy, which is composed of a notion of observational equivalence between code fragments and a notion of diversity between executions. The resulting formalization of software redundancy is general, since it focuses at the code level, and abstracts over implementation details or peculiar programming language features.

In the previous chapter, we introduced the various forms of redundancy that can be deliberately introduced, or intrinsically present in a software system. We also presented the numerous techniques that exploit software redundancy for several purposes: fault tolerance, test oracle generation, self-optimization, and automatic repair. In this section, we propose a formalization of software redundancy that captures an abstract and general notion of redundancy, and that constitutes the basis for defining a meaningful and quantitative measure of redundancy that we will present and exploit in Chapter 4.

Informally, we say that a system is redundant when it provides the same functionality in different ways. In particular, a software system is redundant when it produces functionally *equivalent* results through *different* executions.

Two executions are functionally equivalent when, starting from the same initial state and taking the same input values, they produce equivalent results and they lead to equivalent system states. Two states or two results are equivalent if they are either identical or observationally indistinguishable. More in detail, two states (or results) are identical if they are exactly the same state. However, we consider a broader notion of equivalence that we refer to as *observational* equivalence, as shown in Figure 3.1. We say that two states (or results) are observationally equivalent when they can not be

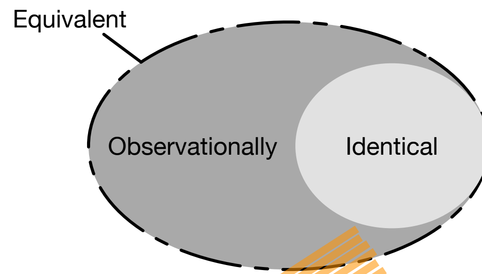


Figure 3.1. Informal visualization of functional equivalence

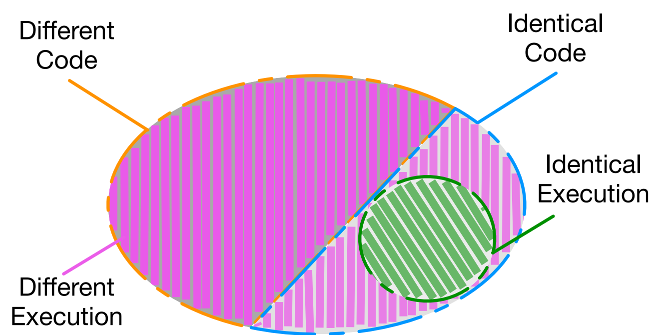


Figure 3.2. Informal visualization of execution diversity

distinguished by an external observer, that is, informally, through the system's public interface. Thus, two identical states are also observationally equivalent, but there might also be equivalent states that are not identical.

Two executions are different when they either execute different code elements or, as a minimum requirement, execute the same elements in different ways. The same code fragment can be executed in different ways due to changes in the execution environment. For instance, many programming languages provide functions to influence either the timing or the scheduling of the execution. This is the case, for example, of the sleep(int milliseconds) and setTimeout(function, milliseconds) methods in Java and Javascript respectively, which control the timing of the execution of other functions. Such methods do not alter the sequence of instructions executed by a thread, which are the exactly same, but rather change the interleaving with other threads. For instance, put(key, value) and sleep(500); put(key, value) execute the same functional code with different scheduling and the timing. Previous studies have shown that such methods are useful to create method sequences that are functionally equivalent but with different executions, and can be effectively exploited to reveal, analyze, or avoid concurrency faults like race conditions and deadlocks [EFN+02, KLNB+09, NBTU08]. In conclusion, we are characterizing the execution of different code as well as the exact same code

linkage: `int x; int y;`	

`int tmp = x;`
`x = y;`
`y = tmp;`
	`x ^= y;`
	`y ^= x;`
	`x ^= y;`

linkage: `AbstractMultimap<K,V> map; K key; V value;`	

`map.put(key, value);`	`List list = new ArrayList();`
	`list.add(value);`
	`map.putAll(key, list);`

Figure 3.3. Examples of redundant code fragments

but with a different execution schedule, as *different executions*. Figure 3.2 summarizes such notion of execution diversity with respect to functional equivalence as previously described.

We define redundancy as a relation between two code fragments within a larger system. A code fragment is any portion of code together with the necessary linkage between that code and the rest of the system. A fragment can be seen as the in-line expansion of a function with parameters passed by reference, where the parameters are the linkage between the fragment and its context. The linkage can be seen as the portion of the state that encapsulates all the possible side-effects of the execution of the code fragment. Figure 3.3 presents two examples of redundant code fragments. The figure indicates the fragments with the linkage between them and the rest of the system by listing the variables in the fragments that refer to variables in the rest of the system. All other variables are local to the fragments.

The example at the top of the figure is a code fragment that swaps two integer variables. The code fragment on the left-hand side uses a temporary variable, while the code fragment on the right side swaps the same variables by using the bitwise xor operator. The example at the bottom of the figure refers to the container class AbstractMultimap<K,V> of the Google Guava library in which one can add an individual key-value mapping for a given key or multiple mappings for the same key with methods put and putAll respectively. The methods implement different code and different algorithms, as indicated in Figure 2.2 at page 11. In both the examples reported in Figure 3.3 the code fragments are different but compute the same functionality, thus illustrating our definition of redundancy: two fragments are redundant when they are functionally equivalent and at the same time their executions are different. We now formalize these two constituent notions.

3.1 An Abstract and General Notion of Redundancy

We propose to formalize the notion of software redundancy at the code level. For instance, referring to the code fragments in Figure 3.3 we want to express the notion that it is possible to substitute every call to put(key, value) with a call to put(key, list) (where list contains only the element value) and the software system would preserve its original functionality. In essence, we want to express the notion that replacing a fragment A with a redundant fragment B within a larger system does not change the functionality of the whole system, that is executing B produces the same results as executing A without any noticeable difference in the future behavior of the system. In other words, we want B to have the same result and equivalent side-effects—or more in general equivalent state changes—as A.

Other studies on semantically equivalent code adopt a purely functional notion of equivalence, and therefore assume no visible state changes [DF94]. Yet others consider state changes to be part of the input/output transformation of code fragments, but then accept only identical state changes [JS09]. Instead, we would still consider two fragments to be equivalent even if they produce *different* state changes, as long as the *observable effects* of those changes are identical.

With the expression "observable effects" we indicate that the two code fragments have the same effect from the perspective of an external observer, that is, any element that interacts with the component. For instance, two container implementations may use different algorithms to sort the elements present in the data structure. Such algorithms may have asymptotically different performance profiles, and they may store data using different internal structures. Nevertheless, they still may be redundant from an external viewpoint if they both return a sorted set as expected by an external observer, who has no access to the internal state and therefore may not notice any difference in the result, or in any case tolerate performance differences.

This notion of equivalence is inspired by the testing equivalence of De Nicola and Hennessy [DH84] and the weak bisimilarity of Hennessy and Milner [HM80]. We now formulate an initial definition of equivalence between code fragments.

3.1.1 Basic Definitions

To formalize the intuitive notion of software redundancy at the code level, we need an abstract and general model of software systems. We model a software system as a state machine, and we denote with $\mathbb{S}$ the set of states, and with $\mathbb{A}$ the set of all possible actions of the system. Under this model, the execution of a code fragment C starting from an initial state S_0 amounts to a sequence of actions $\alpha_1, \alpha_2, \ldots, \alpha_k \in \mathbb{A}$. This sequence of actions may depend on the initial state and induces a sequence of state transitions $S_0 \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} S_k$. In our notion we only consider code fragments with sequential and terminating—and therefore finite but perhaps unbounded—executions, and without loss of generality we consider the input as being part of the initial state.

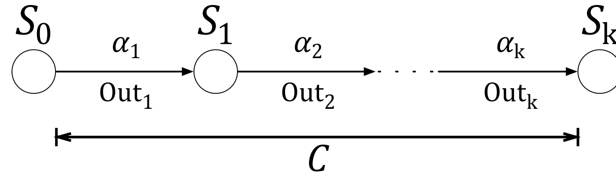


Figure 3.4. Abstract representation of the execution of a generic code fragment C

We use $\mathbb{O}$ to denote the set of all possible outputs, that is, the set of all externally observable effects of an execution. The set of externally observable effects may also include I/O operations such as reading or writing files, sockets, etc. We use $Out(S, \alpha) \in \mathbb{O}$ to denote the output corresponding to the execution of action α starting from state S, and, generalizing, we denote with $Out(S_0, C) \in \mathbb{O}^*$ the output of the sequence of actions $\alpha_1, \alpha_2, \ldots, \alpha_k$ corresponding to the execution of C from state S_0. Note that the cardinality of the output set does not have to necessarily match the cardinality of performed actions, that is, some actions may not produce any output but only a change of state. This abstract model of the execution of a software system is represented in Figure 3.4.

Our model of software systems is abstract, general, and applicable to any code fragment. In Figure 3.5 we represent the execution of some redundant code fragments using as reference the examples in the first row of Figure 3.3. The two redundant code fragments C_A and C_B swap the values of two variable x and y. The fragment C_A uses a temporary variable, whereas C_B uses the bitwise xor operator. The execution of the two different code fragments starts from an identical initial state S_0. Since the fragments are implemented with different algorithms, the two executions perform different actions and thus induce different state changes. Nevertheless, since the fragments are functionally equivalent, the final state S_3 reached by the two computations is the same. In this example, the complexity of the code fragments is modest, and claiming the equivalence of their final state is trivial. In general, we would like to include the possibility that the final states reached by the two computations may differ, so long as this difference is not observable. We now formalize such notion of *observational equivalence* between program states.

3.1.2 Observational Equivalence

We want to express the notion that two code fragments are functionally equivalent when, starting from the same initial state and with the same input, the two code fragments compute the same results and their executions lead to equivalent system states. In particular, the resulting states should be deemed as equivalent even if their internal representation differs, but such difference can not be exposed by any noticeable difference in the functionality of the system. Intuitively, it should not be possible to distinguish the two supposedly equivalent code fragments by executing any code

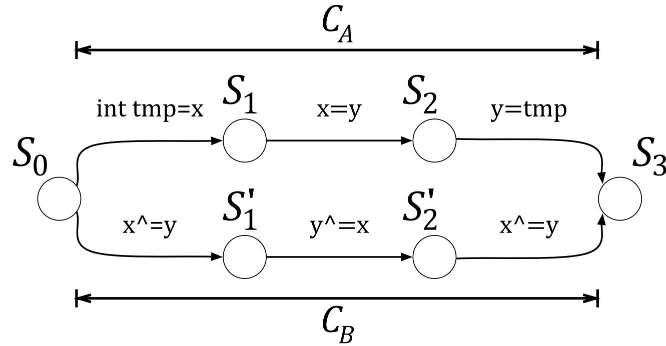


Figure 3.5. Representation of the execution of the redundant code fragments at the top of Figure 3.3

fragment that interacts with them.

More formally, we say that two code fragments C_A and C_B are observationally equivalent from an initial state S_0 if and only if, for every code fragment C_P, which we refer to as *probing code*, the output $Out(S_0, C_A; C_P)$ is the same as $Out(S_0, C_B; C_P)$, where $C_A; C_P$ and $C_B; C_P$ are code fragments obtained by concatenating C_A and C_B with the probing code C_P, respectively. Figure 3.6 abstractly represents this notion of observational equivalence.

This definition requires that the two code fragments and the follow-up probing code produce exactly the same output. This in turn does not take into account the intended semantics of the system whereby different output sequences may be equally valid and therefore should be considered equivalent. For example, consider the case of the HashSet<E> class of the Java Class Library that implements an unordered set of elements. Suppose to instantiate an HashSet of integers that in state S_0 represents the set $\{10\}$. Consider now a fragment C_A that adds element 20 to the set, and a supposedly equivalent fragment C_B that also adds 20 to the set but with a different internal state transformation: C_A leaves the set in a state such that an iteration would first go through 10 and then 20, while C_B causes the same iteration to first go through 20 and then 10. In the end, although semantically and functionally equivalent, C_A and C_B would not be considered observationally equivalent according to the definition above, since a probing code that iterates through the elements of the set would expose a difference.

To account for the semantics of the system, we have to define an equivalence relation between output sequences that is more specific than the identity relation. We thus consider a more general definition of equivalence that requires the output of the two fragments and the follow-up probing code to be equivalent according to an oracle relation that embodies the semantics of the system. Let $T_{S_0,C_A,\cdot}$ be a family of equivalence relations (hereafter *oracles*) that, depending on S_0 and C_A, and for every valid probing code C_P defines an equivalence oracle $T_{S_0,C_A,C_P} \subseteq \mathbb{O}^* \times \mathbb{O}^*$ that essentially says whether

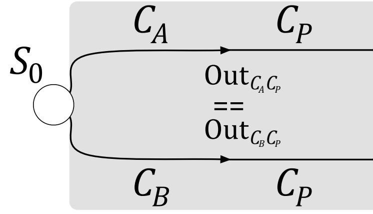


Figure 3.6. Observational equivalence between two code C_A and C_B

two output sequences are both correct (and therefore equivalent) for the execution of C_A from S_0 with probing code C_P. Notice that such oracles are not symmetric with respect to C_A and C_B, since they define an equivalence between output sequences *for the semantic of a specific code fragment*, in this case C_A. Therefore we say that C_B *can replace* C_A when the output of C_B and C_A are equivalent according to T_{S_0,C_A,C_P}, and vice-versa C_A can replace C_B when the outputs are equivalent according to T_{S_0,C_B,C_P}. Finally, we say that C_A and C_B are *observationally equivalent* in a semantically meaningful sense from an initial state S_0 when for every probing code fragment C_P they can replace each other. More specifically, the output sequences $Out(S_0, C_A; C_P)$ and $Out(S_0, C_B; C_P)$ are equivalent according to both oracles T_{S_0,C_A,C_P} and T_{S_0,C_B,C_P}. Notice that the former and more strict definition is a special case in which the oracles reduce to the identity relation.

The readers should notice that in our notion of observational equivalence, the quantity and length (calculated as number of actions for each probing code) of probing code C_P executed are not explicitly defined and might be not bounded. Moreover, our notion of behavioral equivalence is not decidable in the most general computational model, and even considering a bounded notion of observability (that is, with a bounded sequence of actions in the probing code) the verification of the equivalence relations may be computationally very expensive. However, it should be clear that our objective in formulating a notion of equivalence is not to facilitate the automated checking of such a property. Rather, we want to provide the ability to specify an equivalence relation regardless of the relation that may or may not exist at the implementation level. In other words, as for other forms of specifications (for example, assertions) the specification expresses an intent to which the implementation should adhere.

3.1.3 Software Redundancy

Our definition of software redundancy combines two notions: functional equivalence and execution diversity. We previously defined a semantically meaningful notion of observational equivalence. We now define a notion of execution diversity and we then put these two notions together to ultimately define a notion of software redundancy.

We say that the executions of two code fragments C_A and C_B from an initial state S_0 are *different* if the sequence of actions $\alpha_{A,1}, \alpha_{A,2} \ldots$ induced by C_A differs from the

sequence $\alpha_{B,1}, \alpha_{B,2} \ldots$ induced by C_B. For instance, let us consider the two equivalent sequences of actions in Figure 3.5. We consider those as different executions because the actions performed by the two code fragments during the execution are completely different. We explore and experimentally evaluate several alternative approaches for quantifying the difference between sequences of actions in Chapter 4.

In summary, two code fragments C_A and C_B are *redundant* in state S_0 when they produce observationally equivalent results, and execute different actions or the same actions but in different order. Two code fragments C_A and C_B are always redundant if they are redundant in every valid state S (every syntactically valid application of the fragments with every valid input).

This model of redundancy is built upon our practical experience in developing various concrete techniques to capture and exploit software redundancy [CGG+14, CGM+13, CGPP15, GGM+14], and generalizes to software redundancy in other contexts, such as N-version programming [ALRL04] and recovery blocks [Ran75].

We formulated an abstract and general model of software redundancy that abstracts from a specific technique to capture the essence of software redundancy. This is the first and necessary step for understanding the nature of redundancy in software systems, and using it systematically to improve the reliability of software systems.

From this abstract model, we derive a concrete method to characterize the redundancy of a system, and to obtain a measure that would captures the attainable benefits of the redundancy present within a system. In Chapter 4, we present a definition and implementation of a practical and useful measure of software redundancy that is one of the main contribution of this dissertation.

3.2 Behavioral Equivalence and Execution Diversity

Our formalization of redundancy is built upon two distinctive notions: a notion of behavioral equivalence, and a notion of execution diversity. Researchers have already presented definition, approaches, and techniques that study and explore these two constituent notions of redundancy. In Section 3.2.1 we present the most significant definitions of behavioral equivalence in process algebra, and some recent definitions used in software engineering literature. In Section 3.2.2 we present the most influential work on execution diversity.

3.2.1 Behavioral Equivalence

The notion of behavioral equivalence is one of the most fundamental and pervasive notions in computer science. In particular, researchers in the field of process algebra defined several equivalence relations [BR83, DH84, HM80, Hoa80, HBR81, Mil82, Par81]. In this section, we survey the definitions that inspired our notion of observation equivalence among code fragments.

Trace equivalence. The classic definition of behavioral equivalence was proposed by Hoare [Hoa80], who defines a notion of equivalence between processes based on the concept of traces. A trace is defined as a finite sequence of symbols recording the actual or potential behavior of a process from its beginning up to some moment in time. Since any process P is characterized by the set of all possible traces, two processes P and Q are *trace equivalent* if they have the same sets of possible traces, that is, they expose the same behaviors.

The main limitation of trace equivalence is that it does not discriminate between a process that terminates and a process that enters in a deadlock state. *Completed trace equivalence* extends the notion of trace equivalence to cope with this case [JAB01]. A completed trace of a process P is a sequence of actions that leads to some process Q that has no further possible action, that is, whose set of actions is the empty set. An empty set of actions expresses the termination of a process, and a trace that terminates is not equivalent to a trace that leads to a deadlock (where the set of actions is non-empty). Two processes P and Q are completed trace equivalent if they are trace equivalent and have the same completed traces.

Brookes et al. proposes an extension to trace equivalence referring to *failures* instead of traces [BHR84]. A failure is defined as a state in which a process may be unable to perform any other transitions as the result of a non-deterministic decision. Two processes are *failure trace equivalent* if they have identical failing state sets.

Testing equivalence. Close to the notion of failure equivalence, De Nicola and Hennessy formalize the notion of *testing equivalence* [DH84]. Testing equivalence of communicating processes is built upon the notion of observers. intuitively, an observer is an external agent that tests the processes. To decide if a process passes a test, we need a specification of the subset of the possible states that are considered successful states. A computation is deemed as *successful* if it contains at least one successful state and *unsuccessful* if it does not contain any successful state. Based on these notions, two or more processes are equivalent (with respect to the set of observations applied) if and only if they pass or fail the same set of tests.

Strong and weak bisimilarity. Bisimulation is defined as a binary relation between labelled transition systems. A bisimulation is a relation that associates two states, s_P and s_Q belonging to two transition systems P and Q under consideration, if any possible action in s_P can be executed in s_Q and vice-versa. Two systems P and Q are bisimilar if and only if for each state in P and Q there exists a bisimulation that relates them.

The difference between strong and weak bisimulation concerns the set of actions considered in the relation. *Strong bisimulation* takes into consideration all possible actions in a labelled transition system, including internal actions that are actions that hide the internal behavior of a system. Thus, strong bisimulation requires a complete equivalence between systems [Par81]. Weak bisimulation (better known as *observational*

equivalence) takes into consideration only *visible* actions, that is, all but the internal possible actions of the systems. Therefore, observational equivalence abstracts from the internal representation by comparing systems on the basis of their observable behavior [HM80].

Behavioral equivalence in software engineering literature. The definitions of behavioral equivalence have been extended to cope with practical software engineering problems. Doong and Frankl defined a notion of *observational equivalence* for object-oriented programming languages [DF94]. Their notion applies to pairs of objects O_1 and O_2 of a class C, is inspired by the notion of weak bisimulation, and presents some similarities with the notion of observational equivalence that we presented in Section 3.1. Doong and Frankl's equivalence is defined as follows: if C is a primitive type, two objects O_1 and O_2 are considered equivalent if they have identical values. Conversely, if C is a user-defined class, two objects O_1 and O_2 are equivalent if and only if for any sequence of methods S that returns two objects O_1' and O_2', O_1' and O_2' are observationally equivalent.

Jiang and Su proposed a definition of functional equivalence between code fragments based on the input/output behavior of the fragments [JS09]. Two code fragments are equivalent if they compute the same output values when executed with the same input values. The technique compares only code fragments that accept the same set of input data types. Two fragments are then considered *functionally equivalent* if there exists at least one permutation of the input variables for which the outputs are equivalent on all the generated input

3.2.2 Execution Diversity

Researchers have exploited quantitative distance metrics on program executions in different areas: verification of the correctness of an implementation with respect to its requirements [vHR12], intrusion detection [GRS06], test case reduction [HAB13], field failure replication [JO12, RZF+13] fault localization techniques [JHS02, RR03]. In this section, we survey the most relevant metrics that are closely related to our notion of redundancy, and in particular with our notion of execution diversity. We distinguish two classes of distance metrics: static and dynamic metrics.

Static distance metrics. Static distance metrics are defined on the basis of some kind of static analysis on either artifacts related to the code, such as models of the specifications, or directly the code itself.

A relevant representative of static distance metrics is the notion of simulation distance developed by Černý et al. [vHR12], who propose a new notion of correctness that they define as as a real-value distance function between an implementation and the specification. The more an implementation does not satisfy its requirements, the higher is its distance from its specifications. Such quantitative definition of correctness can

also serve as a distance metric between code fragment executions: the more the two fragments are distant, the more the code is likely to be different.

The approach of Černý et al. abstracts from low-level details and is therefore general, but requires to derive precise finite state machine models of both the requirements and the implementation. Imprecise models can lead to inexact distances and may result in both false positives and false negatives. The approach is limited by the high cost of extracting precise models from the code and the requirements, and by definition does not capture dynamic aspects of the execution, such as the different execution scheduling.

Dynamic distance metrics. Dynamic distance metrics are defined on some artifacts extracted from the executions of the system under analysis, such as behavioral models or execution traces.

Gao et al. proposed an intrusion detection technique based on the behavioral difference between two executions [GRS06]. They define a distance metric between sequences of system calls inspired from DNA alignment algorithms, and use such metrics to identify malicious attacks. A significant distance between the original and the current traces could indicate a previously unseen behavior that may be due to a malicious attack.

Hemmati et al. defined an approach to minimize the number of test cases generated through model-based test case generation [HAB13]. They propose to adjust the size of the test suite based on the selection of the most diverse subset of test cases. They evaluated many similarity-based metrics inspired from information retrieval and bioinformatics.

Dynamic distance metrics have been proposed also in the context of spectra-based fault detection techniques [JHS02, RBDL97]. Spectra-based techniques record the execution information of a program in certain aspects (for example, statements or paths) to track program behavior. When the execution fails, these approaches use this information to identify the suspicious code that may be responsible for the failure. Renieris and Reis propose a fault localization technique based on a distance metric between traces [RR03]. They measure the distance between correct and failing program executions as the sequence of statements that are executed in the failing run but not in the successful run, and use the metric to select a successful run as similar as possible to the failing one.

Chapter 4

A Measure of Software Redundancy

In this chapter we present a novel method to measure redundancy that is based on our abstract notion of software redundancy but is practical and can be efficiently computed. The method uses a specific projection of a finite set of execution traces that logs memory changes, and measures a specific form of edit distance between traces. Our experimental analysis shows that our measure distinguishes code that is only minimally different from truly redundant code where the algorithmic nature of the computation differs. We also show that the measurement can help predict the effectiveness of techniques that exploit redundancy.

In the previous chapter, we introduced an abstract and general notion of software redundancy that conjugates a notion of functional equivalence between code fragments with a notion of execution diversity. In this chapter, we introduce a practical approach that approximates the redundancy of a system. In particular, we define a measure that reflects the attainable benefits of the redundancy present within a system.

Intuitively, we would like a more informative measure that ranks the *amount* of redundancy of differently designed code fragments. Such a meaningful and significant measure of redundancy presents two main challenges related to the decidability of the measure and the possibility of quantifying the difference between code fragments. First, our abstract notion of redundancy presented in the previous chapter is not decidable, since it subsumes a basic form of equivalence between programs that amounts to a well-known undecidable problem by Rice's theorem [Ric53]. We therefore need to either limit the expressiveness of the model or somehow accept an incomplete or imprecise decision procedure. Second, our model expresses a binary decision: two code fragments are either redundant or not. We must therefore enrich the model with a form of distance and we must define a corresponding measurement method.

The problem of the decidability of equivalence has been studied extensively from a theoretical perspective independently from its relation to the notion of redundancy [DH84, Hoa80, HM80]. We experienced the problem of quantifying equivalence from a practical

perspective, and specifically in relation to redundancy, when we identified equivalent code fragments that we used to annotate potential sources of redundancy in code fragments of significant size and complexity [CGPP10, CGM+13, CGPP15]. We overcome the problem by approximating the abstract and undecidable notion of equivalence by relying on a bounded notion of observational equivalence inspired by the definition of Doong and Frankl [DF94]: Two states are observationally equivalent when a finite sequence of public method calls invoked on the two states produces indistinguishable results. We developed a method to automatically test the equivalence of code fragments in the specific context of test oracles using a bounded search approach [CGG+14] focusing on the problem of establishing whether two system states were equivalent or not.

We would then like to define a measure that discriminates code fragments whose executions differ substantially. As previously discussed in Chapter 2, the assumption that different code fragments provide strong guarantees to increase the reliability of a system is not realistic, even if the code fragments are designed and implemented independently. In fact, as shown by Knight and Leveson, two independent teams may ultimately rely on the same identical algorithms, and therefore the two versions may be susceptible to correlated failures [KL86]. For this reason, an ideal measure of execution difference should consider both the executed code and the algorithmic nature of the computations. In other words, we would like to somehow measure the difference between the algorithms implemented by two fragments.

We discuss in details this bounded-search method to decide equivalence in Section 4.1.2. In Section 4.1 we introduce a non-binary measure of redundancy, which is one of the main contribution of this dissertation. In Section 4.2 we present the results of an extensive experimental analysis that indicate that our measure of redundancy indeed distinguishes code that is only minimally different, from truly redundant code, and that distinguishes low-level code redundancy from high-level algorithmic redundancy. We also show that the measurement is significant and useful for the designer, as it can help predict the effectiveness of techniques that are built upon software redundancy.

4.1 A Practical Measure of Redundancy

Redundancy is present when two code fragments induce different executions with the same functional effect on a system. In this section, we extend this abstract binary condition to a more general and useful measure of the redundancy of two code fragments, where by "useful" we mean a measure that is indicative of either some interesting property or some design objective related to redundancy. For example, if we use redundant fragments in an N-version programming scheme to increase the reliability of a system, then a useful measure of redundancy correlates with the gain in reliability. In Section 4.1.1 we formulate a general non-binary measure of redundancy. In Section 4.2 we specialize the general measure with different metrics that we experimentally evaluate.

4.1.1 A Non-Binary Measure of Redundancy

We define a useful measure of redundancy by first turning the binary condition into a richer non-binary metric. In particular, we introduce non-binary measurements of redundancy of two code fragments by combining a measure of the *degree of equivalence* of the fragments with a measure of the *degree of diversity* between their executions. We define the two measures with respect to a starting execution state, and generalize the measure to a code fragment by aggregating the results computed over a set of representative initial states.

We denote the degree of equivalence between the execution of two code fragments C_A and C_B from a state S as $e_S(C_A, C_B) \in [0, 1]$. This degree of equivalence is based on the notion of observational equivalence that we introduced in Section 3.1.2: C_A and C_B are observationally equivalent if and only if the output $Out(S_0, C_A; C_P)$ is the same as $Out(S_0, C_B; C_P)$ for every probing code C_P. Intuitively, the degree of equivalence $e_S(C_A, C_B)$ can be seen as the probability that a probing code C_P would not expose any difference between the executions of C_A and C_B from any state S.

We denote the degree of diversity between the executions of C_A and C_B from a state S as $d_S(C_A, C_B) \in [0, 1]$. Such degree of diversity is based on the abstract notion of execution diversity that we presented in Section 3.1.3. In essence, the degree of diversity quantifies the difference between sequences of actions or, more concretely, between executions when starting from a state S.

Both degrees are normalized in the range $[0, 1]$. $e_S(C_A, C_B) = 1$ indicates two functionally equivalent code fragments while $e_S(C_A, C_B) = 0$ indicates completely different functionalities. Similarly, $d_S(C_A, C_B) = 0$ indicates two identical execution traces while $d_S(C_A, C_B) = 1$ indicates completely different executions.

Given these degrees of equivalence and diversity, we define a general measure of redundancy of the execution of two code fragments C_A and C_B from a state S as the product of the degrees of equivalence and diversity: $R_S = e_S(C_A, C_B) \cdot d_S(C_A, C_B)$.

This measure of software redundancy is the first attempt towards the definition of measures that can help software engineers successfully exploit software redundancy. Our formulation of redundancy is driven primarily by our experience in exploiting software redundancy [CGM+13, CGG+14, CGPP15]. Our formula to compute a measure of redundancy may thus be suboptimal in other contexts. Nevertheless, the choice of computing the product of the two constituent elements satisfies an intuitive and yet fundamental property of redundancy: two code fragments are not redundant at all if they either implement different functionalities ($R_S = 0 \cdot d_S(C_A, C_B) = 0$), or execute the same identical code ($R_S = e_S(C_A, C_B) \cdot 0 = 0$). In Section 4.2 we present an extensive set of experimental results that confirm the consistency and usefulness of this measure in the context of software engineering.

The two constituent degrees of the redundancy measure are computed on an execution that start from a state S. To compute an ideal measure of redundancy, we should compute the expected value of R_S for *all* possible states S. An exhaustive set of initial set

input: code fragments C_A, C_B		
	repeat n times:	
1:	$S \leftarrow$ *choose from* $\mathbb{S}$	// sample the state space
2:	$e \leftarrow e_S(C_A, C_B)$	// measure equivalence
3:	$d \leftarrow d_S(C_A, C_B)$	// measure difference
	$R_S \leftarrow e \cdot d$	
4:	**return** *aggregate all* R_S	// R_S in expectation

Figure 4.1. General algorithm to measure redundancy.

leads to a perfect measure of redundancy, but is not feasible. In practice, we aggregate over a finite sample of the state space.

Figure 4.1 presents the algorithm for measuring the redundancy of two code fragments. The algorithm iterates over the state space sample of size n of the two input code fragments C_A and C_B. At each iteration, (i) we sample the state space by choosing a valid state S, that is, a state in which both code fragments can be executed without violating their specifications (line 1), (ii) execute both code fragments C_A and C_B form state S, (iii) measure the degree of equivalence by applying our definition of observational equivalence, that is by generating probing codes (line 2), (iv) measure the degree of diversity by obtaining their execution traces and by measuring their dissimilarity (line 3), (v) compute the degree of redundancy by multiplying the two degrees of equivalence and diversity, (vi) aggregate the degrees of redundancy produced for the different states to compute the redundancy of the two code fragments (line 4).

4.1.2 Sampling the State Space

The first step of the algorithm samples the state space of the input fragments to identify the system states from which to execute the two code fragments C_A and C_B. In a general system and without particular references to a specific programming language, this first step amounts to generating a *concrete state* for the system from which we can execute the two fragments (either one). A concrete state is the set of global variables and function parameters that are required to execute the code fragments (again, either one of them).

As previously discussed, we ideally want an exhaustive set of the execution space that captures all possible behaviors of the code fragments under analysis. We can generate a set of significant concrete states by exploiting various approaches, depending on the available artifacts. For instance, if we had models of the system, such as a finite state machine, we could sample the set of admissible states. Or if we had test cases released by the developers, we could use those tests to produce concrete states. In our experiments, we assume we only have the code itself, and we rely on automatically generated test inputs to produce the concrete states.[1]

[1]In the remaining of this section we use the term “test case” to indicate a “test input”, that is a set of

```
218 | @Test
219 | public void test10_RandoopTest7() throws Throwable {
...
285 |  ArrayListMultimap var82 = ArrayListMultimap.create(10, 0);
286 |  ArrayListMultimap var83 = ArrayListMultimap.create();
287 |  var83.clear();
288 |  ArrayListMultimap var86 = ArrayListMultimap.create();
289 |  ArrayListMultimap var89 = ArrayListMultimap.create();
290 |  var89.clear();
291 |  boolean var91 = var89.isEmpty();
292 |  boolean var92 = var86.putAll((Multimap) var89);
293 |  List var94 = var86.removeAll((Object) "hi!");
294 |  boolean var95 = var83.putAll((Object) (short) (-1), (Iterable) var94);
295 |  boolean var96 = var3.put((Object) var82, (Object) var94);
296 |  int var97 = var82.size();
297 |  ArrayListMultimap var98 = ArrayListMultimap.create((Multimap) var82);
298 |  var82.clear();
299 | }
```

Figure 4.2. Example of random test case generated with Randoop for the ArrayList-Multimap class of Google Guava. The invocation of the method put(key, value) is highlighted at line 295.

We chose to rely mainly on automatic test case generation since it has the main advantage of generating a large corpus of test cases without significant effort. Moreover, it is possible to easily evaluate the thoroughness of the generated set through proxy measures such as code coverage criteria. Since in our experiments we target Java programs, we use Randoop [PLEB07] and EvoSuite [FA13] to generate test cases. To completely automate the process, we let the test generator insert the code fragments directly into the test case. That is, we generate many test cases and select those that already contain C_A, and, for those with multiple instances of C_A, we consider as an initial state the state right before each invocation of C_A. Figure 4.2 shows an example of test case automatically generated with Randoop[2] for method put(key, value) of the class ArrayListMultimap of the Google Guava library. The invocation of the method under analysis is highlighted at line 295.

In some experiments, we generate test cases following the category-partition approach [OB88]. In particular, we manually categorize the input to the system so as to represent classes of equivalent inputs. We then compile a list of tests that cover each category, and then use these tests as operational definitions of the initial state. The advantage of this manual sampling is that it leads to a more representative distribution over the state space than randomly generated suites, although it requires a significant effort even to produce a modest set of test cases.

input values for the code under test, regardless of the availability of proper oracles.

[2]https://github.com/randoop/randoop

4.1.3 Measuring Observational Equivalence

We measure the degree of observational equivalence $e_S(C_A, C_B)$ by directly applying its definition: Two states are observationally equivalent when all finite sequences of public method calls invoked on the two states produces indistinguishable results. We thus generate a large number of probing code fragments C_P that we execute right after C_A and C_B, respectively, and compare the results of each pair of executions to compute the percentage of executions that produce the same observable results.

In essence, probing code fragments amount to random test cases, specifically for the variables corresponding to the linkage of the two fragments. We therefore modified Randoop to implement a specialized, random-test generator that starts from a pool of variable descriptors, each indicating name and type. For example, considering the second example in Figure 3.3, the generator would start from the three variables: AbstractMultimap map, String key and Object value. At each step, the generator selects a variable from the pool, say map, together with a public method to call on that variable, say isEmpty(), and adds that call to the test. If the method returns a primitive value or if the variable is itself of a primitive type, the generator adds a statement to output the value. If the method returns another object, the generator assigns the result of the call to a newly declared variable, and adds a descriptor for the new variable (name and type) to its pool of variables. The generator also adds the necessary code to catch and output any exception that might be raised at each step of the test. The generator terminates and outputs the generated test after a given number of steps. Figure 4.3 shows an example of a generated probing code fragment for a code fragment similar to the second example of Figure 3.3.

