Automatic Workarounds: Exploiting the Intrinsic Redundancy of Software Systems

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
Alessandra Gorla

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

July 2011

Dissertation Committee

Prof. Cesare Pautasso	Università della Svizzera Italiana, Switzerland
Prof. Paola Inverardi	University of L'Aquila, Italy
Prof. Rogério de Lemos	University of Kent, United Kingdom

Dissertation accepted on 01 July 2011

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

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

Prof. Michele Lanza
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.

Alessandra Gorla
Lugano, 01 July 2011

To my parents

Abstract

Despite the best intentions, the competence, and the rigorous methods of designers and developers, software is often delivered and deployed with faults. In order to cope with imperfect software, researchers have proposed the appealing notion of *self-healing* for software systems. The ambitious goal is to create software systems capable of detecting and responding "autonomically" to functional failures so as to completely or at least partially compensate for those failures. In this dissertation we take a principled approach to the concept of self-healing software. We believe that self-healing can only be an expression of some form of redundancy, meaning that, to automatically fix a faulty behavior, the correct behavior must be already present somewhere, in some form, within the software system. One approach is to *deliberately* design and develop redundant systems, and in fact this kind of deliberate redundancy is the essential ingredient of many fault tolerance techniques. However, this type of redundancy is also generally expensive and, as it turns out, not always effective.

Our intuition is that modern software systems naturally acquire another type of redundancy, which is not introduced deliberately but that rather occurs *intrinsically* as a by-product of modern modular software design. This thesis develops this intuition and researches ways to use intrinsic redundancy to achieve some level of self-healing. We first demonstrate that software systems are indeed intrinsically redundant. Then we develop a way to express and exploit this redundancy to tolerate faults with a technique called Automatic Workarounds. In essence, an Automatic Workaround amounts to replacing some failing operations with alternative operations that are semantically equivalent in their intended effect, but that ultimately avoid the failure. We develop the notion of Automatic Workarounds in the context of Web applications, and in particular we implement a browser extension as a prototype to evaluate the ability of the technique to deal with several known issues of three popular Web libraries. Our studies show that workarounds are frequently used by developers to address failures before a permanent fix is available, and a manual analysis of the workarounds that are publicly reported in issue trackers show that a large portion of them are instances of intrinsic redundancy, and are therefore easy to find automatically with our technique. We prove the effectiveness of our technique by showing that it automatically finds a valid solution for most of the issues that we analyzed, and furthermore that it finds valid solutions for a large portion of the open issues for which no workaround was known.

Contents

Figures

Tables

Chapter 1

Introduction

This dissertation explores the use of the intrinsic redundancy of software for several purposes, and in particular it proposes a technique to exploit redundancy to mask faults at runtime.

Redundancy is the key element of most of hardware and software fault tolerance techniques. In hardware systems, redundancy is commonly used to tolerate physical faults. RAID, for example, is a very successful technology that overcomes faults as well as the performance limitations of disk storage by bundling and controlling an array of disks so as to present them as a single, more reliable storage device [PGK88]. The positive results obtained by using redundancy at the hardware level led researchers to apply the same principles to software, in particular in an attempt to address development faults in safety critical systems. N-version programming and recovery blocks are the most successful techniques developed in this line of research. These techniques are based on a similar redundancy as RAID, in the sense that they assume that faults are not correlated, and therefore that several independently developed versions of the same component are unlikely to fail on the same input [Avi85; Ran75]. Thus, to the extent that this assumption is correct, all versions can be executed in parallel (N-version) or in sequence (recovery blocks) and the output of the system is determined by the output of the majority of executions (N-version) or by the first non-failing execution (recovery blocks).

The use of redundancy has been recently explored in several areas of software engineering, such as software testing. In this case, multiple implementations of the same component can be used as an implicit oracle: each test is executed by all implementations, and if all executions produce identical results then the test is considered passed, while any inconsistent behavior of the different implementations is considered a failed test [DF94; Got03]. Formal specifications (i.e., contracts, algebraic specifications, assertions, etc.) are another form of redundancy that has been extensively exploited in software testing, typically to determine the correctness of the test executions [DF94; CLOM08].

Redundancy is also the key element of several software fault localization techniques. In fact, most of the well-known techniques in this area require multiple redundant executions of a test suite to pinpoint the likely faulty statements in the code [Zel99; ZH02]. Yet another technique for fault localization is to rely on specifications or models of correct executions to identify the root cause of the failures [MP08; MPPar].

Regardless of the type of redundancy (code, specifications, models, etc.) and its use (fault tolerance, testing, fault localization, etc.) most of the techniques mentioned so far exploit a redundancy that is *deliberately introduced within the software system* at design time. However, this form of "deliberate" redundancy can be problematic. First, because it introduces additional development costs. This cost overhead is particularly significant in techniques such as N-version programming. Second, because, as it turns out, faults exhibit some level of correlation even across components that are developed completely independently. In other words, as demonstrated in several studies, the assumption that faults occur independently in different versions is not realistic [KL86; BKL90; Hat97].

This thesis proposes to exploit a particular type of redundancy that is *intrinsic* in software.

We argue that software is inherently redundant for three main reasons. First, the modern development process naturally induces developers to use third-party software components that already implement the functionalities they need. It is usually possible to find several components that offer similar functionalities. Second, despite the best intentions of modularization and reuse, developers often implement logically similar functions multiple times and in different ways. Finally, even when and in fact *because* software is extensively modularized, redundancy occurs as a by-product of the design of highly configurable and versatile libraries or frameworks. Regardless of the reason, the fact is that *software is redundant*, as demonstrated by some studies on functionally equivalent code [GJS08; JS09].

Containers are a typical example of reusable components that are intrinsically redundant. It is usually possible, for example, to add several elements to a container by either adding them one after the other, or by adding them all at the same time. Since according to the specifications these two operations should lead to the same result, we call them "*equivalent sequences*" of operations.

The goal of this thesis is to study intrinsic redundancy and its applications in different areas that span from software testing to fault localization, focusing in particular on fault handling.

Intrinsic redundancy can be exploited to build test oracles. In fact, if two different operations are expected to lead to the same result, i.e., if they are equivalent sequences, it is possible to check their correctness by comparing the result of their execution. Doong and Frankl first and Gotlieb later [DF94; Got03] explored this research area with good results. The same idea can be exploited to implement a fault handling

mechanism at runtime. In fact, if we know that two operations are equivalent, and at runtime one of the two fails, then we can try to execute the second operation in an attempt to mask the fault in the first one, and achieve the expected result. When it is possible to find an *equivalent sequence* that does not fail, then we call it a *workaround*.

The major contribution of this thesis is the definition of a technique that exploits intrinsic redundancy to find workarounds automatically and at runtime. The idea behind this technique, which we refer to as Automatic Workarounds, is to have a layer that monitors the execution of the components in the application, and tries a new sequence of operations whenever a failure occurs. The mechanism amounts to the following steps. (1) The healing layer detects the failure, and (2) brings the components that have been affected by the failure back to a consistent state to avoid any side-effect. Later, the layer identifies the list of sequences of operations that are equivalent to the operations that caused the failure (3), and needs to select the one that is more likely to work and execute it (4). Finally it has to evaluate the equivalent sequence, and understand whether it is a valid workaround (5). If a valid workaround has not been found, the self-healing layer can select the next more likely working equivalent sequence, and keep repeating these steps for as many attempts as possible.

To evaluate the Automatic Workarounds framework, we decided to focus on a specific type of applications, that is Web applications using Javascript libraries.

The main reason why we selected this type of applications is that it is hard to assess the quality of such applications during development, and many faults escape the testing process. First of all Web applications operate in very heterogeneous environments, and some problems may occur only under specific conditions (e.g. a particular combination of browser, operating system and connection speed). Moreover, the evolution of Web APIs used within a Web application is out of any control of the Web application developers, and a new API release may cause failures unexpectedly. Thus, a mechanism that can deal with failures at runtime is necessary in this context.

The second reason why we focused our work on Web applications is because these applications have characteristics that can ease the implementation of our technique. First of all, since Web applications are highly interactive by nature, we can exploit this characteristic by asking the user to report failures in the visited page. Thus, in this domain we do not need to implement a failure detector. Second, we can restrict our study on stateless applications in order to avoid the rollback of the failing components, which is the major challenge of the Automatic Workarounds technique.

The next sections in this introductory chapter go through the terminology used throughout the rest of the dissertation (Section 1.1), the main research hypothesis and contributions (Section 1.2), and the structure of the document (Section 1.3).

1.1 Terminology

As this thesis explores the use of intrinsic redundancy for different applications that span from software engineering to fault tolerance, we need to define some key terms that are used in the two communities. In fact, the two communities sometimes use the same terms with slightly different meanings, as it is the case for the concepts of *fault*, *failure* and *error*.

The fault-tolerance community typically refers to the terminology defined by Avizienis et al. [ALRL04]:

> "... a service *failure* means that at least one (or more) external state of the system deviates from the correct service state. The deviation is called an *error*. The adjudged or hypothesized cause of an error is called a *fault*. Faults can be internal or external of a system."

while the software engineering community typically refers to the IEEE standard document that provides the following definitions [IEE90]:

Fault : A defect in a hardware device or component or an incorrect step, process, or data definition in a computer program.

Error : A human action that produces an incorrect result. For example, an incorrect action on the part of a programmer or operator.

Failure : The inability of a system or component to perform its required functions within specified requirements.

Thus, the fault tolerance discipline distinguishes between a human action (a *mistake*), its manifestation (a hardware or software *fault*), the result of the fault (a *failure*), and the amount by which the result is incorrect (the *error*). Software engineering, instead, distinguishes between the human action (an *error*), its manifestation (a *fault*), and the result of the fault (a *failure*). Sometimes in software engineering the terms *bug* and *issue* are used as synonyms for fault.

The two communities may also refer to the same concept using different terms. This is the case of *oracle*, used in software engineering, and *adjudicator*, used in fault tolerance. Both terms refer to the element that can judge the correctness of the execution, that is, the element that can reveal the presence of a fault in the code.

Throughout the rest of the document we use the mentioned terms with their software engineering connotation. More precisely, we refer to the *fault* as the defect in the software, the *failure* as the visible result of the fault, and the *oracle* as the element that can detect the failure. The concept of error is not relevant for this thesis, therefore we avoid the use of this term to avoid confusion.

1.2 Research Hypothesis and Contributions

The main research hypothesis of this thesis is that

> *Software is intrinsically redundant, in the sense that the same operations can be performed in multiple different ways. This type of redundancy can be captured, represented and exploited for several purposes, in particular for fault-handling at runtime.*

The first part of the hypothesis describes the main intuition of this thesis, which is that software systems, and software modules in particular, are intrinsically redundant. Although some recent studies have shown that software contains a lot of semantically equivalent fragments [GJS08; JS09], nobody ever defined the concept of intrinsic redundancy. Moreover these studies do not point out the value of such redundancy.

The second part of the research hypothesis is about how to exploit intrinsic redundancy. There are a few studies on how to possibly use this redundancy, most of them limited to software testing [DF94; Got03]. Our idea is that intrinsic redundancy can be useful to mask the presence of faults at runtime. This intuition comes from the observation of what users do in practice. When they are using a faulty application, and they run into a failure, they look for alternative ways allowed by the application to do what they were planning to do. What the second part of the research hypothesis suggests is that intrinsic redundancy may allow us to automatically identify and execute alternative operations when a failure occurs at runtime.

This thesis makes two major contributions:

Intrinsic redundancy: The first contribution is the notion of intrinsic redundancy in software systems. Although we are not the first ones to use this type of redundancy, we are the first ones to define it, to emphasize its value, and to propose how to capture and use it systematically. We experimentally demonstrate that intrinsic redundancy exists, and is quite pervasive especially in Web APIs. Moreover, we propose a classification of the equivalent sequences, and a way to represent them in the form of program-rewriting rules.

Automatic Workarounds: The second contribution of this thesis is the Automatic Workarounds, a technique that allows to mask the presence of faults in software by looking for alternative ways to execute failing operations. More precisely, we propose a technique that identifies redundancy and exploits it to find workarounds automatically at runtime. We propose a general architecture for component based software, and a more detailed one for Web applications. Moreover, we define several priority strategies for this technique. These strategies aim to help the mechanism in better selecting the equivalent sequences, such that the ones that are more likely to serve as workarounds are executed first. In the architecture for Web applications we also include an automated oracle that can filter

out some of the non valid attempts to find a workaround. Finally, we evaluate the effectiveness of the technique on a set of known issues of popular Web APIs such as Google Maps, YouTube and JQuery. For each issue we provide the code (if it is not publicly available) to reproduce the problem.

1.3 Structure of the Dissertation

The remainder of this dissertation is structured as follows:

- Chapter 2 provides an overview of several techniques related to software redundancy. We define a taxonomy that we use to classify techniques that can identify redundancy in software, and techniques that rely on some form of redundancy to either test programs, or handle failures at runtime.
- Chapter 3 focuses on *intrinsic* redundancy. This is a particular type of redundancy that is naturally present in software, and can be exploited for different purposes, in particular to avoid failures at runtime.
- Chapter 4 presents the Automatic Workarounds technique, that exploits intrinsic redundancy in software to find workarounds at runtime after a failure occurrence. In particular, it explains how the technique can be applied in the context of Web applications.
- Chapter 5 describes the prototype implementation that we used to validate the Automatic Workarounds technique.
- Chapter 6 presents the empirical results of the experiments we ran on Google Maps, YouTube and JQuery to show the effectiveness of the Automatic Workarounds technique.
- Chapter 7 summarizes the contributions of this dissertation, and discusses the future directions of this work.