Our specialized test generator might generate test cases that might time-out due to blocking operations, for example those methods that require any interaction with the user. Moreover, we compare the output of the generated probing code fragments using a simple equality test that in general leads to a conservative form of observational equivalence, as discussed in Section 3.1.2. Still, this method is both practical and efficient, and it is also exact (no false negatives) for all the subjects considered in our evaluation (Section 4.2).

4.1.4 Measuring Execution Diversity

We measure the degree of execution diversity $d_S(C_A, C_B)$ by measuring the differences between the executions of the two code fragments C_A and C_B starting from an initial state S. An execution is a particularly rich source of information, since it gives access to a wide variety of dynamic data. Recording and measuring the distance between two rich execution traces may become quickly impractical on large executions. We therefore measure the difference on a *projection* of the executions. A projection of an execution trace $\alpha_1, \alpha_2, \ldots, \alpha_k$ logs a subset of the information associated with each action α_i. We define and experiment with many distance measures by combining various dissimilarity

```
1  // ... testing code to set up initial state...
2
3  // Code fragment A
4  boolean result = map.put(var1, var2);
5
6  //LINKAGE: boolean result; ArrayListMultimap map; Object var1; Object var2;
7
8  // generated probing code:
9  System.out.println(result);
10 boolean x0 = map.isEmpty();
11 System.out.println(x0);
12 map.clear();
13 java.util.Map x1 = map.asMap(); // x1 added to the pool
14 int x2 = map.size();
15 System.out.println(x2);
16 int x3 = x1.size();
17 System.out.println(x3);
18 java.util.Set x4 = x1.entrySet(); // x4 added to the pool
19 java.util.Iterator x5 = x4.iterator(); // x5 added to the pool
20 boolean x6 = x4.isEmpty();
21 System.out.println(x6);
22 try {
23   x5.remove();
24 } catch (java.lang.IllegalStateException e) {
25   System.out.println(e);
26 }
```

Figure 4.3. Example of a generated probing code executed immediately after one of the code fragments (A) under measurement. The linkage between the code fragment and the probing code is highlighted at line 6.

measures with various projections. In particular, we experimented with two categories of projections, code and data projections.

In *code projections*, we log some aspects of the code executed in each action α_i. Dynamic analysis, combined with debug information, offers a rich variety of data to log. We experimented with a number of code projections, for example the signature of the method being executed, the source code line identifier, or the line of code jointly with its depth in the call stack or with the full call stack.

In *data projections*, we log the read and write operations performed in each action α_i on either object fields or static fields. The individually logged read and write entries consist of an address and a data item. The address identifies the field being read or written, whereas the data item identifies the read or written value. We specialize this projection by encoding different information in the address and data item. For the address, we log the field name and the class name or type of the field, as well as a number of their combinations. For the value, we log either its string representation for basic types or arrays of basic types, or no value at all.

We experimented with various combinations of code and data projections, and with slightly more elaborate variants of each. For example, for write operations can log

the old value as well as the new value. Table 4.1 summarizes the most significant projections that we used in our experiments. The examples refer to the simple code fragment boolean x = map.put(k,v). We evaluate the effectiveness of these projections in Section 4.2.

Given two execution logs, we want to measure their dissimilarity. For this reason, it is crucial to obtain a clean execution log, that is a log in which we prune all entries corresponding to platform-related actions that are irrelevant and potentially confusing for our purposes. For example, in our implementation, which is in Java, we discard all the actions of the class loader. For different code fragments, the class loader may load a class at different points of the execution, and still the two executions are equivalent.

We use the logs to compute the measure of difference between code fragments. Let $L_{S,A}$ and $L_{S,B}$ be the logs of the execution of fragments C_A and C_B from state S. We compute the diversity measure $d_S(C_A, C_B) = 1 - \mathit{similarity}(L_{S,A}, L_{S,B})$ where *similarity* is a normalized similarity measure. Intuitively, the normalization of the similarity measures takes into account the length of the logs, but in general each measure has its own specific normalization procedure. In the application of the similarity measure, we consider each entry as an atomic value that we simply compare (equals) with other entries.

Given a pair of executions logs, we use different similarity measures to compute their degree of diversity. A similarity measure is usually defined on either sets or sequences of elements. Set-based similarity measures do not consider the order of elements, while sequence-based measures do. For example, suppose that from the execution of two fragments of code we extract the following abstract execution logs $L_{S,A} = [A, B, C, D]$ and $L_{S,B} = [C, D, B, A]$. A set-based similarity measure assumes that the two logs $L_{S,A}$ and $L_{S,B}$ are identical since the elements in the vectors are the same. In contrast, a sequence-based similarity measure would recognize that the two logs are dissimilar, since the order of elements in the two vectors is different. Table 4.2 lists the most effective similarity measures we experimented with, divided into set-based and sequence-based.

4.1.5 Aggregating the Measures of Redundancy

Ideally we would like to obtain the *expected* redundancy of two fragments C_A and C_B from the developer's point of view. However, we also would like to use a general aggregation method, independently from the particular distribution of the considered input samples. We therefore simply compute the average of the redundancy measures over the sample of initial states. We also experimented with other intuitive aggregation functions, such as minimum, maximum, and other quantiles, without noticing any improvement over the simple average.

Type	Projection	Example from an actual execution[a]
Code	statement	ArrayListMultimap.put(LObject;LObject;)Z@66
		AbstractListMultimap.put(LObject;LObject;)Z@95
		AbstractMultimap.put(LObject;LObject;)Z@200
	depth, statement	3:ArrayListMultimap.put(LObject;LObject;)Z@66
		4:AbstractListMultimap.put(LObject;LObject;)Z@95
		5:AbstractMultimap.put(LObject;LObject;)Z@200
Data	type, value	Ljava/util/Map;→{}
		Ljava/util/Set;→[]
		Ljava/util/HashMap;→{}
		I→1
		I←1
	field, value	map→{}
		map→{}
		entrySet→[]
		this$0→{}
		modCount→1
		expectedModCount←1
	class, field, value	AbstractMultimap.map→{}
		HashMap.entrySet→[]
		HashMap$EntrySet.this$0→{}
		HashMap$HashIterator.modCount→1
		HashMap$HashIterator.expectedModCount←1
	class, type, value	AbstractMultimap.Ljava/util/Map;→{}
		HashMap.Ljava/util/Set;→[]
		HashMap$EntrySetLjava/util/HashMap;→{}
		HashMap$HashIterator.I→1
		HashMap$HashIterator.I←1
	field, old value	map→{}
		entrySet→[]
		this$0→{}
		modCount→1
		expectedModCount 0←1
	type, no value	Ljava/util/Map;→
		Ljava/util/Set;→
		Ljava/util/HashMap;→
		I→
		I←

[a]Code fragment: boolean x = map.put(k,v);
Legend: → and ← represent *read* and *write* operations respectively
Abbreviations:
ArrayListMultimap stands for com.google.common.collect.ArrayListMultimap
AbstractMultimap stands for com.google.common.collect.AbstractMultimap
HashMap stands for java.util.HashMap
Object stands for java.lang.Object

Table 4.1. Projections used to derive action logs

	Metric		Abbreviation
Sequence-based	Levenshtein	[Nav01]	(Lev)
	Damerau–Levenshtein	[Nav01]	(DamLev)
	Needleman–Wunsch	[Sel74]	(Need)
	Cosine similarity	[CRF03]	(Cos)
	Jaro	[Jar95]	(Jaro)
	Jaro–Winkler	[Jar95]	(JaroW)
	Q-Grams	[CRF03]	(qGrams)
	Smith–Waterman	[DEKM99]	(SmithW)
	Smith–Waterman–Gotoh	[DEKM99]	(SmithG)
	Overlap coefficient	[CC10]	(Ovlp)
Set-based	Jaccard	[CRF03]	(Jaccard)
	Dice	[CRF03]	(Dice)
	Anti-Dice	[CRF03]	(ADice)
	Euclidean	[CC10]	(Euclid)
	Manhattan	[CC10]	(Man)
	Matching coefficient	[CC10]	(MC)

Table 4.2. Similarity measures applied to execution logs. The abbreviations on the right side identify the measures in the experimental evaluation in Section 4.2.

4.2 Experimental Validation

We experimentally validate the measurement method by studying its consistency and significance. In particular: (Q1) we evaluate the *consistency* of our measures by verifying the stability of the measurements with respect to both common semantic-preserving code transformations (such as refactoring) and to sampling of the state-space from a domain of semantically similar inputs; (Q2) we evaluate the *significance* of the measurements in terms of the support to developers in the process of making design decisions related to the redundancy of their system. Thus, we judge the significance and utility by correlating the measurements with some uses of redundancy.

4.2.1 Experimental Setup

We conduct a series of experiments with a prototype implementation of the measurement method described in Section 4.1. We consider a set of subject systems (described below) for which we consider a number of pairs of code fragments (also detailed below). For each system, we generate a set of test cases either manually using the category partition method [OB88], or automatically with either Randoop [PLEB07] or Evosuite [FA13] depending on the experiment. We execute the test cases generated for each system on all the pairs of redundant code fragments to sample the input space as described

in Section 4.1.2. For each pair of fragments and for each initial state (test case) we measure the observational equivalence as discussed in Section 4.1.3. For each pair of fragments and initial state, we trace the executions of the two fragments using the DiSL instrumentation framework [MVZ+12]. For each of these traces, we then consider the projections described in Section 4.1.4, and for the resulting logs we compute a number of similarity measures using a modified version of the SimMetrics library.[3] Finally, we compute the redundancy measure for each initial state, and aggregate with its overall average and standard deviation.

We experiment with two sets of programs:

- *Benchmark 1* is a set of different implementations of two search and four sorting algorithms taken from various websites. We chose these subjects to create a *ground-truth* for our measure to help us evaluate the consistency and significance of our measure. In fact, all the algorithms within the same category are functionally equivalent, and yet their execution differs. This is exactly our notion of redundancy. Table 4.3 lists the implemented algorithms and the number of implementations. We incorporated each code fragment within a Java class, as reported in Appendix A.2. In the case of Benchmark 1, we consider each whole system as a code fragment and we compare code fragments of the same category. For example, we may compare one implementation of bubble sort with one of quicksort, or two implementations of binary search.

- *Benchmark 2* is a set of classes of the Google Guava library. The set of Guava classes contains methods that can be substituted with other code fragments that are equivalent according to the documentation. We consider all pairs of fragments consisting of a method (C_A) and an equivalent fragment (C_B). Table 4.4 lists all the subject class with the number of methods for which we have equivalent fragments, and the total number of equivalent fragments.

The readers can inspect the artifacts of the experiments in Appendix A, in particular all the generated plots in Section A.1 and all the implementations of the programs of Benchmark 1 in Section A.2.

4.2.2 Internal Consistency (Q1)

We check the internal consistency of the measurement method with three experiments: two experiments to validate two fundamental properties of the redundancy measure, namely non-reflexivity and stability, and a third experiment to verify the consistency of the observational-equivalence method.

[3] `https://github.com/Simmetrics/simmetrics`, the modifications are to reduce the space complexity of several measures. For example, the Levenshtein distance in SimMetrics uses $O(n^2)$ space, which is necessary to output the edit actions that define the distance. However, since we only need to compute the numeric distance, we use a simpler algorithm that uses $O(n)$ space.

Algorithm	**Implementations**
Binary search	4
Linear search	4
Bubble sort	7
Insertion sort	3
Merge sort	4
Quicksort	3

Table 4.3. Benchmark 1: Different implementations of search and sorting algorithms

Class	**Methods**	**Equivalences**
ArrayListMultimap	15	23
ConcurrentHashMultiset	17	29
HashBimap	10	12
HashMultimap	15	23
HashMultiset	17	29
ImmutableBimap	11	17
ImmutableListMultimap	11	16
ImmutableMultiset	9	27
Iterators	1	2
LinkedHashMultimap	15	23
LinkedHashMultiset	18	30
LinkedListMultimap	15	23
Lists	8	20
Maps	12	29
TreeMultimap	13	21
TreeMultiset	17	29

Table 4.4. Benchmark 2: Guava Classes, number of considered methods and equivalent implementations considered

Non-reflexivity The first experiment checks the obvious condition that a fragment is not redundant with itself, since the two executions should be identical. We conduct this experiment using the programs of Benchmark 1. We run each program twice and we compute the redundancy measure between the two execution traces. For each combination of program, execution projection, and similarity measure used we obtain a redundancy measure of 0.0.

Stability With the second experiment we check that the measurements are stable with respect to semantic-preserving program transformations, as well as with semantically irrelevant changes in the input states. In essence, measurements for semantically equivalent code fragments should be 0.0 or very close to it. We experiment with Benchmark 1 and we divide the stability experiment in two sub-experiments.

In a first set of experiments, we apply refactoring operations on the program under analysis. In principle, an ideal method to measure redundancy should be stable with respect to semantic-preserving code transformations. We thus apply all the automatic refactoring operations available within the Eclipse IDE (Extract to Local Variable, Extract Method, Inline Expression and Change Name) to all the eligible expressions in the programs, modifying only one expression for each refactored variant. We then measure the redundancy between the original and the four refactored programs.

Figure 4.4 and Figure 4.5 show the results of an indicative subset of the experiments. In particular, we show the redundancy of one implementation of binary search and one implementation of linear search from Benchmark 1 against their respective refactored variants. The experiment produces consistent data for all the other implementations in the benchmark. As in the first experiment, each figure is divided into data and code projections. Figure 4.4 shows the results for binary search, while Figure 4.5 shows the results for linear search. The Y-axis in the plots corresponds to the redundancy measure obtained for a specific refactoring operation, identified by the color of the bar. The X-axis indicates the similarity measure used to compute the redundancy.

We notice immediately that code projections are inconsistent, and are negatively affected by essentially all refactoring operations under every similarity measure. By contrast, data projections have an excellent consistency and stability, and correctly report zero or near-zero redundancy under all refactorings and with all similarity measures. An analysis of the results reveals that data projections based on type rather than field name or class name are particularly robust for some refactoring activities, such as “Extract Method” and “Change Name”. Data projections are less robust with respect to others that may change the execution of read/write actions. For example, if we apply the “Extract to Local Variable” operator to the variable in a for loop condition that checks the length of an array field, then that changes the number of field accesses and thus the data projections.

In a second set of experiments, we measure the redundancy between different execution traces of the same identical program executed with different test cases, that

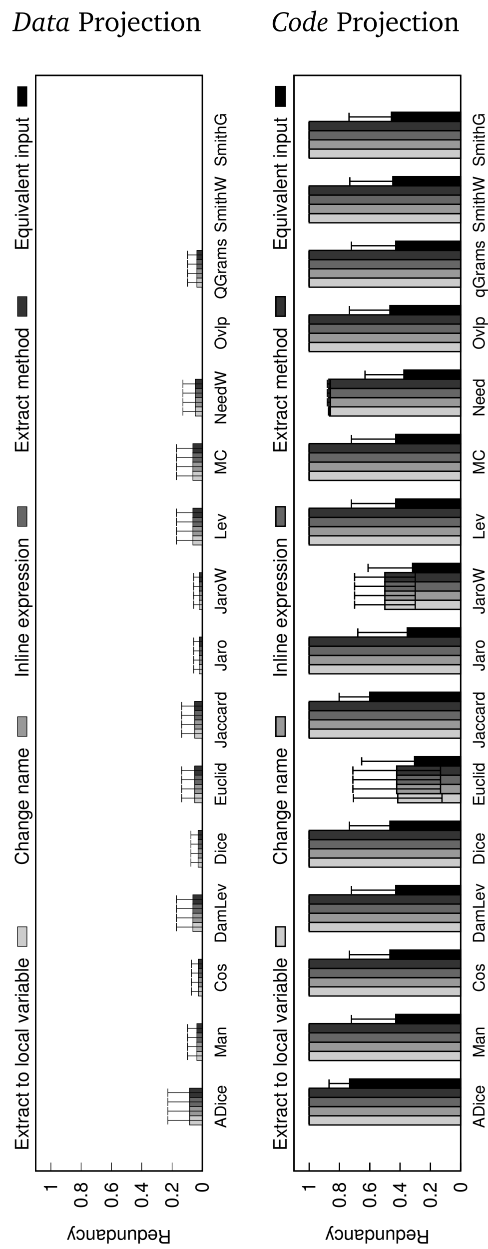


Figure 4.4. Stability of the redundancy measure on an implementation of binary search (Benchmark 1).

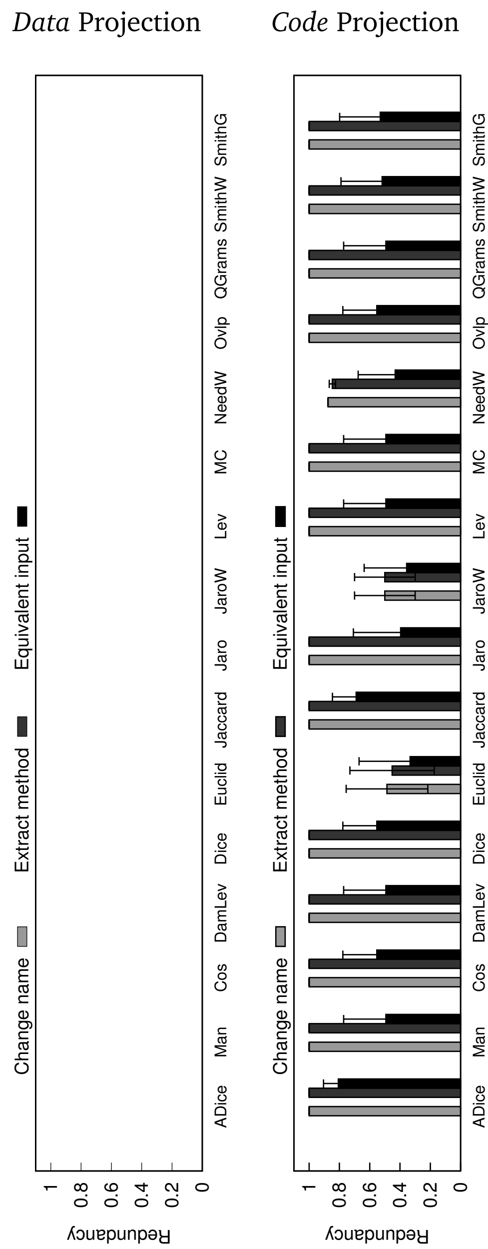


Figure 4.5. Stability of the redundancy measure on an implementation of linear search (Benchmark 1).

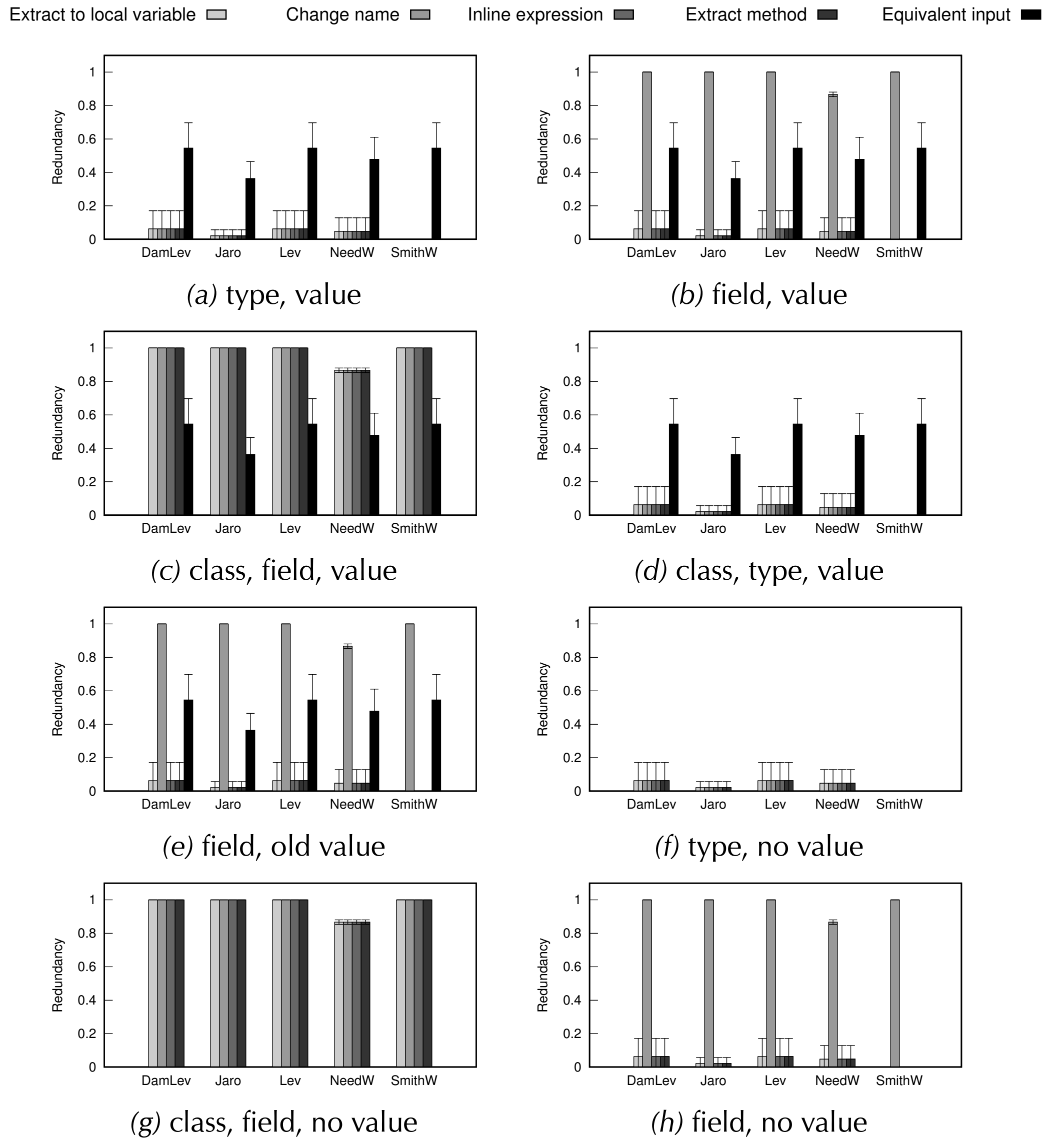
represent different but semantically equivalent initial states. This experiment aims to verify the stability of the measurement with respect to small (insignificant) variations in the initial state. We experimented with test cases generated with the category partition approach and measured the redundancy of pairs of executions with test cases that belong to the same choice of categories. We report the results of these experiments also in Figure 4.4 and Figure 4.5 in the rightmost bar in each group (darkest color, labeled "Equivalent input"). The results are in line with those of the other consistency checks, that is, data projections have excellent consistency and stability, while code projections are inconsistent.

Figure 4.6 summarizes the redundancy measures obtained on the stability experiments by employing all data projections with a selected subset of similarity measures. The results indicate that data projections that keep track of read and written values (plots *(a)–(e)*) are not robust with respect to equivalent inputs, and data projections that logs class or field names (plots *(b)*, *(c)*, *(e)*, *(g)*, *(h)*) are not robust with respect to name changing operations. Our experiments show that the most robust data projection traces only the type and the operation on the fields (plot *(f)*).

Observational equivalence With the third consistency experiment, we focus specifically on the measure of the degree of equivalence, which corresponds to the probability that a probing code would not reveal a difference (see Section 4.1.3). We evaluate the consistency of observational equivalence with our specialized random test generator with an increasingly higher limit on the length of the probing code from 50 to 300 without any noticeable difference on the outcomes of the experiments. For this experiment, we cannot use the programs from Benchmark 1, since we know that those are all equivalent. We therefore focus on the particular case of the *ArrayListMultimap* class taken from Benchmark 2. Table 4.5 lists all the methods that define our first code fragment (C_A) for *ArrayListMultimap*. For each one of them, we report in the table the degree of observational equivalence between the method and all the corresponding programs in Benchmark 2 as the average over several fragments C_B and over all initial states. The degree of equivalence is exactly 1.0 for all methods, which is what we expected, except for the method *keys*.

A closer examination indicates that *keys* is paired with a fragment that uses *keySet*. The two methods are similar but not completely equivalent, since *keys* returns duplicates when the multimap contains multiple values for the same key while *keySet* does not. To better analyze the case, we repeated the experiment with 106 probing codes C_P of variable length up to 300 method invocations starting from 7 different initial states $S_1, \ldots, S_7$. The results show that the measure correctly quantifies the differences in the sequences of actions, and that the results are consistent across initial states and probing codes.

In summary, the results of the experiments demonstrate the internal consistency and robustness of the measurement, and identify the best projections. In the next



(a) type, value

(b) field, value

(c) class, field, value

(d) class, type, value

(e) field, old value

(f) type, no value

(g) class, field, no value

(h) field, no value

Figure 4.6. Comparison among data projections on binary search (Benchmark 1).

Method (C_A)	**Equivalence (average)**
clear()	1.00
containsEntry(Object,Object)	1.00
containsKey(Object)	1.00
containsValue(Object)	1.00
create()	1.00
create(int,int)	1.00
isEmpty()	1.00
keys()	0.61[a]
put(Object,Object)	1.00
putAll(Multimap)	1.00
putAll(Object,Iterable)	1.00
remove(Object,Object)	1.00
removeAll(Object)	1.00
replaceValues(Object,Iterable)	1.00
size()	1.00

[a] For the details of the measure see Table 4.6

Table 4.5. Observational equivalence for the methods of the class ArrayListMultimap from the Google Guava library (Benchmark 2).

Initial state	**generated C_P**	**failed C_P**	**Equivalence**
S_1	13	6	0.54
S_2	16	5	0.69
S_3	16	5	0.69
S_4	15	5	0.67
S_5	14	6	0.57
S_6	16	7	0.56
S_7	16	7	0.56
Average:	15.14	5.86	0.61 (median: 0.57)

Table 4.6. Equivalence measures of methods *keys()* and *keySet()* of class ArrayListMultimap (Benchmark 2).

experiments, we use *data* projections that trace the type and the operation performed on the fields.

4.2.3 Significance (Q2)

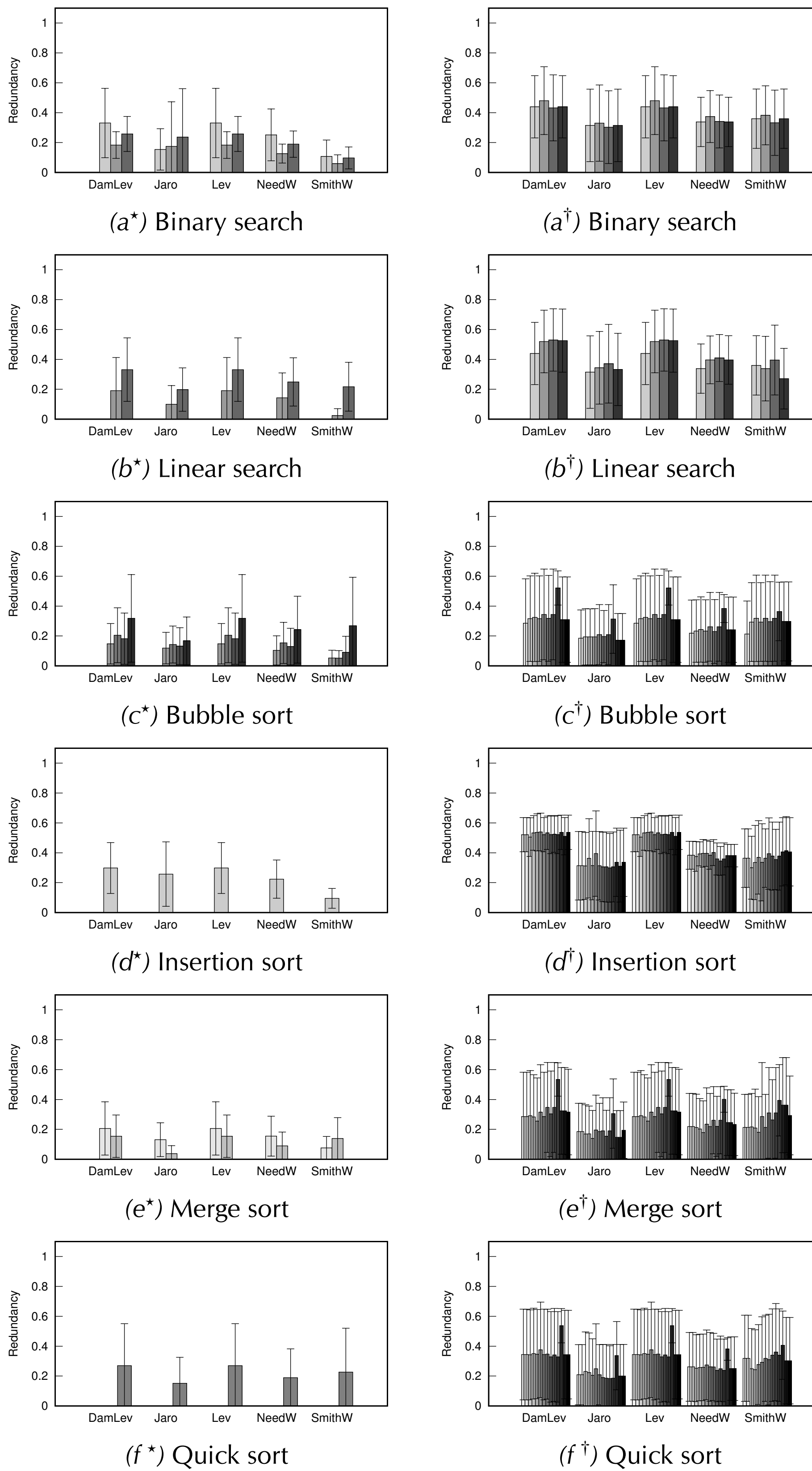
We evaluate the significance of our redundancy measure as the ability to identify differences at various levels of abstractions. Specifically, we want to show that our measure distinguishes code that is only minimally different, from truly redundant code where the algorithmic nature of the computation differs. Furthermore, we evaluate the ability of our measure to predict the effectiveness of a particular technique that exploit redundancy.

Low- and high-level redundancy We assess the ability of the measure to identify redundancy at various levels of abstractions, and more specifically low-level code redundancy versus high-level algorithmic redundancy. With the expression *low-level redundancy* we refer to different implementations of the same algorithm, for example several implementation of the bubble-sort algorithm. Whereas, with the expression *high-level redundancy* we refer to implementations of different algorithms that solve the same task. For example, in the case of sorting, implementations of the bubble sort, insertion sort, and merge sort algorithms.

In this experiment, we assess the ability of our measure to assign higher level of the redundancy measure to different algorithms belonging to the same category, rather than to different implementations of the same algorithm. We conduct this series of experiments on the case studies of Benchmark 1.

Figure 4.7 compares the measurements of the redundancy between programs that implement exactly the same algorithm (left-hand side column, marked with *), and the measurements of the redundancy between programs that implement different algorithms (right-hand side column, marked with †). Each plot shows groups of values representing the average and standard deviation over the measurements between each selected implementation. The Y-axis represents the redundancy measure obtained, while the X-axis represents a subset of the most representative similarity functions used. For example, histogram $a^\star$ shows the redundancy measures obtained by measuring one implementation of binary search against the other three implementations under analysis. On the other hand, histogram $a^\dagger$ shows the redundancy measure obtained by measuring redundancy between one implementation of binary search against all four implementations of the linear search algorithm.

In the case of measurements between different implementation of the same algorithms (histograms $a^\star$–$f^\star$), we observe that in general the redundancy measures are low, which makes sense, since all the fragment pairs implement the same algorithm and can only have low-level code differences. Moreover, notice that in all the case studies, some of the results are zero and therefore do not appear in the histogram. For example, the



(a⋆) Binary search

(a†) Binary search

(b⋆) Linear search

(b†) Linear search

(c⋆) Bubble sort

(c†) Bubble sort

(d⋆) Insertion sort

(d†) Insertion sort

(e⋆) Merge sort

(e†) Merge sort

(f ⋆) Quick sort

(f †) Quick sort

Figure 4.7. Redundancy between implementations of the same algorithm (⋆) and between different algorithms (†).

third histogram ($c^\star$) shows the case of the seven implementations of bubble sort (see Table 4.3), two of which have zero redundancy with respect to the selected one, and therefore the histogram shows only four bars for each similarity measure.

Histograms $a^\dagger$–$f^\dagger$ show the redundancy between fragments that implement different algorithms. Here each histogram represents one particular algorithm, and shows the comparisons between each implementation of that algorithm and every other implementation of another algorithm in the same category (sorting and searching). Here the measures are relatively high, indicating a high degree of redundancy, which makes sense, since all the fragment pairs implement different algorithms, and therefore should have significant differences.

To better analyze the significance of the differences between redundancy measurements on the same and different algorithms, we report in Table 4.7 the results of the statistical tests for the measurements in Figure 4.7. The interested reader can find the complete set of statistical test results in Appendix A.3.

To determine if there exists a statistically significant difference between redundancy measurements among the same algorithms or among different algorithms, we employ the Wilcox-Mann-Whitney U-test as suggested also in previous work [AB11]. We use the U-test because it is a non-parametric test and we cannot make assumptions on the underlining distribution of the data. Our null hypothesis H_0 states that there is no difference between the redundancy measured among different implementations of the same algorithm, and redundancy measured among implementations of different algorithms that solve the same task (searching or sorting). We apply the widely-used significance level of $\alpha = 0.05$ for rejecting the null hypothesis H_0.

In addition to the significance U-test, we also apply Vargha and Delaney's $\hat{A}_{12} \in [0, 1]$ non-parametric effect size measure to assess the magnitude of the improvement [VD00]. In our context, $\hat{A}_{12}$ measures the probability that the measurements of algorithm A yields higher redundancy values than those of algorithm B. If the two measures are equivalent, then $\hat{A}_{12} = 0.5$. If the redundancy measure is greater between two different algorithms, then $\hat{A}_{12} > 0.5$.

In Table 4.7 we report the $\hat{A}_{12}$ measure and its p-value for each benchmark (column *Algorithm*) and for each similarity measure applied (column *Similarity*). In any benchmark and for every similarity measure, the difference between the two measurements is statistically significant, with a p-value smaller than 0.001. We thus reject the null hypothesis H_0 and state that there is statistically significant difference between the two measurements. The effect size measure $\hat{A}_{12}$ shows that first, the measures obtained among different algorithms are statistically higher than the ones obtained among different implementation of the same algorithm, since $\hat{A}_{12} > 0.5$. Second, the magnitude of their difference can be generally considered as "medium", since $0.64 \leq \hat{A}_{12} < 0.71$ [VD00].