Chapter 2

State of the Art

The use of software redundancy has been explored in several research areas such as software testing, fault localization, and in particular fault handling. This chapter aims to provide an overview of existing techniques that are somehow related to software redundancy. We start with a description of the taxonomy we use, and we then survey the existing techniques that can identify redundancy in software together with the techniques that use software redundancy for testing and for either handling or tolerating faults at runtime.

The use of redundancy has been explored in several areas such as software testing, fault localization, and in particular fault-handling. In fact, despite mature design and development methods, despite rigorous testing procedures, efficient verification algorithms, and many other software engineering techniques, the majority of non-trivial software systems are deployed with faults. However the reliability of such systems can be improved by allowing them to prevent or alleviate the effect of faults, and perhaps even to correct the faults at runtime. These are essentially the goals of much research in the area of fault tolerance [Pul01; ALRL04; KK07] and more recently in autonomic computing [KC03; KM07; Hor01].

There are important differences between the approaches to reliability found in the fault tolerance and autonomic computing literature. First, at a high level, fault tolerance is a more focused area, while autonomic computing covers a larger set of objectives. The term autonomic computing refers to the general ability of a system to respond to various conditions such as performance degradation (self-optimizing systems), changes in the configuration of the environment (self-configuring systems), architectural changes to satisfy new requirements (self-adaptive systems), runtime functional failures (self-healing systems), and security issues (self-protecting systems). Fault tolerance, instead, deals with conditions that are caused or affected by faults, thus it has the same objective of a single sub-area of autonomic computing, i.e. self-healing.

Another difference is in the nature of the intended application domain. Fault tolerance research has been driven primarily by highly specialized and safety-critical systems, whereas autonomic computing—specifically, self-healing—is targeted towards general purpose components or loosely coupled systems in which the effects of failures are less destructive. These two application domains also have significant differences in the levels of costs and ultimately in the type of designs that are considered acceptable.

Yet another difference is that fault tolerance research explicitly addresses both hardware and software faults with different techniques that may be hardware or software, while self-healing research does not distinguish different classes of faults and has so far studied mostly software techniques.

Despite their differences, fault tolerance and self-healing share the same goal of isolating or at least limiting the effects of faults. In particular, we see that most of the techniques in both areas share the central intuition of exploiting some form of *redundancy* present in the system.

A system is redundant when it is capable of executing the same, logically unique functionality in multiple ways or in multiple instances. The availability of alternative execution paths or alternative execution environments is the primary ingredient of practically all systems capable of avoiding or tolerating failures. For example, a fairly obvious technique to overcome non-deterministic faults, such as hardware faults, is to run multiple replicas of the system, and then simply switch to a functioning replica when a catastrophic failure compromises one of the replicas. In fact, redundant hardware has been developed since the early sixties to tolerate development faults as well as manufacturing faults in circuits [LV62; PGK88; DG04]. An analogous form of redundancy is at the core of many widely studied replication techniques used to increase the availability of data management systems [EASC85].

Similarly, redundancy has been used extensively to tolerate software faults [Pul01], although software poses some special challenges and also provides new opportunities to exploit redundancy. For example, while simple replication of components can handle some classes of production faults typical of hardware design, it cannot deal with many failures that derive from development and integration problems that occur often in software systems. On the other hand, software systems are amenable to various forms of redundancy generally not found in hardware systems. It is possible, for example, to overcome deterministic problems by replacing faulty components with alternative redundant software components that have been deliberately designed and implemented in different ways, and by different people [Avi85; STN+08; SM05]. The availability of such replacement components is a form of redundancy, and is also generally applicable to hardware systems. However, the nature of software components and their interactions may make this technique much more effective, in terms of time and costs, for software rather than hardware components.

Yet another example is that of *micro-reboots*, which exploit another form of redundancy rooted in the execution environment rather than in the code. In this case, the

system re-executes some of its initialization procedures to obtain a fresh execution environment that might in turn make the system less prone to failures [CKZ+03; Zha07].

Redundancy has been used in software testing as well, in particular to address the oracle problem, that is one of the major challenges of this research area. Since it is usually infeasible to manually judge the correctness of test executions, testers are used to automate the evaluation process by comparing the outcome of a test execution with the expected result that they provide as an oracle. However, precomputing the oracle is a time consuming activity as well, as it may require to manually perform the computation that the software under test is supposed to do. Moreover, sometimes the output is just too complex to compute, and the tester may not have an idea of what the correct result is. According to Weyuker's definition, these programs are *non-testable* [Wey82]. As a solution she proposed to provide an independently written program intended to fulfill the same specification as the program under test, and to run both programs on identical sets of input data and to compare the final results.

Formal specifications are another form of redundancy that has been exploited in software testing for the purpose of having an oracle. Several techniques use program assertions, contracts, and algebraic specification to check the execution of test cases [CLOM08; MFC+09; DF94].

To survey the existing testing, self-healing and fault tolerance techniques that use redundancy to either reveal or to tolerate and avoid software faults, we define a taxonomy that is described in Section 2.1. We then proceed with the analysis of several techniques, first in Section 2.2 where we discuss techniques based on the deliberate use of redundancy, and then in Section 2.3 where we discuss techniques that are based on intrinsic redundancy.

2.1 A Taxonomy for Redundancy

Fault tolerance, autonomic computing and software testing are well-established research areas, and they have all been surveyed extensively. However existing taxonomies focus on specific techniques and applications, and they consider only single areas. For example, Huebscher and McCanne define a taxonomy to review many techniques developed in the context of autonomic computing [HM08], but this taxonomy does not allow to relate them to fault tolerance. At the other end of the spectrum, De Florio and Blondia compile an extensive survey of software fault tolerance techniques [FB08]. In particular, they discuss some techniques related to redundancy (for instance, N-version programming), but primarily review domain-specific languages and other linguistic techniques to enhance the reliability of software systems at the application level. Another related survey is one by Littlewood and Strigini [LS04], who examine the benefits of redundancy—specifically *diversity*—to increase system reliability in the presence of faults that pose security vulnerabilities. Yet another example of a focused survey is the widely cited work by Elnozahy et al. on rollback-recovery

Intention:	deliberate intrinsic
Type:	code data environment
Triggers and oracles:	preventive (implicit oracle) reactive: implicit oracle explicit oracle
Faults addressed by redundancy:	interaction: Malicious development: Bohrbugs Heisenbugs

Table 2.1. Taxonomy for redundancy based mechanisms

protocols [EAmWJ02].

In the software testing context, the survey of Baresi and Young on software testing oracles, which provides a thorough analysis of testing techniques that deal with the oracle problem, is another example of taxonomies that focus on a single area [BY01].

The taxonomies defined by Ammar et al. [ACMF00] and Avizienis et al [ALRL04] are particularly interesting. Ammar et al. propose an extensive survey of the different aspects of fault tolerance techniques, and, in this context, distinguish *spatial*, *information*, and *temporal* redundancy. This taxonomy focuses on the dimensions of redundancy, and matches well the differences of redundant techniques for handling hardware as well as software faults. Avizienis et al. propose a fault taxonomy that has become a de-facto standard.

The taxonomy proposed in this thesis aims to provide a unifying framework for the use of redundancy for both software testing and fault handling, considering both fault tolerance and self-healing research areas. It is based upon four key elements: the *intention* of redundancy, the *type* of redundancy, the *nature of triggers and oracles* that can activate redundant mechanisms and use their results, and lastly the *class of faults* addressed by the redundancy mechanisms. The nature of the triggers and oracles is applicable only for fault handling techniques. Table 2.1 summarizes this high-level classification scheme that is explained in details in the following paragraphs.

Intention. Redundancy can be either *deliberately* introduced in the design or *intrinsically* present in the system. Some techniques deliberately add redundancy to the system to either reveal faults or to handle them on a deployed system. This is the case, for example, of N-version programming that replicates the design process to produce redundant functionalities to mask failures in single modules [Avi85]. Pseudo-oracles are another example of redundancy that is deliberately added to the software, as they share the idea of N-version programming although with the purpose of detect-

ing faults [Wey82].

Other techniques exploit redundancy latent in the system intrinsically. This is the case, for example, of the micro-reboots technique that reboots a set of components to tolerate non deterministic failures [CKZ+03]. Although techniques from both categories can be applied to different classes of systems, deliberately adding redundancy impacts on development costs, and is thus exploited more often in safety critical applications, while intrinsic redundancy has been explored more often in other application domains.

The classification described here is similar to the one discussed by Brito et al., who distinguish between implicit and explicit redundancy [BLR08]. However their classification refers to the intention of incorporating redundancy at the architectural level.

Type. A system is redundant when some elements of its *code*, its input *data*, or its execution *environment* (including the execution processes themselves) are partially or completely replicated. Some techniques rely on redundant computation that replicates the functionality of the system to detect and heal a faulty computation. For example, pseudo-oracles and N-version programming compare the results of equivalent components to reveal a fault in the first case, and to produce a correct result in the second case. This is a case of *code* redundancy. Other techniques rely on redundancy in the data handled during the computation. For example, so-called data diversity relies on redundancy in the data used for the computation, and not on the computation itself, which is based on the same code [AK88]. Yet other techniques rely on redundancy that derives from different reactions of the environment. For example environment perturbation techniques change memory management strategy, and change other elements in the environment to avoid failures [QTZS07]. Different types of redundancy apply to different types of systems and different classes of faults.

This classification based on the type of replicated elements is similar to Ammar's classification in spatial, information, and temporal redundancy [ACMF00] that applies better when considering techniques based on redundancy to handle both hardware and software faults.

Triggers and oracles. When redundancy is exploited for fault handling, redundant components can be either triggered *preventively* to avoid failures, or exploited *reactively* in response to failures. In the first case, the system must decide when and where to act to maximize the chance of avoiding failures. Examples of the preventive use of redundancy are rejuvenation techniques that reboot the system before failures occur [HKKF95]. In the second case, the system must at a minimum detect a failure, and therefore decide how to exploit the redundancy of the system in order to cope with the failure. Fault tolerance refers to the component that can detect a failure as the *adjudicator*. As we already mentioned in Section 1.1, we use the software engineering ter-

minology instead, that refers to the same component as the *oracle*. We further classify a technique by distinguishing oracles that are either *implicitly* built into the redundant mechanisms, or *explicitly* designed by the software engineers for a specific application. For example, N-version programming reveals the presence of faults automatically by comparing the results of executing redundant equivalent code fragments; the result is chosen with a majority vote between the different executions, and therefore amounts to an implicit oracle. On the other hand, recovery-blocks require explicit oracles that check for the correctness of the results to trigger alternative computations [Ran75].

Faults. Redundancy may be more or less effective depending on the types of faults present in the system. In our taxonomy, we indicate the primary class of faults addressed by each mechanism. In particular, we refer to the fault taxonomy proposed by Avizienis et al. and revised by Florio et al. [ALRL04; FB08]. Avizienis et al. distinguish three main classes of faults: *physical*, *development*, and *interaction* faults. Physical faults are hardware faults caused either by natural phenomena or human actions. Examples of physical faults are an unexpected loss of power, or a physical damage to the hardware. Development faults are introduced during the design of the system. Incorrect algorithms or design bottlenecks are some examples of development faults. Interaction faults derive from the incorrect interaction between the system and the environment. For example, incorrect settings that cause bad interactions with the system or malicious actions that aim to violate the system security. In our taxonomy, we focus on software faults only, and thus we consider development and interaction faults, and not physical faults that are related to hardware problems. We further distinguish development faults that consistently manifest themselves under well-defined conditions ("Bohrbugs") from development faults that cause software to exhibit non-deterministic behavior (Heisenbugs) [GT07; Gra86], and we refer to interaction faults introduced with malicious objectives [ALRL04].

The following sections provide a classification of several techniques that aim to either test software or to handle faults at runtime. This classification is done according to the taxonomy listed in Table 2.1. Table 2.2 summarizes the classification of the fault handling techniques that rely on redundancy, while Table 2.3 summarizes the classification of the software testing techniques.

2.2 Deliberate Redundancy

Deliberately adding redundancy is common practice in the design of computer systems at every level, from single registers in a processor to entire components in a computer, and even to entire computers in a data center. In this section, we survey software testing and fault-handling techniques that deliberately introduce redundancy into software systems at the code, data, and environment levels.