The effect size is medium for two plausible reasons: First, our sampling of the state space might be sub-optimal. We compare different—although semantically equivalent—

Algorithm	Similarity	$\hat{A}_{12}$	p-value
Binary search	Jaro	0.67	≤ 0.001
	Lev	0.67	≤ 0.001
	NeedW	0.66	≤ 0.001
	SmithW	0.66	≤ 0.001
Linear search	Jaro	0.67	≤ 0.001
	Lev	0.67	≤ 0.001
	NeedW	0.68	≤ 0.001
	SmithW	0.66	≤ 0.001
Bubble sort	Jaro	0.63	≤ 0.001
	Lev	0.65	≤ 0.001
	NeedW	0.65	≤ 0.001
	SmithW	0.66	≤ 0.001
Insertion sort	Jaro	0.59	≤ 0.001
	Lev	0.61	≤ 0.001
	NeedW	0.61	≤ 0.001
	SmithW	0.66	≤ 0.001
Merge sort	Jaro	0.67	≤ 0.001
	Lev	0.67	≤ 0.001
	NeedW	0.67	≤ 0.001
	SmithW	0.66	≤ 0.001
Quicksort	Jaro	0.66	≤ 0.001
	Lev	0.65	≤ 0.001
	NeedW	0.66	≤ 0.001
	SmithW	0.64	≤ 0.001

Table 4.7. Statistical significance of the results in Figure 4.7. We omit from the results the *DamLev* similarity metric since it is identical to *Lev*.

algorithms that handle some corner cases in similar ways. For example, both search and sort algorithms perform an initial check on the size of the array, and terminates if true. The traces extracted from empty lists will all appear as identical, despite the fact that are obtained from different algorithms. Second and more important, some implementations of the same algorithm might be largely different. For example, more than half of the bubble sort implementations contain various optimizations to lower the number of comparisons, and that are not present in the standard algorithm. In particular, one version implements a sliding search window to minimize the swaps that result in the highest redundancy measure among bubble sort implementations (Figure 4.7 ($c^{\star}$), the darkest bar). Such optimizations naturally introduce noise in our measurements, nevertheless our measure of redundancy can successfully distinguish low- and high-level redundancy also in a statistically meaningful way and with a good effect size.

Predictive ability The last series of experiments assess the significance of the measurement in terms of its predictive ability. We evaluate the ability of our measure as indicator of the effectiveness of a specific technique that exploit redundancy by analyzing the redundancy of some equivalent fragments that we used as automatic workarounds with a technique intended to increase the reliability of systems [CGM+13]. The Automatic Workarounds technique recovers from failures by substituting a failing method with an equivalent sequence at runtime. An equivalent sequence expresses alternative methods as rewriting rules that capture the redundancy of a pair of method sequences. In our experiment, we consider all possible equivalent sequences that are used with varying degrees of success as workarounds. We then measure the redundancy for each pair, and observe how that correlates with the success ratio.

Table 4.8 reports for the two systems analyzed *Carrot* and *Caliper* (column *System*), the list of all the methods we analyzed (column *Method* (C_A)) paired with its equivalent sequence (column *Equivalent Sequence* (C_B)). For each pair of code fragments $< C_A, C_B >$ the table reports the success of the equivalent sequence as a potential workaround (column *Success ratio*) measured as the frequency the fragment C_B was indeed a successful workaround over the total number of substitutions in the experiments in response to a failure in the fragment C_A. The last column *Redundancy* reports the redundancy measurements between the code fragments C_A and C_B as average and standard deviation using the data projection that trace the type and the operation performed on the fields, and the Levenshtein distance as similarity measure. For each subject system we sort the equivalent pairs by their success ratio to highlight the correlation with the measure of redundancy.

The most obvious cases in our experimental results are when the two code fragments (C_A and C_B) are either not redundant at all or completely redundant. When there is no redundancy, as in the case of LinkedHashMultimap.create() in Caliper or Lists.newArrayList() in Carrot, the equivalent sequence is also completely ineffective to obtain workarounds. Conversely, when we obtain a measure of complete redundancy in the case of Iterators.forArray(a) in Caliper, the equivalence is always effective as a workaround.

The redundancy measure is also a good indicator of the success of a workaround in the other, non extreme cases. Consider for example the case of ImmutableMultiset.of(Object..c) in Carrot where the first equivalent alternative has a higher redundancy measure and a higher success ratio than the second one: 0.56 ± 0.07 and 0.59 for the first sequence, and 0.24 ± 0.12 and 0.31 for the second one. This case shows that the redundancy measure can be an effective predictor to select or rank alternative fragments for use as workarounds.

Overall we obtain a positive correlation of 0.9523 using Pearson's r linear correlation coefficient (p-value less than 0.001) from which we conclude that our redundancy measure is indeed a good indicator and predictor for the designers, as it can help predict the effectiveness of techniques that are built upon software redundancy.

System	Method (C_A)	Equivalent Sequence (C_B)	Success ratio	Redundancy
Caliper	Iterators.forArray(a)	Arrays.asList(a).iterator()	3/3 (100%)	1.00 ± 0.00
	LinkedHashMultiset.retainAll(Collection c)	foreach(o in map) if(o not in c) map.remove(o);	1/2 (50%)	0.61 ±0.01
	ArrayListMultimap.putAll(Object k,Collection c)	foreach(o in c) put(k,o);	8/41 (20%)	0.37 ±0.32
	LinkedHashMultimap.putAll(Object k, Collection c)	foreach(o in c) put(k,o);	0/1 (0%)	0.00 ±0.00
	LinkedHashMultimap.create()	create(100,100)	0/207 (0%)	0.12 ±0.15
	LinkedHashMultimap.create(int,int)	create()	0/202 (0%)	0.12 ±0.15
	LinkedHashMultimap.isEmpty()	size() = = 0 ? true : false	0/34 (0%)	0.00 ±0.00
Carrot	ImmutableMultiset.of(Object..c)	foreach(o in c) builder().setCount(o,count(o in c))	13/22 (59%)	0.56 ±0.07
	ImmutableMultiset.of(Object..c)	builder().add(..c).build()	7/22 (31%)	0.24 ±0.12
	ArrayListMultimap.putAll(Object k,Collection c)	foreach(o in c) put(k,o);	1/13 (8%)	0.37 ±0.32
	ImmutableMultiset.of(Object o)	builder().add(o).build()	0/1 (0%)	0.32 ±0.14
	Lists.newArrayList()	new ArrayList()	0/24 (0%)	0.00 ±0.00
	Lists.newArrayList()	new ArrayList(10)	0/24 (0%)	0.00 ±0.00
	Lists.newArrayListWithCapacity(int c)	new ArrayList()	0/20 (0%)	0.00 ±0.00
	Lists.newArrayListWithCapacity(int c)	new ArrayList(c)	0/20 (0%)	0.00 ±0.00
	Maps.newHashMap()	Maps.newHashMapWithExpectedSize(16)	0/54 (0%)	0.00 ±0.00
	Maps.newHashMap()	new HashMap()	0/54 (0%)	0.00 ±0.00
	Maps.newHashMap()	new HashMap(16)	0/54 (0%)	0.00 ±0.00

Table 4.8. Correlation between redundancy measure and the effectiveness of Automatic Workarounds.

4.3 Limitations and Threats to Validity

Our measure of redundancy is limited primarily by the fact that the model considers only single-threaded code fragments. Notice in fact that the model, as well as the measure of dissimilarity, is based on the notion of an execution consisting of *one* sequence of actions. One way to model multi-threaded code would be to linearize parallel executions, although that might be an unrealistic oversimplification.

Another limitation of our measure of redundancy stems from the use of dynamic analysis. Our approach exploits dynamic analysis to measure both the degree of equivalence and the degree of execution diversity. Therefore, any incompleteness or inability to successfully execute a code fragment negatively impacts our measurements.

The issue arises from the choice of using automatic test generation as main approach to both sample the state space and generate code probes to assess observational equivalence. To limit such issues, we use code coverage criterion to evaluate the exhaustiveness of the generated code. Such proxy measure gives an overview of the effectiveness of the generated test suite in exploring the code under analysis. To increase the reliability of our measurements, in some experiments we augmented the automatically generated tests with some hand-written test cases.

On the other hand, the code under analysis might include blocking functions, for example to obtain inputs from the user, or interactions with the execution environment, for example for I/O operations, and it might prevent the correct execution of the test. We envision the use of our measure to assess the redundancy level directly by developers to evaluate the design choice made. It is thus reasonable to assume the project to provide the necessary mechanism to stub and mock the troublesome calls and effectively run the code.

We acknowledge potential problems that might limit the validity of our experimental results. The internal validity depends on the correctness of our prototype implementations, and may be threatened by the evaluation setting and the execution of the experiments. The prototype tools we used are relatively simple implementations of well defined metrics computed over execution logs and action sequences. We collected and filtered the actions of interests with robust monitoring tools and we carefully tested our implementation with respect to the formal definitions.

Threats to external validity may derive from the selection of case studies. We present results obtained on what we would refer to as “ground truth,” that is, on cases with clear and obvious expectations that would therefore allow us to check the significance and robustness of the proposed metrics.

Chapter 5

Automatic Identification of Equivalences

In this chapter we present a technique to automatically identify methods or sequences of methods that are functionally equivalent to a target input method. In particular, the technique synthesizes sequences of method invocations that are equivalent to a target method within a finite set of execution scenarios. Our experimental analysis shows that the proposed approach correctly identifies equivalent method sequences in the majority of the cases where redundancy was known to exist, with acceptable precision and performance.

In the previous chapters, we presented how deliberate and intrinsic redundancy find many useful applications. In the context of intrinsic redundancy, the issue of effectively exploiting redundancy is exacerbated by the unavailability at design time of the set of the alternative implementations. Due to the nature of this form of redundancy, the various alternatives stem spontaneously in the system and there is thus no documentation. While redundancy can be effectively measured with the approach discussed in the previous chapter, there are no techniques that can effectively identify equivalent method sequences. As a result, the developers have to manually identify where are the alternative implementations.

Similarly, equivalent method sequences find many useful applications, from the automatic generation of test inputs [CKTZ03], to the design of automatic repair techniques [CGM$^+$13, CGPP10], and the automatic generation of test oracles [CGG$^+$14]. As for the previously discussed applications, the equivalence is exploited automatically, but must be identified manually.

The manual identification of equivalent method sequences is a non-trivial and error-prone activity that may represent a showstopper to the practical applicability of all discussed techniques. In this chapter, we propose a technique that can automate the identification of equivalent method sequences.

Deciding whether two methods are equivalent for all the possibly infinite execution

scenarios is an undecidable problem. However the decidability of equivalence becomes computationally feasible by limiting the number of scenarios to a finite set. We synthesize equivalent methods or combinations of methods by examining the program behavior on a finite set of execution scenarios. We refer to the synthesized methods or combinations of methods as *likely-equivalent*, to indicate that they may behave differently for inputs not considered in the synthesis process.

Given a target method and an initial set of execution scenarios, our technique automatically synthesizes method sequences that are *likely-equivalent* to the target method. The synthesized method sequences are equivalent with respect to the set of execution scenarios, and are expected but not guaranteed to be equivalent in the general case. The synthesis proceeds in two phases: In the first phase, the search goal is to synthesize a candidate method sequence to be likely-equivalent to the target method; In the second phase, the search goal is to synthesize a counterexample showing that the candidate method sequence is not equivalent to the target method on some previously unexplored scenarios. The two phases iterate, with the counterexamples added to the execution scenarios, until the second phase fails to find a new counterexample. At this point, the synthesized method sequence is deemed as likely-equivalent to the target method.

Our technique is fully automatic and requires only as few as one test input (the initial execution scenario) that may be either provided by the developers or generated automatically. Our experiments indicate that the technique is effective in synthesizing equivalent method sequences, and at the same time is reasonably efficient. On 266 methods belonging to 23 different classes for which equivalent method sequences were known to exist, our approach synthesizes 74% of the known equivalences, finding one or more equivalent sequences for each target method, with limited false positives and with an execution time that significantly outperforms the manual approach.

We introduce and discuss the basic and essential characteristics of search-based techniques in Section 5.1. In Section 5.2 we introduce an approach to automatically identify equivalent method sequences, which is one of the main contribution of this dissertation. In Section 5.3 we present SBES, our tool that implements our automatic approach for Java, and discuss the technical challenges in detail. In Section 5.4 we present the results of an extensive experimental analysis that indicate that our tool SBES indeed correctly identify a large amount of equivalences, with reasonable precision and performance.

5.1 Background

Search-based software engineering is an emerging field that consists in applying meta-heuristic optimization algorithms to software engineering problems [Har07]. In recent years, search-based engineering has produced interesting results, especially in the area of automatic test case generation [McM04]. Search-based approaches can be partition

in three main categories based on their search strategies: local, global, or memetic.

Local Search Algorithms A local search algorithm considers only the neighborhood of a candidate to generate the solution [Arc09]. An example of local search algorithm is the hill-climbing algorithm [RN03]. Hill-climbing usually starts with a random candidate solution, and evaluates all the candidate neighbors with respect to their fitness for the search objective. It then searches either the first neighbor that has improved the fitness, or the best neighbor. The algorithm proceeds by considering recursively the neighborhood of the new candidate solution. A local search algorithm can easily get stuck in local optima, that is solutions that optimize the fitness function solely in the considered neighborhood. Local optima are typically overcome by either restarting the search with new random values, or with some other form of escape mechanism (for example, by accepting a worse solution temporarily as performed by Simulated Annealing [RN03]).

Global Search Algorithms *Global search* algorithms try to overcome local optima to ultimately find more globally optimal solutions. Among the several global search algorithms proposed in literature, Genetic Algorithms (GAs) play a dominant role mostly because of their effectiveness [HM10, McM04].

GAs are inspired by the natural laws of evolution discovered by Charles Darwin, and in particular by the "survival of the fittest" principle. Informally, GAs look for approximate solutions to optimization problems whose exact solutions cannot be obtained at acceptable computational cost. In a nutshell, a GA aims to either minimize or maximize the value of a *fitness function* that quantifies the distance of the candidate solutions from the optimal solution. Each candidate solution of the problem is encoded in what we refer to as a *chromosome*. A *population* is a set of chromosomes that iteratively evolves through *generations* by means of genetic operators. Genetic operators select and evolve candidate solutions to produce new "fitter" chromosomes. The genetic operators commonly employed in GAs have two objectives. First, they select chromosomes with the best fitness values, that is those candidate solutions to be preserved in the next generation. Second, they create new chromosomes by introducing variations in a candidate solution, for example through chromosome mutation and crossover. GAs terminate when they either find the desired solution or exhaust the time budget for the search, and return the best solution found during the evolution process.

GAs have been successfully used to generate test cases for both procedural [PHP99] and object-oriented software systems [FA13, Ton04]. To apply GAs on the problem of generating test cases, the problem itself is reformulated as an optimization problem. For example, in the case of test case generation, the problem of generating test cases for a fragment of code is transformed in searching for inputs that maximize the coverage metrics associated with the chosen test adequacy criterion, for instance branch coverage.

To apply GAs to an optimization problem, it is necessary to define:

1. a representation of a candidate solution as chromosome,
2. a fitness function, defined on the basis of the chosen candidate representation, and
3. a set of manipulation operators.

For example, let us consider again the problem of test case generation for object-oriented systems. When generating test cases, a *chromosome* is a combination of invocations of constructors and methods, terminated with the invocation of the method under test. Primitive values are generated randomly, while the objects needed for the final call are generated by invoking their constructors. Intermediate method calls are introduced in a chromosome to change the internal state of an object. Figure 5.1 shows an example of such kind of chromosomes for the Java programming language.

The manipulation operators used in GAs are categorized in *mutation* and *crossover* operators. Mutation operators are applied to a single chromosome at a time. Some mutation operators are general, for example the mutation of a primitive value, while others are designed specifically to manipulate method sequences by inserting, removing, or replacing method calls. The *crossover* operator combines pairs of chromosomes, for instance by swapping their suffixes. Figures 5.2 and 5.3 show some examples of mutations and crossover operators respectively.

The fitness function commonly employed in the literature to maximize branch coverage is the sum of the *approach level* and the *branch distance*. The approach level rewards those executions that get close to the target branch, referring to the control flow graph, while the branch distance quantifies heuristically the distance of a condition from the opposite boolean value [McM04]. For example, if the predicate is TRUE, then the branch distance informs on how far the input used in the test case is from an input that would make that predicate FALSE. By evolving iteratively over generations, GAs produce test cases with increasing fitness values, until either all branches are covered, or a time limit is reached.

Memetic Algorithms A Memetic Algorithm (MA) hybridizes global and local search such that the individuals of a population in a global search algorithm have the opportunity to improve their fitness value through local search. For example, after selection and crossover have been applied in GA, each individual has the opportunity to improve its fitness by applying a local search algorithm to reach an optimum local value. Harman and McMinn analyzed the effects of global and local search, and concluded that MAs achieve better performance since they can exploit the local information available in the surroundings of candidate solutions [HM10].

The use of MAs for test case generation was originally proposed by Wang and Jeng in the context of procedural code [WJ06], and later extended by Arcuri and Yao for object-oriented code [AY07]. In particular, Arcuri and Yao compared GA, hill climbing,

```
1 Stack s = new Stack();
2 s.push(0);
3 int result = s.pop();
```

Chromosome A

```
1 Stack s = new Stack();
2 s.push(-81527);
3 s.push(2);
4 int result = s.pop();
```

Chromosome B

Figure 5.1. Examples of chromosomes for the Stack class of the Java standard library.

```
1 Stack s = new Stack();
2 s.push(1234566);
3 int result = s.pop();
```

Chromosome A′

```
1 Stack s = new Stack();
2 s.push(-81527);
3 s.push(2);
4 Collection c = new Collection();
5 c.add(12);
6 c.add(-4);
7 s.addAll(c);
8 int result = s.pop();
```

Chromosome B′

Figure 5.2. Examples of mutations applied on the chromosomes of Figure 5.1. The mutations are located at line 2 on Chromosome A, and at lines 4–7 on Chromosome B.

```
1 Stack s = new Stack();
2 s.push(1234566);
3 s.push(2);
4 Collection c = new Collection();
5 c.add(12);
6 c.add(-4);
7 s.addAll(c);
8 int result = s.pop();
```

Chromosome A″

```
1 Stack s = new Stack();
2 s.push(-81527);
3 int result = s.pop();
```

Chromosome B″

Figure 5.3. Examples of crossover applied on the mutated chromosomes of Figure 5.2. The single crossover point is applied between lines 2 and 3.

and a MA defined as the combination of GA and hill climbing, for generating unit tests that optimize branch coverage for container classes. Their evaluation showed that MA is more effective in generating high-coverage test cases than global and local searches. Liaskos and Roper also confirmed that the combination of global and local search algorithms leads to improved coverage when generating test cases for object-oriented classes [LR08]. Baresi et al. proposed a hybrid evolutionary search in their TestFul test case generation tool. The global search aims to maximize coverage on a class as a whole, while the local search targets the optimization of individual yet-to-be-covered branch conditions [BLM10, MLB09].

The local search in the MA techniques described so far are mainly focused on the

local optimization of numerical data types and covering specific testing targets (for example, a single branch) within a single test case. To overcome such limitations, Fraser et al. propose MAs for whole test suite generation by combining GAs with hill climbing [FAM15]. Their approach exploits local search both at the method call level and at the test suite level.

At the *method call* level, the local search optimizes primitives, strings, arrays, and reference values. For primitive types, the original value is modified through the application of some predefined patterns. For example, on integer values the first local move in the neighborhood is the addition of +1 or −1 to the current value. Then, the successful modification is recursively applied until a (sub-)optimal value is reached. For strings, the local search is performed only if the string variable has some impact of the overall fitness value. To determine such condition, the variable is modified with some random mutations. If the fitness values changes due to such mutations, the local search on the string variable is performed. The local search tries to optimize the string by inserting, changing, and deleting characters. For arrays, the first step of local search is to optimize the length of the array itself. To compute the optimal length, the local search removes assignments to array slots. That is, each element of the array of length n is removed, starting from position $n-1$ and moving to position 0, until the fitness values remains unchanged. Once the search has found the best length, on each assignment to the array is performed a local search, depending on the type of the array. For reference types, the neighborhood of a complex type in a sequence of calls is huge, such that exhaustive search is not feasible. Therefore, the local search consists of applying a predefined number of random mutations to the statement. The mutations applied are the replacement of a method call with another random call returning the same type, the replacement of a parameter, and a new random call on the returning object.

At the *test suite* level, the local search tries to optimize the tests with respect to the overall goal of achieving high code coverage. First, the local search minimizes the reuse of primitive variables by generating multiple variables with identical values. Such transformation is beneficial for a later improvement of the individual through a step of local search on those variables. Second, the local search is used to ensure that each target branch is covered twice. The fitness function defined in EvoSuite requires each branch to be covered by two test cases to preserve one of the tests that cover the branch. Therefore, if a branch is covered only once in a test suite, the local search duplicates the individual that covers the branch, ultimately improving the fitness value. Last, the local search improves a test suite by adding test cases that execute yet-to-be-covered branches. In fact, during the evolution of the population EvoSuite collects all the test cases that cover a particular branch. If a test suite is not able to cover a peculiar branch, EvoSuite applies a local improvement by adding the according test case, improving the overall fitness.

Fraser et al. show through their evaluation that MAs can improve the effectiveness of the generation process, up to 53% more covered branches. The effectiveness of

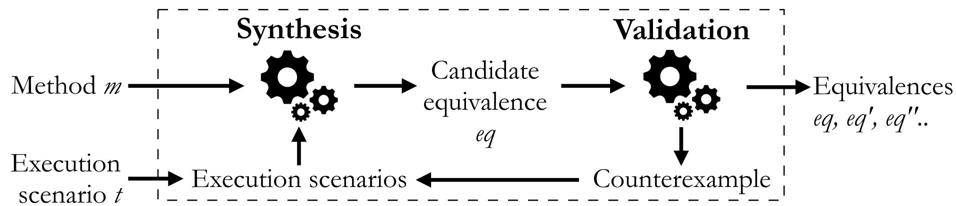


Figure 5.4. High-level representation of our approach to synthesize equivalences.

memetic algorithms largely depends on the frequency at which EvoSuite performs the local search. If the local search occurs too often, it steals search budget from the global search. On the other hand, if the local search occurs infrequently, it does not bring much benefits. Fraser et al. showed through an empirical investigation that the optimal configuration for EvoSuite with MAs requires the execution of local search every 100 generations, with a search budget of 25 seconds [FAM15].

5.2 Automatic Synthesis of Equivalent Method Sequences

We exploit search-based algorithms to synthesize a sequence of method invocations that is likely-equivalent to a target method m by means of a two-phase iterative process, as shown in Figure 5.4. We start with an initial non-empty set of execution scenarios that represent a sample of the input space of m. The initial execution scenarios may be as simple as a single test case. In the first phase, we invoke the search algorithm to generate a likely-equivalent candidate *eq* for the given set of scenarios. Limiting the number of scenarios to a finite set introduces potential spurious results. In the second phase, we validate *eq* by using the search algorithm to find a counterexample, which corresponds to an execution scenario for which *eq* and m behave differently. If we find a counterexample, we add it to the set of execution scenarios, and we iterate through the first phase looking for a new candidate *eq*. Otherwise, we have successfully synthesized a method sequence *eq* that is likely-equivalent to m. Method m may be equivalent to many different method combinations. Therefore, once an equivalent method sequence has been synthesized, we incrementally remove the methods used in the synthesized sequence from the search space, and we iterate looking for further equivalences.

The algorithm for identifying a likely-equivalent method sequence for a target method m is detailed in Figure 5.5. The algorithm needs a non-empty set of execution scenarios for m. If the method comes with one or more test cases, the algorithm uses them, otherwise it generates an initial set of execution scenarios (line 1), and then iterates over the two phases (lines 2-13).

The first phase is detailed with function FIND-EQUIVALENT (lines 14-29). The search-based algorithm generates a sequence of method invocations that is likely-equivalent to the input method m for the current set of execution scenarios. The algorithm iteratively

```
input: target method m
 1: execScenarios := LOAD-INITIAL-TS
 2: while time < overall-time-limit do
 3:   candidate := FIND-EQUIVALENT(m,execScenarios)
 4:   if candidate is NIL then
 5:     return NIL
 6:   end if
 7:   counterex := FIND-COUNTEREXAMPLE(m,candidate)
 8:   if counterex is NIL then
 9:     PRINT(candidate)
10:     REMOVE-CALLS(candidate)
11:   end if
12:   add counterex to execScenarios
13: end while

14: function FIND-EQUIVALENT(m,execScenarios
15:   while time < time-limit do
16:     candidate := SYNTHESIZE-EQUIVALENT-CANDIDATE
17:     candidateFound := true
18:     for each e in execScenarios do
19:       if ¬EQUIVALENT(m,candidate,e) then
20:         candidateFound := false
21:         break
22:       end if
23:     end for
24:     if candidateFound then
25:       return candidate
26:     end if
27:   end while
28:   return NIL
29: end function

30: function FIND-COUNTEREXAMPLE(m,candidate)
31:   while time < time-limit do
32:     counterex := SYNTHESIZE-COUNTEREXAMPLE
33:     if ¬EQUIVALENT(m,candidate,counterex) then
34:       return counterex
35:     end if
36:   end while
37:   return NIL
38: end function
```

Figure 5.5. General algorithm to synthesize an equivalence.

generates a candidate sequence of method invocations (line 16), and evaluates the equivalence of the synthesized sequence with m for all the executions e in the set of execution scenarios (lines 18-23). If the candidate is equivalent to m for all the execution scenarios, the phase terminates and returns the candidate (line 25). Otherwise, the algorithm discards the candidate and generates a new one, which will then be evaluated for all the execution scenarios. The function EQUIVALENT compares the object attributes and the return values obtained by executing the original method m and the candidate method sequence on a given execution scenario. If no candidate equivalent sequence is found within a given time bound, the first phase terminates (line 28), and the whole algorithm terminates as well (line 5).

The second phase is detailed in function FIND-COUNTEREXAMPLE (lines 30-38). During this phase the algorithm validates the candidate through the exploration of new scenarios in the attempt to violate the equivalence between m and the candidate equivalent sequence synthesized after the first phase. The search for a counterexample terminates when either a counterexample is found (line 34), or the search budget expires (line 37). If this process produces a counterexample, then the candidate is deemed as not equivalent to m.

The algorithm iterates from the first phase adding the counterexample to the execution scenarios. The main iteration (line 2-13) terminates when a timeout expires and the algorithm fails in synthesizing an equivalent sequence (line 5). If the algorithm cannot produce new counterexamples, it prints the likely-equivalent sequence (line 9), removes the method calls used in the synthesis of the candidate (line 10), and iterates to synthesize new equivalent sequences.

5.3 SBES: A Tool to Synthesize Equivalent Method Sequences

We implemented the algorithm in Figure 5.5 in a Java prototype tool called SBES (Search-Based Equivalent Synthesis). The key idea of our implementation is to leverage search-based test case generation techniques to synthesize equivalent method sequences.

The intuition of exploiting search-based test case generation to synthesize equivalent method sequences stems from the observation that the two problems have some commonalities. First, they both aim to generate fragments of code. In fact, the goal of test case generators is to produce a fragment of code that covers some structural elements in the code and potentially reveals the presence of a fault in the code. Similarly, we want to obtain a method or a combination of methods that are likely-equivalent with respect to a given method. Second, and more relevant, the problem of synthesizing an equivalent sequence can be easily translated in the problem of satisfying an adequacy criterion. The idea is to translate the problem of identifying equivalent sequences to the problem of searching for test cases whose execution covers a branch: if the test case generator produces a test case that covers a specific branch, this implies that we have synthesized either an equivalence or a counterexample. We target branch coverage

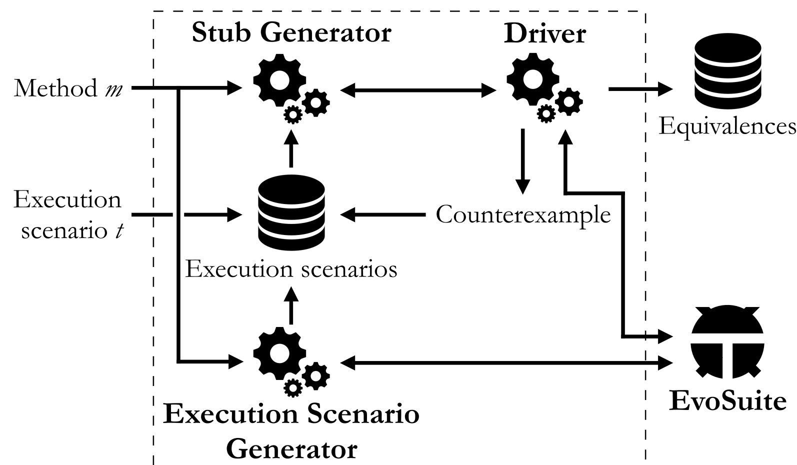


Figure 5.6. Main components of SBES

since it is both supported by search-based test case generators [FA13, LHG13, Ton04] and computationally feasible.

Figure 5.6 shows the main components of SBES, which exploits EvoSuite as search-based engine. The Execution Scenarios Generator generates a set of execution scenarios by invoking EvoSuite. The Stub Generator creates a modified version of the target class by removing the target method m to enable the synthesis of equivalent sequences. The Driver iteratively invokes EvoSuite to synthesize equivalent sequences and to search for counterexamples. EvoSuite natively supports the generation of test cases with method calls, constructors, arrays of random length, and primitive values. We modified EvoSuite to better deal with arrays of given length and values. EvoSuite does not generate some arithmetic operators, loops, and conditional statements, and thus our current prototype implementation cannot synthesize equivalent sequences that contain these constructs.

In the next sections we detail the generation process, with a particular focus on how we transformed the identification of equivalences into a search problem.

5.3.1 Initialization: Execution Scenarios

The starting point of our technique are the execution scenarios. Execution scenarios can be either provided by developers, typically in the form of a test suite for the target method m, or can be generated automatically with tools such as Randoop and EvoSuite [FA13, PLEB07].

In the context of Java programs, an execution scenario is a sequence of method invocations that generates objects by means of constructors, operates on such objects by means of public and protected methods, and terminates with an invocation of method m. The following test cases are two examples of valid execution scenarios for the pop() method of the Stack<T> class:

```
1 public void test01() {
2   Stack<Integer> s = new Stack<>();
3   s.push(1);
4   Integer result = s.pop();
5 }
```

```
1 public void test02() {
2   Stack<Integer> s = new Stack<>();
3   s.push(1);
4   s.push(1);
5   Integer result = s.pop();
6 }
```

In our experiments we used test suites when available, and we generated the execution scenarios with EvoSuite otherwise [FA11]. EvoSuite generates and evolves test suites in the attempt to cover a set of target branches through the invocation of any accessible method. Since the tool may generate method invocations that call the target method m indirectly, we modified EvoSuite to force every generated execution to include an explicit call to m as its last statement.

As in these examples, most of the classes in Java libraries are implemented using generic types. Ignoring generic types exacerbates the combinatorial explosion of methods and parameters, since type erasure substitutes generics with the base class java.lang.Object. Yet, by considering generic types we must concatenate method calls that both satisfy and adhere to the generic types specified at class instantiation time, increasing the complexity of both synthesis and counterexample processes.

In those cases where the execution scenarios declare and use concrete classes rather than generic types, we exploit such information. For example, given the previous execution scenarios we can replace the generic type T with the Integer type. This generic-to-concrete transformation is extremely useful in case of complex generic types. For example, suppose that we wish to synthesize equivalences for method Multimap<K,V>.put(key,value), with the following initial execution scenario: Multimap<Integer,String> m = new Multimap();m.put(15, "String"). Since in the execution scenario the generic types K and V are replaced with Integer and String respectively, we can safely replace all the occurrences of the generic types with the concrete classes. By resolving generic types, the search engine obtains more information to guide the search towards better individuals, without wasting time to find syntactically valid concrete classes.

5.3.2 First Phase: Candidate Synthesis

The goal of the first phase is to synthesize a sequence of method invocations that is equivalent to the target method m on a set of execution scenarios. We reformulate the synthesis goal in a test case generation problem by introducing an artificial method that contains a single branch. If the test case generator, in our case EvoSuite, can produce a test case then we have synthesized a fragment of code whose behavior is equivalent to m. The definition of the branch condition is crucial for the synthesis process and asserts the equivalence of method m and the synthesized sequence with respect to all the execution scenarios. The equivalence considers object states reached by the execution as well as the return values. To allow for the comparison, rely on a stub class.

The Stub Generator creates a stub for the target class, namely the class that includes the declaration of the method m. The stub class encloses all the execution scenarios, and evaluates the equivalence between the target method and the synthesized candidate sequence. The Driver iteratively synthesizes method sequences by invoking EvoSuite, and uses the stub class to evaluate whether the generated sequence is equivalent to the target method m.

Given a class C that declares the target method m, the Stub Generator produces a new stub class C_Stub that contains additional fields and methods. The additional fields store the results computed as well as the states reached by both the original method m and the currently generated test case. In particular, the additional fields are the following:

expected_states is an array containing one object of type C for each execution scenario. This data structure stores the state of these objects after the execution of each scenario.

expected_results is an array containing the return values of each execution scenario on the target method m.

actual_states is an array containing one object of type C for each execution scenario. This data structure stores the state of the objects after the execution of the synthesized method sequence on each scenario.

actual_results is an array containing the return values of the execution of the synthesized method sequence on each scenario.

The additional methods included in the stub class have the following functionalities:

class constructors: the constructor of the stub class invokes each scenario on the objects stored in expected_states, and stores the results in expected_results. It also invokes all the methods of each scenario on the objects stored in actual_states. These latter invocations do not include calls to the target method m.

methods proxies: the stub class declares every method originally declared in class C and every method that C inherits from any of its superclasses. Each of these methods are simply proxies for the invocation of the corresponding original method of C on every object in the actual_states array and they return the corresponding return values of such executions in the form of an array.

set_results: it is a utility method that stores the return values of the synthesized sequence in actual_results.

method_under_test: it is the target method for the search-based test case generator. It contains a single branch, whose condition asserts the equivalence of method

m and the synthesized sequence with respect to all the execution scenarios. The equivalence considers both the object states, as stored in expected_states and actual_states, and the return values, as stored in expected_results and actual_results.

Figure 5.8 shows the automatically generated stub for the Stack class of the Java standard library. Given the two execution scenarios presented in Section 5.3.1, the stub class declares two arrays of length 2 for the expected and the actual object states, and two arrays of length 2 for the expected and the actual execution results. The constructor at line 7 executes both scenarios, and stores states and results in the expected_states and expected_results arrays, respectively. The actual_states array contains the object states obtained by applying each execution scenario up to the invocation of the target method (for example, pop() in the running example).

Method push (line 25–31) is an example of how our prototype implementation generates proxy methods. A proxy method redirects the invocations of the original methods of the Stack class to each and every object stored in actual_states, which represents the initial state of an execution scenario. SBES generates a proxy method for every method that was originally declared in the Stack class, with the exception of the target method, in this example the pop() method.

The artificial method method_under_test at line 185 is the main driver for the synthesis of a candidate equivalent sequence. By generating an execution that covers the TRUE branch of this method, we obtain a method sequence that is equivalent to the target method in all the considered scenarios. We generate such sequence with EvoSuite [FA11]. The default strategy for branch coverage requires EvoSuite to generate a test case to cover both the TRUE and the FALSE branches for every possible basic condition. Since we are interested in a test case that covers the compound branch, we modified Evosuite so that its only goal is to cover the compound TRUE branch of method_under_test, so as to execute line 190.

The generation of likely-equivalent method sequences is guided by the fitness function that quantifies the distance of each candidate sequence from satisfying the condition at lines 186–189. Since the condition is a conjunction of atomic clauses, the fitness function is the sum of the branch distances for each single clause, so that the overall distance is zero when all the clauses evaluate to TRUE. In turn, the branch distances for the atomic clauses are computed as numeric, object or string distances, depending on the involved types. When the distance involves objects, the search-based algorithm cannot guide the evolution, since comparing objects with the boolean method equals flattens the fitness landscape [HHH+04]. To overcome this problem, we resort to an object distance that quantifies the difference between two objects, similarly to what ARTOO [CLOM06] and RECORE [RZF+13] implement.