	Intention	Type	Oracle	Faults
N-version programming [Avi85; Dob06; LMX05; GPSS04]	deliberate	code	reactive implicit	development
Recovery blocks [Ran75; Dob06]	deliberate	code	reactive explicit	development
Self-checking programming [LBK90; Dob06; YC75]	deliberate	code	reactive expl./impl.	development
Self-optimizing code [DMM04; NG07]	deliberate	code	reactive explicit	development
Exception handling, rule engines [Cri82; Goo75; Cab09; BGP07; MMP06; FM06]	deliberate	code	reactive explicit	development
Wrappers [PRRS01; CMP09; DPT09; SRFA99; FX01; FO07]	deliberate	code	preventive	Bohrbugs malicious
Data structure repair [DR03; EGSK07; HC10]	intrinsic	code	reactive explicit	development
Robust data structures [TMB80; CPW72]	deliberate	data	reactive implicit	development
Data diversity [AK88]	deliberate	data	reactive expl./impl.	development
Data diversity for security [NTEK+08]	deliberate	data	reactive implicit	malicious
Rejuvenation [GHKT96; HKKF95; GT07]	deliberate	environment	preventive	Heisenbugs
Environment perturbation [QTZS07; NBZ07]	deliberate	environment	reactive explicit	development
Process replicas [CEF+06; BCL07]	deliberate	environment	reactive implicit	malicious
Dynamic service substitution [STN+08; TBFM06; SM05; MB08]	intrinsic	code	reactive explicit	development
Fault fixing using genetic programming [WTNF09; AY08; DW10]	intrinsic	code	reactive explicit	Bohrbugs
Deviation from specifications [DZM09; WPF+10; PKL+09]	intrinsic	code	reactive expl./impl.	devel., malicious
Healing multithreaded programs [KLNB+09; NBTU08]	intrinsic	environment	reactive explicit	Heisenbugs
Checkpoint-recovery [EAmWJ02]	intrinsic	environment	reactive explicit	Heisenbugs
Reboot and micro-reboot [CKZ+03; Zha07]	intrinsic	environment	reactive explicit	Heisenbugs

Table 2.2. A taxonomy of redundancy for fault tolerance and fault handling techniques.

	Intention	Type	Faults
Pseudo-oracle [Wey82]	deliberate	code	development
Assertion and contract based oracle [MFC+09; PLEB07; CL02]	deliberate	code	development
Symmetric testing [DF94; Got03]	intrinsic	code	development
Metamorphic testing [CKTZ03; CCY98; MSK09]	intrinsic	data	development
Testing multithreaded programs [EFN+02]	intrinsic	environment	Heisenbugs

Table 2.3. A taxonomy of redundancy for testing techniques.

2.2.1 Deliberate Code Redundancy

Deliberate software redundancy has been widely exploited at the code level. Classic techniques explore the use of N-version programming and recovery-blocks to tolerate software faults. Similarly, pseudo-oracles and assertion based oracles were introduced to reveal faults. Other techniques introduced the concepts of self-checking and self-optimizing programming to overcome a wider variety of faults as well as performance issues. Recently some techniques proposed various forms of registries to identify healing procedures, mostly in the context of BPEL processes. A different form of deliberate code redundancy, defined in various contexts, is represented by wrappers. Wrappers add redundant code to detect and correct interaction problems such as incompatibilities of formats or protocols between software components.

N-version programming. The technique was originally proposed by Avizienis et al., and is one of the classic techniques to design fault tolerant software systems [Avi85]. N-version programming relies on several programs that are designed independently and executed in parallel. The results are compared to identify and correct wrong outputs. The multiple versions must differ as much as possible in the use of design and implementation techniques, programming languages, and tools. A general voting algorithm compares the results, and selects the final one based on the output of the majority of the programs. Since the final output needs a majority quorum, the number of programs determines the number of tolerable failures: a three-versions system can tolerate at most one faulty result, a five-versions system can tolerate up to two faulty results, and so on. In general, in order to tolerate k failures, a system must consists of $2k + 1$ versions.

The original N-version mechanism has been extended to different domains, in particular recently to the design of Web- and service-based applications. Looker et al. define *WS-FTM*, a mechanism that supports the parallel execution of several independently-

designed services. The different services implement the same functionality, and their results are validated on the basis of a quorum agreement [LMX05]. Dobson implements N-version programming in WS-BPEL, by executing services in parallel, and by having a voting algorithm on the obtained responses [Dob06]. Gashi et al. describe and evaluate another typical application of N-version programming to SQL servers [GPSS04]. In this case, N-version programming is particularly advantageous since the interface of an SQL database is well defined, and several independent implementations are already available. However, reconciling the output and the state of multiple, heterogeneous servers may not be trivial, due to concurrent scheduling and other sources of non-determinism.

N-version programming is a relevant instance of *deliberate, code-level redundancy*, since it requires the design of different versions of the same program. The technique relies on a general built-in consensus mechanism, and does not require explicit oracles: The voting mechanism detects the effects of faults by comparing the results of the program variants, and thus acts as a *reactive, implicit* oracle. N-version programming has been investigated to tolerate development faults, but, if used with distinct hardware for the different variants, it can tolerate also some classes of physical faults. This makes the technique particularly appealing in domains like service-oriented applications, where services are executed on different servers and may become unavailable due to server or network problems.

Recovery-blocks. This technique was originally proposed by Randell, and relies on the independent design of multiple versions of the same components [Ran75]. Differently from N-version programming, in this case the various versions are executed sequentially instead of in parallel. When the running component fails, the technique executes an alternate (redundant) component. If the alternate component fails as well, the technique selects a new one, and in the case of repeated failures, this process continues as long as alternate components are available. The recovery-blocks mechanism detects failures by running suitable acceptance tests, and relies on a rollback mechanism to bring the system back to a consistent state before retrying with an alternate component.

As for N-version programming, the core ideas behind recovery blocks have been extended to different domains, and in particular to Web- and service-based applications. In their work that extends N-version programming to WS-BPEL, Dobson exploits also the BPEL *retry* command to execute an alternate service when the current one fails [Dob06]. As in the classic recovery-block technique, alternate services are statically provided at design time.

The recovery-blocks technique is another classic implementation of *deliberate code-level redundancy*, since it relies on redundant designs and implementations of the same functionality. However, recovery blocks differ from N-version programming in that they rely on *reactive, explicit* oracles to detect failures and trigger recovery actions. In fact,

recovery blocks detect component failures by executing explicitly-designed acceptance tests. Like N-version programming, recovery-blocks target development faults, but unlike N-version programming, they are not ideal for physical faults, as they do not exploit parallel execution.

Self-checking programming. Further extending the main ideas of N-version programming and recovery blocks, Laprie et al. proposed self-checking programming, which is a hybrid technique that augments programs with code that checks its dynamic behavior at runtime [LBK90]. A self-checking component can be either a software component with a built-in acceptance test suite, or a pair of independently designed components with a final comparison. Each functionality is implemented by at least two self-checking components that are designed independently and executed in parallel. If the main self-checking component fails, the program automatically checks the results produced by the alternative component to produce a correct result. At runtime, components are classified as "acting" components, which are in charge of the computation, and "hot spare" components, which are executed in parallel to tolerate faults of the acting components. An acting component that fails is discarded and replaced by a hot spare. This way, self-checking programming does not require any rollback mechanism, which is essential with recovery blocks. The core idea of self-checking software goes back to 1975, when Yau et al. suggested software redundancy to check for the correctness of system behavior in order to improve system reliability [YC75].

Similarly to previous techniques, Dobson applies also the self-checking programming technique to service oriented applications, by calling multiple services in parallel and considering the results produced by the hot spare services only in case of failures of the acting one [Dob06].

Self-checking programming is yet another example of *deliberate code-level redundancy*, since it is based on redundant implementations of the same functionalities. Self-checking programming uses *reactive* oracles that can be *implicit* or *explicit* depending on the design of the self-checking components. Components with a built-in acceptance test suite implement reactive, explicit oracles, while components with a final comparison of parallel results implement reactive, implicit oracles. Similarly to N-version programming and recovery blocks, self-checking programming has been introduced to tolerate *development faults*.

Pseudo-oracles. N-version programming principles have been explored in software testing as well. In fact, as multiple versions of the same component can be used to mask the presence of faults, they can be used first of all to reveal the presence of faults in one of the versions. Back in 1982, Weyuker described the problem of testing "non-testable" programs, that is programs for which either an oracle does not exist, or it is theoretically possible, but practically too difficult to compute [Wey82]. Among the solutions she proposed is the idea of pseudo-oracles, that is to have an independently

written program intended to fulfill the same specifications of the program under test. Running both programs on the same input data may reveal the presence of faults if the independently computed outputs are not identical. More precisely, to automatically identify the faulty program at least three versions are required.

Since this technique shares the same principles of N-version programming, it is yet another example of *deliberate code* redundancy, and as N-version programming, it is meant for detecting any *development* fault.

Self-optimizing code. Development faults may affect non-functional properties such as performance. The term self-optimization, used within the larger study of self-managed systems, refers to an automatic reaction of a system that would allow it to compensate for and recover from performance problems. Some techniques to self-optimization rely on redundancy. Diaconescu et al. suggest implementing the same functionalities with several components optimized for different runtime conditions. Applications can adapt to different performance requirements and execution conditions at runtime by selecting and activating suitable implementations for the current contexts [DMM04].

Naccache et al. exploit a similar idea in the Web services domain [NG07]. They enhance Web service applications with mechanisms that choose among several implementations of the same service interfaces depending on the required quality of service. To maintain the required performance characteristics in Web services applications, the framework automatically selects a suitable implementation among the available ones.

These self-optimizing techniques *deliberately* include *code* redundancy. In fact the presence of different components and Web services at design time is required to allow these frameworks to work at runtime. The oracles are *reactive and explicit*, since the frameworks monitor the execution and, when the quality of service offered by the application falls below a given threshold, select another component or service.

Exception handling and rule engines (Registries). Exception handling is a classic mechanism that catches pre-defined classes of errors and activates recovery procedures (exception handlers) explicitly provided at design time [Goo75; Cri82].

Rule engines extend classic exception handling mechanisms by augmenting service-based applications with a *registry* of rule-based recovery actions. The registry is filled by developers at design time, and contains a list of failures each one with corresponding recovery actions to be executed at runtime. Both Baresi et al. [BGP07] and Pernici et al. [MMP06] propose registry-based techniques. They enhance BPEL processes with rules and recovery actions. In both cases, failures are detected at runtime by observing violations of some predetermined safety conditions, although the two techniques differ in the way they define rules and actions.

Recently Cabral proposed the automatic exception handling technique, which improves the classic exception handling mechanism by automating some basic recovery

actions [Cab09]. His studies suggest that many exceptions can be handled with recovery actions that are not application specific, and can thus be executed automatically. For instance, if a `DiskFullException` occurs at runtime, removing temporary files on the hard drive can solve the problem. Relying on pre-defined recovery actions can ease the developers task of dealing with non application specific exceptions.

Mechanisms that rely on exception handlers and registries add redundant *code deliberately*, and rely on *explicit oracles*, which are managed as exceptions. Recovery actions address *development faults*.

Wrappers. The term *wrapper* indicates elements that mediate interactions between components to solve integration problems. Wrappers have been proposed in many contexts. Popov et al. propose wrappers in the context of the design of systems that integrate Commercial Off-The-Shelf (COTS) components to cope with integration problems that derive from incomplete specifications [PRRS01]. Incompletely specified COTS components may be used incorrectly or in contexts that differ from the ones they have been designed for. The wrappers proposed by Popov et al. detect classic mismatches and trigger appropriate recovery actions, for example they switch to alternative redundant components. Chang et al. require developers to release components together with sets of healing mechanisms that can deal with failures caused by common misuses of the components [CMP08; CMP09]. Failure detectors and so-called healers are designed as exceptions that, when raised, automatically execute the recovery actions provided by the developers. Similarly, Denaro et al. apply adapters, provided by developers, to avoid integration problems among Web services [DPT09].

Salles et al. propose wrappers for off-the-shelf components for operating systems. With wrappers, Salles et al. improve the dependability of OS kernels that integrate COTS components with different dependability levels [SRFA99]. Fetzer et al. introduce "healers" to prevent some classes of malicious faults [FX01]. Healers are wrappers that embed all function calls to the C library that write to the heap, and perform suitable boundary checks to prevent buffer overflows.

Fuad et al. [FO07; FDO06] present a technique that adds self-healing abilities in non self-healing systems by introducing wrappers to deal with runtime failures. Failures that cannot be handled at runtime cause the termination of the application, and produce a log file that can help the administrators implement a wrapper to handle the future occurrences failures.

Wrappers *deliberately* insert redundant *code* to *prevent* failures. They have been proposed to deal with both *Bohrbugs* and *malicious* attacks.

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

Demsky et al. were the first to develop a framework that allows developers to specify consistency constraints for data structures. They rely on these constraints to

automatically detect and repair constraint violations at runtime [DR03; DR05]. Later they improved their work by integrating their framework with Daikon in order to automatically infer likely consistency constraints [DEG+06; ECGN01].

Elkarablieh et el. presented a technique similar to the one proposed by Demsky et al., and they implemented it in a tool called Juzi [EGSK07; EK08]. The technique relies on program assertions to automatically detect consistency constraints in complex data structures, and it exploits dynamic symbolic execution to automatically repair the fault. Later, Hussain et al. presented an improved technique that similarly relies on dynamic symbolic execution to repair complex data structures. Thanks to their improvements, this technique can handle generic repairs, instead of focusing on a few types of faults as the previous work [HC10].

These techniques check data structures consistency relying on constraints, which are redundant specifications that have been added to *code deliberately*. All these techniques rely on *reactive, explicit* oracles to trigger the fixing algorithm, and they all focus on *Bohrbugs*.

Assertion and contract-based oracles. Assertions and contracts are partial specifications that can be added to code to verify properties at runtime [Mey88; Ros95]. They are redundant in respect to code, and they are added deliberately. Many unit-testing techniques rely on them to judge the correctness of the execution of a test case. Several unit-testing frameworks such as JUnit, which can automate the test cases execution [JUn11], are very popular. Junit can automate part of the testing process. However, the most critical parts, that is, writing the test cases and the assertions, have to be done manually by the testers. Thus, several techniques propose solutions to completely automate the generation of unit test cases. Cheon et al., for instance, propose a technique that uses formal specifications provided by the developers in the form of pre- and post- conditions to generate test oracles. This technique combines the JML runtime assertion checker with JUnit to execute Java methods, and automatically decides whether they are working correctly [CL02].

Similarly, Randoop generates random unit tests for Java classes by taking as input a set of classes under tests and a set of contracts. Randoop tries to generate a test suite containing contract-violating tests that exhibit scenarios where the code under test leads to the violation of an API contract, thus revealing a fault either in the implementation or in the contract [PLEB07; PE07].

Autotest is yet another technique that works in this direction. It automates the generation of test cases and oracles, by using Eiffel contracts, which are already present in the software, as test oracles, and by generating objects and routine arguments to exercise all given classes. Eiffel contracts specify the expected behavior of a class, which can be monitored during execution [MFC+09; MCLL07].

As for data structure repair, these testing techniques rely on assertions and contracts, which are redundant specifications that have been *deliberately* added to *code*.

These techniques can be effective in revealing *Bohrbugs*.

Costs and efficacy of code redundancy. As our survey shows, deliberate code redundancy has been exploited primarily to cope with development faults, and has been recently extended to cope with performance and security faults. Different techniques try to mitigate the additional design and execution costs by trading recovery and oracle design costs for execution costs. N-version programming and pseudo-oracles come with high design and execution costs, but work with inexpensive and reliable implicit oracles. Recovery blocks reduce execution costs, but increase the cost of designing oracles. Self-checking components support a flexible choice between the two techniques at the price of complex execution frameworks.

Software execution progressively consumes the initial explicit redundancy, since failing elements are discard and substituted with redundant ones. The efficacy of explicit redundancy is controversial. Supporters of explicit redundancy recognize the increased reliability of properly designed redundant systems [Hat97]. Detractors provide experimental evidence of the limited improvements of the reliability of redundant over non redundant systems. For example, Brilliant et al. indicate that, in N-version programming, the amount of faults increases unexpectedly, and the correlation is higher than predicted, thus reducing the expected reliability gain [KL86; BKL90].

2.2.2 Deliberate Data Redundancy

To overcome the limitations of code redundancy, and in particular to avoid the costs of developing several versions of the same components, redundancy has been deliberately added to both data and, more recently, the runtime environment. Deliberate data redundancy has been proposed to increase the dependability of data structures, to reduce the occurrences of failures caused by specific input-dependent conditions (e.g., corner cases in data structures) and very recently to cope with some classes of security problems.

Robust data structures and software audits. Connet et al. introduce a preliminary form of data redundancy in the early seventies [CPW72]. Their technique augment systems with so called software audits that check for the integrity of the system itself at runtime. Taylor et al. exploited a form of deliberate data redundancy to improve the reliability of data structures [TMB80]. Taylor et al. propose data redundancy consisting of additional code to trace the amount of nodes in data structures and of additional node identifiers and references to make data structures more robust. Their approach uses the redundant information to identify and correct faulty references. These techniques exploit *data* redundancy that is *deliberately* added to the programs to tolerate *development* faults. The redundant information implicitly enables failures detection, thus oracles are *reactive and implicit*.

Data diversity. Knight et al. apply deliberate data redundancy to cope with failures that depend on specific input conditions [AK88]. Knight's technique is effective with software that contains faults that result in failures with particular input values, but that can be avoided with slight modifications of the input. The technique relies on data "re-expressions" that can generate logically equivalent data sets. Re-expressions are *exact* if they express the same input in a different way, thus producing the expected output; They are *approximate* if they change the input and thus produce a different output but within an accepted range. Data diversity is implemented in the form of either "retry blocks" that borrow from the idea of recovery blocks, or "N-copy programming" that redefine N-version programming for data. Therefore, data diversity can work with both *reactive* and *implicit* oracles. As for recovery blocks and N-version programming, data diversity addresses development faults.

Data diversity for security. Recently Knight et al. extended the conceptual framework of data diversity to cope with security problems [NTEK+08]. They apply data diversity in the form of N-variant systems to provide high-assurance "conjectures" against a class of data corruption attacks. Data are transformed into variants with the property that identical concrete data values have different interpretations. In this way attackers would need to alter the corresponding data in each variant in a different way while sending the same inputs to all variants. The only available implementation runs in parallel on the different data sets, and executions are compared. Thus this technique relies on *data* redundancy *deliberately* added to tolerate *malicious* faults. Since the technique relies on the parallel execution and the comparison of results, the oracle is *implicit*.

Despite early attempts tracing back almost 30 years, deliberate data redundancy has not been exploited as thoroughly as code redundancy. In particular it has been exploited only to tolerate faults, and not to reveal them with software testing. Most techniques focus on development faults. Recent work indicates space for applications to non-functional faults as well.

2.2.3 Deliberate Environment Redundancy

Deliberate environment redundancy is the most basic form of redundancy, and has been used extensively to increase reliability in the face of purely hardware faults, for example in the case of database replication. Deliberate environment redundancy consists of deliberately changing the environment conditions and re-executing the software system under the new conditions. Thus, this form of redundancy impacts on the program execution rather then on the program structure. We only mention this well-known and widely studied application of environment redundancy in passing here because we intend to focus specifically on software faults. Therefore we describe in detail

only some more recent uses of environment redundancy that are more significant for this class of faults.

Rejuvenation. The first notable attempt to deliberately modify the environment conditions to avoid failures tracks back to the nineties, when Wang et al. proposed software rejuvenation. Rejuvenation is a technique that works with environment diversity, and relies on the observation that some software systems fail due to "age," and that proper system re-initializations can avoid such failures [WHV+95; GT07]. Wang et al. focused on memory-management faults, such as memory leaks, memory caching, and weak memory reuse, that can cause the premature termination of the program execution. Rejuvenation amounts to cleaning the volatile state of the system periodically, whenever it does not contain useful information. The same research group improved software rejuvenation by combining it with checkpoints: By rejuvenating the program every *N* checkpoints, they can minimize the completion time of a program execution [GHKT96].

Software rejuvenation is a *deliberate* change to the *environment*, since the memory state is cleared intentionally by re-executing some global initialization procedures, thereby presenting a new environment to the system. Rejuvenation acts independently from the occurrence of failures, thus it can be both reactive or preventive. However it does not rely on an oracle that explicitly identifies a failure, and thus we classify it as *preventive* from the oracle viewpoint. Techniques such as software rejuvenation work well for *Heisenbugs*.

Environment perturbation. Using the analogy of allergies in humans and other animals, and specifically of the treatment of such allergies, Qin et al. suggested a rollback mechanism, called Rx, that partially re-executes failing programs under modified environment conditions to recover from deterministic as well as non deterministic faults [QTZS07; QTSZ05]. The mechanism is based on environment changes that include different memory management strategies, shuffled message orders, modified process priority, and reduced user requests. These changes in the execution environment can prevent failures such as buffer overflows, deadlocks and other concurrency problems, and can avoid interaction faults often exploited by malicious requests.

Similarly, Exterminator dynamically patches buffer overflows by allocating extra memory, and dangling pointers by deferring object deallocations [NBZ07]. Differently from Rx, however, it detects failures by relying on a probabilistic debugger, thus it does not come with the cost of providing an explicit oracle.

As software rejuvenation, environment perturbation is based on *deliberate* environment redundancy, since the applied changes explicitly create different environments where the programs can be re-executed successfully. However, contrary to rejuvenation, these techniques rely on *reactive* oracles to start proper recovery actions. In particular, the environment changes in Rx are triggered by *explicit* exceptions or by

sensors that monitor the system execution. Exterminator, instead, relies on *implicit* oracles provided by its probabilistic debugger. These techniques work mainly with *Heisenbugs*, but can be effective also in the presence of *Bohrbugs* and *malicious* faults.

Process replicas. The main concepts of N-version programming have been extended to environment changes to cope with malicious faults. Cox et al. propose to execute N-variants of the same code under separate environment conditions and compare their behavior to detect malicious attacks [CEF+06]. The aim is to complicate the attackers' task by requiring malicious users to simultaneously compromise all system variants with the same input to actually succeed in the attack. The framework provided by Cox starts from the original program, and automatically creates different variants by partitioning the address space, and by tagging the instructions. Partitioning the address space can prevent memory attacks that involve direct reference to absolute addresses, while tagging the instructions (that is, prepending a variant-specific tag to all instructions) can detect code injection. Bruschi et al. improve Cox' process replicas with a new mechanism that also detects attacks that attempt to overwrite memory addresses [BCL07].

Techniques based on process replicas *deliberately* add redundancy to the execution *environment*, since the variants are obtained through explicit, though automatic, changes. The tagging mechanism proposed by Cox acts on the program, and thus also creates redundancy in the code. Process replicas do not require explicit oracles, but instead rely on *reactive, implicit* mechanisms in the same way that N-version programming derives a single output value, by executing the variants in parallel, and then by comparing execution results at runtime. Process replicas target *malicious faults*, and do not seem well suited to deal with development faults.

In general, deliberate redundancy in the environment execution has been exploited only recently, and only to tolerate faults. This type of redundancy seems well suited to deal with Heisenbugs and some classes of interaction faults, especially malicious faults, that are particularly difficult to detect and remove.

2.3 Intrinsic Redundancy

While deliberate redundancy has been exploited since the seventies and in many contexts, implicit redundancy has been explored only recently, with some promising results. Implicit redundancy at the code level usually stems from the complexity of system modules, which in turn result in a partial overlap of functionality within different program elements. Implicit redundancy at the environment level comes from the complexity of the execution environment, which typically does not behave deterministically to all requests, and therefore may allow for different but functionally equivalent behaviors.

2.3.1 Intrinsic Code Redundancy

Implicit redundancy at the code level has been exploited both in specific application domains, mostly in the context of service oriented applications and more general of dynamically bound components, and with specific technologies, namely genetic programming. Implicit code redundancy has been explored in both software testing and fault handling.

Dynamic service substitution. Popular services are often available in multiple implementations, each one designed and operated independently, each one also possibly offering various levels of Quality of Service, but with every one complying with an equivalent, common interface. In fact, this is more or less the vision of service-oriented computing. Some researchers propose to take advantage of the available, independent implementations of the same or similar service to increase the reliability of service-oriented applications, especially for failures that may be caused by malfunctioning services or by unforeseen changes in the functionalities offered by the current implementation. Subramanian et al. enhance BPEL with constructs to find alternative service implementations of the same interfaces in order to overcome unpredicted response or availability problems [STN+08]. Taher et al. enhance runtime service substitution by extending the search to services implementing similar interfaces, and by introducing suitable converters to use services that, although different, are sufficiently similar to admit to a simple adaptation [TBFM06]. Sadjadi et al. further simplify the substitution of similar service implementations by proposing transparent shaping to weave alternative services invocation at runtime, thus avoiding manual modification of the original code [SM05]. Mosincat and Binder define an infrastructure to handle dynamic binding of alternative services that can handle both stateless and stateful Web services [MB08].

In summary, service substitution amounts to exploiting available redundant *code* that is *intrinsically* available. The substitution is triggered in reaction to faults thanks to *explicit oracles*, and allows systems to tolerate both *development* faults and physical faults.

Fault fixing using genetic programming. Recently both Weimer et al. and Arcuri et al. investigated genetic programming as a way of automatically fixing software faults [WTNF09; FNWLG09; AY08]. Both techniques assume the availability of a set of test cases to be used as oracle. When the software system fails, the runtime framework automatically generates a population of variants of the original faulty program. Genetic algorithms evolve the initial population guided by the results of the test cases that select a new “correct” version of the program. As Weimer et al. and Arcuri et al. techniques apply mutations to the faulty statements by copying similar statements defined elsewhere in the code, Wong et al., who lately defined a similar technique, rely

on a set of predefined mutation operators [DW10].

Genetic programming does not require the deliberate development of redundant functionality, but exploits the *intrinsic* redundancy of *code* to produce variants of the original programs, and selects a "correct" variant. Genetic techniques react to failures detected by test suites, and thus rely on *reactive and explicit* oracles to identify and correct *Bohrbugs*.

Deviation from specifications or previously observed behavior. Several recent techniques rely on either specifications or previously observed behavior to fix faults dynamically. Dallmeier et al. developed a tool called Pachika that automatically builds behavioral models of a set of passing and failing test cases of Java classes [DZM09; DLWZ06]. By comparing the model of the passing runs to the model of the failing run, Pachika tries to come up with possible fixes that amount to modifications of the model of the failing runs to make it compatible with the model of the passing runs. However, the supported modifications are limited to inserting and removing transitions from the finite state model, thus only some types of faults can be fixed.