Our object distance computes the similarity between two objects by recursively comparing all the object fields through the following cases:

Primitive: distance computed as absolute difference between the two numbers.

```
1   class Stack_Stub {
2     Stack expected_states[2] = new Stack[2];
3     int expected_results[2] = new int[2];
4     Stack actual_states[2] = new Stack[2];
5     int actual_results[2] = new int[2];
6
7     public Stack_Stub() {
8       // execution scenario 1
9       expected_states[0] = new Stack();
10      expected_states[0].push(1);
11      expected_results[0] = expected_states[0].pop();
12      actual_states[0] = new Stack();
13      actual_states[0].push(1);
14
15      // execution scenario 2
16      expected_states[1] = new Stack();
17      expected_states[1].push(1);
18      expected_states[1].push(1);
19      expected_results[1] = expected_states[1].pop();
20      actual_states[1] = new Stack();
21      actual_states[1].push(1);
22      actual_states[1].push(1);
23    }
24
25    public int[] push(int[] item) {
26       int result[2];
27       for (int i = 0 ; i < 2 ; i++) {
28          result[i] = actual_states[i].push(item[i]);
29       }
30       return result;
31    }
```

[...] *// all other methods in Stack*

```
179   public void set_results(int result[]) {
180      for (int i = 0 ; i < 2 ; i++) {
181        actual_results[i] = result[i];
182      }
183   }
184
185   public void method_under_test() {
186      if (distance(expected_states[0], actual_states[0]) == 0 &&
187          distance(expected_states[1], actual_states[1]) == 0 &&
188          distance(expected_results[0], actual_results[0]) == 0 &&
189          distance(expected_results[1], actual_results[1]) == 0) {
190        ; // target
191      }
192   }
193
194 }
```

Figure 5.8. The synthesis stub generated for the Stack class.

String: distance computed as Levensthein similarity measure between the two strings.

Array: recursively computed as pairwise distance between each array element. In case of arrays of different size, each missing element is replaced with the maximum possible value for the type. In case of array of object values, the missing elements are treated as null.

Object: recursively computed as distance among all the fields. If one object is null, the distance is set to a configurable value that represents infinity.

The readers should notice that such notion of equivalence based on the identity between objects is stronger and in fact *implies* the notion of behavioral equivalence, as discussed in Chapter 3.

The Driver component of our tool controls all the elements described so far, and drives the whole process towards the synthesis of a candidate equivalent sequence by invoking EvoSuite to generate a sequence of method invocations that tries to cover the target branch in method_under_test, after saving the results of the execution by calling set_results.

In an attempt to find an equivalent sequence for method pop of class Stack, the driver may generate the following sequence of method calls:

```
1  Stack_Stub x0 = new Stack_Stub();
2  int x1[] = x0.remove(0);
3  x0.set_results(x1);
4  x0.method_under_test();
```

which in turn can be automatically transformed into the candidate sequence:

$$\text{stack.pop()} \equiv \text{stack.remove(0)}$$

This candidate expresses the equivalence between pop(), which removes the object on top of the stack and returns such object, and remove(0), which removes the first element in the stack and returns it. This equivalence holds only because the first and the last elements in the two scenarios considered above are the same (two integer values equal to 1). This equivalence, though, does not hold in other scenarios. The next section describes how the second phase can invalidate such a spurious candidate.

5.3.3 Second Phase: Candidate Validation

The second phase validates the candidate equivalence synthesized in the first phase by considering other execution scenarios. This phase aims to identify a scenario for which the equivalence does not hold.

Similarly to the first phase, the prototype automatically generates a method_under_test containing a single branch asserting the *non equivalence* between method *m* and the synthesized candidate sequence. The prototype then automatically includes such method in the declaration of class C. For instance, Figure 5.9 illustrates how our SBES prototype automatically transforms class Stack for the counterexample phase.

```
1  public class Stack_Stub2 extends Stack {
2
3    public void method_under_test() {
4      Stack clone = deepClone(this);
5      int expect = this.pop();
6      int actual = clone.remove(0);
7      if (distance(this,clone) > 0.0 || distance(expect,actual) > 0.0) {
8        ; // target
9      }
10   }
11
12 }
```

Figure 5.9. The validation stub generated for the Stack class.

Similarly to the first phase, we exploit EvoSuite to automatically generate an execution covering the target branch (line 13), hence generating a counterexample for the equivalence. The original method pop is applied on object this, while the candidate sequence is applied to a clone of object this. We rely on a deep clone library to create exact copies of the current state of the object. This operation is crucial to avoid spurious results, since not all classes may contain a sound and complete implementation of the optional method clone().

For the Stack example, EvoSuite might produce the following method sequence in an attempt to cover the TRUE branch of method_under_test:

```
1  Stack x0 = new Stack();
2  x0.push(2);
3  x0.push(1);
4  x0.method_under_test();
```

which indeed provides a scenario that shows that the equivalence between pop() and remove(0) does not hold. In this scenario, the first and the last elements in the Stack are different, and consequently the two operations have different effects on the Stack. As described in the algorithm in Figure 5.5, the process iterates, and the synthesis of a new candidate takes into account also the new execution scenario:

```
1  Stack s = new Stack();
2  s.push(2);
3  s.push(1);
4  int result=s.pop();
```

In the second iteration of the first phase for this example, the stub considers three scenarios, and the size of all the arrays and the branch conditions to cover are updated accordingly.

The new iteration of the first phase may generate the following method sequence that covers the new target branch:

```
1  Stack_Stub x0 = new Stack_Stub();
2  int x1[] = x0.size();
3  int x2[] = ArrayUtils.add(x1, -1);
4  int x3[] = x0.remove(x2);
```

```
5 | x0.set_results(x3); x0.method_under_test();
```

thus producing the following new candidate equivalence:

$$\text{stack.pop()} \equiv \begin{array}{rcl} \text{int x0} & = & \text{stack.size();} \\ \text{int x1} & = & \text{x0-1;} \\ \text{result} & = & \text{stack.remove(x1)} \end{array}$$

In this new iteration, the search for a counterexample times out, and the synthesis process outputs the likely-equivalent sequence. Since a method can be equivalent to a code fragment that combines more than one method, SBES incrementally removes the methods used in the currently synthesized sequence from the stub. At each iteration SBES repeats the search process for each newly generated stub to obtain further equivalent sequences, when more exist.

5.4 Experimental Validation

The evaluation of our technique aims to answer the following research questions:

Q1 recall: Can the proposed approach correctly identify equivalent method sequences?

Q2 precision: How often does the proposed approach identify non-equivalent method sequences as equivalent?

Q3 performance: How efficiently can the proposed approach identify equivalent method sequences and counterexamples?

Q4 role of counterexamples: How often do counterexamples correctly discard method sequences that are not equivalent to the target one?

Research questions Q1 and Q2 deal with the effectiveness of the proposed approach by considering its ability to retrieve known equivalences (*recall*) and to report them with few false positives (*precision*). Q3 deals with the efficiency and the practical applicability of the approach. Q4 validates the need for the second phase of the approach to generate counterexamples and eliminate candidate sequences that were at first wrongly identified as equivalent.

To answer Q1 and Q2 we resort to the standard recall and precision metrics:

$$Recall = \frac{\text{true positives}}{\text{true positives+false negatives}}$$

$$Precision = \frac{\text{true positives}}{\text{true positives+false positives}}$$

Recall is defined as the ratio between the number of equivalent sequences correctly synthesized with the approach (*true positives*) and the total number of equivalent

sequences, which include both the ones correctly synthesized (*true positives*) and the ones that the approach fails to synthesize (*false negatives*).

Precision is defined as the ratio between the number of equivalent sequences correctly synthesized with the approach (true positives) and the total number of sequences deemed as equivalent, which include both the equivalent ones (*true positives*) and the non-equivalent ones erroneously identified as equivalent by the approach (*false positives*).

To answer Q3, we measure performance as the time required to *synthesize* an equivalent sequence and the time required to find a *counterexample*, since these two measures directly influence the overall performance of our approach. We use these two values to compute the optimal timeouts. In fact, we acknowledge a synthesized sequence as likely-equivalent to the target method when no counterexamples are found within a given timeout. The maximum time required to find a counterexample indicates the optimal value for the counterexample timeout: a smaller value would lead to missing some counterexamples, a larger value would cause time waste. Similarly, the maximum time required to synthesize a sequence indicates the optimal synthesis timeout, that is, the value that avoids missing sequences without wasting time.

For Q4, we measure the role of counterexamples as the number of method sequences identified as candidates for equivalence that the counterexamples discard as false positives. This number corresponds to the number of iterations between the second and the first phase, since discarding a sequence results in re-executing the first phase.

5.4.1 Experimental Setup

Table 5.1 shows the case studies used in our evaluation. For each library we use (column *Case Study*), the table reports all the classes we analyze (column *Class*) with their number of public API methods (*Class Methods*), target methods for our evaluation (*Target Methods*), and number of manually identified equivalences (*Equivalences*). Intuitively, the number of public methods in a class defines the lower bound of the space of possible elements that SBES has to deal with while searching for a candidate equivalence or a counterexample.

The first target of our experiments is class Stack, taken as a representative for the various containers available in the Java standard library.[1] Class Stack is challenging because it contains many non trivial equivalent method sequences. Our second set of classes is taken from Google Guava, and in particular from its collection of container classes.[2] Guava is particularly difficult to analyze due to complex generic classes and equivalence sequences that require deep method concatenations. We also considered a set of classes from Graphstream, a library to model and analyze dynamic graphs.[3]

[1] http://docs.oracle.com/javase/7/docs/api/java/util/Stack.html
[2] https://github.com/google/guava
[3] http://graphstream-project.org/

Case Study	Class	Class Methods	Target Methods	Equivalences
Java	Stack	50	15	45
Graphstream	Path	31	2	5
	Edge	36	9	20
	SingleNode	72	5	12
	MultiNode	76	5	12
	Vector2	29	5	21
	Vector3	39	6	22
Guava	ArrayListMultimap	25	15	18
	ConcurrentHashMultiset	27	16	16
	HashBasedTable	25	16	13
	HashMultimap	24	15	13
	HashMultiset	26	16	19
	ImmutableListMultimap	30	11	20
	ImmutableMultiset	32	8	20
	LinkedHashMultimap	24	15	13
	LinkedHashMultiset	26	16	19
	LinkedListMultimap	24	24	17
	Lists	17	8	16
	Maps	32	9	12
	Sets	30	10	25
	TreeBasedTable	27	15	17
	TreeMultimap	26	14	12
	TreeMultiset	35	20	34
Total		**778**	**266**	**421**

Table 5.1. Case studies analyzed in our SBES experiments.

Our experiments cover 23 classes and 266 methods. In particular, we use 15 methods of class Stack, 201 methods of 16 classes of Guava, and 32 methods belonging to 6 classes of Graphstream. Stack, Guava, and Graphstream represent different application domains, are developed and maintained by different third party subjects, and include all the language characteristics that we can currently handle with EvoSuite, which constrains our prototype implementation.

We run the experiments by feeding the prototype with the class under analysis, the target method, and an initial scenario. The target method is the method of the class under analysis for which we would like to synthesize equivalent method sequences. The initial scenario consists of one test case that was either extracted from the existing test suite, or generated automatically with EvoSuite, depending on the availability of the test suites. If the initial scenario contains concrete types for generic ones, we exploit our generics-to-concrete transformation. We configure EvoSuite to use Memetic Algorithms with local search operators applied every 75 generation as previously identified [MGG15].

We execute SBES with an overall budget of 20 attempts for the first phase, regardless

of how many times the technique refines the synthesized candidates by identifying counterexamples. We give a search budget of 180 seconds to the first phase and a search budget of 360 seconds to the second phase.

To answer Q1 and Q2 we compare the sequences that we synthesize automatically against the set of sequences that we previously identified with manual inspection within the limits of the current prototype. In theory, the amount of equivalent sequences would be infinite, since we can easily combine simple equivalent sequences to obtain new ones. For example, method pop() is equivalent to remove(size()-1), but is also equivalent to push() pop() remove(size()-1). In our experiments we considered only *minimal* equivalences that we informally identify as those that cannot be derived by suitably combining simpler equivalences or adding method calls with a globally null effect, as the pair push() pop() in the previous example.

In our experiments, we synthesize equivalent method sequences for single methods only. Synthesizing equivalent sequences for method sequences does not change the problem, but simply increases the size of the experiment. Our automatic synthesis is limited by EvoSuite that can deal with method calls, constructors, primitive values and arrays, but not with all the arithmetic operators, loops and conditional statements. These limitations are inherited from EvoSuite, and do not belong to the approach that can synthesize equivalent sequences for general method sequences, potentially exploiting all language constructs.

The execution environment provides a listener that logs detailed information about the timing of the events. Each iteration consists of creating a stub, compiling and executing it. The listener logs the compilation and execution time, recording the execution time of both the prototype and EvoSuite. These data allowed us to compute all the performance metrics discussed above.

5.4.2 Effectiveness (Q1, Q2)

In this section, we discuss the experimental results. We ran our prototype on 266 methods of 23 classes taken from the Stack Java class, the Google Guava library, and the Graphstream library. We automatically synthesized 312 equivalent method sequences, which represent more than 74% of the 421 sequences that we manually identified ahead by inspecting the classes documentation. We considered only the *minimal* equivalences, and we excluded those that could not be found due to the limitations of our prototype.

Table 5.2 presents a sample of the equivalent sequences that we synthesized automatically. SBES can synthesize both simple equivalences, for example methods that can be replaced interchangeably, and complex equivalences that include non trivial combinations of method calls, as in the case of Collection c = new Collection(); c.add(e); addAll(c); that is equivalent to addElement(e).

Table 5.3 summarizes the experiment results. For each of the analyzed methods, the table shows the following information:

Original sequence	Synthesized sequence
java.util.Stack	
addElement(Object e)	add(e)
	push(e)
	add(e,size())
	Collection c=new Collection(); c.add(e); addAll(c);
clear()	removeAllElements()
	setSize(0)
	Collection c=new Collection(); retainAll(c);
e = pop()	e=peek(); index=size()-1; removeElementAt(index);
e = set(int i, Object o)	e=remove(i); insertElementAt(o,i)
org.graphstream.graph.implementations.Edge	
getSourceNode()	temp=getTargetNode(); getOpposite(temp)
getTargetNode()	temp=getSourceNode(); getOpposite(temp)
org.graphstream.graph.implementations.SingleNode	
getAttribute(String s)	getAttribute(s,Object.class)
org.graphstream.ui.geom.Vector2	
fill(double d)	Vector2 v=new Vector2(); v.set(d,d); copy(v);
	Point2 p=new Point2(d,d); set(p.x, p.y);
	scalarAdd(d)
org.graphstream.ui.geom.Vector3	
copy(Vector3 v)	Point3 p=new Point3(); p.move(v); set(p.x,p.y,p.z)
com.google.common.collect.ArrayListMultimap	
create()	create(new LinkedListMultimap())
	create(0,0)
put(Object k, Object v)	putAll(k, new ImmutableSet(v, 0))
com.google.common.collect.LinkedListMultimap	
removeAll(Object k)	replaceValues(k, new LinkedList())
s = size()	Collection c=values(); s = c.size();

Table 5.2. Sample sequences synthesized with SBES.

Case Study	Class	Equiv	Synthesized TP	FP	Prec	Rec
Java	Stack	45	32	7	0.82	0.71
Graphstream	Path	5	5	2	0.71	1.00
	Edge	20	20	1	0.95	1.00
	SingleNode	12	12	0	1.00	1.00
	MultiNode	12	12	0	1.00	1.00
	Vector2	21	21	3	0.87	1.00
	Vector3	22	22	4	0.84	1.00
Guava	ArrayListMultimap	18	12	3	0.80	0.67
	ConcurrentHashMultiset	16	6	2	0.75	0.38
	HashBasedTable	13	2	8	0.20	0.15
	HashMultimap	13	13	1	0.92	1.00
	HashMultiset	19	19	5	0.79	1.00
	ImmutableListMultimap	20	2	0	1.00	0.10
	ImmutableMultiset	20	3	0	1.00	0.15
	LinkedHashMultimap	13	12	3	0.80	0.92
	LinkedHashMultiset	19	19	6	0.76	1.00
	LinkedListMultimap	17	11	0	1.00	0.65
	Lists	16	15	1	0.94	0.94
	Maps	12	8	0	1.00	0.67
	Sets	25	21	0	1.00	0.84
	TreeBasedTable	17	3	10	0.24	0.18
	TreeMultimap	12	8	2	0.80	0.67
	TreeMultiset	34	34	10	0.78	1.00
Total		**421**	**312**	**68**	**0.82**	**0.74**

Table 5.3. Q1, Q2: Effectiveness of SBES.

1. the number of *minimal* equivalent sequences identified with manual inspection (column Equiv), which we use as baseline,
2. the amount of *true* equivalent sequences automatically synthesized (column TP),
3. the amount of candidate sequences *wrongly* identified as equivalent (column FP),
4. the precision (Prec) and the recall (Rec).

Table 5.3 shows that in most of the cases where a target method has multiple equivalent sequences our approach can synthesize a substantial fraction—if not all—of the equivalences. This is a very interesting result, since all of the practical applications of redundancy typically benefit from a high level of redundancy [CGM+13, CGG+14, CGPP15, CGP09]. In those case studies where SBES was not able to synthesize all the manually identified equivalences, we identified three major issues that limit the its effectiveness: wrong handling of return values, sub-optimal search configuration, and failure in the generation of stubs.

Wrong return handling We observed that sometimes the correct equivalent sequence was indeed synthesized during the evolution of the individuals, but the objects holding the correct results were not used as parameters of the set_results method. As a result, the actual_result array was not set with the proper objects, and the search did not stop.

Similarly, we observed cases where the solution was actually correctly synthesized in an individual, but some additional and further method invocations were disrupting either the return value or the objects stored in the actual_state data structure.

These are indeed limitations of the current approach that can be overcome by improving the evolution process to make use of any object available in the current method sequence, instead of arbitrarily choosing one, and to better locate the method_under_test location. For example, one can design a new local search operator that optimizes where to place the set_results and method_under_test methods.

Sub-optimal search configuration The optimal EvoSuite configuration for our code synthesis problem might differ from the recommended configuration for test case generation. One major issue of search-based approaches, in particular Genetic Algorithms, is the tendency throughout the generations in producing individuals of increasing size but without a corresponding increase of fitness function. This *bloating* phenomenon is addressed by EvoSuite through the limitation on the size of an individual. When a test case exceeds a predefined limit, it is removed from the population. Such countermeasure has been demonstrated effective for test case generation [FA15, FA16], but limits the effectiveness of SBES. To successfully identify some equivalent sequences, SBES requires to concatenate a long series of methods and parameters. Unfortunately, EvoSuite often removes the candidate with a partially correct sequence because it exceeds the predefined bloat limit. As a result, EvoSuite tends to generate solutions that lack of diversity and that contain only a partial correct solution.

This issue is particularly relevant in the cases of immutable data structures, such as ImmutableListMultimap and ImmutableMultiset where SBES successfully synthesized only 10% and 15% of the equivalences manually identified. In the experiments with the two immutable classes, the majority of the equivalences require to concatenate from 3 to 12 methods. However, EvoSuite quickly reaches the bloat limit and removes the candidate from the population.

On the one hand, it is possible to increase the bloat limit, thus potentially allowing EvoSuite to generate the required test cases. On the other hand, however, increasing the size of the individual directly affects the performance of the evolutionary algorithm, potentially lowering the effectiveness of the approach even in the successful cases. One promising solution is to allow EvoSuite to temporary exceed the bloat limit if the fitness function of the candidate improves in the following generations. This modification can be applied in conjunction with a minimization phase where we remove source code lines that do not directly influence the fitness function of the individual.

Case Study	Class	Equiv	Type Erasure		Generics Support	
			TP	FP	TP	FP
Java	Stack	45	22	4	32	7
Guava	ArrayListMultimap	15	7	1	13	1
	ConcurrentHashMultiset	16	5	0	9	1
	HashBasedTable	16	3	6	3	8
	HashMultimap	15	7	0	9	2
	HashMultiset	16	6	0	15	3
	ImmutableListMultimap	11	1	1	2	1
	ImmutableMultiset	8	3	0	1	0
	LinkedHashMultimap	15	6	1	9	1
	LinkedHashMultiset	16	5	1	19	2
	LinkedListMultimap	15	6	2	10	1
	Lists	8	18	0	17	3
	Maps	9	6	0	5	0
	Sets	10	12	2	15	0
	TreeBasedTable	15	0	8	4	8
	TreeMultimap	14	4	1	9	3
	TreeMultiset	20	12	5	32	13
Total		**266**	**123**	**32**	**204**	**54**

Table 5.4. Contribution of the generic-to-concrete transformation on the effectiveness of SBES.

Failure of the stub generation In HashBasedTable, TreeBasedTable, and TreeMultiset, the counterexample phase fails to discard spurious candidate sequences due to the inability of EvoSuite to generate a syntactically valid test case. The validation phase, thus, fails in invalidating even the most trivial spurious candidate.

As discussed in Section 5.3.3, the validation phase creates a stub that extends the class under analysis, as shown in Figure 5.9. The Java programming language force every sub-class to re-declare all non-default the constructors. In HashBasedTable, TreeBasedTable, and TreeMultiset, the constructors declarations lead to compilation errors due to the use of non-visible inner classes.

A practical solution for this issue is to generate a stub that does not extend the target class, but rather requires an object of the class as additional parameter of the method_under_test. The subsequent challenge is to then guide EvoSuite to efficiently generate and use the additional object.

As discussed in the evaluation setup, we ran SBES with both generics-to-concrete transformation and Memetic Algorithms. It is thus interesting to evaluate if and to what extent these enhancements contribute to increase the effectiveness of SBES.

Table 5.4 reports the comparison between SBES with and without generics-to-concrete transformation on Java and Guava case studies, since Graphstream does not

Class	GA		MA^{10}		MA^{50}		MA^{75}		MA^{85}		MA^{100}	
	TP	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**	**TP**	**FP**
ArrayListMultimap	13	1	12	2	14	1	12	3	13	0	11	3
ConcurrentHashMultiset	9	1	7	2	5	2	6	2	7	4	4	3
HashBasedTable	3	8	4	7	3	6	2	8	5	8	6	7
HashMultimap	9	2	12	1	12	1	13	1	12	2	12	2
HashMultiset	15	3	16	7	15	4	19	5	17	4	16	5
ImmutableListMultimap	2	1	2	1	0	0	2	0	1	0	1	2
ImmutableMultiset	1	0	4	0	5	0	3	0	4	0	0	0
LinkedHashMultimap	9	1	12	1	10	1	12	3	11	2	11	2
LinkedHashMultiset	19	2	16	4	15	4	19	6	16	3	18	5
LinkedListMultimap	10	1	9	1	10	0	11	0	10	1	11	3
Lists	17	3	18	2	14	2	15	1	12	4	16	3
Maps	5	0	6	0	7	0	8	0	8	0	6	0
Sets	15	0	16	0	17	0	21	0	15	0	19	2
TreeBasedTable	4	8	1	7	4	6	3	10	9	4	4	9
TreeMultimap	9	3	7	1	11	2	8	2	7	1	9	1
TreeMultiset	32	13	26	11	28	10	34	10	28	4	30	9
Total	**172**	**47**	**142**	**36**	**170**	**39**	**188**	**50**	**175**	**37**	**174**	**56**

Table 5.5. Comparison between Genetic Algorithms (GA) and Memetic Algorithms (MA) on the effectiveness of SBES.

use generic types. The table reports for each class the number of known equivalent sequences (column *Equiv*). The effectiveness of SBES without generic type support (columns *Type Erasure*) and with generic-to-concrete transformation (columns *Generics Support*) is reported in terms of correct equivalences synthesized (columns *TP*) and false positives (column *FP*). The generic-to-concrete transformation is indeed beneficial for the synthesis phase of SBES, leading to 66% more correct equivalent sequences. An increase in the effectiveness in the synthesis phase negatively impacts the effectiveness of counterexample phase, which leads to an equivalent percentage of additional false positives. Under this perspective, solving the uncovered limitations of the generation of the validation stub become even more significant.

Table 5.5 reports the comparison between SBES configured with EvoSuite using Genetic Algorithms (GA) and Memetic Algorithms (MA). The comparison was performed on the Guava case study, since the Guava benchmark is composed of complex equivalent sequences that can benefit the most from the local search operators applied by EvoSuite. EvoSuite requires in input the frequency at which to apply the local search. Since the frequency it is a problem-dependent variable, we performed several runs by applying local optimization every 10, 50, 75, 85, and 100 generations. Table 5.5 shows for each target class (column *Class*) the effectiveness of SBES with EvoSuite configured with GAs (columns *GA*) or MAs (columns *MA*) with the chosen local search frequencies. As for the previous experiments, we report the effectiveness of SBES in terms of true and false

positives (columns *TP* and *FP*, respectively).

MAs outperform GAs in terms of true equivalences synthesized with an increment of 9%, however the effectiveness of memetic algorithms largely depends on the frequency at which EvoSuite performs the local search. If the local search occurs too often, it steals search budget from the global search. On the other hand, if the local search occurs infrequently, it does not bring much benefits. With 10 generations we obtained the worst result, since we synthesized 30 equivalences less than the run with GAs (-17%). We obtained consistently better results for the other configurations up to the optimal rate of once every 75 generations. After this threshold, local search seems not to be frequent enough, since the effectiveness decreased again (-7% w.r.t. the optimal configuration).

In summary, despite the discussed limitations and issues, precision and recall are high, almost always close to or equal to one, indicating that the proposed approach can retrieve a large portion of the known equivalent sequences with an acceptable number of false positives. Therefore, we can answer positively to **Q1, Q2**: The proposed approach can correctly identify one and often more than one equivalent method sequences, with recall and precision which are close or equal to one in the majority of the cases.

5.4.3 Efficiency (Q3)

Table 5.6 reports the efficiency metrics. Column *Synthesis* shows the time required to synthesize an equivalent sequence, while column *Validation* reports the time for the counterexample generation. In particular, column *Max* shows the optimal timeout that can be set to the validation phase without altering the effectiveness of the approach, that is the precision and recall values reported in Table 5.3. Column *Synthesis* report the median of the values computed over the runs across the target methods of each class. Under column *Validation*, column *Median* reports the median of the values computed over the runs across the target methods of each class, while column *Max* reports the worst computation time experienced in the experiments during the validation phase. The table reports only the counterexample timeout because the synthesis timeout is always lower than the counterexample one, and thus the counterexample timeout represents an upper bound for the performance of the approach.

The execution time is acceptable and compatible with the typical usage scenarios in which redundancy is needed. In fact, even when equivalent sequences are used at runtime, for example in self-healing applications, the synthesis of equivalent sequences can be carried out in advance, offline. Hence, we can answer positively to research question **Q3**: The proposed approach requires a total execution time that is compatible with the typical application scenarios, where redundancy can be identified offline.

Case Study	Class	Synthesis	Validation	
			Median	Max
Java	Stack	13.0s	9.0s	190s
Graphstream	Path	26.0s	15.0s	60s
	Edge	17.0s	6.0s	7s
	Node	15.0s	-	-
	MultiNode	20.0s	-	-
	Vector2	13.0s	9.0s	36s
	Vector3	76.5s	7.0s	29s
Guava	ArrayListMultimap	28.7s	20.5s	191s
	ConcurrentHashMultiset	46.5s	16.0s	192s
	HashBasedTable	26.5s	9.0s	12s
	HashMultimap	28.0s	17.2s	121s
	HashMultiset	31.2s	13.5s	116s
	ImmutableListMultimap	24.5s	91.0s	194s
	ImmutableMultiset	32.0s	-	-
	LinkedHashMultimap	29.7s	20.7s	191s
	LinkedHashMultiset	37.0s	13.5s	191s
	LinkedListMultimap	27.0s	94.0s	195s
	Lists	13.0s	99.0s	189s
	Maps	16.0s	66.0s	89s
	Sets	18.0s	15.0s	45s
	TreeBasedTable	27.5s	69.2s	169s
	TreeMultimap	30.0s	75.2s	104s
	TreeMultiset	27.0s	31.5s	197s

Table 5.6. Q4: Efficiency of the approach

5.4.4 Counterexamples (Q4)

Table 5.7 reports the data about the effectiveness of the counterexamples: column *False Positives* indicates the amount of sequences that were erroneously identified as equivalent and were not automatically discarded with a counterexample. Column *Discarded* indicates the amount of overfitted candidate solutions that are identified as non-equivalent by a counterexample, and column *Efficiency* indicates the percentage of sequences automatically discarded with counterexamples. The table indicates that counterexamples are extremely effective in identifying and removing many method sequences erroneously proposed as equivalent, from 89% in the worst case to 100% in the best case.

We can thus positively answer research question **Q4**: Counterexamples can discard a relevant amount of method sequences erroneously identified as equivalent to the target one, and can thus reduce significantly the number of reported false positives.

Case Study	Class	False Positives	Discarded	Efficiency
Java	Stack	7	258	97.35%
Graphstream	Path	2	23	92.00%
	Edge	1	29	96.67%
	Node	0	0	-
	MultiNode	0	0	-
	Vector2	3	76	96.20%
	Vector3	4	95	95.95%
Guava	ArrayListMultimap	3	76	96.05%
	ConcurrentHashMultiset	2	244	99.18%
	HashBasedTable	8	78	89.74%
	HashMultimap	1	89	98.87%
	HashMultiset	5	104	95.19%
	ImmutableListMultimap	0	54	100.00%
	ImmutableMultiset	0	0	-
	LinkedHashMultimap	3	70	95.71%
	LinkedHashMultiset	6	92	93.41%
	LinkedListMultimap	0	105	100.00%
	Lists	1	74	98.64%
	Maps	0	96	100.00%
	Sets	0	91	100.00%
	TreeBasedTable	10	106	90.56%
	TreeMultimap	2	65	96.92%
	TreeMultiset	10	138	92.75%

Table 5.7. Q5: Effectiveness of counterexamples

5.4.5 Limitations and Threats to Validity

Our SBES tool is limited primarily by technological limitation of the EvoSuite test case generator. EvoSuite deals with method calls, constructors, primitive values and arrays, but not with all the arithmetic operators, loops and conditional statements. Not synthesizing loops and conditionals limits the overall effectiveness of our approach. In fact, a substantial amount of equivalences contains loops. As a representative example, in the Java Stack class the methods putAll(Collection) and removeAll(Collection) are equivalent to invoking the put or remove method on each element of the collection.

A possible solution for this problem is to exploit grammar-based genetic programming (GGP) approaches. GGPs use an input grammar to define the structure of individuals. We could enhance EvoSuite to exploit GGPs with an input grammar that incorporates loops and conditionals. However, GPPs pose two major challenges. First, it is not straightforward to create a grammar for the Java programming language: Intuitively, we would like to create a grammar that is as comprehensive as possible to maximize the number of synthesized equivalences. On the other hand, a non-trivial and complex grammar hampers the efficiency of the approach, since the search algorithm has to deal with complex cases and conditions. Second, the synthesis of loop conditions is challenging, and might lead to infinite iteration, thus impacting on the overall efficiency—and in turn effectiveness—of the GGPs.

Another limitation of SBES stems from our implementation of the object distance. Our object distance deems two object as equivalent if and only if their internal fields are identical. This implementation limits the effectiveness of our technique under certain circumstances. For example, if a class creates an internal data structure only the first time it is accessed (lazy initialization), our implementation may deem two semantically equivalent object as different. Our technique would thus benefit from a better object distance based on observational equivalence, as discussed in Chapter 3 and Chapter 4.

The main challenge with observational equivalence is to balance the trade-off between completeness and efficiency. Theoretically, we need to provide and infinite amount of infinite sequences of methods to pinpoint any observable difference. Practically, and especially within the fitness function evaluation, we must rely on an efficient and bounded notion of observability. As a result, we need to limit the bound to increase the efficiency—and in turn—the effectiveness of the approach. On the other hand, we need to increase the bound to improve the precision of the approach, in terms of amount of generated false positive. The evaluation and improved implementation of the object distance based on observational equivalence thus need further investigation and evaluation.

We acknowledge potential problems that might limit the validity of our experimental results. The main threats that affect the validity of the empirical study described above is the authors' bias. We have manually identified the reference set of equivalent sequences used to assess the effectiveness of the approach. Such task was carried out before

running SBES to avoid any influence from SBES output. Moreover, we cross-checked the equivalent sequences identified to verify that no equivalent sequence was missed and that the identified equivalent sequences were correct.

Threats to external validity may derive from the selection of the case studies. We have validated our approach on 23 classes and 266 methods, taken from three real-world subjects. Different results could be obtained for different systems. We have chosen three subjects, java.util, Graphstream, and Google Guava, that were known to contain some degree of equivalence in their implementation. By construction, our approach will not synthesize any result on systems that do not include any equivalence at all.

The selection of the subjects used in the experiments was driven exclusively by prior knowledge about the presence of equivalence. Hence, we expect our approach to behave similarly on other systems having a comparable degree and kind of equivalence. On the other hand, specific implementation details might affect the performance of search-based generators in finding candidate method sequences or counterexamples. For instance, the use of generic types represented a technological obstacle that required some tool adaptation.

Chapter 6

Conclusion

This dissertation presents the first systematic investigation of software redundancy. Redundancy has been extensively exploited by researchers and practitioners as a key ingredient of numerous techniques such as fault tolerance, software testing, and automatic program repair. Despite its widespread use, there are no formal definitions of the concept of software redundancy, nor approaches and tools to quantitatively and qualitatively assess the redundancy present in software systems.

This thesis introduces a formal definition of software redundancy whereby two code fragments are considered redundant when they provide the same functionality through different executions. More specifically, two code fragments are redundant when their execution produces results indistinguishable to an external observer, while the executions differ in the set of executed instructions, in the order in which the instructions are executed, or both in the set of instruction and their order.

On the basis of this definition of redundancy, we develop a measure of redundancy that is both practical and significant. Our measure considers a finite set of initial states obtained from a finite set of executions, and quantifies the degree of redundancy as a combination of a degree of observational equivalence between computed results and application states, and a degree of difference between execution traces. We demonstrate the significance and therefore usefulness of this measure as an indicator of redundancy. In particular, we show experimentally that our measure can distinguish between shallow and deep high-level redundancy, where not only the code is different but also the implemented algorithm differs. Moreover, we demonstrate that our measure is a good predictor of the effectiveness of techniques that exploit redundancy.

In the thesis, we also report an approach to automatically identify redundancy in software systems, with particular emphasis on the redudant method calls. In particular, our approach automatically identifies method sequences whose behavior is equivalent to a given target method. Our approach efficiently explores the solution space by restricting the possible executions to a finite set, and then validate the candidate solutions on additional executions. We demonstrate the effectiveness and efficiency of our approach

on 266 target methods taken from three real-world subject systems. We show that our technique correctly identifies a large amount of equivalences where redundancy was known to exist, with adequate precision and performance.

Contributions

The major contributions of this thesis are a formalization and measurement of software redundancy, and a technique to automatically identify redundancy at the method level. This thesis provides the first formalization of *what* software redundancy is and *how* one can measure the redundancy present in a software system. The technique to automatically identify equivalent functionalities provides a way to investigate *where* a software system is intrinsically redundant. The specific contributions of this thesis are:

- **A formalization of software redundancy.** We define and abstract and general notion of redundancy at the code level. Two code fragments are redundant when they produce observationally equivalent results, and differ in their execution. Two code fragments are observationally equivalent when, starting from the same initial state and with the same input, it is not possible to distinguish the two code fragments by executing any code fragment that interacts with them. Two executions are different when the actions on the application state are different, or they are the same actions but in different order.