Following the same idea, Wei et al. rely on Autotest to automatically detect failures in Eiffel classes, and they propose automatic fixes by comparing the outcome of the passing runs to the outcome of the failing run [MFC$^+$09; WPF$^+$10]. They use class specifications expressed in the form of contracts, that consist of preconditions, post-conditions, assertions and invariants as oracles. As for Pachika, they build finite state machines representing the correct and the failing executions, and by comparing them they propose fixes. Since they can support more complex modifications to the faulty classes, they can successfully deal with more types of faults than Pachika.

Although with a different target, ClearView is similar to the previous techniques in the sense that it relies on dynamic analysis to detect failures and fix faults at runtime. This framework relies on dynamically inferred invariants computed with Daikon to automatically detect buffer overflows and illegal control flows caused by malicious code injections [PKL$^+$09; ECGN01]. After an invariant violation, ClearView releases a patch that enforces the invariant to hold.

All the described techniques *intrinsically* exploit *code* redundancy, because they are changing the code in the application to reproduce a behavior that has been already observed in previous correct executions. ClearView is the only technique that relies on *reactive implicit* oracles to trigger the fixing operation. All other techniques need *reactive explicit* oracles. This class of fault handling techniques can deal with all the types of faults that span from development to interaction.

Symmetric Testing. Doong and Frankl first, and Gotlieb later have proposed to check the correctness of test cases execution by exploiting the symmetries that are intrinsically present in programs. Doong and Frankl presented ASTOOT, a technique that relies on algebraic specifications of components to generate self-checking test cases.

Each test case consists of a pair of sequences of operations on the same component, and a tag that specifies if the sequences are expected to have the same effects on the component. Tests are then executed by invoking these sequences of operations, and by running an equivalence check that is provided by the tester [DF94].

Similarly, Gotlieb exploits symmetries of the program to automatically check the test cases correctness. Differently from ASTOOT, however, they consider operation permutations as the only possible symmetry to exploit. They automatically generate test inputs to run the test cases, and they check the symmetries provided by the user to decide whether the test case execution is correct [Got03]

Techniques that rely on symmetries in programs for testing purposes exploit *intrinsic code* redundancy, as they rely on redundancy that has not been intentionally included in the application under test. These techniques are suitable for *development* faults.

2.3.2 Intrinsic Data Redundancy

Intrinsic data redundancy has not yet been explored thoroughly. In fact, only metamorphic testing techniques exploit this type of redundancy to assess the quality of software.

Metamorphic testing. Similarly to symmetric testing, metamorphic testing exploits the symmetries in programs to automatically generate test cases that do not require explicit oracles. However, while symmetric testing techniques exploit code redundancy, these techniques rely on data redundancy, and for this reason they are closely related to data diversity [AK88].

Given an original input, the objective of metamorphic testing is to use alternate means of computing the same input using the same function, and run the function under test on both inputs expecting the same results. As an example, consider the sine function. For any two inputs x_1 and x_2 such that $x_1 + x_2 = \pi$, we know that $sin(x_1) = sin(x_2)$ holds. This property can be exploited in metamorphic testing by creating a second input $x_2 = \pi - x_1$ from any provided real number x_1, and by then executing the sin function on both inputs expecting the same result [CCY98; CTZ03; CKTZ03; MSK09].

These techniques exploit *intrinsic data* redundancy to automate the testing activity. As data diversity, these techniques work well for revealing development faults. However their effectiveness is limited to some classes of applications (e.g., mathematical libraries).

2.3.3 Intrinsic Environment Redundancy

Techniques that exploit intrinsic redundancy in the environment extend and formalize the common experience of non-deterministic behavior of the execution environment:

Simple system reboots are often used as last resource to overcome unexpected failures.

Testing and healing of multithreaded programs. Concurrency problems are usually hard to detect and reproduce, because they are nondeterministic. Edelstein et al. propose a technique that addresses this problem by exploiting environment redundancy [EFN+02]. More precisely they can reveal concurrency faults such as race conditions and deadlocks in Java programs by randomly inserting `sleep()` invocations in existing test suites, and by checking that the test execution produces the same result. The `sleep()` operation causes the current thread to suspend execution for a specified period, and it does not affect the functional behavior of the Java application if locks are defined properly.

The same research group later realized that the same idea can be exploited to heal a Java application from detected concurrency problems, too. In fact, if a deployed and running application is affected by either race conditions or deadlocks, it is possible to automatically insert a sleep statement in the proper place in the code to avoid failures [NBTU08; KLNB+09].

These techniques exploit *environment* redundancy that is *intrinsically* present in Java applications to handle a particular type of *development* faults, that is concurrency faults. The healing phase is triggered by *explicit reactive* oracles.

Checkpoint and recovery. Checkpoint and recovery techniques periodically save consistent states to be used later as safe roll backs [EAmWJ02]. When the system fails, it is brought back to a consistent state and re-executed, to solve temporary problems that may have been caused by accidental, transient conditions in the environment.

These techniques exploit *intrinsic* redundancy in the *environment*, since a system would re-execute the same code without trying to modify the environment, but instead relying on spontaneous changes in the environment to avoid the conditions that created the failure. Notice that this is different from other explicit techniques, such as that of the Rx method by Qin et al. that deliberately changes the environment before re-executing the code [QTZS07]. Checkpoint-and-recovery requires *reactive and explicit* oracles to determine if the system has failed, and therefore to roll back to a consistent state. This technique is effective in dealing with *Heisenbugs* that depend on transient execution conditions, but does not work well for *Bohrbugs* that persist in the code and in the execution environment.

Reboot and micro-reboot. The classic brute-force but surprisingly effective technique of simply rebooting the systems was refined by Candea et al., who propose local micro-reboots to avoid the high costs of complete reboots [CKZ+03]. Zhang et al. extended the same technique to service-based applications [Zha07]. Although intuitively simple, micro-reboots require a careful modular design of the systems as well as an adequate runtime support for reboot operations that do not affect the overall execution. These

techniques exploit *intrinsic* redundant behavior of the execution environment to overcome *Heisenbugs*. As in the other cases, they operate when triggered by *reactive, explicit* oracles, since they react to system failures explicitly notified by the oracles.

2.4 Techniques to Identify Redundancy

Although redundancy has been used in several ways and for different purposes, there are a few existing techniques that aim to identify redundancy in software. Most of the work done so far focuses on the identification of *code clones*, that is, on the identification of syntactically equivalent (and almost equivalent) code snippets.

Little work has been done to identify semantic equivalence, instead. On the other hand, several types of specifications can be useful to identify semantically equivalent operations. Examples of such specifications include finite state machines, program invariants, and algebraic specifications. Thus, any technique that can infer these specifications can be useful to identify semantically equivalent operations. Here we briefly survey some techniques that can infer finite state machines, programs invariants, and algebraic specifications thanks to dynamic analysis.

Code clone detection. Redundancy is not always a good thing, especially when it comes in the form of code clones. Clones in code are mainly caused by the bad practice of developers copying and pasting code snippets to quickly duplicate functionalities. Several studies show that, on average, 10% of code in software is cloned [KSNM05]. Since code clones make the software harder to maintain, several techniques have been proposed to detect syntactically equivalent and similar code snippets.

Code clone detection techniques can be classified among the ones that rely on parametrized string matching algorithms to identify code sections that are equivalent except for a systematic change of parameters [Bak95; Bak97], techniques that split code into tokes, and then scan them looking for duplicated token subsequences [KKI02; LLMZ04], and finally techniques that parse programs as abstract syntax trees and look for similar sub-trees [BYM+98; JMSG07; KDM+96; MLM96; WSGF04].

Code clones detection techniques can identify *intrinsic code* redundancy. In fact developers do not deliberately insert code duplicates in code, but it is just a consequence of a potentially bad programming practice.

Discovery of semantic equivalence via dynamic analysis. Several techniques rely on dynamic analysis to infer specifications that can represent semantic equivalence.

Henkel et al. discover algebraic specifications of Java classes that implement containers. More precisely they can infer axioms that express which operations, or sequences of operations, are equivalent to each other [HRD07].

Similarly Dallmeier et al. infer finite state machines for Java components, and can thus represent how the component changes state after the execution of each

method [DLWZ06]. Thanks to the models they can infer, it is possible to identify the likely semantic equivalent methods in a component, albeit the high level abstraction they use may not give precise results. Ghezzi et al. can produce similar models, although with a different abstraction that makes the semantic equivalence representation more precise [GMM09]. Mariani et al. propose a slightly different technique to infer finite state machine models that represent the sequences of method invocations that have been observed at runtime [MPPar].

Although not originally with this intent, Daikon can also identify equivalent methods in Java classes. In fact, the technique by Ernst et al. can automatically infer likely method invariants and likely pre- and post-conditions. By comparing the information collected for different methods it is possible to identify semantically equivalent methods [ECGN01].

Differently from the previous techniques, Su et al. can mine functionally equivalent code fragments at different granularity levels. Instead of focusing on programs or functions as a whole, they extract code fragments from each function, they produce inputs thanks to an existing random testing technique, and they execute these fragments and compare the generated outputs to identify which fragments are equivalent [JS09].

All the techniques we mentioned rely on dynamic analysis to identify *intrinsic code* redundancy. In fact, they all look for functionally equivalent elements that developers unintentionally inserted in their code.

In this chapter we have seen how different types of redundancy can be exploited for different purposes, in particular to handle faults in deployed systems. Although most of the existing techniques rely on deliberate redundancy, many recent techniques try to exploit a form of implicit redundancy. In the next chapter we will describe a particular type of intrinsic redundancy, and we will discuss how it can be represented and exploited.

Chapter 3

Intrinsic Redundancy

Deliberate code redundancy impacts on development costs, and several studies have shown that this redundancy is not always effective, given the high probability that multiple design diverse versions of the same component fail on the same input [BKL90]. In this chapter we focus on the concept of intrinsic redundancy that is a redundancy that is naturally present in modular software. We argue that software modules usually provide multiple ways to either perform exactly the same operation, or to perform alternative operations that bring to almost the same result. We propose to exploit this type of redundancy for testing, fault localization and in particular for fault-handling.

Redundancy is widely recognized as an important means for both asserting the quality of a program during development, and for tolerating existing faults once the system is deployed. As we discussed in the previous chapter, a lot of techniques deliberately insert redundancy in code to gain reliability, despite the known problems of increasing development costs, and the known limitations of the effectiveness due to the presence of correlated faults.

However, in software engineering redundancy in code is hardly considered a positive feature, since it may cause maintenance problems, especially when developers do not add it to their code deliberately. This is the reason why techniques for code clones detection have been successful [Bak97; KKI02]. Although redundancy is not negative per se in software testing, some negative aspects of unintentionally inserting redundancy are visible in this area, too. When software evolves, so does its test suite. Testers provide new test cases for the new functionalities, and fix the old ones if necessary, but they hardly remove existing test cases, even if some of them are equivalent. As a consequence, the number of test cases keeps growing, and with limited time constraints it might be impossible to run the whole test suite for regression testing. Thus, a lot of effort has been done to identify and remove the redundant test cases, and save time for test execution [JH03; FW07].

Although recently researchers have started to investigate the possibility of exploit-

ing redundancy that has not been deliberately inserted into code, the idea that intrinsic redundancy can be used to test software and to tolerate existing faults is not widely spread yet.

The few existing studies in the software testing research limit the exploitation of intrinsic redundancy to very specific software (e.g. containers and mathematical libraries) [DF94; Got03]. Moreover, the known studies on using intrinsic redundancy to address fault-handling rely mainly on characteristics of emergent application domains, such as service-based applications. The results are encouraging, though, and suggest that implicit code redundancy should be exploited also in classic application domains to increase reliability.

We already said that the main advantage of using intrinsic redundancy is that it does not incur serious additional development costs. However, it should be clear that this approach still relies on code redundancy, although it does so *implicitly*, and its effectiveness is bound to the existence of such redundancy. So the main question is whether intrinsic redundancy actually exists. Beside all the studies on the presence of syntactically similar code fragments in software [KSNM05], some recent studies show that the presence of semantically equivalent fragments in software is quite high [GJS08; JS09]. Thus, these studies suggest that implicit redundancy does exist in real projects.

Our intuition is that redundancy can be unintentionally included in software for three main reasons. First, the modern development process induces developers to reuse third-party software components that already implement the functionalities they need. Several components provide similar functionalities, and it is quite common that developers decide to include multiple similar components in their project in order to be able to select the most suitable one depending on the situation. For instance, developers can rely on existing libraries implementing several different sorting algorithms. Depending on the situation, they may decide to use one algorithm instead of another one to improve performance. For example, in presence of small sets of elements, and in presence of lists that are already sorted, except for few elements, developers may prefer the component that implements the *bubble sort* algorithm. In other parts of the code, instead, they may prefer another component that implements the *quick sort* algorithm, instead. Thus, in this case intrinsic redundancy comes from the availability of several third-party components that implement the same or similar functionalities (i.e. sorting algorithms).

The second reason why redundancy may be unintentionally included in software is the lack of software reuse. Developers sometimes just ignore the functionalities that are already available in the project, and they might implement the same functionalities multiple times. For instance, in the same project one developer might rely on a third-party component to sort some elements, while another might implement the sorting algorithm himself because he ignores the availability of the third-party component. Similarly, two developers, or even the same developer, might implement logically

- `add(a)` Adds the element `a` to the container.
- `addAll(a1...an)` Adds elements `a1...an` to the container.
- `remove(a)` Removes the element `a` from the container.
- `removeAll()` Removes all the elements from the container.
- `isEmpty()` Returns `true` whether the container is empty.
- `size()` Returns the number of elements currently in the container.
- `toString()` Returns a string representation of the container.

Figure 3.1. Some operations of a sample container

similar functions multiple times and in different ways. The poor quality (or the lack) of documentation may increase this problem. The lack of communication and trust among developers is another cause of little software reuse, which consequently leads to the introduction of intrinsic redundancy.

Finally, the third reason for which redundancy is intrinsically present in software stands in the design of reusable components. We argue that components that are designed to be reusable and versatile have redundant interfaces to meet different needs, and they offer several ways to perform the same operations. Containers, for instance, are a typical example of reusable components, and they usually offer different ways to add items (e.g. it is possible to add one item at a time and add several items at the same time). Moreover, it is possible to obtain the same result by either adding an item a to a container or by adding items a and b to a container and removing item b right after.

In the case of reusable components, intrinsic redundancy stands within the component itself, since it implements the same or similar functionalities in multiple ways in order to meet different needs. This type of intrinsic redundancy in code is the one that is more spread, and it is the one we are mainly interested in.

Regardless of the reason, the fact is that software *is* intrinsically redundant, and software modules comprise several sequences of operations that are functionally equivalent, since their execution leads to the same effects. Considering again the sample container example, whose operations are listed in Figure 3.1, we can say that the following operations are equivalent according to the specifications:

```
addAll(a,b)  ≡  add(a); add(b)
     add(a)  ≡  addAll(a,b); remove(b)
```

Since these sequences of operations lead to the same results, we refer to them as *"equivalent sequences"*. Some sequences of operations can be equivalent to others only

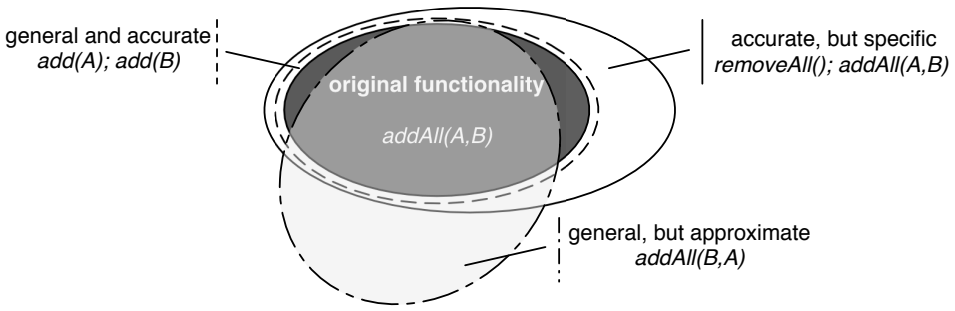


Figure 3.2. Equivalent sequences

under certain conditions, and can be equivalent to some others except for marginal details. To consider these cases we define two dimensions for the equivalent sequences, namely *generality* and *accuracy*:

- **Generality**: Equivalent sequences may represent intrinsic redundancy that is *specific* to an instance rather than *general*. In fact, some sequences of operations are equivalent only under certain conditions, that is they are specific, while others may be always equivalent, that is they are general.

- **Accuracy**: Sequences of operations can lead exactly to the same result of other operations, that is they are *accurate*. Others may be equivalent with an acceptable approximation, that is they are *approximate*.

For instance, the operation `addAll(A,B)`, which adds the elements `A` and `B` to a container, has `add(A); add(B)` as a *general* and *accurate* equivalent sequence. In fact, it is always the case that adding two elements to a container is equivalent to adding the first one, and adding the second one right after. If we assume that the order of the elements in the container is not relevant, then `addAll(B,A)` is a *general*, but *approximate* equivalent sequence. If fact, even if it might be acceptable to have the elements added to the container in a reverse order, the effect of the execution is not the same. Finally, we can also say that removing all the elements before adding `A` and `B` to the container (`removeAll();add(A,B)`) is equivalent to just adding `A` and `B`. However, this is true only when the container is initially empty. Since this equivalent sequence does not hold under all the conditions, then we say that it is *accurate*, but *specific*.

In this thesis we do not consider differences in non functional behavior. Thus, if two sequences of operations are leading to the same result, we consider them accurate equivalent sequences, even if one performs better than the other.

However, we limit the scope of this thesis to *accurate equivalent sequences*, and we leave the study of approximate equivalent sequences as future work. Thus, from now on whenever we refer to equivalent sequences we implicitly say that they are accurate. Still, we can differentiate among general and specific equivalent sequences.

In the following sections we present a component of the Apache Tomcat servlet container, and we use it as a case study (Section 3.1). We first show that this component is intrinsically redundant, as it offers a lot of equivalent sequences that can be easily identified from the Tomcat documentation. We then provide a classification of the equivalent sequences (Section 3.2), and we conclude this chapter by describing how this redundancy can be identified (Section 3.3), and how it can be exploited (Section 3.4).

3.1 Tomcat Web Applications Loader Case Study

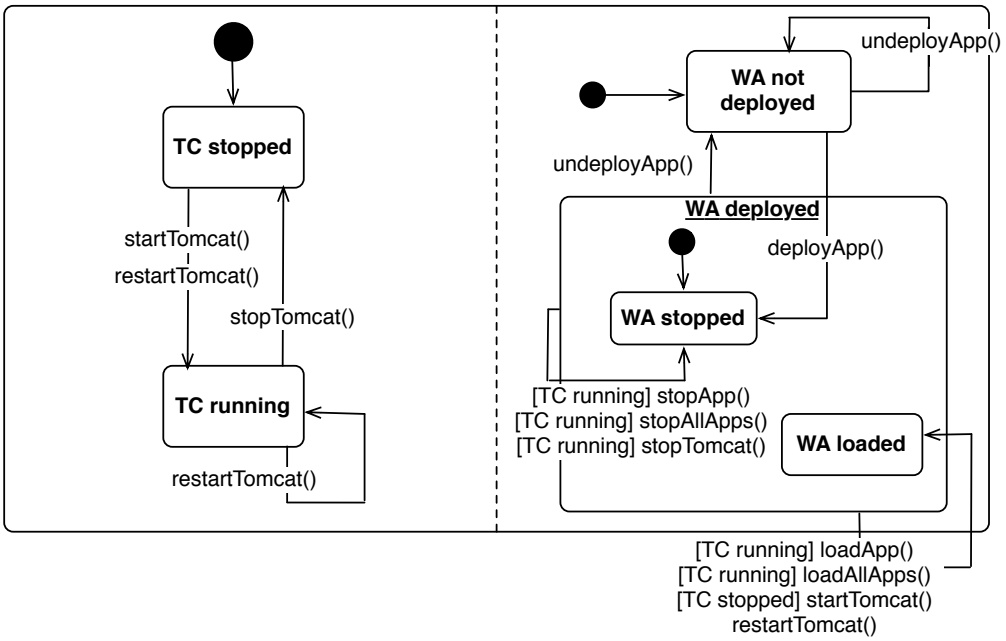
Apache Tomcat [1] is an open source Servlet and JSP container that is widely used because of its portability and stability. Moreover it can be easily integrated with Web servers through different connectors. Tomcat has been designed to be very modular, and each module deals with specific functionalities like deploying Web applications, supporting proxies, logging, and SSL support, just to mention a few.

Similarly to the sample container that we mentioned before, all these modules are intrinsically redundant, as they all provide several ways to achieve the same functionalities. In this section we focus on one of these components, that is the Web applications loader. This module is in charge of deploying, loading and stopping Web applications in Tomcat. The Statechart in Figure 3.3 is a partial specification of this module, and we modeled it following the module description on the Tomcat website. A short description of the main operations that this module implements is provided below the specification.

The Statechart shows that it is possible to deploy and load a Web application in at least two different ways, i.e. when Tomcat is running, and before its startup. In the first case the Web application has to be loaded explicitly to make it available after the deployment. In the second case, instead, it is not required to load the Web application after the deployment, since Tomcat automatically loads all the deployed Web applications at startup. Another example of intrinsic redundancy in this module is the restart operation. It is in fact possible to restart Tomcat by either invoking the restart command, or by stopping the servlet container and then by starting it again.

Intuitively, a lot of sequences of operations are semantically equivalent according to the specification of this module. Figure 3.4 lists some of them, grouped by the conditions under which they hold.

[1] `http://tomcat.apache.org`



`startTomcat()`	Start the servlet container. All deployed Web applications are loaded
`stopTomcat()`	Stop the servlet container. All running Web applications are stopped
`restartTomcat()`	The servlet container is stopped and started
`deployApp(app)`	Deploy a Web application
`undeployApp(app)`	Undeploy a Web application
`loadApp(app)`	Load a Web application
`stopApp(app)`	Stop a Web application
`loadAllApps()`	Load all deployed Web applications
`stopAllApps()`	Stop all running Web applications

Figure 3.3. Statechart specification of the Tomcat Web applications loader

Tomcat stopped and WA not deployed	
startTomcat()	≡ restartTomcat()
startTomcat()	≡ startTomcat(), stopTomcat(), startTomcat()
startTomcat()	≡ startTomcat(), restartTomcat()
startTomcat()	≡ startTomcat(), restartTomcat(), stopTomcat(), startTomcat()
restartTomcat()	≡ startTomcat()
restartTomcat()	≡ startTomcat(), stopTomcat(), startTomcat()
undeployApp()	≡ deployApp(), undeployApp()
deployApp()	≡ deployApp(), undeployApp(), deployApp()
deployApp(), startTomcat()	≡ **startTomcat(), deployApp(), loadApp()**
...	...

Tomcat running and WA not deployed	
restartTomcat()	≡ stopTomcat(), startTomcat()
restartTomcat()	≡ stopTomcat(), restartTomcat()
restartTomcat()	≡ restartTomcat(), stopTomcat(), startTomcat()
deployApp()	≡ deployApp(), undeployApp(), deployApp()
deployApp()	≡ deployApp(), loadApp()
deployApp()	≡ deployApp(), stopApp(), loadApp()
...	...

Tomcat stopped and WA deployed	
startTomcat()	≡ restartTomcat()
startTomcat()	≡ startTomcat(), stopTomcat(), startTomcat()
startTomcat()	≡ startTomcat(), restartTomcat()
startTomcat()	≡ startTomcat(), restartTomcat(), stopTomcat(), startTomcat()
startTomcat()	≡ **startTomcat(), loadApp()**
undeployApp()	≡ deployApp(), undeployApp()
...	...

Tomcat running and WA deployed	
restartTomcat()	≡ stopTomcat(), startTomcat()
restartTomcat()	≡ stopTomcat(), restartTomcat()
restartTomcat()	≡ restartTomcat(), stopTomcat(), startTomcat()
stopApp()	≡ stopAllApp()
stopApp()	≡ loadAllApp(), stopAllApp()

Figure 3.4. Some equivalent sequences derived from the Statechart in Figure 3.3

Tomcat 6.0 Issue

The Tomcat Web applications loader has been affected by a severe issue for a long time [2], from version 6.0.0 to version 6.0.10. The problem used to occur when a Web application containing a JSP file was deployed before the Tomcat server has started. Deploying the Web application after the server startup was not causing any problem, instead. However, the issue was manifesting again by stopping and restarting the server. On Tomcat startup the following error message was printed:

```
Feb 15, 2011 4:27:00 PM org.apache.catalina.core.StandardService start
INFO: Starting service Catalina
Feb 15, 2011 4:27:00 PM org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/6.0.0
Feb 15, 2011 4:27:00 PM org.apache.catalina.core.StandardHost start
INFO: XML validation disabled
Feb 15, 2011 4:27:00 PM org.apache.catalina.startup.HostConfig deployWAR
INFO: Deploying web application archive ELResolverTest.war
Feb 15, 2011 4:27:00 PM org.apache.catalina.core.StandardContext start
SEVERE: Error listenerStart
SEVERE: Context [/ELResolverTest] startup failed due to previous errors
...
Feb 15, 2011 4:27:00 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 627 ms
```

The Tomcat Manager (Figure 3.5), and the error message on the console were showing that the Web application was deployed correctly, but not loaded. The application could be loaded by explicitly invoking the `start` command though.