- **A practical measurement method, based on our formalization, to quantify the redundancy of differently designed code fragments.** Our method computes redundancy as product of a measure of the degree of equivalence of the fragments and a measure of the degree of diversity between their execution. Our method measures equivalence by employing a bounded notion of observability that is implemented as approximation of the probability of identifying differences between the two fragments. Our method measures execution differences by computing a specific form of edit distance between specific projections of a finite set of execution traces that log memory changes.

- **Empirical evidence of the usefulness and significance of the measure of redundancy.** We present a set of experimental results that show that our measure distinguishes code that is only minimally different, from truly redundant code, and distinguish low-level code redundancy from high-level algorithmic redundancy. The results also confirm that our practical measure can help predict the effectiveness of techniques that exploit redundancy, such as the Automatic Workarounds technique [CGM+13].

- **A technique to automatically identify equivalent sequences of methods.** The technique efficiently synthesizes sequences of method invocations that are equivalent to a given target method within a finite set of executions. The technique

exploits genetic algorithms to efficiently explore the space of possible solutions, and employs a two-phase iterative approach to prune the solution space from spurious results.

- **Empirical evidence of the effectiveness of the technique in automatically identifying equivalent sequences of methods.** We implemented our automatic identification technique in a prototype for Java. We evaluated the prototype on 266 target belonging to 23 classes from three real-world libraries. The experiment confirm that our approach is valid, and that our technique can successfully synthesize a large amount of equivalent method sequences with good precision and performance.

Future Directions

The work presented in this dissertation uncovered or touched upon problems and ideas that remain open for future research. We would like to conclude this dissertation with a set of new challenges opened by the results presented in this dissertation: enriching the model and measure of redundancy, improving the automatic identification of redundancy, studying the nature of intrinsic redundancy as a phenomenon of software development processes, and investigating new applications of intrinsic redundancy.

- **Enriching the model and measure of software redundancy.** The model and corresponding measure of redundancy proposed in this dissertation are correct, useful, and significant for the developer, but they are currently limited to single-threaded, procedural code fragments. One way to enrich our model and measure of redundancy is to consider multi-threaded code. To extend in this direction, the model needs to account for a notion of execution that goes beyond one single sequence of actions. One possibility is to require a partial order over the events in the execution traces, for example by adapting and extending the definitions and algorithms proposed by Terragni et al. [TCZ15]. Another potential extension includes the evaluation of the measure of redundancy to consider entire components or systems. To obtain a practical measure of redundancy for entire components, we need to evaluate the overall scalability of the measurement method. In particular, the dissimilarity metrics used in our practical method might become too expensive to be precisely applied on large execution traces. Moreover, a useful measure of redundancy should take into account how critical is a sub-component while assessing the redundancy level of the overall system.

- **Improving the identification of equivalences.** Our automatic identification of equivalent method sequences is effective at identifying equivalent methods or combination of methods, but does not deal with conditionals and suffers from spurious results.

We can enhance our method towards the synthesis of conditionals. In fact, a substantial amount of equivalent method sequences manually extracted from the code documentation contains loops. To automatically synthesize loops and branches our plan is to investigate the use of grammar-based genetic programming approaches. The main obstacle for a profitable use of grammar-based genetic programming is the complexity of the evolutionary computation in case of non-trivial input grammars. We thus need to rely on incomplete grammars, which should focus on conditional statements, rely on heuristics to decide when to use grammar-based genetic programming, or both.

Although the evaluation results show that the counterexample phase can discard most of the spurious results, i.e., candidate solutions that are equivalent only when applied to the execution scenarios, the counterexample phase can be improved to reduce the number of false positives, thus improving the precision of the technique. A possible solution is the systematic exploration of both the original and the candidate methods to expose any divergence in their behavior, for example by using symbolic execution as investigated by Ramos and Engler [RE11, RE15].

Another future research direction is the extension of the technique to automatically identify *redundant* method sequences by integrating our measure of redundancy.

- **Characterizing intrinsic redundancy.** A formal notion of redundancy and a technique to automatically identify equivalent method sequences opens the opportunity to systematic study the pervasiveness and the nature of intrinsic redundancy in modern software systems. In essence, we plan to characterize the intrinsic redundancy of software systems by correlating its presence with some relevant aspects of the software development process. We also plan to see whether different kinds of intrinsic redundancy exist and which are their essential features. Our ultimate goal is to comprehend intrinsic redundancy as a phenomenon, to harness its power *by design.*

- **Investigating new applications of intrinsic redundancy.** In Chapter 2.2, we discuss the application of intrinsic redundancy in the context of software reliability, and in particular for the design of mechanisms for runtime failure recovery and automated oracles, focusing on functional properties. The notion of intrinsic software redundancy could also be extended to *non-functional* properties, where it might find many new applications, for example in performance optimization and security.

  Redundant code fragments execute different sequences of actions that may lead to notable differences in runtime behavior, and such differences can be exploited to alleviate performance problems. The redundancy intrinsically present in software systems offers new opportunities for automatically improving performance and resource consumption at runtime. The key idea is to devise a "profile" of the

redundant code that captures non-functional differences among the alternatives, for instance in terms of timing, memory or energy consumption, or network utilization. This "non-functional profile" can then be updated and exploited at runtime, while efficiently monitoring the system execution, to adapt the behavior to meet performance or resource usage requirements.

Redundant code fragments may also provide different security levels that could be exploited to tackle security issues and overcome runtime problems. Recent work has investigated the possibility of exploiting some form of explicit redundancy to mitigate security issues. N-variant systems increase application security by executing different synthesized variants of the same program in parallel [CEF+06]. Orchestra tackles security issues by creating multiple variants of the same program based on various compiler optimizations [SJGF09]. Replicated browsers tackles security problems by executing different browsers in parallel [XDK12]. Intrinsic redundancy offers a promising alternative to implement new security mechanisms by defining a security profile of redundant code fragments and by efficiently executing the various alternatives to identify divergences in their runtime behavior.

Appendices

Appendix A

Measure of Software Redundancy: Experiments

A.1 Redundancy Measurements

A.1.1 Identity

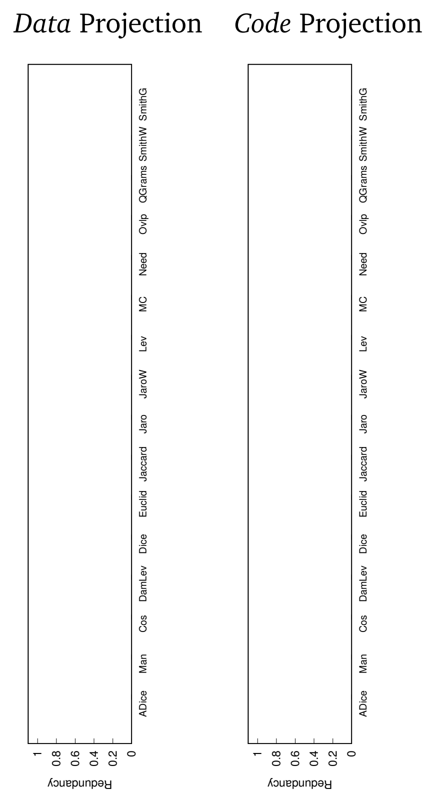


Figure A.1. Measurements obtained on two identical execution traces

A.1.2 Refactoring

Data Projection *Code* Projection

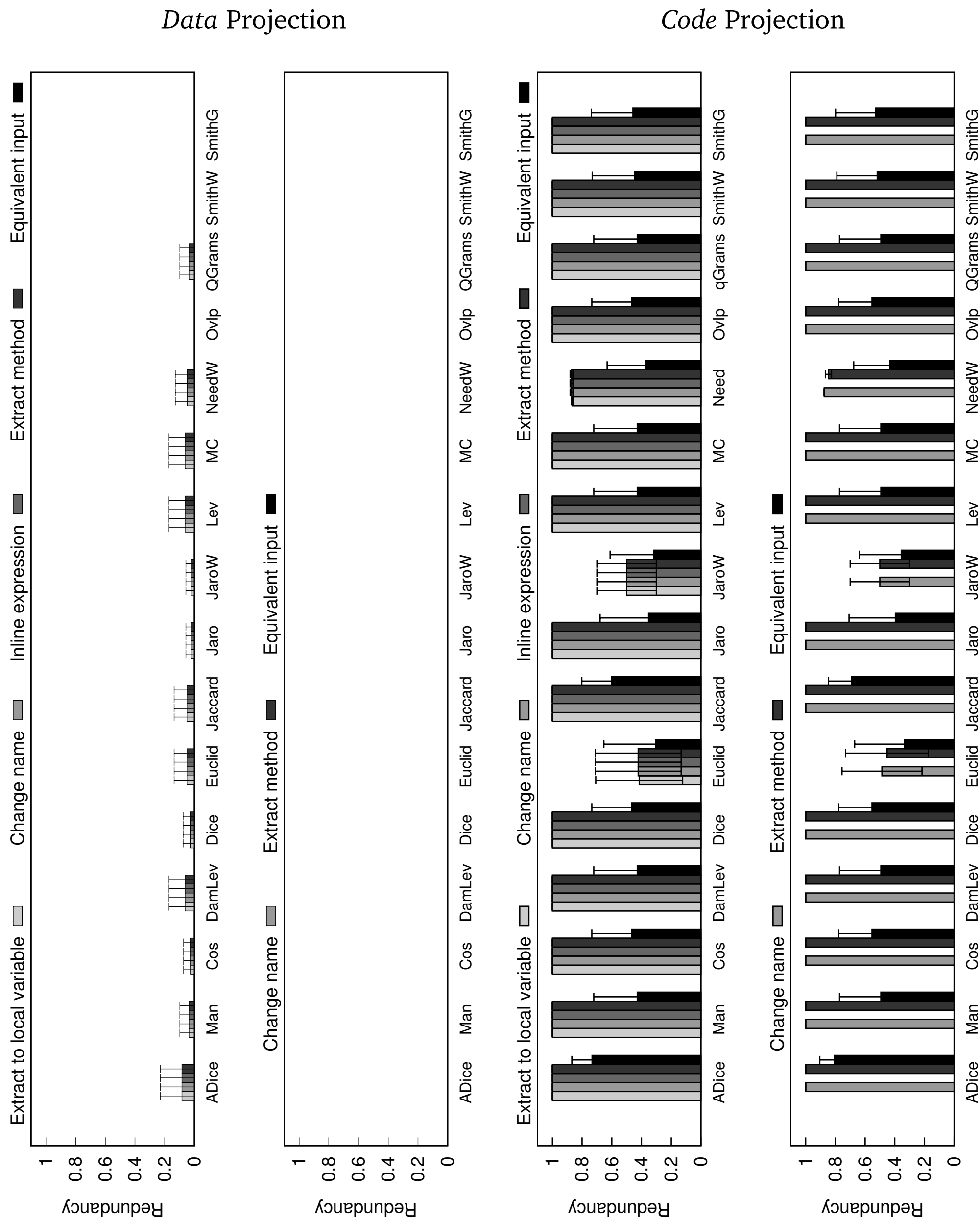


Figure A.2. Measurements after refactoring: binary and linear search

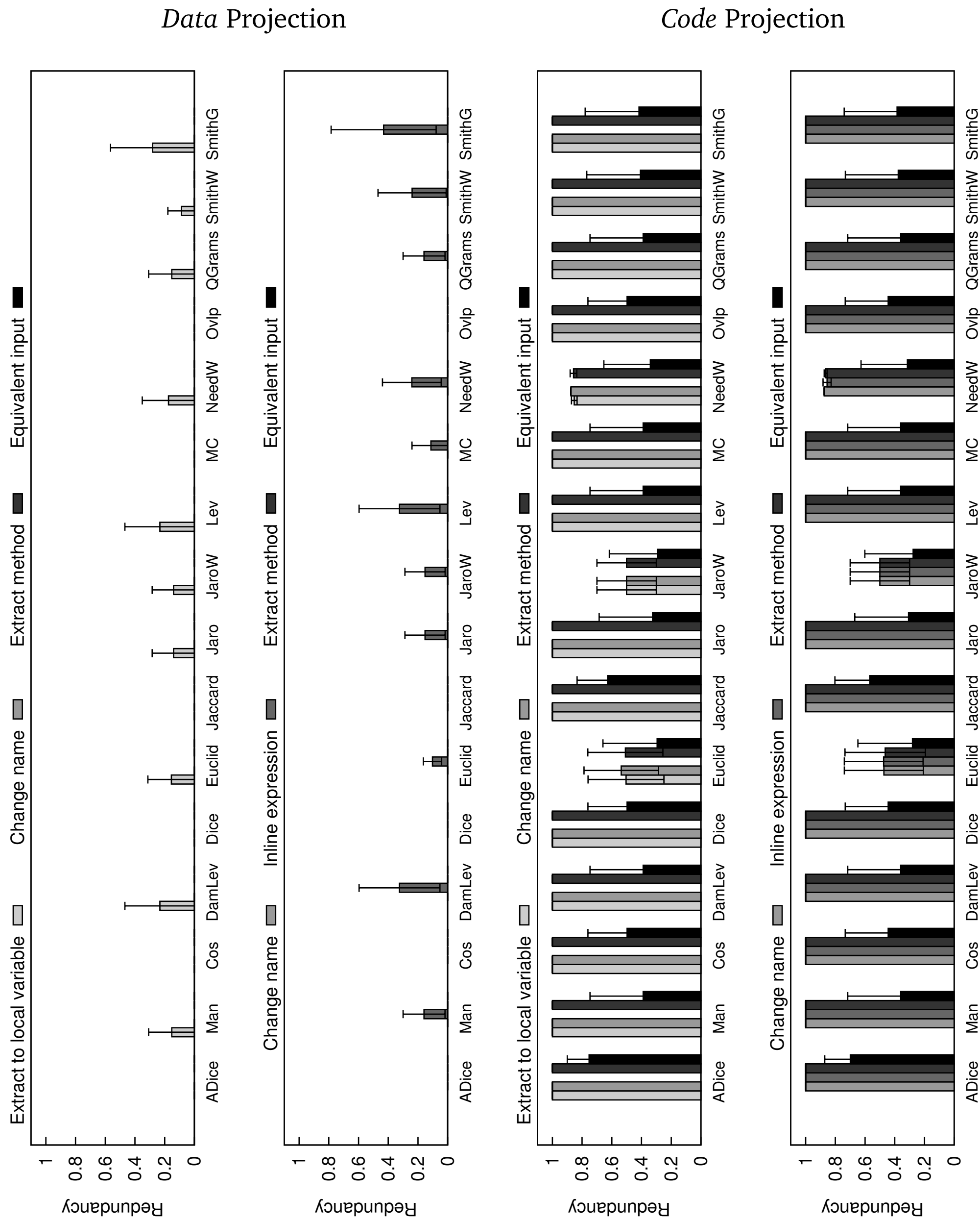


Figure A.3. Measurements after refactoring: bubble sort and insertion sort

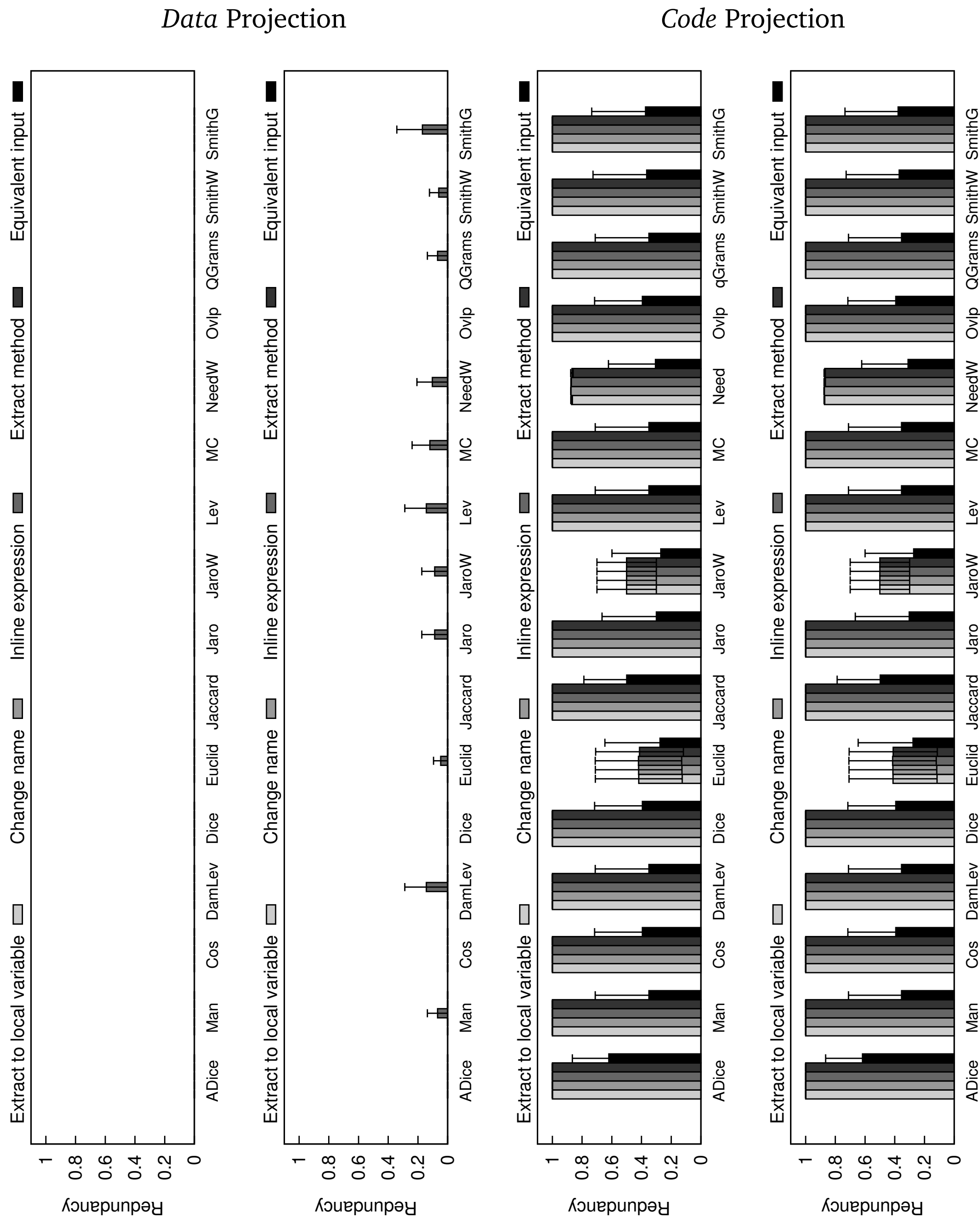


Figure A.4. Measurements after refactoring: merge sort and quick sort

A.1.3 Ground-truth Benchmark Results

Data Projection

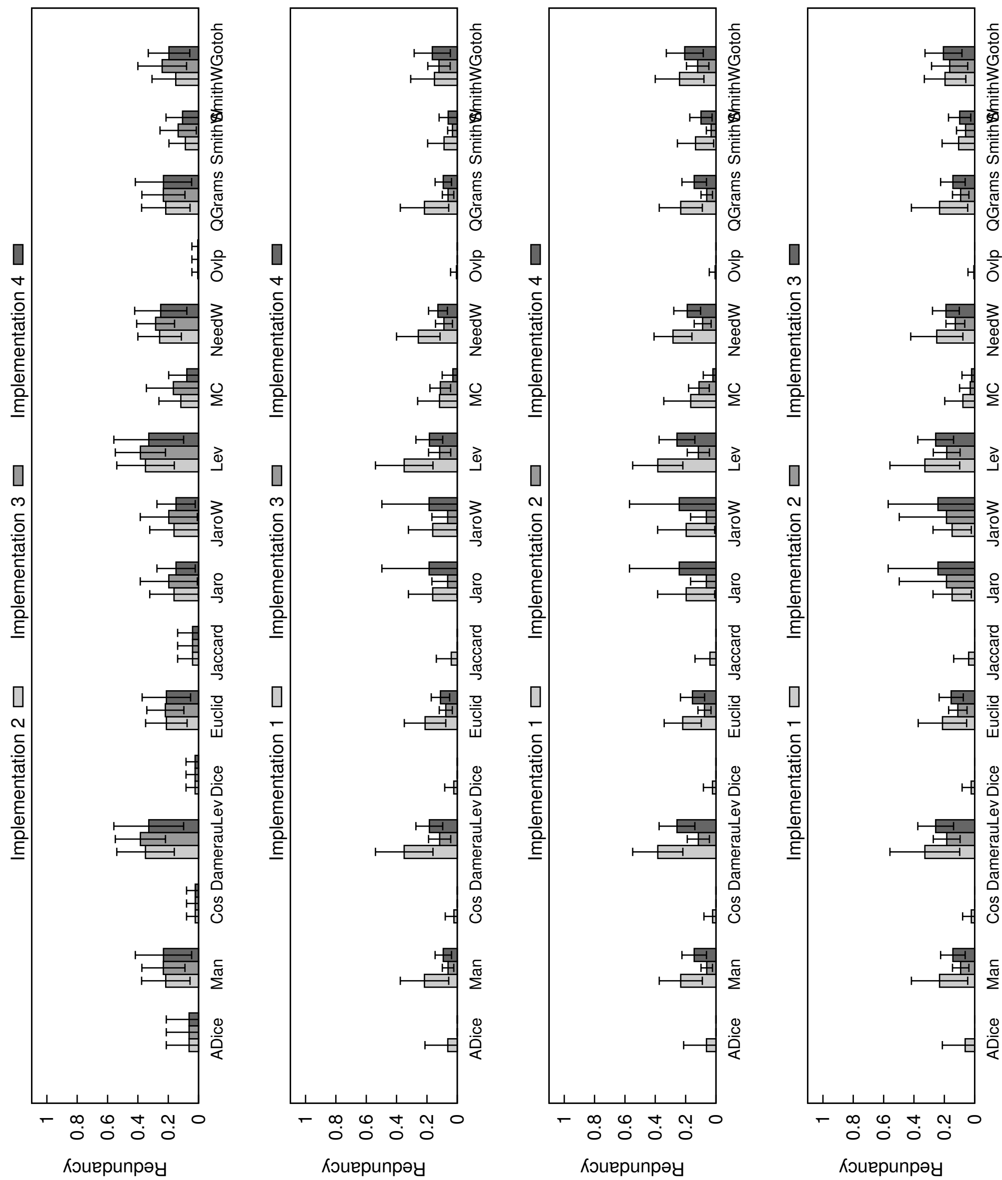


Figure A.5. Ground-truth binary algorithm data projections

Code Projection

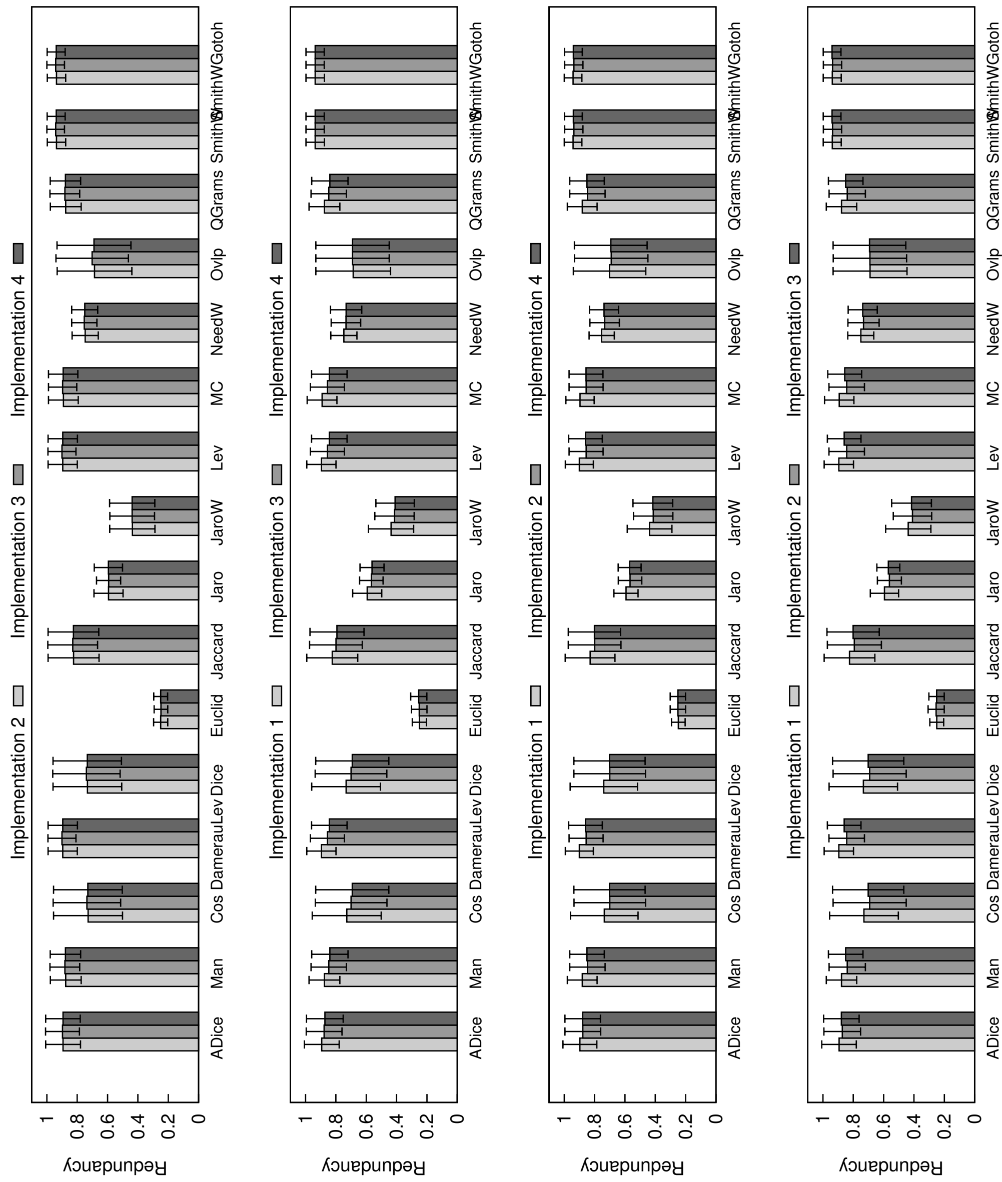


Figure A.6. Ground-truth binary algorithm code projections

Data Projection

Figure A.7. Ground-truth linear algorithm data projections

Code Projection

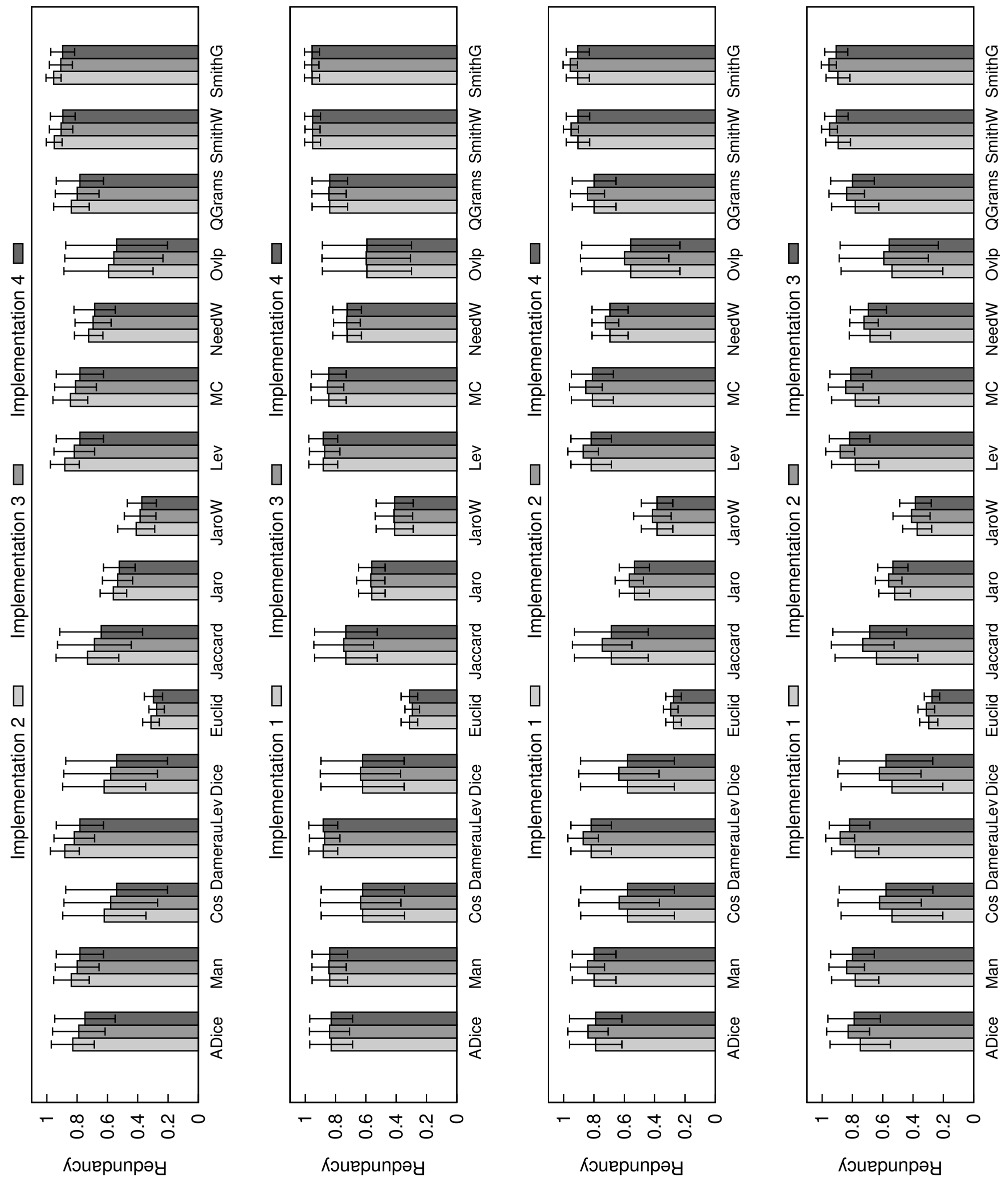


Figure A.8. Ground-truth linear algorithm code projections

Data Projection

Figure A.9. Ground-truth linear implementation vs every binary implementation, data projections

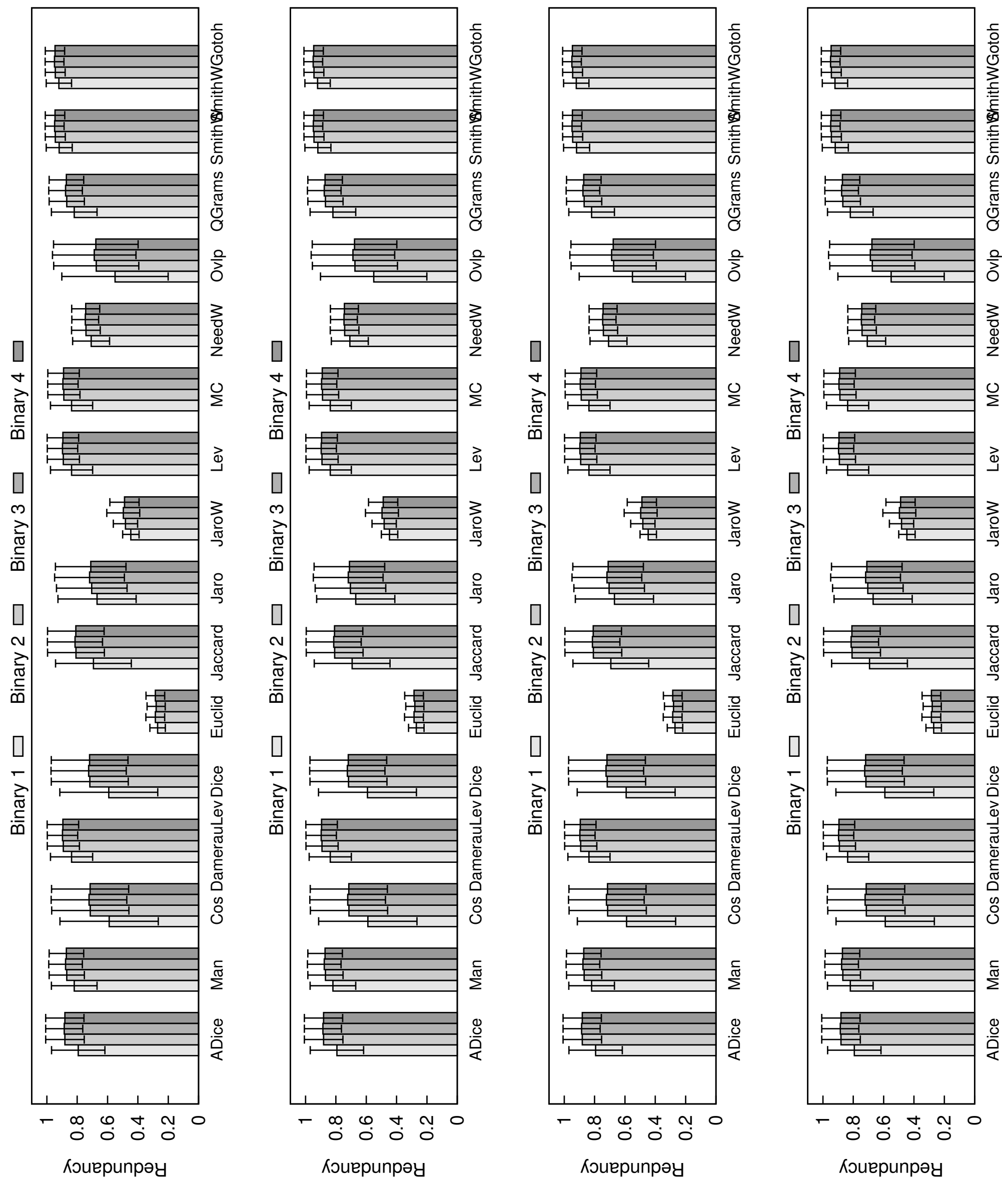


Figure A.10. Ground-truth linear implementation vs every binary implementation, code projections

Figure A.11. Ground-truth binary implementation vs every linear implementation, data projections