Looking at the log files, it was clear that the problem was due to a missing initialization, as a null pointer exception was thrown:

```
Feb 15, 2011 4:27:00 PM org.apache.catalina.core.StandardContext listenerStart
SEVERE: Exception sending context initialized event to listener instance of class eltest.ChipsListener
java.lang.NullPointerException
  at eltest.ChipsListener.contextInitialized(ChipsListener.java:18)
  at org.apache.catalina.core.StandardContext.listenerStart(StandardContext.java:3826)
  at org.apache.catalina.core.StandardContext.start(StandardContext.java:4335)
  at org.apache.catalina.core.ContainerBase.addChildInternal(ContainerBase.java:759)
  at org.apache.catalina.core.ContainerBase.addChild(ContainerBase.java:739)
  at org.apache.catalina.core.StandardHost.addChild(StandardHost.java:524)
  at org.apache.catalina.startup.HostConfig.deployWAR(HostConfig.java:824)
  at org.apache.catalina.startup.HostConfig.deployWARs(HostConfig.java:713)
  at org.apache.catalina.startup.HostConfig.deployApps(HostConfig.java:489)
  at org.apache.catalina.startup.HostConfig.start(HostConfig.java:1137)
  at org.apache.catalina.startup.HostConfig.lifecycleEvent(HostConfig.java:310)
  at org.apache.catalina.util.LifecycleSupport.fireLifecycleEvent(LifecycleSupport.java:119)
  at org.apache.catalina.core.ContainerBase.start(ContainerBase.java:1021)
  at org.apache.catalina.core.StandardHost.start(StandardHost.java:718)
  at org.apache.catalina.core.ContainerBase.start(ContainerBase.java:1013)
  at org.apache.catalina.core.StandardEngine.start(StandardEngine.java:442)
  at org.apache.catalina.core.StandardService.start(StandardService.java:450)
  at org.apache.catalina.core.StandardServer.start(StandardServer.java:709)
  at org.apache.catalina.startup.Catalina.start(Catalina.java:551)
```

[2] Tomcat issue: ID 40820, `https://issues.apache.org/bugzilla/show_bug.cgi?id=40820`

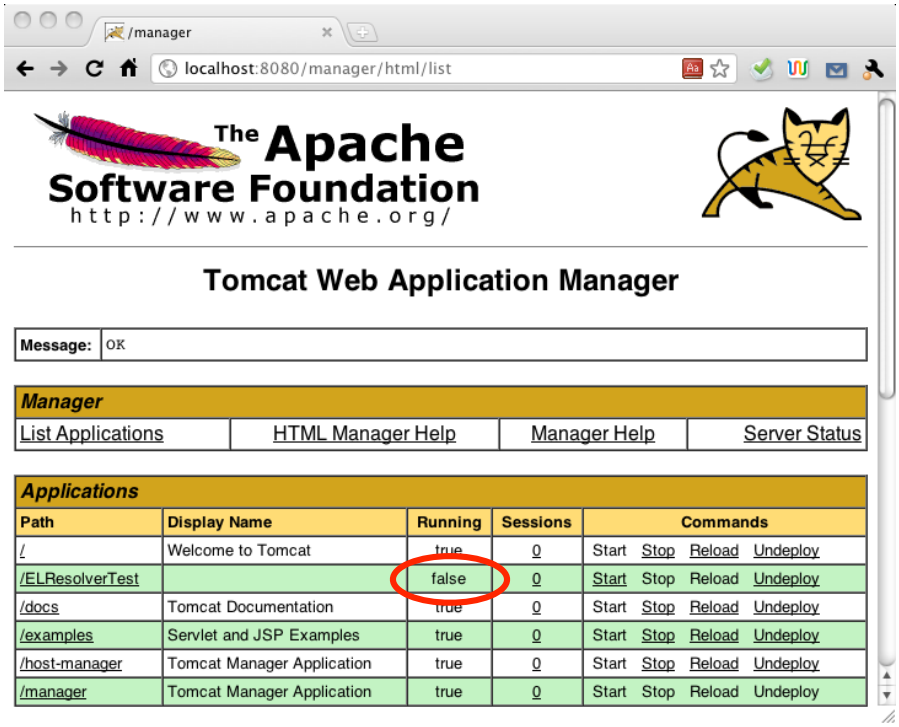


Figure 3.5. Tomcat manager screenshot

```
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
  at java.lang.reflect.Method.invoke(Method.java:597)
  at org.apache.catalina.startup.Bootstrap.start(Bootstrap.java:287)
  at org.apache.catalina.startup.Bootstrap.main(Bootstrap.java:412)
Feb 15, 2011 4:27:00 PM org.apache.catalina.core.ApplicationContext log
INFO: ContextListener: contextInitialized()
Feb 15, 2011 4:27:00 PM org.apache.catalina.core.ApplicationContext log
INFO: SessionListener: contextInitialized()
```

Figure 3.6 shows the sequence of operations that leads to the failure during the Tomcat startup: the method `Catalina.start()` initializes the Web server instance, and deploys the Web applications found on the server by invoking the method `HostConfig.deployApps()`. After the Web application deployment, Tomcat calls the method `StandardContext.start()` that loads the class `JspRuntimeContext`, and initializes the listeners of the Web application.

During this initialization there is no `JspFactory` initialized, and thus a `NullPointer` exception is raised. Tomcat catches the exception, and completes the server startup without loading the Web application containing the JSP file.

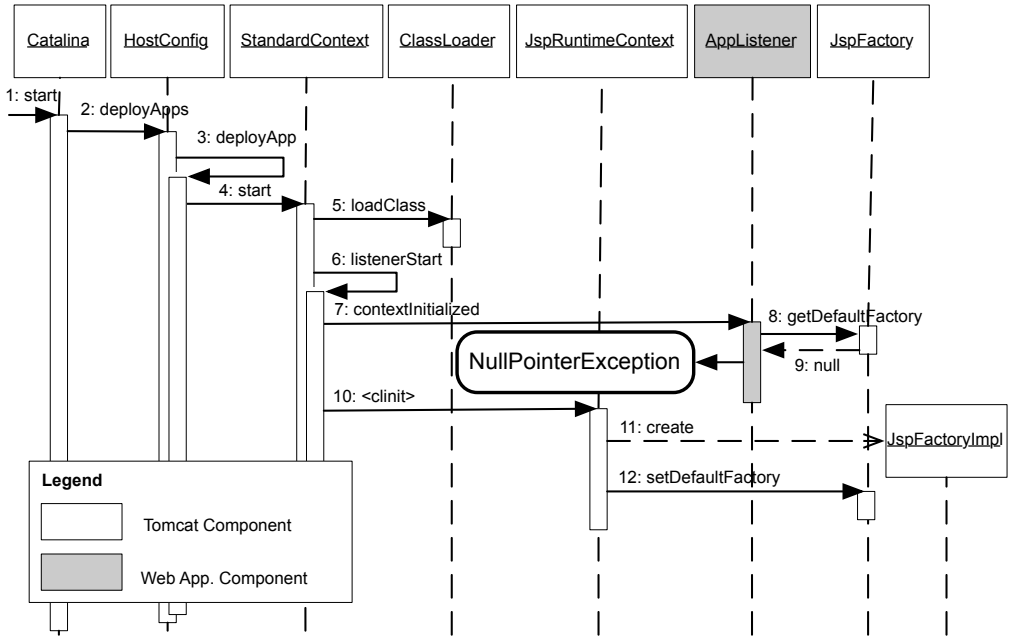


Figure 3.6. The failing Tomcat bootstrap sequence

3.2 Redundancy as Equivalent Sequences

In the previous section we have described the module responsible of deploying and loading the Web applications in Tomcat. We have seen that software modules, such as the one we described, can be intrinsically redundant, and this redundancy can be expressed as a list of equivalent sequences, as the ones listed in Figure 3.4.

Beside Tomcat, we analyzed the specification and the issue tracker of several projects, such as OpenOffice (`http://www.openoffice.org`), YouTube (`http://code.google.com/apis/youtube`), Flickr (`http://www.flickr.com/services/api`), Picasa (`http://code.google.com/apis/picasaweb`), Facebook (`http://developers.facebook.com`), Yahoo! Maps (`http://developer.yahoo.com/maps`), Microsoft Maps (`http://www.microsoft.com/maps/developers`), Google Maps (`http://code.google.com/apis/maps`) and JQuery (`http://jquery.com`), and we identified three general classes of equivalent sequences, namely *functionally null*, *invariant* and *alternative operations*. This classification is orthogonal to the one presented at the beginning of the chapter, which considers generality and accuracy.

We now provide an intuitive description of each class, using the Tomcat module to provide examples.

Functionally null operations. It is often the case that programming languages provide functions that manage either the timing or the scheduling of other functions. This

is the case, for example of the `sleep`, `wait` and `notify` operations in Java, which control the threads execution, and the `setTimeout` operation in Javascript, which delays the execution of other Javascript operations of a specified amount of milliseconds. Although it is not common to find functionally null operations implemented in software components, we ran into few cases. For example `JQuery`, a Javascript framework that easily enhances HTML pages with Javascript functionalities, offers the `delay()` function that sets a timer to delay the execution of an animation effect.

Functionally null operations should not affect the functionality of an application, thus they can be consistently combined with any other operation to produce sequences that are semantically equivalent to the operation itself. The obtained equivalent sequences are always general and accurate, because, as we said at the beginning of the chapter, we do not distinguish among non functional behavior differences.

If we consider the Tomcat module example described in Section 3.1, we can see that it is possible to obtain the same result by for example loading a Web application in Tomcat and by delaying the load operation of few milliseconds:

$$\texttt{loadApp(app)} \equiv \texttt{sleep(500);loadApp(app);}$$

Previous studies have shown that functionally null operations are useful to create potentially equivalent sequences that can be exploited in testing and debugging to reveal and analyze concurrency faults like race conditions and deadlocks [EFN+02]. By randomly inserting `sleep()` invocations in existing Java test suites, and by re-executing them, it is possible to pinpoint faults if the executions do not produce the same results [3]. The same research group recently showed that functionally null operations can be useful to avoid race conditions and deadlocks, following the same intuition. By inserting a `sleep()` invocation in the right place concurrency problems can be avoided at runtime [NBTU08; KLNB+09].

Invariant operations. Some of the operations that software modules provide have no functional effect when they are executed under certain conditions or when executed together with other operations. These are typically pairs of operations in which the second one reverses the effects of the execution of the first one. Longer and more complex combinations of operations are also possible, as long as the overall functional effect of the combination is null.

Examples of invariant sequences are `add` and `remove` operations for containers, or `zoom-in` and `zoom-out` for graphical elements.

Software modules usually have several single, pairs or groups of these operations. In the Tomcat deployment module, for example, there are several invariant operations such as

[3] The `sleep` method in Java stops the execution of a thread for a specified amount of milliseconds

`loadApp(app)` – when `app` is already loaded.
`deployApp(app)` and `undeployApp(app)` – when `app` is not deployed yet.
`startTomcat()` and `stopTomcat()` – when Tomcat is not running.
`loadApp(app)` and `stopApp(app)` – when `app` is not loaded yet.

Since they do not have (or they should not have) any functional effect, invariant operations can be used in testing just as null operations. They can be inserted in existing test cases with the expectation to obtain the same results. If this does not happen, then they reveal the presence of faults in the code. Like functionally null operations, invariant operations may also affect the scheduling of concurrent operations, and thereby solve concurrency problems. Moreover, some operations within an invariant sequence may partially reset the state of the application, and therefore identify and solve initialization problems.

Invariant operations are usually *specific*. Most of the times, in fact, the equivalence holds only for a particular state. For instance, deploying and undeploying a Web application in Tomcat is an invariant operation only if the Web application has not been deployed yet.

Invariant operations are application-specific, but they can be easily identified by developers and users.

Alternative operations. Software modules often offer operations that lead to the same effects of other operations. Moreover some operations can be used in different ways to achieve the same goal. Alternative operations are sets of operations that produce the same results as other (different) sets of operations. An example of intrinsic redundancy that can be expressed in terms of alternative operations is the one about containers that we mentioned earlier in this chapter. Adding several elements to a container should bring to the same result as of adding one element at a time, respecting the order.

If we consider the Tomcat Web applications deployment module, it is possible to identify several alternative operations. For instance

$$\texttt{restartTomcat()} \equiv \texttt{stopTomcat(); startTomcat()}$$
$$\texttt{loadAllApps()} \equiv \texttt{loadApp(a}_1\texttt{); loadApp(a}_2\texttt{); ...loadApp(a}_n\texttt{);}$$

Alternative operations can generate equivalent sequences by substitution, and they tend to engage more redundant code in software modules. Alternative operations are application-specific and can be identified by gaining a basic understanding of the semantics of the application.

Alternative operations can be either specific, if they apply under specific conditions, or general if they are always valid.

3.3 Identifying Intrinsic Redundancy

The intrinsic redundancy of software systems can be identified and represented in the form of equivalent sequences either automatically or manually. Equivalent sequences can be derived automatically from formal specifications that provide a sound and complete description of the semantic of the software systems. Formal specifications can be given during development, especially for critical applications, or can be derived dynamically, during system execution.

Developers can write equivalent sequences manually as an additional form of specification for their software, or can identify equivalent sequences from specifications written in natural language. According to our experience, users who are familiar with the software can easily write a set of correct equivalent sequences in few hours. It is often the case that specifications mention whether operations do not have any functional effect, and they thus correspond to null operations, or whether operations are equivalent to other operations, and thus represent alternative operations. Moreover, invariant operations usually have similar names, for example `startTomcat` – `stopTomcat`, `enableDragging` – `disableDragging`, and thus can be easily spotted. For the experiments reported in this thesis, we derived the equivalent sequences manually. We discuss the size and complexity of the set of sequences generated for the experiments, and the required effort in Chapter 6.