Code Projection

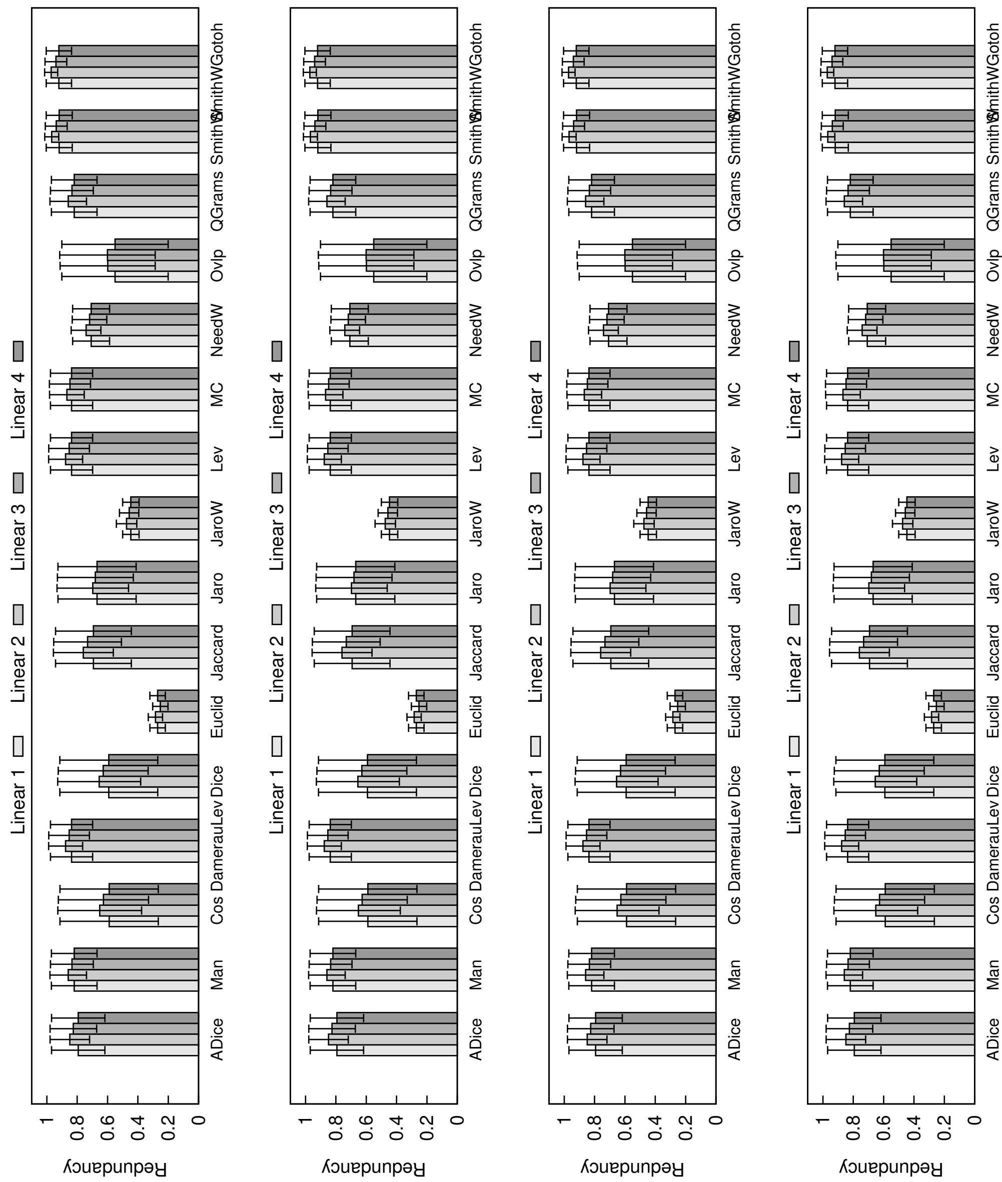


Figure A.12. Ground-truth binary implementation vs every linear implementation, code projections

Data Projection

Figure A.13. Ground-truth bubble sort algorithm data projections

Data Projection

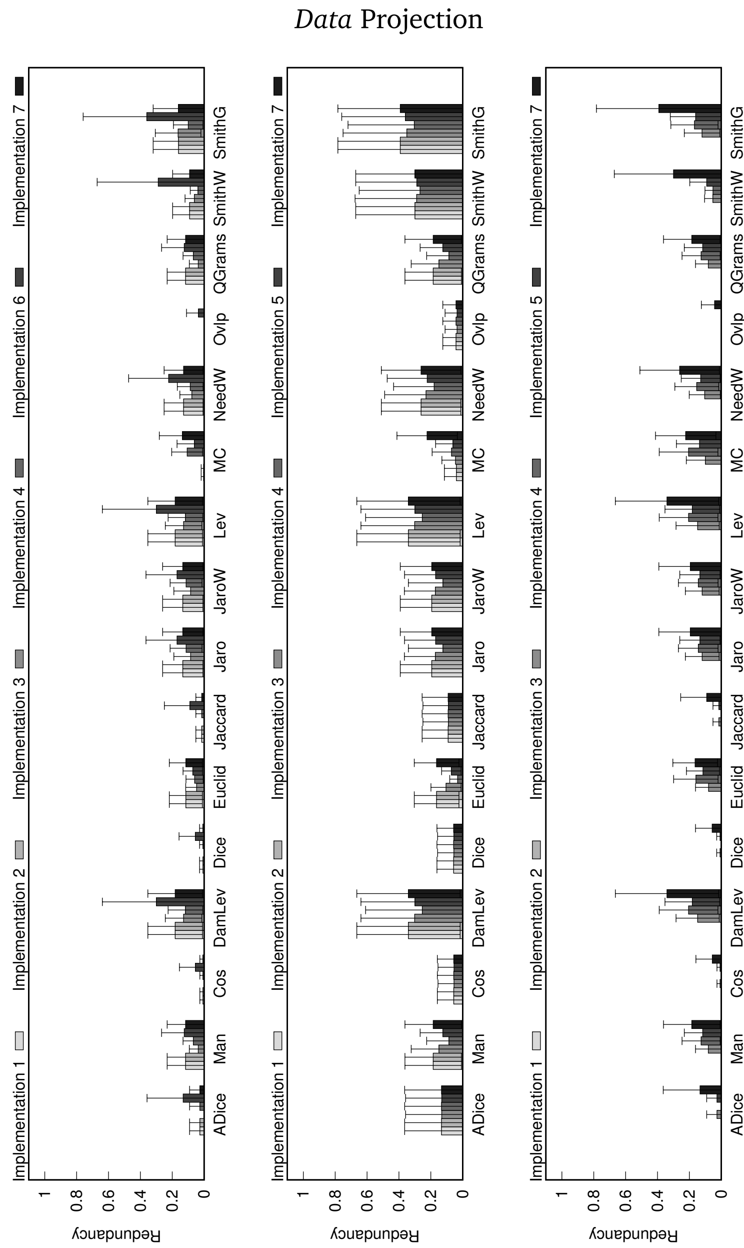


Figure A.14. Ground-truth bubble sort algorithm data projections (cont.)

Code Projection

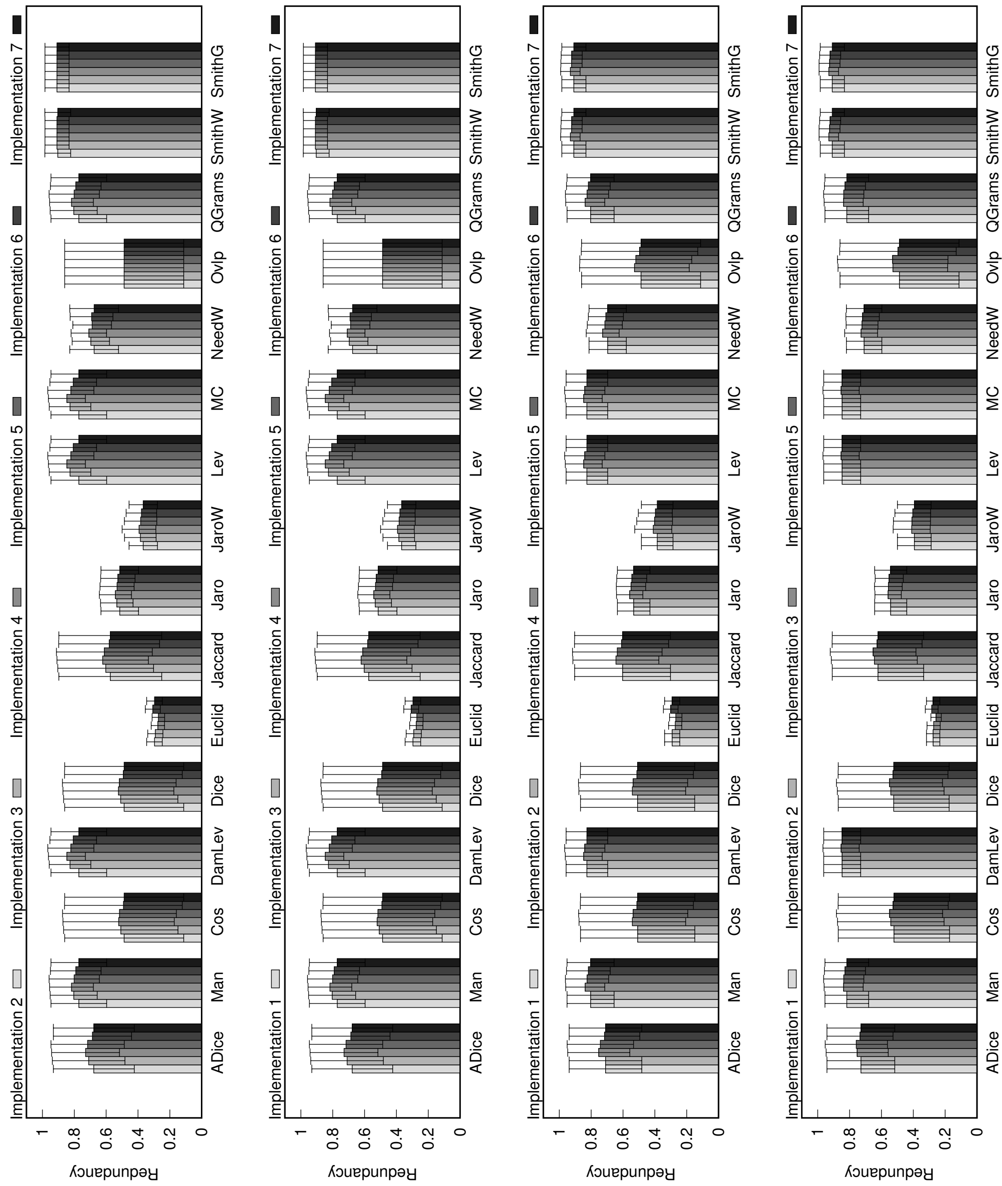


Figure A.15. Ground-truth bubble sort algorithm code projections

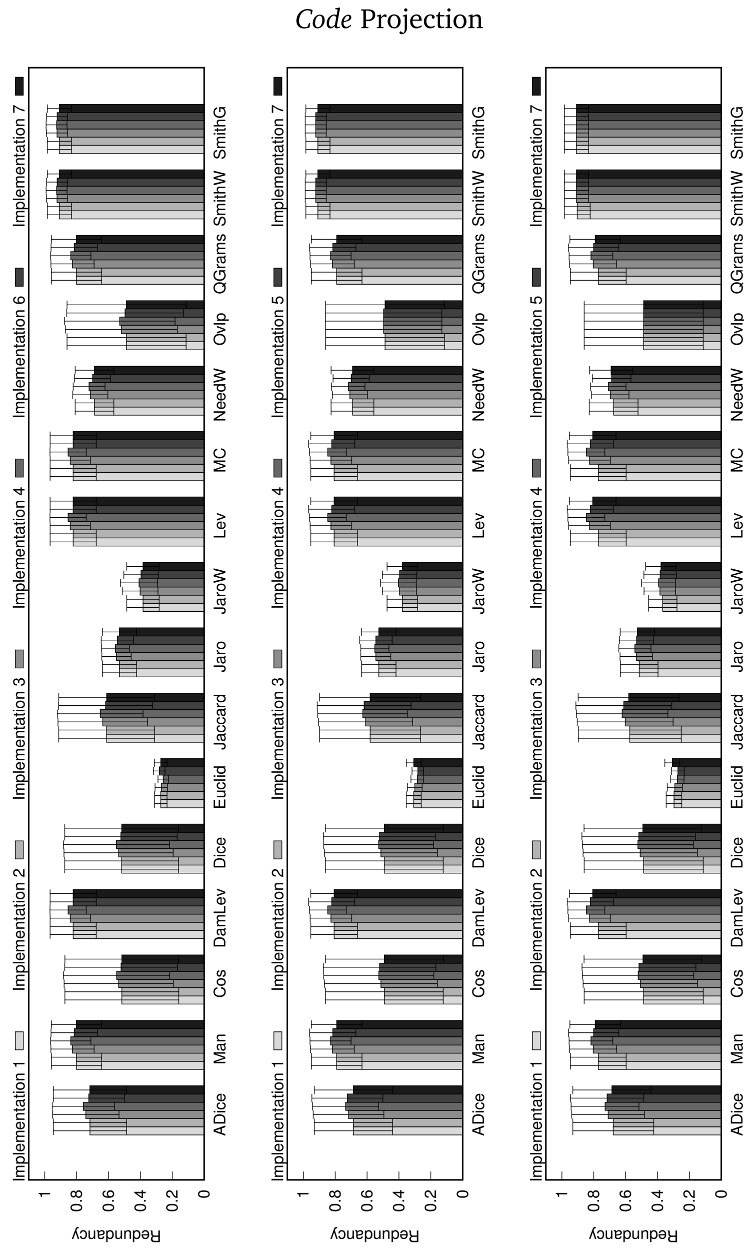


Figure A.16. Ground-truth bubble sort algorithm code projections (cont.)

Data Projection

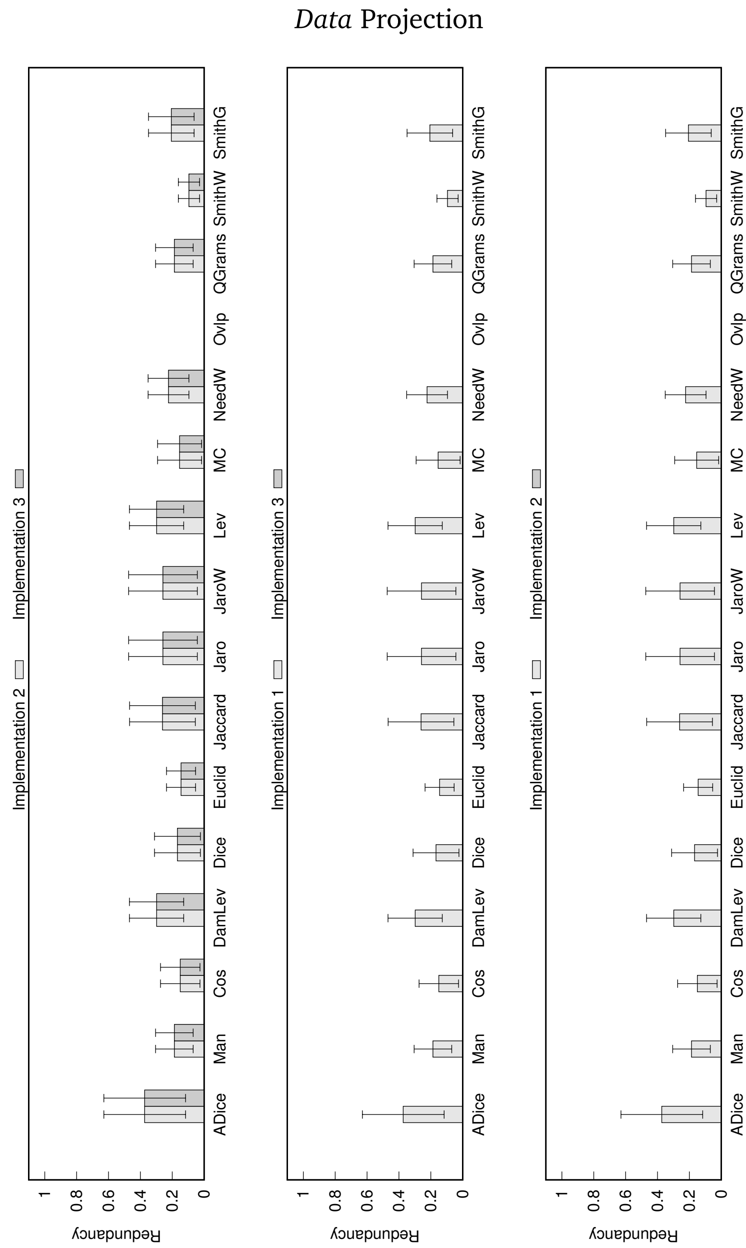


Figure A.17. Ground-truth insertion sort algorithm data projections

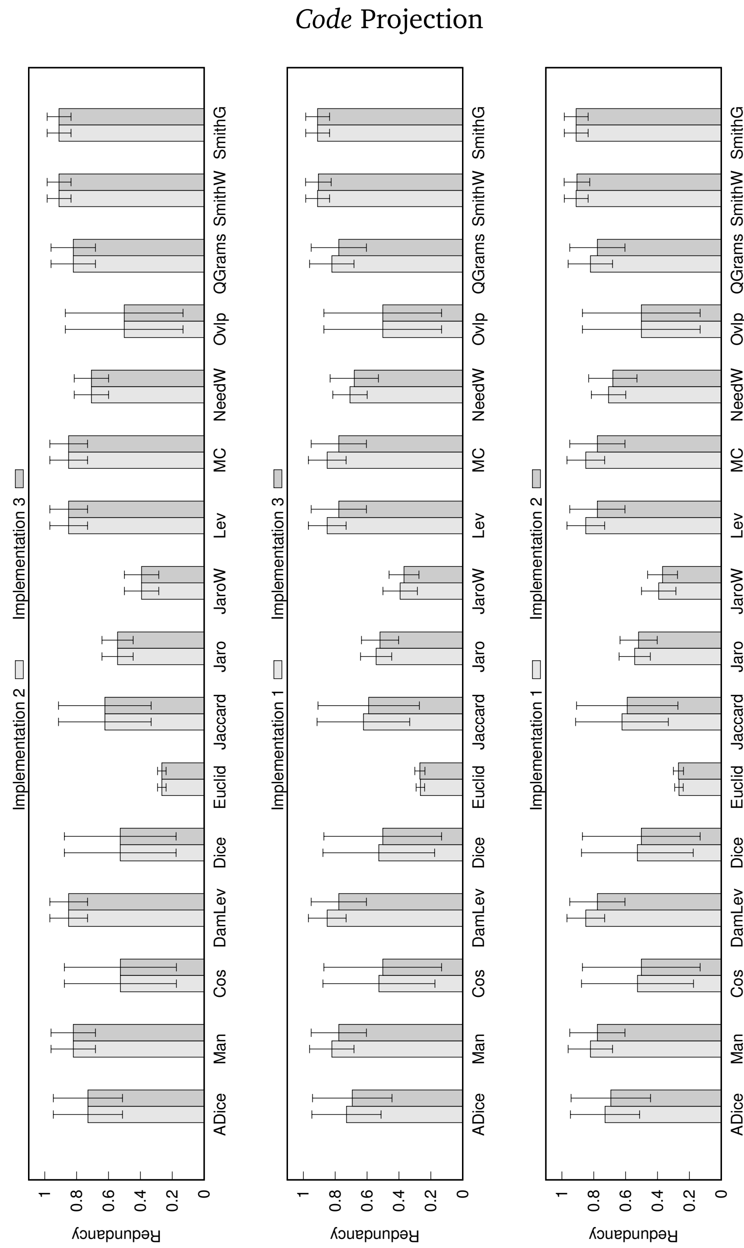


Figure A.18. Ground-truth insertion sort algorithm code projections

Data Projection

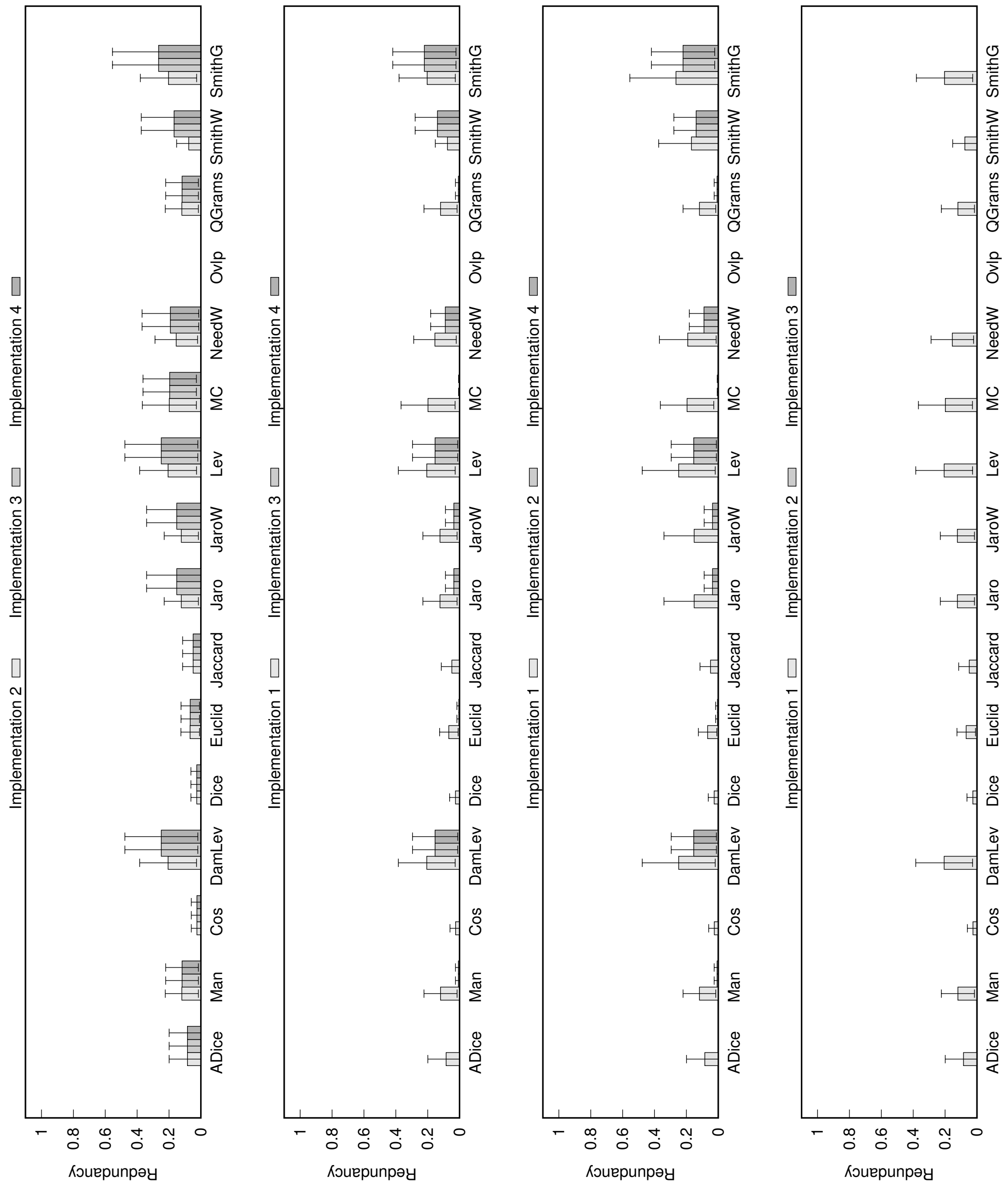


Figure A.19. Ground-truth merge sort algorithm data projections

Code Projection

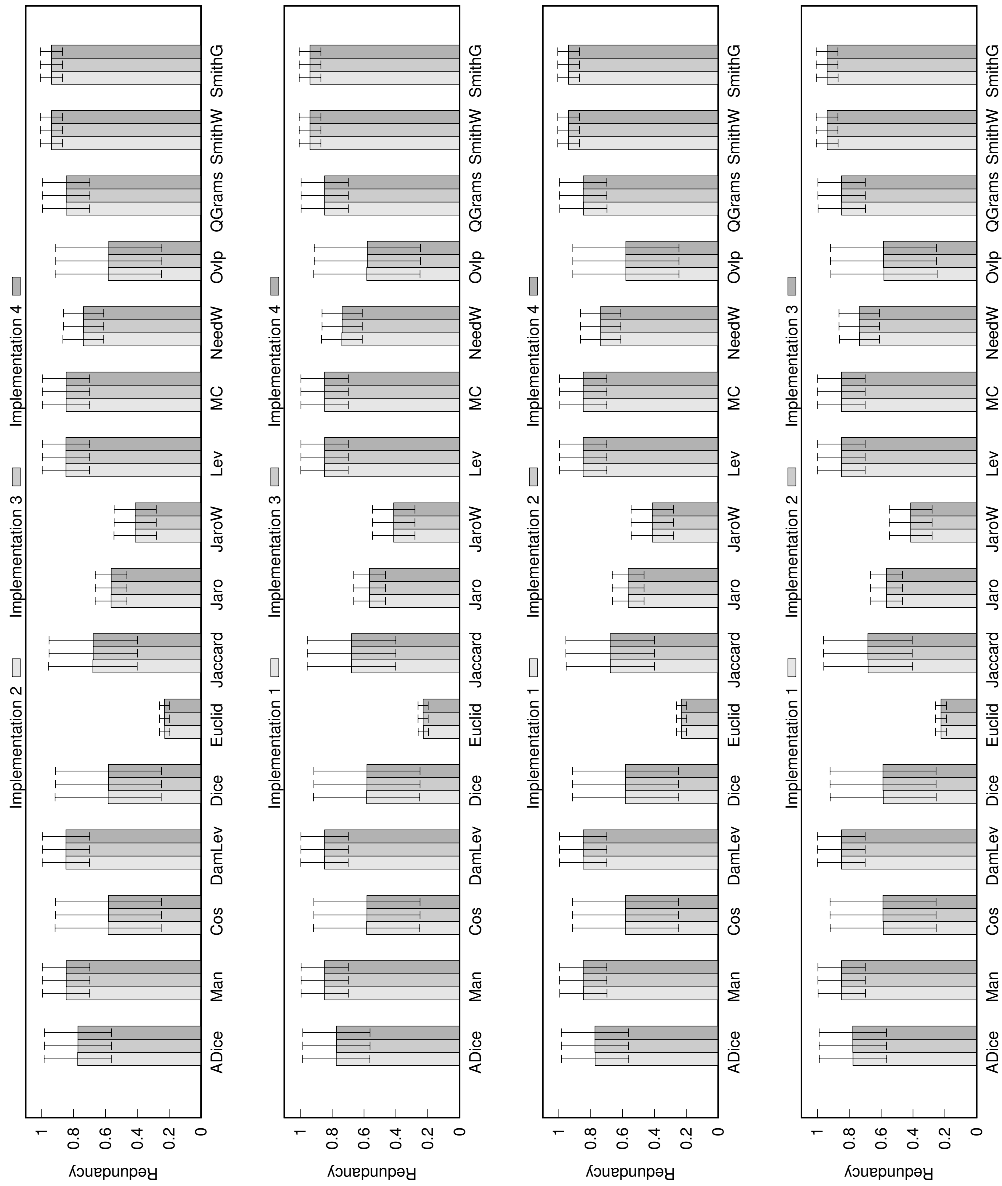


Figure A.20. Ground-truth merge sort algorithm code projections

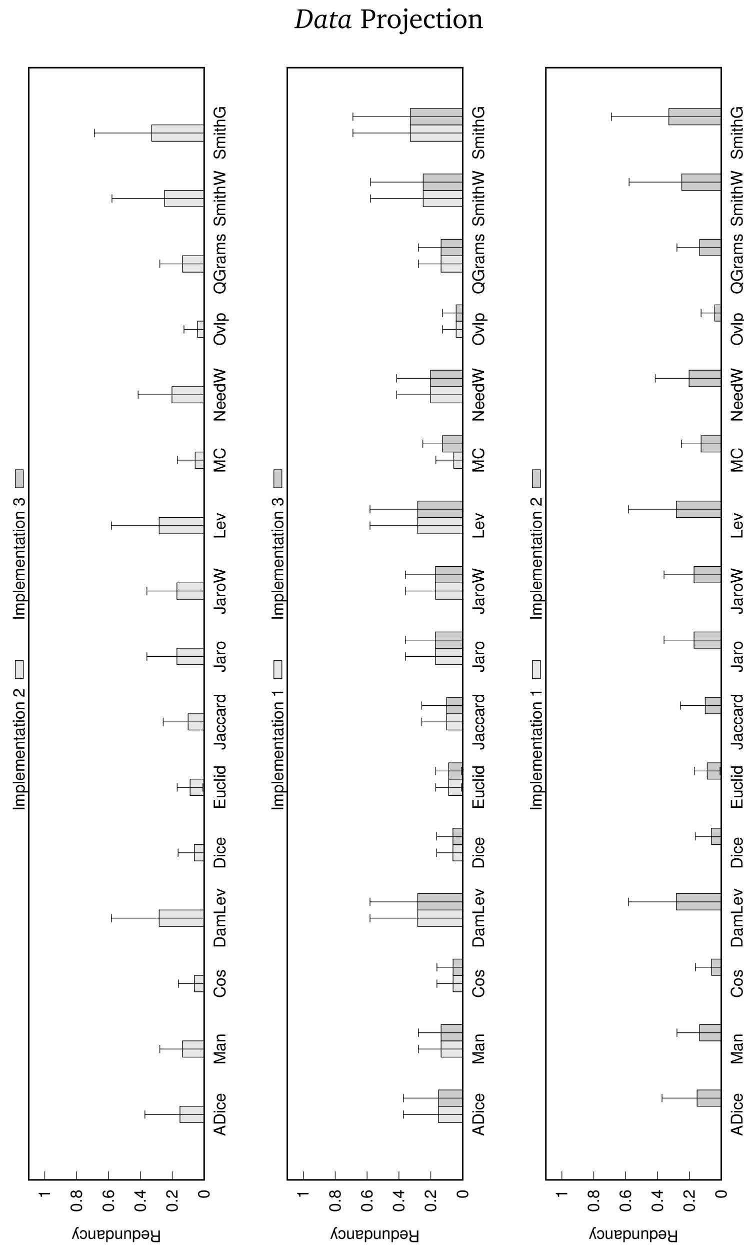


Figure A.21. Ground-truth quick sort algorithm data projections

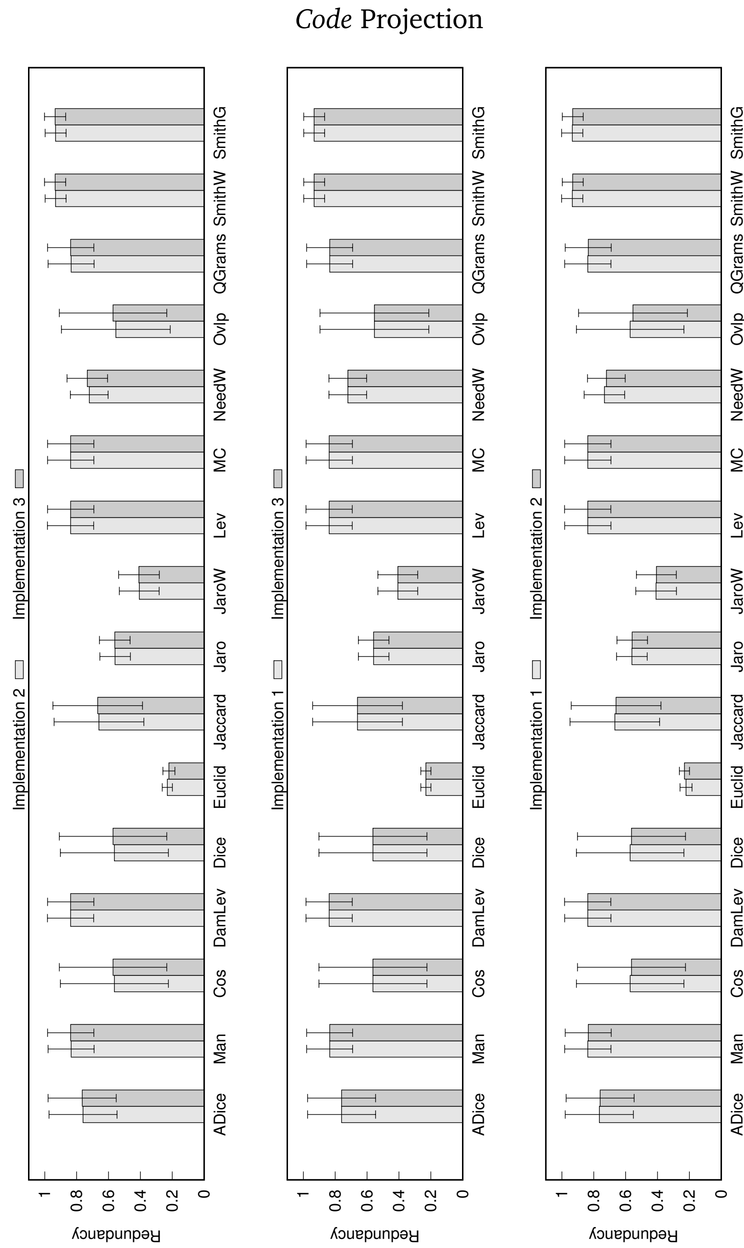


Figure A.22. Ground-truth quick sort algorithm code projections

Data Projection

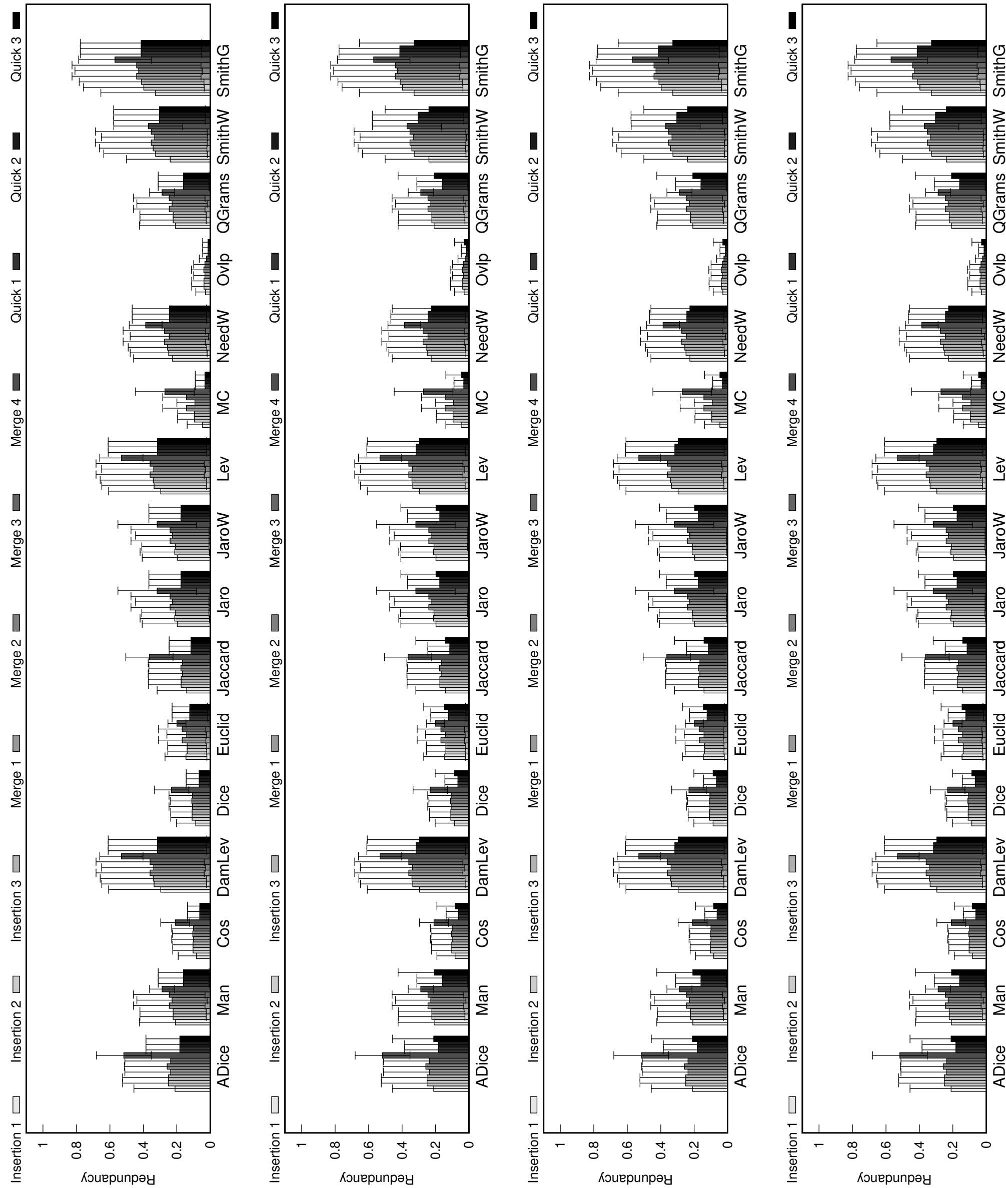


Figure A.23. Ground-truth bubble sort implementation vs every other sorting algorithm implementation, data projections

Data Projection

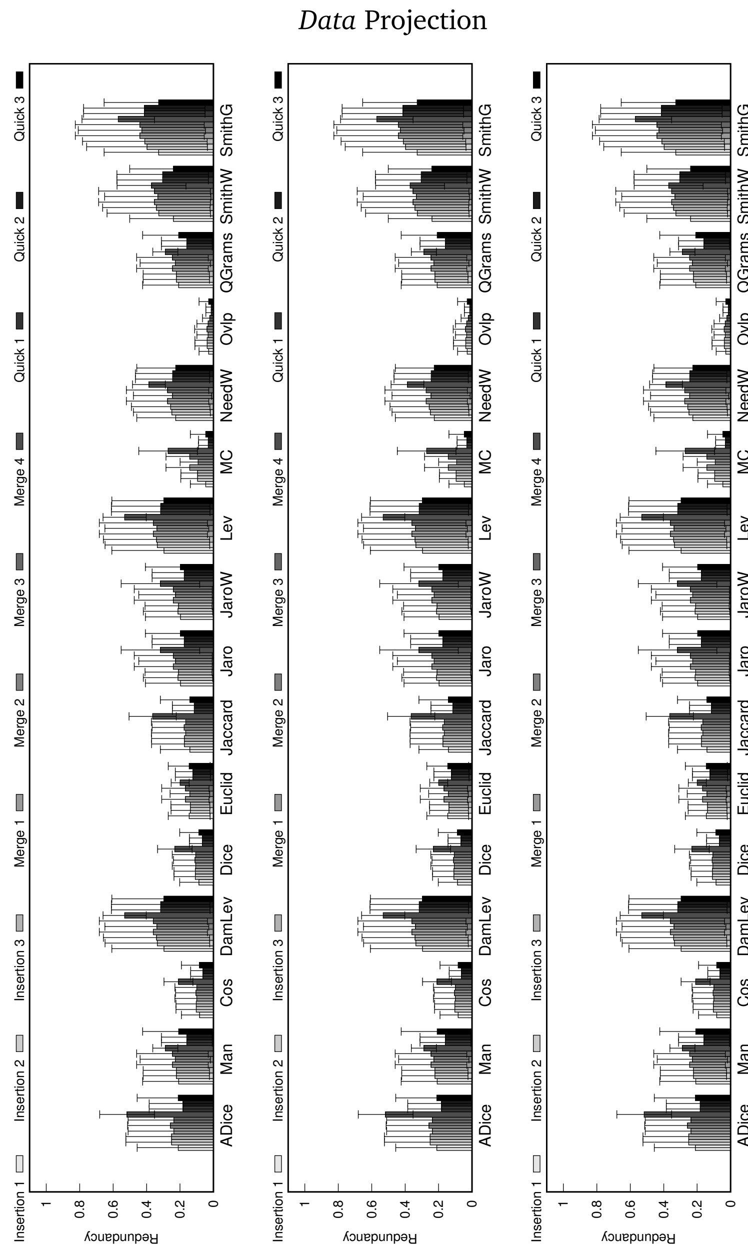


Figure A.24. Ground-truth bubble sort implementation vs every other sorting algorithm implementation, data projections (cont.)

Code Projection

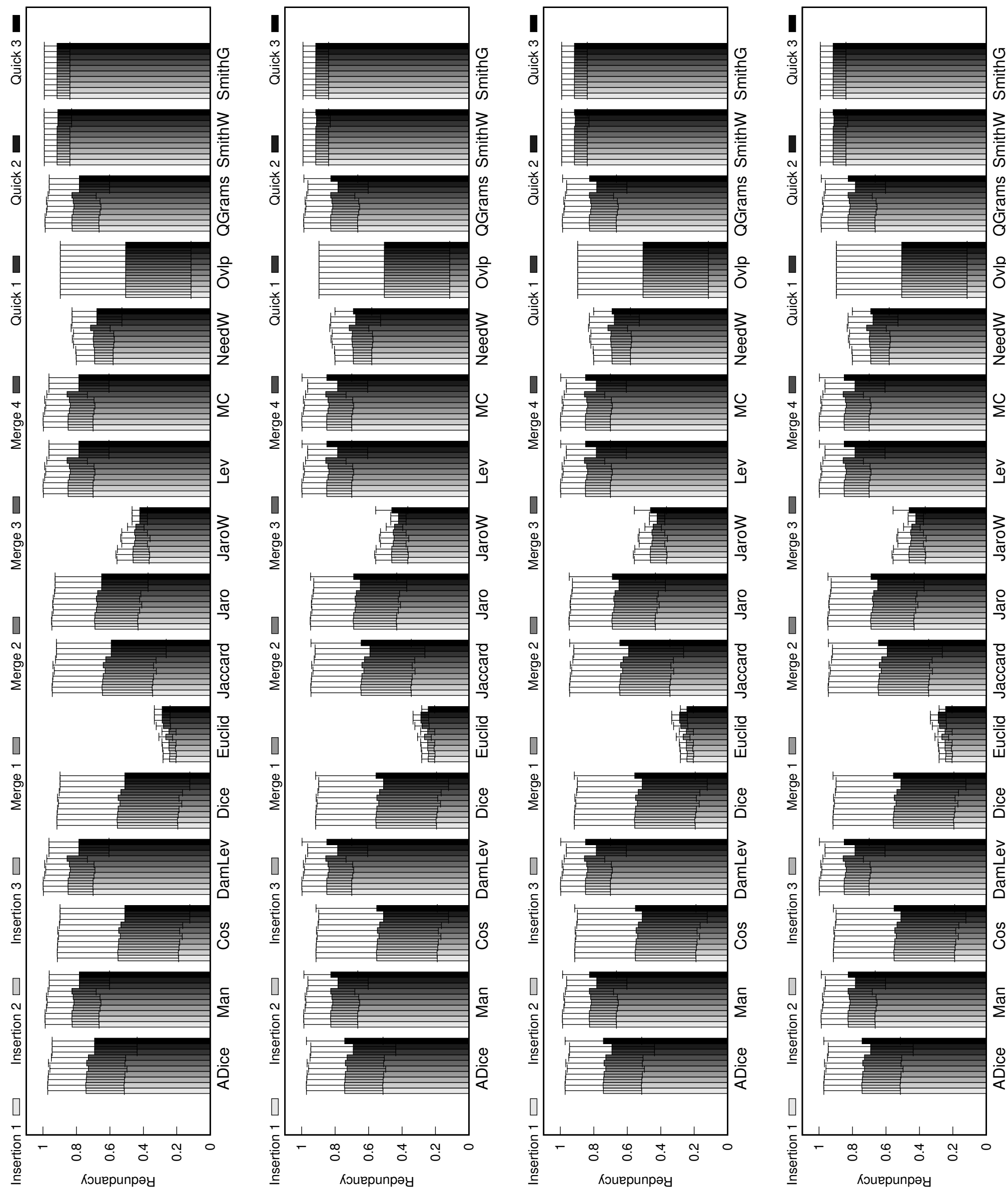


Figure A.25. Ground-truth bubble sort implementation vs every other sorting algorithm implementation, code projections

Code Projection

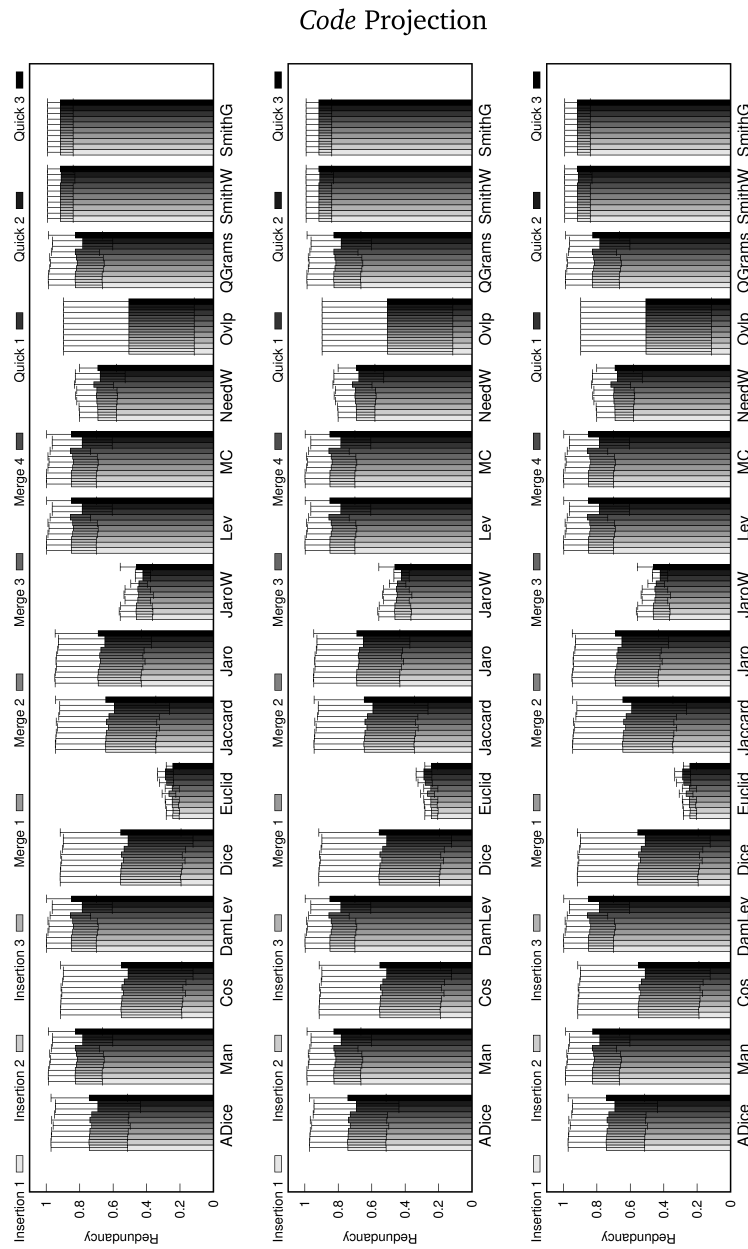


Figure A.26. Ground-truth bubble sort implementation vs every other sorting algorithm implementation, code projections (cont.)

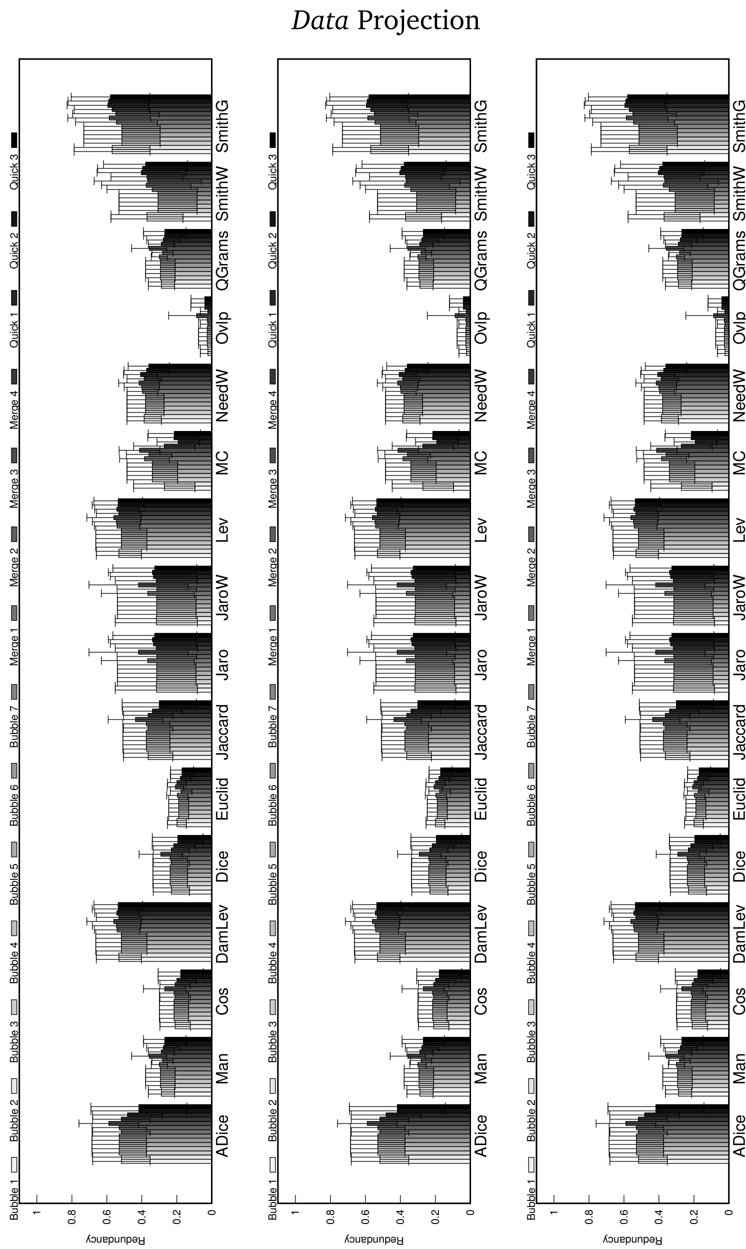


Figure A.27. Ground-truth insertion sort implementation vs every other sorting algorithm implementation, data projections

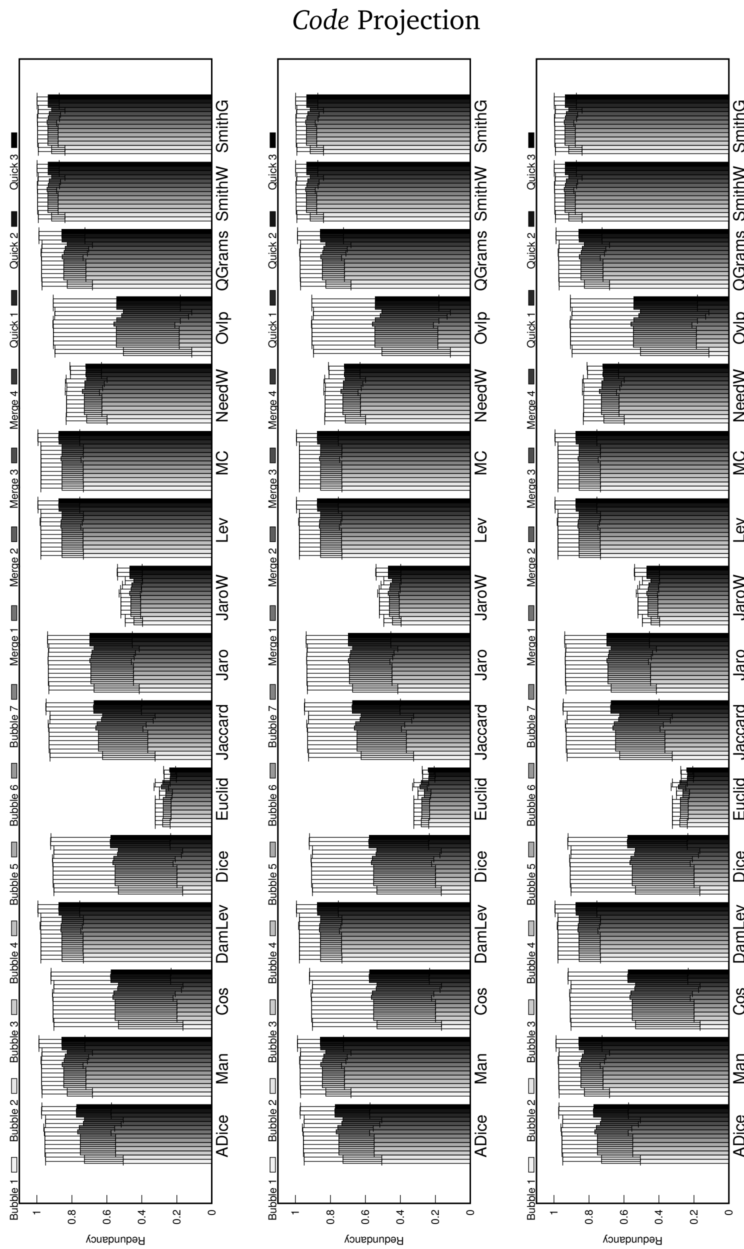


Figure A.28. Ground-truth insertion sort implementation vs every other sorting algorithm implementation, code projections

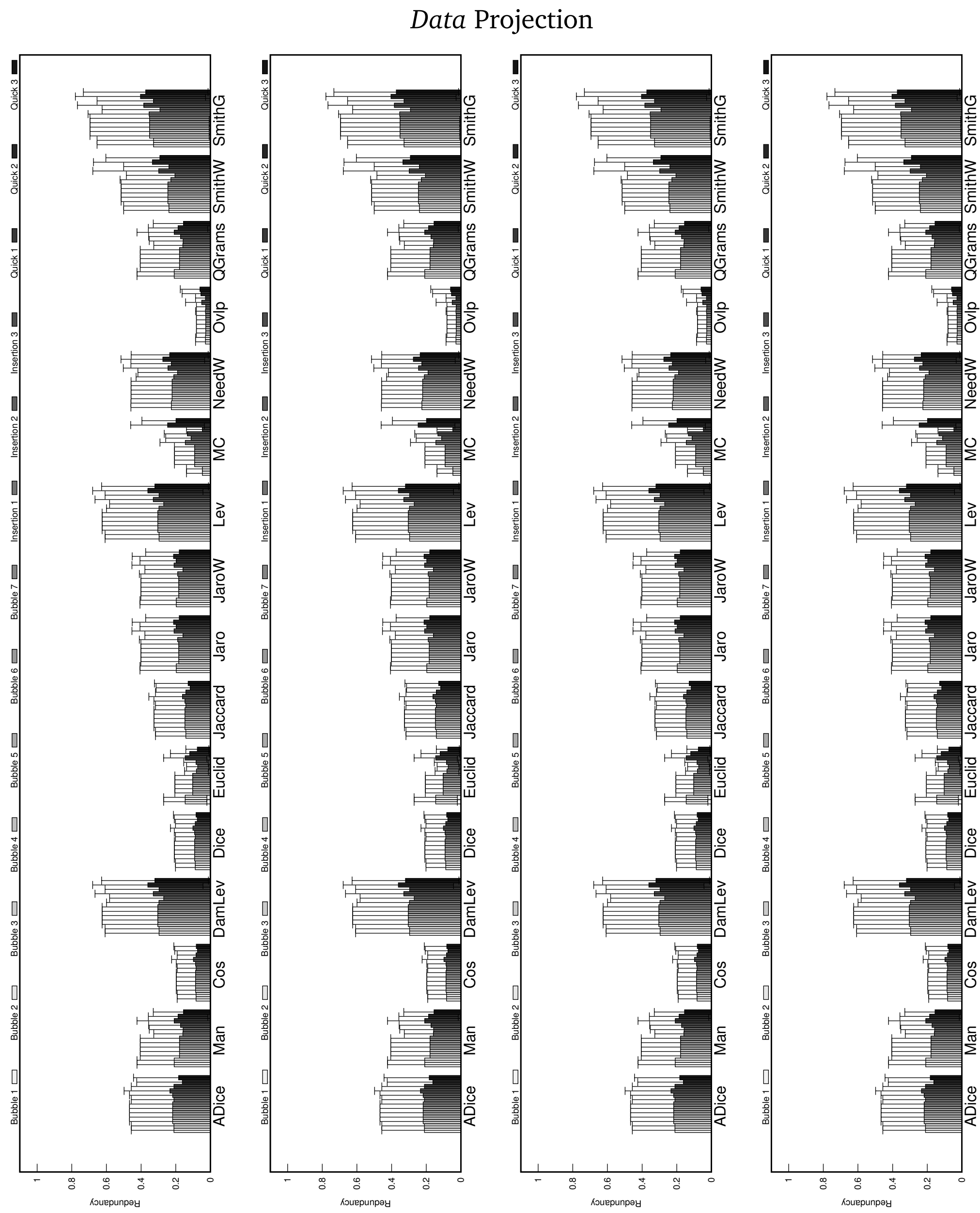


Figure A.29. Ground-truth merge sort implementation vs every other sorting algorithm implementation, data projections

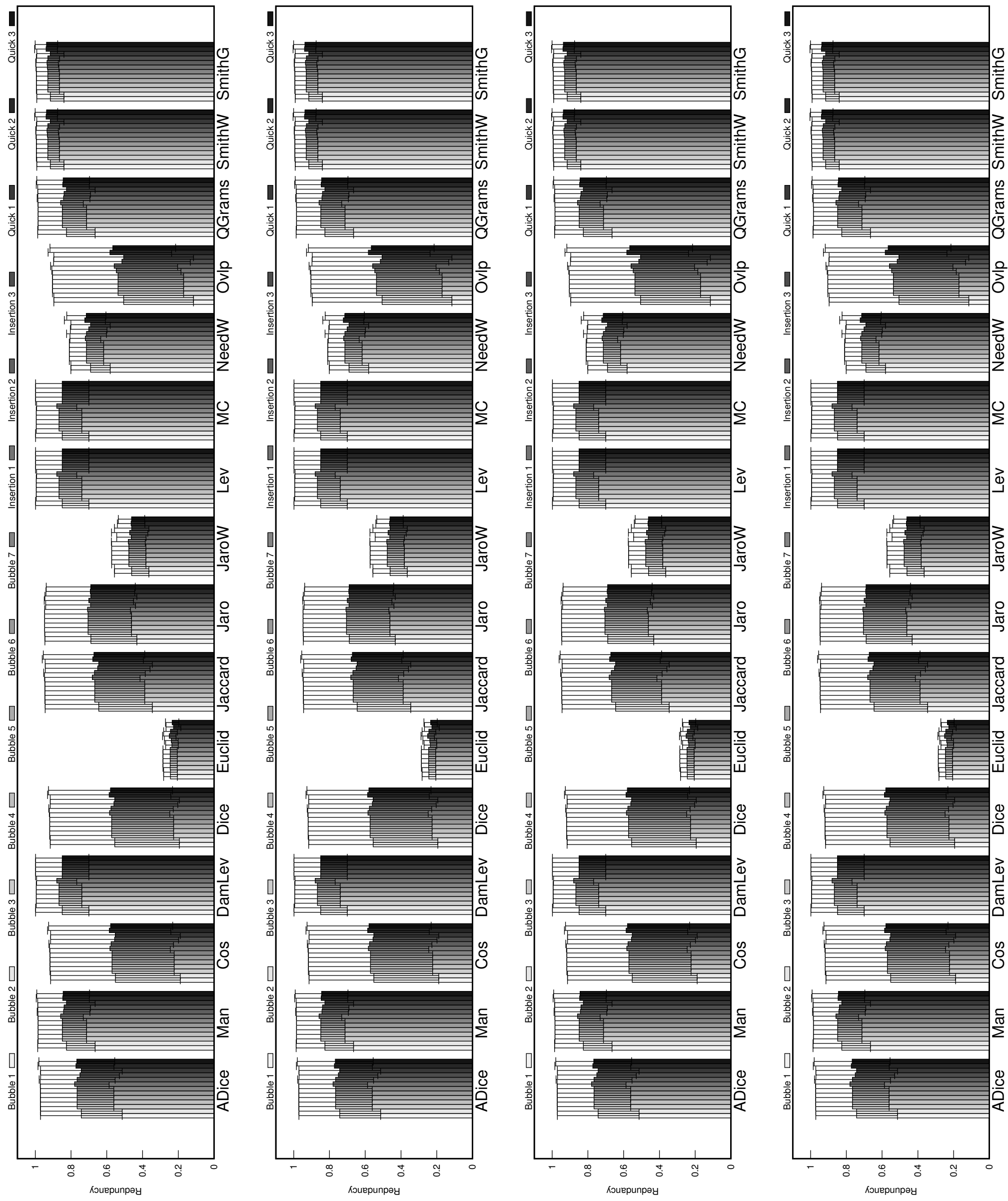


Figure A.30. Ground-truth merge sort implementation vs every other sorting algorithm implementation, code projections

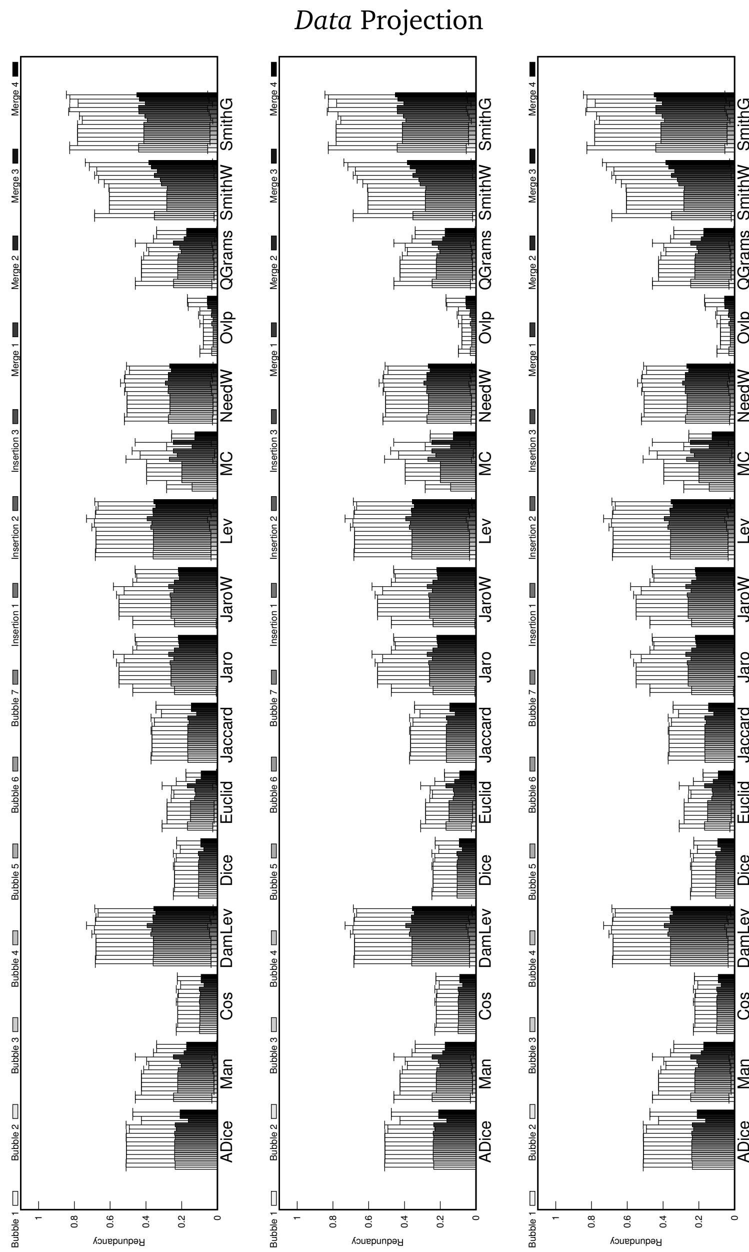


Figure A.31. Ground-truth quick sort implementation vs every other sorting algorithm implementation, data projections

Code Projection

Figure A.32. Ground-truth quick sort implementation vs every other sorting algorithm implementation, code projections

A.2 Benchmark Implementations

A.2.1 Binary search

```
1  private int array[];
2
3  public int search(int key) {
4    return search(key, 0, array.length);
5  }
6
7  private int search(int key, int lo, int hi) {
8   if (hi <= lo) {
9      return -1;
10  }
11  int mid = lo + (hi - lo) / 2;
12  if ( array[mid] > key) {
13    return search(key, lo, mid);
14  }
15  else if (array[mid] < key) {
16    return search(key, mid + 1, hi);
17  }
18  else {
19    return mid;
20  }
21 }
```

Figure A.33. Implementation 1 of the binary search algorithm

```
1  private int array[];
2
3  public int search(int key) {
4    return search(key, 0, array.length);
5  }
6
7  private int search(int key, int lowerbound, int upperbound) {
8    int position;
9    position = (lowerbound + upperbound) / 2;
10   while ((array[position] != key) && (lowerbound <= upperbound)) {
11     if (array[position] > key) {
12       upperbound = position - 1;
13     } else {
14       lowerbound = position + 1;
15     }
16     position = (lowerbound + upperbound) / 2;
17   }
18   if (lowerbound <= upperbound) {
19     return position;
20   } else
21   return -1;
22 }
```

Figure A.34. Implementation 2 of the binary search algorithm

```
1  private int array[];
2
3  public int search(int key) {
4    return search(key, 0, array.length);
5  }
6
7  private int search(int key, int low, int high) {
8    while (high >= low) {
9      int middle = (low + high) / 2;
10     if (array[middle] == key) {
11       return middle;
12     }
13     if (array[middle] < key) {
14       low = middle + 1;
15     }
16     if (array[middle] > key) {
17        high = middle - 1;
18     }
19   }
20   return -1;
21 }
```

Figure A.35. Implementation 3 of the binary search algorithm

```
1  private int array[];
2
3  public int search(int key) {
4    return search(key, 0, array.length);
5  }
6
7  private int search(int key, int lo, int hi) {
8    int middle = (lo + hi) / 2;
9    while (lo <= hi) {
10     if (array[middle] < key)
11       lo = middle + 1;
12     else if (array[middle] == key)
13       return middle;
14     else
15       hi = middle - 1;
16
17     middle = (lo + hi) / 2;
18   }
19   return -1;
20 }
```

Figure A.36. Implementation 4 of the binary search algorithm

A.2.2 Linear search

```
1  private int array[];
2
3  public LinearSearch(int array[]) {
4    this.array = array;
5  }
6
7  public int search(int key) {
8    for (int i = 0; i < array.length; i++) {
9      if (key == array[i]) {
10       return i;
11     }
12   }
13   return -1;
14 }
```

Figure A.37. Implementation 1 of the linear search algorithm

```
1  private int array[];
2
3  public LinearSearch2(int array[]) {
4    this.array = array;
5  }
6
7  public int search(int toSearch) {
8    int foundIndex = 0;
9    for (int i = 0; i < array.length; i++) {
10     if (array[i] == toSearch) {
11       foundIndex = i;
12     }
13   }
14   return foundIndex;
15 }
```

Figure A.38. Implementation 2 of the linear search algorithm

```
1  private int array[];
2
3  public LinearSearch3(int array[]) {
4    this.array = array;
5  }
6
7  public int search(int key) {
8    int index = 0;
9    while(index < array.length) {
10     if(array[index] == key) {
11       return index;
12     }
13     if(array[index] < key) {
14       return -1;
15     }
16     index++;
17   }
18   return -1;
```

Figure A.39. Implementation 3 of the linear search algorithm

```
1  private int array[];
2
3  public LinearSearch2(int array[]) {
4    this.array = array;
5  }
6
7  public int search(int toSearch) {
8    int foundIndex = 0;
9    for (int i = 0; i < array.length; i++) {
10     if (array[i] == toSearch) {
11       foundIndex = i;
12     }
13   }
14   return foundIndex;
15 }
```

Figure A.40. Implementation 4 of the linear search algorithm

A.2.3 Bubble Sort

```
1  private int array[];
2
3  public BubbleSort(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    for (int i = 1; i < array.length; i++) {
9      for (int j = 0; j < array.length - i; j++) {
10       if (array[j] > array[j + 1]) {
11         int temp = array[j];
12         array[j] = array[j + 1];
13         array[j + 1] = temp;
14       }
15     }
16   }
17   return array;
18 }
```

Figure A.41. Implementation 1 of the bubble sort algorithm

```
1  public int array[];
2
3  public BubbleSort2(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    for (int i = 0; i < array.length - 1; i++) {
9      for (int j = 1; j < array.length - i; j++) {
10       if (array[j - 1] > array[j]) {
11         int tmp = array[j - 1];
12         array[j - 1] = array[j];
13         array[j] = tmp;
14       }
15     }
16   }
17   return array;
18 }
```

Figure A.42. Implementation 2 of the bubble sort algorithm

```
1  private int array[];
2
3  public BubbleSort3(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    boolean doMore = true;
9    while (doMore) {
10     doMore = false; // assume this is last pass over array
11     for (int i = 0; i < array.length - 1; i++) {
12       if (array[i] > array[i + 1]) {
13         // exchange elements
14         int temp = array[i];
15         array[i] = array[i + 1];
16         array[i + 1] = temp;
17         doMore = true; // after an exchange, must look again
18       }
19     }
20   }
21   return array;
22 }
```

Figure A.43. Implementation 3 of the bubble sort algorithm

```
1  private int array[];
2
3  public BubbleSort4(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    int n = array.length;
9    boolean doMore = true;
10   while (doMore) {
11     n--;
12     doMore = false; // assume this is our last pass over the array
13     for (int i = 0; i < n; i++) {
14       if (array[i] > array[i + 1]) {
15         // exchange elements
16         int temp = array[i];
17         array[i] = array[i + 1];
18         array[i + 1] = temp;
19         doMore = true; // after an exchange, must look again
20       }
21     }
22   }
23   return array;
24 }
```

Figure A.44. Implementation 4 of the bubble sort algorithm

```
1  private int array[];
2
3  public BubbleSort5(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    int newLowest = 0; // index of first comparison
9    int newHighest = array.length - 1; // index of last comparison
10   while (newLowest < newHighest) {
11     int highest = newHighest;
12     int lowest = newLowest;
13     newLowest = array.length; // start higher than any legal index
14     for (int i = lowest; i < highest; i++) {
15       if (array[i] > array[i + 1]) {
16         // exchange elements
17         int temp = array[i];
18         array[i] = array[i + 1];
19         array[i + 1] = temp;
20         if (i < newLowest) {
21           newLowest = i - 1;
22           if (newLowest < 0) {
23             newLowest = 0;
24           }
25         } else if (i > newHighest) {
26           newHighest = i + 1;
27         }
28       }
29     }
30   }
31   return array;
32 }
```

Figure A.45. Implementation 5 of the bubble sort algorithm

```
1  public int array[];
2
3  public BubbleSort6(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    for (int i = array.length - 1; i >= 0; i--) {
9      for (int j = i - 1; j >= 0; j--) {
10       if (array[j] > array[i]) {
11         int tmp = array[i];
12         array[i] = array[j];
13         array[j] = tmp;
14       }
15     }
16   }
17   return array;
18 }
```

Figure A.46. Implementation 6 of the bubble sort algorithm

```
1  private int array[];
2
3  public BubbleSort7(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    for (int i = 0; i < array.length - 1; i++) {
9      for (int j = 1; j < array.length - i; j++) {
10       if (array[j - 1] > array[j]) {
11         int temp = array[j - 1];
12         array[j - 1] = array[j];
13         array[j] = temp;
14       }
15     }
16   }
17   return array;
18 }
```

Figure A.47. Implementation 7 of the bubble sort algorithm

A.2.4 Insertion Sort

```
1  private int array[];
2
3  public InsertionSort(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    for (int i = 0; i < array.length; i++) {
9      int value = array[i];
10     int j = i - 1;
11     while (j >= 0) {
12       if (array[j] <= value) {
13         break;
14       }
15       array[j + 1] = array[j];
16       j--;
17     }
18     array[j + 1] = value;
19   }
20   return array;
21 }
```

Figure A.48. Implementation 1 of the insertion sort algorithm

```
1  private int array[];
2
3  public InsertionSort2(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    for (int i = 1; i < array.length; i++) {
9      int value = array[i];
10     int j = i;
11     while ((j > 0) && (array[j - 1] > value)) {
12       array[j] = array[j - 1];
13       j--;
14     }
15     array[j] = value;
16   }
17   return array;
18 }
```

Figure A.49. Implementation 2 of the insertion sort algorithm

```
1  public InsertionSort3(int array[]) {
2    this.array = array;
3  }
4
5  public int[] sort() {
6    for (int i = 1; i < array.length; i++) {
7      int value = array[i];
8      int j = i;
9      while ((j > 0) && (array[j - 1] > value)) {
10       array[j] = array[j - 1];
11       j--;
12     }
13     array[j] = value;
14   }
15   return array;
16 }
```

Figure A.50. Implementation 3 of the insertion sort algorithm

A.2.5 Merge Sort

```
1  private int[] array;
2
3  public MergeSort(int[] array) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    sort(0, array.length - 1);
9    return array;
10 }
11
12 private void sort(int low, int high) {
13   if (low < high) {
14     int middle = low + (high - low) / 2;
15     sort(low, middle);
16     sort(middle + 1, high);
17     merge(low, middle, high);
18   }
19 }
20
21 private void merge(int low, int middle, int high) {
22   int[] helper = new int[array.length];
23   for (int i = low; i <= high; i++) {
24     helper[i] = array[i];
25   }
26   int i = low;
27   int j = middle + 1;
28   int k = low;
29   while (i <= middle && j <= high) {
30     if (helper[i] <= helper[j]) {
31       array[k] = helper[i];
32       i++;
33     } else {
34       array[k] = helper[j];
35       j++;
36     }
37     k++;
38   }
39   while (i <= middle) {
40     array[k] = helper[i];
41     k++;
42     i++;
43   }
44 }
```

Figure A.51. Implementation 1 of the merge sort algorithm

```
1  private int array[];
2
3  public MergeSort2(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    sort(0, array.length - 1);
9    return array;
10 }
11
12 private void sort(int low, int high) {
13   if (low < high) {
14     int mid = (low + high) / 2;
15     sort(low, mid);
16     sort(mid + 1, high);
17     merge(low, mid, high);
18   }
19 }
20
21 private void merge(int low, int mid, int high) {
22   int helper[] = new int[array.length];
23   int h = low, i = low, j = mid + 1, k;
24   while ((h <= mid) && (j <= high)) {
25     if (array[h] <= array[j]) {
26       helper[i] = array[h];
27       h++;
28     } else {
29       helper[i] = array[j];
30       j++;
31     }
32     i++;
33   }
34   if (h > mid) {
35     for (k = j; k <= high; k++) {
36       helper[i] = array[k];
37       i++;
38     }
39   }
40   else {
41     for (k = h; k <= mid; k++) {
42       helper[i] = array[k];
43       i++;
44     }
45   }
46   for (k = low; k <= high; k++) {
47     array[k] = helper[k];
48   }
49 }
```

Figure A.52. Implementation 2 of the merge sort algorithm

```
1  public int array[];
2
3  public MergeSort3(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    sort(0, array.length - 1);
9    return array;
10 }
11
12 private void sort(int low, int high) {
13   if (low < high) {
14     int mid = (low + high) / 2;
15     sort(low, mid);
16     sort(mid + 1, high);
17     merge(low, mid + 1, high);
18   }
19 }
20
21 private void merge(int low, int mid, int high) {
22   int[] helper = new int[array.length];
23   int endLeft = mid - 1;
24   int k = low;
25   int numElemen = high - low + 1;
26   while (low <= endLeft && mid <= high) {
27     if ((array[low]) < (array[mid])) {
28       helper[k++] = array[low++];
29     } else {
30       helper[k++] = array[mid++];
31     }
32   }
33   while (low <= endLeft) {
34     helper[k++] = array[low++];
35   }
36   while (mid <= high) {
37     helper[k++] = array[mid++];
38   }
39   for (int i = 0; i < numElemen; i++, high--) {
40     array[high] = helper[high];
41   }
42 }
```

Figure A.53. Implementation 3 of the merge sort algorithm

```
1  public int[] array;
2
3  public MergeSort4(int[] array) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    sort(0, array.length - 1);
9    return array;
10 }
11
12 private void sort(int low, int high) {
13   if (low < high) {
14     int mid = (low + high) / 2;
15     sort(low, mid);
16     sort(mid + 1, high);
17     merge(low, mid, high);
18   }
19 }
20
21 private void merge(int low, int mid, int high) {
22   int[] helper = new int[array.length];
23   int i = low;
24   int j = mid + 1;
25   int k = low;
26   while (i <= mid && j <= high) {
27     if (array[i] <= array[j]) {
28       helper[k] = array[i];
29       i++;
30     } else {
31       helper[k] = array[j];
32       j++;
33     }
34     k++;
35   }
36   while (i <= mid) {
37     helper[k] = array[i];
38     i++;
39     k++;
40   }
41   while (j <= high) {
42     helper[k] = array[j];
43     j++;
44     k++;
45   }
46   for (k = low; k <= high; k++) {
47     array[k] = helper[k];
48   }
49 }
```

Figure A.54. Implementation 4 of the merge sort algorithm

A.2.6 Quick Sort

```
1  private int array[];
2
3  public QuickSort(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    sort(0, array.length - 1);
9    return array;
10 }
11
12 private void sort(int low, int high) {
13   if (low >= high) {
14     return;
15   }
16   int middle = low + (high - low) / 2;
17   int pivot = array[middle];
18   int i = low, j = high;
19   while (i <= j) {
20     while (array[i] < pivot) {
21       i++;
22     }
23     while (array[j] > pivot) {
24       j--;
25     }
26     if (i <= j) {
27       int temp = array[i];
28       array[i] = array[j];
29       array[j] = temp;
30       i++;
31       j--;
32     }
33   }
34   if (low < j) {
35     sort(low, j);
36   }
37   if (high > i) {
38     sort(i, high);
39   }
40 }
```

Figure A.55. Implementation 1 of the quick sort algorithm

```
1  private int array[];
2
3  public QuickSort2(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    sort(0, array.length - 1);
9    return array;
10 }
11
12 private void sort(int low, int high) {
13   if (high <= low || low >= high) {
14     return;
15   } else {
16     int pivot = array[low];
17     int i = low + 1;
18     int tmp;
19     for (int j = low + 1; j <= high; j++) {
20       if (pivot > array[j]) {
21         tmp = array[j];
22         array[j] = array[i];
23         array[i] = tmp;
24         i++;
25       }
26     }
27     array[low] = array[i - 1];
28     array[i - 1] = pivot;
29     sort(low, i - 2);
30     sort(i, high);
31   }
32 }
```

Figure A.56. Implementation 2 of the quick sort algorithm

```
1  private int array[];
2
3  public QuickSort3(int array[]) {
4    this.array = array;
5  }
6
7  public int[] sort() {
8    sort(0, array.length - 1);
9    return array;
10 }
11
12 private void sort(int lowerIndex, int higherIndex) {
13   if (lowerIndex >= higherIndex) {
14     return;
15   }
16   int i = lowerIndex;
17   int j = higherIndex;
18   int pivot = array[lowerIndex + (higherIndex - lowerIndex) / 2];
19   while (i <= j) {
20     while (array[i] < pivot) {
21       i++;
22     }
23     while (array[j] > pivot) {
24       j--;
25     }
26     if (i <= j) {
27       int temp = array[i];
28       array[i] = array[j];
29       array[j] = temp;
30       i++;
31       j--;
32     }
33   }
34   if (lowerIndex < j)
35     sort(lowerIndex, j);
36   if (i < higherIndex)
37     sort(i, higherIndex);
38 }
```

Figure A.57. Implementation 3 of the quick sort algorithm

A.3 Statistical Test Results

Algorithm	Similarity	$\hat{A}_{12}$	p-value
Binary search	ADice	0.69	≤ 0.001
	Man	0.67	≤ 0.001
	Cos	0.69	≤ 0.001
	DamLev	0.66	≤ 0.001
	Dice	0.69	≤ 0.001
	Euclid	0.66	≤ 0.001
	Jaccard	0.69	≤ 0.001
	Jaro	0.67	≤ 0.001
	JaroW	0.67	≤ 0.001
	Lev	0.66	≤ 0.001
	MC	0.69	≤ 0.001
	NeedW	0.66	≤ 0.001
	Ovlp	0.54	≤ 0.001
	QGrams	0.67	≤ 0.001
	SmithW	0.68	≤ 0.001
	SmithG	0.67	≤ 0.001
Linear search	ADice	0.56	≤ 0.001
	Man	0.66	≤ 0.001
	Cos	0.56	≤ 0.001
	DamLev	0.67	≤ 0.001
	Dice	0.56	≤ 0.001
	Euclid	0.68	≤ 0.001
	Jaccard	0.56	≤ 0.001
	Jaro	0.67	≤ 0.001
	JaroW	0.67	≤ 0.001
	Lev	0.67	≤ 0.001
	MC	0.65	≤ 0.001
	NeedW	0.68	≤ 0.001
	Ovlp	0.54	≤ 0.001
	QGrams	0.66	≤ 0.001
	SmithW	0.66	≤ 0.001
	SmithG	0.66	≤ 0.001

Table A.1. Statistical tests - Search benchmark

Algorithm	Similarity	$\hat{A}_{12}$	p-value
Bubble sort	ADice	0.61	≤ 0.001
	Man	0.64	≤ 0.001
	Cos	0.61	≤ 0.001
	DamLev	0.65	≤ 0.001
	Dice	0.61	≤ 0.001
	Euclid	0.56	≤ 0.001
	Jaccard	0.61	≤ 0.001
	Jaro	0.63	≤ 0.001
	JaroW	0.63	≤ 0.001
	Lev	0.65	≤ 0.001
	MC	0.59	≤ 0.001
	NeedW	0.65	≤ 0.001
	Ovlp	0.51	≤ 0.001
	QGrams	0.64	≤ 0.001
	SmithW	0.66	≤ 0.001
	SmithG	0.65	≤ 0.001
Insertion sort	ADice	0.39	≤ 0.001
	Man	0.55	≤ 0.001
	Cos	0.39	≤ 0.001
	DamLev	0.61	≤ 0.001
	Dice	0.39	≤ 0.001
	Euclid	0.53	≤ 0.001
	Jaccard	0.39	≤ 0.001
	Jaro	0.59	≤ 0.001
	JaroW	0.59	≤ 0.001
	Lev	0.61	≤ 0.001
	MC	0.47	≤ 0.001
	NeedW	0.61	≤ 0.001
	Ovlp	0.51	≤ 0.001
	QGrams	0.55	≤ 0.001
	SmithW	0.66	≤ 0.001
	SmithG	0.66	≤ 0.001

Table A.2. Statistical tests - Sorting benchmark

Algorithm	Similarity	$\hat{A}_{12}$	p-value
	ADice	0.56	≤ 0.001
	Man	0.69	≤ 0.001
	Cos	0.56	≤ 0.001
	DamLev	0.67	≤ 0.001
	Dice	0.56	≤ 0.001
	Euclid	0.70	≤ 0.001
	Jaccard	0.56	≤ 0.001
	Jaro	0.67	≤ 0.001
Merge sort	JaroW	0.67	≤ 0.001
	Lev	0.67	≤ 0.001
	MC	0.50	0.440
	NeedW	0.67	≤ 0.001
	Ovlp	0.51	≤ 0.001
	QGrams	0.69	≤ 0.001
	SmithW	0.66	≤ 0.001
	SmithG	0.66	≤ 0.001
	ADice	0.60	≤ 0.001
	Man	0.68	≤ 0.001
	Cos	0.60	≤ 0.001
	DamLev	0.65	≤ 0.001
	Dice	0.60	≤ 0.001
	Euclid	0.66	≤ 0.001
	Jaccard	0.60	≤ 0.001
	Jaro	0.66	≤ 0.001
Quick sort	JaroW	0.66	≤ 0.001
	Lev	0.65	≤ 0.001
	MC	0.58	≤ 0.001
	NeedW	0.66	≤ 0.001
	Ovlp	0.51	≤ 0.001
	QGrams	0.68	≤ 0.001
	SmithW	0.64	≤ 0.001
	SmithG	0.65	≤ 0.001

Table A.3. Statistical tests - Sorting benchmark (cont.)

Bibliography

[AB11] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the International Conference on Software Engineering*, ICSE '11, pages 1–10. ACM, 2011.

[AK88] Paul E. Ammann and John C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing*, 1(1):11–33, 2004.

[Arc09] Andrea Arcuri. Theoretical analysis of local search in software testing. In *Stochastic Algorithms: Foundations and Applications*, volume 5792 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2009.

[Avi85] Algirdas Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[AY07] Andrea Arcuri and Xin Yao. A memetic algorithm for test data generation of object-oriented software. In *Proceedings of IEEE Congress on Evolutionary Computation*, CEC '07, pages 2048–2055. IEEE Computer Society, 2007.

[AY08] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceedings of IEEE Congress on Evolutionary Computation*, CEC '08, pages 162–168. IEEE Computer Society, 2008.

[Bak95] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of The Working Conference on Reverse Engineering*, WCRE '95, pages 86–95. IEEE Computer Society, 1995.

[Bak97] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.

[BBA+13] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 252–261. IEEE Computer Society, 2013.

[BBD+14] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 306–317. ACM, 2014.

[BEHK14] Veronika Bauer, Jonas Eckhardt, Benedikt Hauptmann, and Manuel Klimek. An exploratory study on reuse at google. In *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices*, SER&IPs 2014, pages 14–23. ACM, 2014.

[BGP07] Luciano Baresi, Sam Guinea, and Liliana Pasquale. Self-healing BPEL processes with dynamo and the jboss rule engine. In *International Workshop on Engineering of Software Services for Pervasive Environments*, ESSPE '07, pages 11–20, 2007.

[BHR84] Stephen D. Brookes, Charles Antony Richard Hoare, and Andrew William Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[BKL90] S.S. Brilliant, J.C. Knight, and N.G. Leveson. Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering*, 16:238–247, 1990.

[BLM10] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. Testful: An evolutionary test approach for java. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '09, pages 185–194. IEEE Computer Society, 2010.

[BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms: program transformations for practical scalable software evolution. In *Proceedings of the International Conference on Software Engineering*, ICSE '04, pages 625–634. IEEE Computer Society, 2004.

[BR83] Stephen D. Brookes and William C. Rounds. Behavioural equivalence relations induced by programming logics. In *Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 97–108. Springer, 1983.

[BST00] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the Annual Con-*

ference on USENIX Annual Technical Conference, ATEC ’00, pages 21–33. USENIX Association, 2000.

[BVJ14] Veronika Bauer, Tobias Völke, and Elmar Jürgens. A novel approach to detect unintentional re-implementations. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ICSME ’14, pages 491–495. IEEE Computer Society, 2014.

[CA78] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *International Symposium on Fault-Tolerant Computing*, FTCS ’78, pages 113–119, 1978.

[Cab09] Bruno Cabral. *A Transactional Model for Automatic Exception Handling*. PhD thesis, University of Coimbra, Portugal, 2009.

[CC10] Seung-Seok Choi and Sung-Hyuk Cha. A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics*, 8:43–48, 2010.

[CEF+06] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *Proceedings of the Conference on USENIX Security Symposium*, SEC ’06. USENIX Association, 2006.

[CGG+14] Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. Cross-checking oracles from intrinsic software redundancy. In *Proceedings of the International Conference on Software Engineering*, ICSE ’14, pages 931–942. ACM, 2014.

[CGM+13] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Nicolò Perino. Automatic recovery from runtime failures. In *Proceedings of the International Conference on Software Engineering*, ICSE ’13, pages 782–791. IEEE Computer Society, 2013.

[CGP09] Antonio Carzaniga, Alessandra Gorla, and Mauro Pezzè. Fault handling with software redundancy. In R. de Lemos, J. Fabre, C. Gacek, F. Gadducci, and M. ter Beek, editors, *Architecting Dependable Systems VI*, pages 148–171. Springer, 2009.

[CGPP10] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds for web applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE ’10, pages 237–246. ACM, 2010.

[CGPP15] Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè. Automatic workarounds: Exploiting the intrinsic redundancy of web applications. *ACM Transactions on Software Engineering and Methodologies*, 24(3):16, 2015.

[CH13] Zack Coker and Munawar Hafiz. Program transformations to fix c integers. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 792–801. IEEE Computer Society, 2013.

[CKTZ03] Tsong Y. Chen, F.-C. Kuo, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing and beyond. In *International Workshop on Software Technology and Engineering Practice*, STEP '03, pages 94–100. IEEE Computer Society, 2003.

[CLOM06] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Object distance and its application to adaptive random testing of object-oriented programs. In *Proceedings of the International Workshop on Random Testing*, RT '06, pages 55–63. ACM, 2006.

[CM07] Bruno Cabral and Paulo Marques. Exception handling: A field study in java and .net. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '07, pages 151–175. Springer, 2007.

[CM11] Bruno Cabral and Paulo Marques. A transactional model for automatic exception handling. *Computer Languages, Systems and Structures*, 37(1):43–61, 2011.

[CMP08] Hervè Chang, Leonardo Mariani, and Mauro Pezzè. Self-healing strategies for component integration faults. In *ARAMIS '08: Proceedings of the 1st International Workshop on Automated Engineering of Autonomous and runtiMe evolvIng Systems*. IEEE Computer Society, 2008.

[CMP09] Herve Chang, Leonardo Mariani, and Mauro Pezzè. In-field healing of integration problems with cots components. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 166–176. IEEE Computer Society, 2009.

[CRF03] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string metrics for matching names and records. In *KDD Workshop on Data Cleaning and Object Consolidation*, pages 73–78, 2003.

[Cri82] Flaviu Cristian. Exception handling and software fault tolerance. *IEEE Transactions on Computers*, 31:531–540, 1982.

[CTS08] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the International Conference on Software Engineering*, ICSE '08, pages 281–290. ACM, 2008.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '01, pages 73–88. ACM, 2001.

[DEG+06] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '06, pages 233–244. ACM, 2006.

[DEKM99] Richard Durbin, Sean Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis. Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1999.

[DER10] Ekwa Duala-Ekoko and Martin P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. *ACM Transactions on Software Engineering and Methodology*, 20:1–31, 2010.

[DF94] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.

[DH84] R. De Nicola and M.C.B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83–133, 1984.

[DKM+12] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Gordon Fraser, Sebastian Hack, and Andreas Zeller. Automatically generating test cases for specification mining. *IEEE Transactions on Software Engineering*, 38(2):243–257, 2012.

[DLWZ06] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, WODA '06, pages 17–24. ACM, 2006.

[DMM04] Ada Diaconescu, Adrian Mos, and John Murphy. Automatic performance management in component based software systems. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '04, pages 214–221. IEEE Computer Society, 2004.

[Dob06] Glen Dobson. Using WS-BPEL to implement software fault tolerance for Web services. In *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*, SEAA '06, pages 126–133. IEEE Computer Society, 2006.

[DPT07] Giovanni Denaro, Mauro Pezzè, and Davide Tosi. Designing self-adaptive service-oriented applications. In *ICAC '07: Proceedings of the 4th IEEE International Conference on Autonomic Computing*, page 16. IEEE Computer Society, 2007.

[DPT13] Giovanni Denaro, Mauro Pezzè, and Davide Tosi. Test-and-adapt: An approach for improving service interchangeability. *ACM Transactions on Software Engineering and Methodology*, 22(4), 2013.

[DR03] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '03, pages 78–95. ACM, 2003.

[DR05] Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the International Conference on Software Engineering*, ICSE '05, pages 176–185. ACM, 2005.

[DW10] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '10, pages 65–74. IEEE Computer Society, 2010.

[DXLBM14] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA '14, pages 30–39. ACM, 2014.

[DZM09] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '09. IEEE Computer Society, 2009.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[EFN+02] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded java program test generation. *IBM Software Journal*, 41(1):111–125, 2002.

[EGSK07] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of complex data structures. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '07, pages 64–73. ACM, 2007.

[EK08] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: a tool for repairing complex data structures. In *Proceedings of the International Conference on Software Engineering*, ICSE '08, pages 855–858. ACM, 2008.

[FA11] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM, 2011.

[FA13] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[FA15] Gordon Fraser and Andrea Arcuri. Evosuite at the sbst 2015 tool competition. In *Proceedings of the International Workshop on Search-Based Software Testing*, SBST '15, pages 25–27. IEEE Computer Society, 2015.

[FA16] Gordon Fraser and Andrea Arcuri. Evosuite at the sbst 2016 tool competition. In *Proceedings of the International Workshop on Search-Based Software Testing*, SBST '16, pages 33–36. ACM, 2016.

[FAM15] Gordon Fraser, Andrea Arcuri, and Phil McMinn. A memetic algorithm for whole test suite generation. *Journal of Systems and Software*, 103(0):311–327, 2015.

[FDO06] M. Muztaba Fuad, Debzani Deb, and Michael J. Oudshoorn. Adding self-healing capabilities into legacy object oriented application. In *Proceedings of the International Conference on Autonomic and Autonomous Systems*, ICAS '06. IEEE Computer Society, 2006.

[FO07] M. Muztaba Fuad and Michael J. Oudshoorn. Transformation of existing programs into autonomic and self-healing entities. In *Proceedings of the International Conference on the Engineering of Computer-Based Systems*, ECBS '07, pages 133–144. IEEE Computer Society, 2007.

[FX01] Christof Fetzer and Zhen Xiao. Detecting heap smashing attacks through fault containment wrappers. In *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, pages 80–89. IEEE Computer Society, 2001.

[GGM+14] Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, Mauro Pezzè, and Paolo Tonella. Search-based synthesis of equivalent method sequences. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, FSE '14, pages 366–376. ACM, 2014.

[GH04] Rosa Laura Zavala Gutiérrez and Michael N. Huhns. On building robust web service-based applications. In *Extending Web Services Technologies*, volume 13 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, pages 293–310. Springer, 2004.

[GJS08] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the International Conference on Software Engineering*, ICSE '08, pages 321–330. ACM, 2008.

[GMM09] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *Proceedings of the International Conference on Software Engineering*, ICSE '09, pages 430–440. IEEE Computer Society, 2009.

[Goo75] John B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[Got03] Arnaud Gotlieb. Exploiting symmetries to test programs. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '03, pages 365–374. IEEE Computer Society, 2003.

[GPS07] Ilir Gashi, Peter Popov, and Lorenzo Strigini. Fault tolerance via diversity for off-the-shelf products: A study with sql database servers. *IEEE Transactions on Dependable Secure Computing*, 4(4):280–294, 2007.

[GPSS04] Ilir Gashi, Peter Popov, Vladimir Stankovic, and Lorenzo Strigini. On designing dependable services with diverse off-the-shelf SQL servers. In *Architecting Dependable Systems II*, volume 3069 of *Lecture Notes in Computer Science*, pages 191–214. Springer, 2004.

[GRS06] Debin Gao, Michael Reiter, and Dawn Song. Behavioral distance for intrusion detection. In *Recent Advances in Intrusion Detection*, volume 3858 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2006.

[GT07] Michael Grottke and Kishor S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *IEEE Computer*, 40(2):107–109, 2007.

[HAB13] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology*, 22(1):1–42, 2013.

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

[Hat97] Les Hatton. N-version design versus one good version. *IEEE Software*, 14(6):71–76, 1997.

[HBR81] Charles Antony Richard Hoare, Stephen D. Brookes, and Andrew William Roscoe. A theory of communicating sequential processes. Technical report PRG-16, Oxford University, Programming Research Group, 1981.

[HC10] Ishtiaque Hussain and Christoph Csallner. Dynamic symbolic data structure repair. In *Proceedings of the International Conference on Software Engineering*, ICSE '10, pages 215–218. ACM, 2010.

[HHH+04] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[HK14] Yoshiki Higo and Shinji Kusumoto. How Should We Measure Functional Sameness from Program Source Code? An Exploratory Study on Java Methods. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 294–305. ACM, 2014.

[HKKF95] Yennun Huang, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, FTCS '95, pages 381–390. IEEE Computer Society, 1995.

[HM80] Matthew Hennessy and Robin Milner. On observing nondeterminism and concurrency. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer, 1980.

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

[Hoa80] Charles Antony Richard Hoare. A model for communicating sequential process. Working paper, Department of Computing Science, University of Wollongong, 1980.

[HRD08] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Developing and debugging algebraic specifications for Java classes. *ACM Transactions on Software Engineering and Methodology*, 17:1–37, 2008.

[JAB01] Scott A. SmolkaBergstra Jan A. Bergstra, Alban Ponse. *Handbook of Process Algebra*. Elsevier, 2001.

[Jar95] Matthew A. Jaro. Probabilistic linkage of large public health data files. *Statistics in Medicine*, 14(5-7):491–498, 1995.

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the International Conference on Software Engineering*, ICSE '09, pages 485–495. IEEE Computer Society, 2009.

[JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering*, ICSE '02, pages 467–477. ACM, 2002.

[JO12] Wei Jin and Alessandro Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proceedings of the International Conference on Software Engineering*, ICSE '12, pages 474–484. IEEE Computer Society, 2012.

[JS09] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '09, pages 81–92. ACM, 2009.

[KDM+96] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. In *Reverse Engineering*, pages 77–108. Kluwer Academic Publishers, 1996.

[KH01] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the International Symposium on Static Analysis*, SAS '01, pages 40–56. Springer, 2001.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28:654–670, 2002.

[KL86] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12:96–109, 1986.

[KLNB+09] Bohuslav Křena, Zdeněk Letko, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Shmuel Ur, and Tomáš Vojnar. A concurrency testing tool and its plug-ins for dynamic analysis and runtime healing. In *Runtime Verification*, RV '09, pages 101–114. Springer, 2009.

[KMY91] John P. J. Kelly, Thomas I. McVittie, and Wayne I. Yamamoto. Implementing design diversity to achieve fault tolerance. *IEEE Software*, 8(4):61–71, 1991.

[KNSK13] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 802–811. IEEE Computer Society, 2013.

[KPR04] Vyacheslav Kharchenko, Peter Popov, and Alexander Romanovsky. On dependability of composite web services with components upgraded online. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '04, pages 287–291. Springer, 2004.

[KR09] David Kawrykow and Martin P. Robillard. Improving api usage through automatic detection of redundant code. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '09, pages 111–122. IEEE Computer Society, 2009.

[KSNM05] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '05, pages 187–196. ACM, 2005.

[LBK90] Jean-Claude Laprie, Christian Béounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, 1990.

[LDVFW12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the International Conference on Software Engineering*, ICSE '12, pages 3–13. IEEE Computer Society, 2012.

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

[LLMZ04] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, OSDI'04, pages 289–302. USENIX Association, 2004.

[LMP08] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *ICSE '08: Proceedings of the 30th*

International Conference on Software Engineering. IEEE Computer Society, 2008.

[LMX05] Nik Looker, Malcolm Munro, and Jie Xu. Increasing Web service dependability through consensus voting. In *Proceedings of the International Computer Software and Applications Conference*, COMPSAC '05, pages 66–69. IEEE Computer Society, 2005.

[LNFW12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38:54–72, 2012.

[LR08] Konstantinos Liaskos and Marc Roper. Hybridizing evolutionary testing with artificial immune systems and local search. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshop*, ICSTW '08, pages 211–220, 2008.

[LR15] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '15, pages 166–178. ACM, 2015.

[LR16] Fan Long and Martin Rinard. Prophet: Automatic patch generation via learning from successful patches. In *Proceedings of the Symposium on Principles of Programming Languages*, POPL '15, page to appear. ACM, 2016.

[LV62] Robert E. Lyons and Wouter Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6:200–209, 1962.

[MB08] Adina Mosincat and Walter Binder. Transparent runtime adaptability for BPEL processes. In *Proceedings of the International Conference on Service Oriented Computing*, ICSOC '08, pages 241–255. Springer, 2008.

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

[MFC+09] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *Computer*, 42(9):46–55, 2009.

[MGG15] Andrea Mattavelli, Alberto Goffi, and Alessandra Gorla. Synthesis of equivalent method calls in Guava. In *Proceedings of the 7th International Symposium on Search-Based Software Engineering*, SSBSE '15, pages 248–254. Springer, 2015.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1982.

[MKIE15] Paul Muntean, Vasantha Kommanapalli, Andreas Ibing, and Claudia Eckert. Automated generation of buffer overflow quick fixes using symbolic execution and smt. In *Proceedings of the International Conference on Computer Safety, Reliability, and Security*, SAFECOMP '15, pages 441–456. Springer, 2015.

[MLB09] Matteo Miraz, Pier Luca Lanzi, and Luciano Baresi. Testful: using a hybrid evolutionary algorithm for testing stateful systems. In *Proceedings of the conference on Genetic and Evolutionary Computation*, GECCO '09, pages 1947–1948. ACM, 2009.

[MM01] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '01, pages 107–114. IEEE Computer Society, 2001.

[MMP06] Stefano Modafferi, Enrico Mussi, and Barbara Pernici. SH-BPEL: a self-healing plug-in for WS-BPEL engines. In *Proceedings of the 1st Workshop on Middleware for Service Oriented Computing*, MW4SOC '06, pages 48–53. ACM, 2006.

[MVZ+12] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: A domain-specific language for bytecode instrumentation. In *Proceedings of the International Conference on Aspect-oriented Software Development*, AOSD '12, pages 239–250. ACM, 2012.

[MYR15] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the International Conference on Software Engineering*, ICSE '15, pages 448–458. IEEE Computer Society, 2015.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33:31–88, 2001.

[NBTU08] Yarden Nir-Buchbinder, Rachel Tzoref, and Shmuel Ur. Deadlocks: From exhibiting to healing. In *Proceedings of the International Conference on Runtime Verification*, RV '08, pages 104–118. Springer, 2008.

[NG07] Henri Naccache and Gerald C. Gannod. A self-healing framework for web services. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '07, pages 398–345. IEEE Computer Society, 2007.

[NQRC13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the International Conference on Software Engineering*, ICSE '13, pages 772–781. IEEE Computer Society, 2013.

[OB88] Thomas J. Ostrand and Marc J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[OPM14] Frolin S. Ocariza, Jr., Karthik Pattabiraman, and Ali Mesbah. Vejovis: Suggesting fixes for javascript faults. In *Proceedings of the International Conference on Software Engineering*, ICSE '14, pages 837–847. ACM, 2014.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 1981.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record*, 17(3):109–116, 1988.

[PHP99] Roy Pargas, Mary Jean Harrold, and Robert Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[PKL+09] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, 2009.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering*, ICSE '07, pages 75–84. ACM, 2007.

[PRRS01] Peter Popov, Steve Riddle, Alexander Romanovsky, and Lorenzo Strigini. On systematic design of protectors for employing OTS items. In *Proceedings of the 27th EUROMICRO Conference*, EUROMICRO '01, pages 22–29, 2001.

[PTK13] Jeremy R. Pate, Robert Tairas, and Nicholas A. Kraft. Clone evolution: a systematic review. *Journal of Software: Evolution and Process*, 25:261–283, 2013.

[QML+14] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the International Conference on Software Engineering*, ICSE '14, pages 254–265. ACM, 2014.

[Ran75] Brian Randell. System structure for software fault tolerance. *SIGPLAN Notes*, 10(6):437–449, 1975.

[RBD12] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. Clones: What is that smell? *Empirical Software Engineering*, 17(4-5):503–530, 2012.

[RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '97, pages 432–449. Springer, 1997.

[RE11] David A Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the International Conference on Computer Aided Verification*, CAV '11, pages 669–685. Springer, 2011.

[RE15] David A. Ramos and Dawson R. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *Proceedings of the Conference on USENIX Security Symposium*, pages 49–64. USENIX Association, 2015.

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[RR03] Manos Renieris and Steven P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '03, pages 30–39. IEEE Computer Society, 2003.

[RZF+13] Jeremias Rössler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. Reconstructing core dumps. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '13, pages 114–123. IEEE Computer Society, 2013.

[SDLLR15] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '15, pages 43–54. ACM, 2015.

[Sel74] Peters Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974.

[Sha14] Alex Shaw. Program transformations to fix c buffer overflows. In *Proceedings of the International Conference on Software Engineering*, ICSE '14, pages 733–735. ACM, 2014.

[SJGF09] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the ACM SIGOPS EuroSys European Conference on Computer Systems*, EuroSys '09, pages 33–46. ACM, 2009.

[SM05] Seyed Masoud Sadjadi and Philip K. McKinley. Using transparent shaping and Web services to support self-management of composite systems. In *Proceedings of the International Conference on Autonomic Computing*, ICAC '05, pages 76–87. IEEE Computer Society, 2005.

[SRFA99] Frederic Salles, Manuel Rodrguez, Jean-Charles Fabre, and Jean Arlat. Metakernels and fault containment wrappers. In *International Symposium on Fault-Tolerant Computing*, pages 22–29. IEEE Computer Society, 1999.

[STN+08] Sattanathan Subramanian, Philippe Thiran, Nanjangud C. Narendra, Ghita Kouadri Mostefaoui, and Zakaria Maamar. On the enhancement of BPEL engines for self-healing composite web services. In *Proceedings of the International Symposium on Applications and the Internet*, SAINT '08, pages 33–39. IEEE Computer Society, 2008.

[TBFM06] Yehia Taher, Djamal Benslimane, Marie-Christine Fauvet, and Zakaria Maamar. Towards an approach for Web services substitution. In *Proceedings of the International Database Engineering and Applications Symposium*, IDEAS '06, pages 166–173. IEEE Computer Society, 2006.

[TCZ15] Valerio Terragni, Shing-Chi Cheung, and Charles Zhang. Recontest: Effective regression testing of concurrent programs. In *Proceedings of the International Conference on Software Engineering*, ICSE '15. ACM, 2015.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '04, pages 119–128. ACM, 2004.

[TR15] Shin Hwei Tan and Abhik Roychoudhury. Relifix: Automated repair of software regressions. In *Proceedings of the International Conference on Software Engineering*, ICSE '15, pages 471–482. IEEE Computer Society, 2015.

[VD00] András Vargha and Harold D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal on Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[vHR12] Pavol Černý, Thomas A. Henzinger, and Arjun Radhakrishna. Simulation distances. *Theoretical Computer Science*, 413(1):21–35, 2012.

[Voa98] Jeffrey M. Voas. Certifying off-the-shelf software components. *IEEE Computer*, 31(6):53–59, 1998.

[Wey82] Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25(4):465–470, 1982.

[WJ06] Hsiu-Chi Wang and Bingchiang Jeng. Structural testing using memetic algorithm. In *Proceedings of the Second Taiwan Conference on Software Engineering*, 2006.

[WNGF09] Westley Weimer, ThanVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the International Conference on Software Engineering*, ICSE ’09, pages 364–374. IEEE Computer Society, 2009.

[WPF+10] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA ’10, pages 61–72. ACM, 2010.

[XDK12] Hui Xue, Nathan Dautenhahn, and Samuel T. King. Using replicated execution for a more secure and reliable web browser. In *Proceedings of the Annual Network and Distributed System Security Symposium*, NDSS ’12. The Internet Society, 2012.

[XN03] Tao Xie and David Notkin. Mutually enhancing test generation and specification inference. In *Proceedings of the International Workshop on Formal Approaches to Testing of Software*, FATES ’03, pages 60–69. Springer, 2003.

[YC75] Stephen S. Yau and Ray C. Cheung. Design of self-checking software. In *Proceedings of the International Conference on Reliable Software*, pages 450–455. ACM, 1975.

[ZYR+14] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz Khurshid. Feedback-driven dynamic invariant discovery. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA ’14, pages 362–372. ACM, 2014.