We now discuss the problem of generating equivalent sequences automatically referring to algebraic specifications and finite state machines. Algebraic specifications describe the semantics of the logical functions or methods of the system by means of a set of axioms that indicate the effect of the functions or methods on the state of the system. Equivalent sequences can be derived automatically from algebraic specifications by comparing the semantics of the axioms, in line with the way Doong and Frankl generate equivalent scenarios as automated test oracles [DF94]. Finite state machine specifications represent the intended effect of the operations of a module on the state. Self-loops in the finite state machine correspond to null operations, since these operations should not have any functional effect. Loops that traverse two or more states correspond to invariant operations, since they bring the system to the state before their execution, and thus do not have any functional effect overall. Sequences of transitions that share the same start and end state correspond to alternative operations, since the overall functional effect of these sequences is the same. Equivalent sequences that can be executed from every state of the model are general, sequences that can be executed only from some states are specific.

When formal specifications are not provided as part of the software development process, they can be generated from the execution of the software system. There exist several techniques to generate various forms of specifications from program execution. For example, Henkel, Reichenbach, and Diwan proposed a technique to infer algebraic representations of the execution space [HRD07], while other research

groups proposed different techniques to automatically derive finite state representations [MPPar; DLWZ06; GMM09; BIPT09; DKM+10]. 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 represent also failing behaviors that have been observed. Thus, these models can be used to infer equivalent sequences, but since the obtained result may not be correct, they should be approved by developers before use.

3.4 Exploiting Intrinsic Redundancy

In the previous sections we introduced the concept of intrinsic redundancy, and we presented our hypotheses on why this redundancy is intrinsically present in modular software applications. We now want to illustrate how this redundancy can be exploited, and we use the Tomcat issue described in Section 3.1 for this purpose.

Our intuition is that intrinsic redundancy can be exploited in several ways, namely to reveal the presence of faults, to identify the location of the fault in the failing software, and possibly to avoid failures.

Intrinsic redundancy for testing: Testing activities include the generation and the execution of test cases, and the judgement of the execution results. Checking the result of the execution is one of the major challenges of software testing. Since it is usually infeasible to manually judge the correctness of test executions, testers usually automate the evaluation process by comparing the outcome of a test execution with the expected result that they provided as an oracle. Precomputing the oracle, however, is a time consuming activity as well, as it may require to manually perform the computation that the software under test is supposed to do.

In chapter 2 we described the techniques proposed by Doong and Frankl first and by Gotlieb later to use program symmetries to have implicit oracles [DF94; Got03].

Using intrinsic redundancy expressed in the form of equivalent sequences for testing amounts to the same idea, and would lead to the benefits of testing software modules without having the cost of writing test oracles. In fact, if we know that two sequences are expected to be equivalent, we can execute them on the same preconditions, and verify that the results obtained are the same. If this is not the case, then we can infer that it is very likely that there is a fault somewhere in the software module.

The issue of the Tomcat Web applications loader could have been identified during the development process thanks to this technique, and consequently version 6.0.0 and following releases would have not been affected by this problem.

There are at least three sequences of operations that are equivalent according to the specifications

1. `deployApp(appWithJSP); startTomcat();` that raises an exception.

2. `startTomcat(); deployApp(appWithJSP); loadApp(appWithJSP);`

3. `deployApp(appWithJSP); startTomcat(); loadAllApps(appWithJSP);`

The execution of these sequences of operations should bring to the same result, that is, the Web application should be deployed, Tomcat should be running correctly, and the application should be loaded and reachable. This is not the case, though. The first sequence deploys the application and starts Tomcat, but does not load the Web application, while the other two load the Web application as well. By comparing the observable state of these objects it would have been clear that the module for loading the Web applications had a fault.

Although the idea of exploiting intrinsic redundancy for testing is not new, the previous works limit the applicability to either mathematical libraries or containers, and they mainly rely on formal specifications such as algebraic axioms [DF94; Got03]. Instead, we argue that the idea can be applied to a large set of applications, and can be used to test medium to large size applications such as Tomcat.

Intrinsic redundancy for fault localization: Intrinsic redundancy expressed in the form of equivalent sequences can be useful for fault localization as well. For the same principle of N-version programming, the execution of three equivalent sequences can pinpoint the likely sequence of operation that contains the fault.

For instance, if we consider the three equivalent sequences that we mentioned before, we can say that the sequence of operations that is likely to contain the fault is the first one, since it produces an observable behavior that is different from the other two sequences.

Intrinsic redundancy to avoid failures: Beside revealing the presence of faults, and locating faulty operations, intrinsic redundancy can be used to avoid failures at runtime. Similarly to the recovery-blocks mechanism [Ran75], which swaps the execution to alternative redundant components upon a failure, equivalent sequences can be used as alternative implementations in an attempt to mask the presence of faults in the software. If a sequence of operations leads to a failure, as it is the case for the sequence that deploys a Web application with a JSP file before starting Tomcat

(1) `deployApp(appWithJSP); startTomcat()`

then it is possible to execute any equivalent sequence of operations to try to avoid the problem:

(2) `startTomcat(); deployApp(appWithJSP); loadApp(appWithJSP);`

or

(3) `deployApp(appWithJSP); startTomcat(); loadAllApps();`

When an equivalent sequence is selected at runtime as an alternative to the failing sequence of operations, we do not know whether it is going to fail. Thus, similarly to recovery-blocks, all the equivalent sequences are selected and executed one after the other, until one of them does not fail. An equivalent sequence that does not fail is what we call a *workaround*, and it is an alternative way to perform the failing operations, but without leading to the failure. Once a workaround is found, it can substitute all the occurrences of the failing sequences of operations to avoid any further failure occurrence.

In Tomcat, both the equivalent sequences (2) and (3) proposed as alternatives to the failing sequence of operations can deploy and load the Web application with the JSP file correctly. More precisely, the equivalent sequence (2) can avoid the null pointer exception raised by the failing sequence. The equivalent sequence (3), instead, still raises the null pointer exception after the Tomcat startup, but can mask the effect of the failure after the execution of the `loadAllApps` operation.

This is the basic idea that stands behind our technique, which we call Automatic Workarounds. We provide details about this technique in the next chapter.

Chapter 4

Automatic Workarounds

In this chapter we present a novel technique, called Automatic Workarounds, that exploits intrinsic redundancy in software components to find workarounds automatically at runtime after the occurrence of a failure. The key idea is to dynamically change the executed code with equivalent sequences in an attempt to avoid further failures. We first present the technique as a general framework, and then focus on the Web applications domain. We first motivate the need of such technique in this context, and we then show that the Automatic Workarounds technique is particularly suitable and effective for Web applications.

In the previous chapter we introduced the concept of intrinsic redundancy, and we analyzed the reasons why this type of redundancy is usually present in software modules. We also gave a high level idea of how this redundancy can be used to test software components without incurring the costs of writing test oracles, and how it can be exploited to avoid failures at runtime. We now present the Automatic Workarounds technique as a way to exploit intrinsic redundancy to mask the presence of faults at runtime.

The main intuition behind this technique comes from the observation of what people do in practice when they experience a failure at runtime. When a failure occurs, users usually try to avoid the problem by looking for alternative ways to achieve the desired results. The technique proposed in this thesis tries to automate the process of looking for workarounds by dynamically substituting the executed code at runtime with equivalent sequences, and by re-executing the new code to see whether the equivalent sequence serves as a workaround. More precisely, the technique focuses on handling failures caused by problematic interactions between applications and reusable components such as libraries, services, frameworks, and it exploits the intrinsic redundancy of such components to find workarounds automatically when a failure occurs.

To deploy workarounds at runtime we propose a layer that mediates the interaction between the application and the software libraries used by the application. Figure 4.1

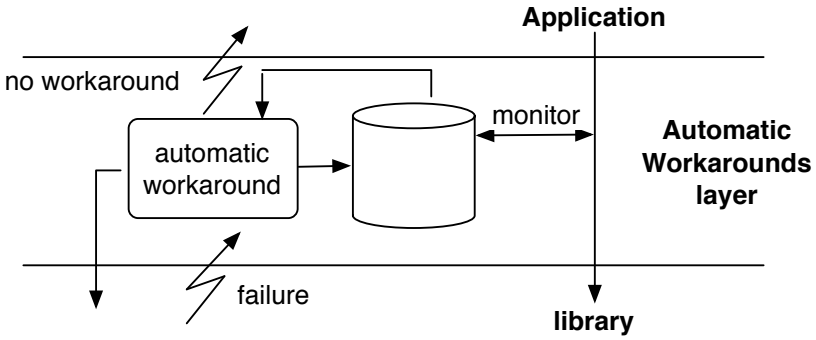


Figure 4.1. High level architecture of the Automatic Workarounds system.

illustrates the role of the Automatic Workarounds layer. The Figure shows the application, at the top of the diagram, calling an operation implemented by a library, at the bottom of the diagram. The Automatic Workarounds layer monitors the calls and maintains a partial history of calls. In addition to monitoring calls, the Automatic Workarounds layer intercepts failure signals, and in response to those, it looks at the history of calls and selects and executes equivalent sequences. The occurrence of a failure and the execution of one or more equivalent sequences to find a valid workaround are invisible to the application. Only when all the equivalent sequences have been executed and fail, the failure is reported up to the application. However, it should be clear that the Automatic Workarounds layer does not aim to fix a fault permanently. It rather looks for a temporary patch to avoid failures.

The architecture described in Figure 4.1 assumes the availability of an oracle that can detect the failures, and a roll back mechanism that can bring the system back to a consistent state. The oracle is necessary to trigger the Automatic Workarounds process, while the rollback mechanism is required to reverse the effects of the failing sequence of operations, thus to avoid side-effects.

The execution of an equivalent sequence is illustrated in Figure 4.2. The sequences of letters symbolize sequences of operations. The figure shows an initial sequence of operations that results in a failure. The automatic recovery executed in response to the failure restores the system by rolling back some operations. The failure handling starts from that point, selects an appropriate equivalent sequence that matches the intended semantics of the failing sequence, and executes the equivalent sequence to completion. If the execution of the equivalent sequence results in another failure, then the process iterates. If the execution of the equivalent sequence does not result in another failure, the system found a valid workaround. Thus, a *workaround* is a sequence of operations that is semantically equivalent to a failing one in its original intent, but does not result in a failure.

We decided to explore Automatic Workarounds in the context of Web applications because the characteristics of these applications can ease the implementation of the

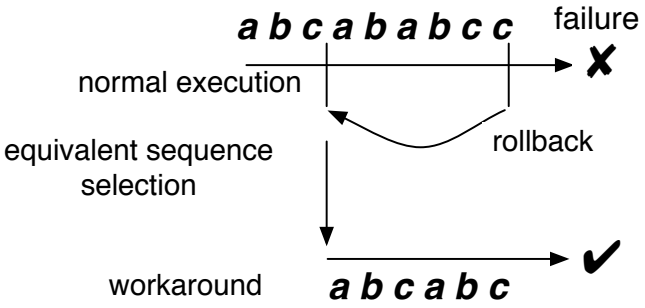


Figure 4.2. Execution of a workaround.

rollback and the failure detection mechanisms. In the next section we describe the challenges that this new domain poses, and we frame the Automatic Workarounds technique in this new context.

4.1 Automatic Workarounds for Web Applications

Web 2.0 brought new features in Web applications, which are now much more similar to desktop applications in terms of functionalities and interaction with users. It is often the case that Web application developers rely on the existing Web APIs to include some functionalities that span from photo sharing, mapping, blogging, advertising and much more [1].

APIs for popular Web applications like Google Maps and Facebook increase the popularity of such applications by allowing other Web applications to interact with their functionalities. Assessing the quality of Web APIs, however, is challenging, since third-party developers can use Web APIs in many different ways and for various purposes. Moreover applications can be accessed by many users through different combinations of browsers, operating systems, and connection speeds. This leads to a combinatorial explosion of use cases, and therefore a growing number of potential incompatibilities that can be difficult to test with classic approaches, especially within tight schedules and constrained budgets.

Furthermore, failures caused by faults in common APIs can affect a large number of users, and fixing such faults requires a time consuming collaboration between third-party developers and API developers. In order to overcome these open problems in the absence of permanent fixes, users and developers often resort to workarounds. However, although many such workarounds are found and documented in on-line support groups, their descriptions are informal, and their application is carried out on a case-by-case basis and often with non-trivial ad-hoc procedures.

[1] see the `www.programmableweb.com` website to see the growing popularity of Web APIs. In May 2011 the web site was pointing to more than 3000 Web APIs.

Thus, in the context of Web applications, where it is particularly difficult for external users to operate on third-party libraries code, the Automatic Workarounds technique can be particularly effective, since it allows to find and execute workarounds at runtime in response to failures caused by faults in the libraries that the Web application depends on.

In order to deploy workarounds effectively, we need to solve four problems. First, we need to detect failures and check for the validity of workarounds. Second, we must be able to execute equivalent sequences repeatedly without compromising the state of the application and without suffering from potential side-effects of the previous failing execution. Third, we need to represent the redundancy of APIs so as to be able to generate equivalent sequences at run time. Fourth, we need to find valid workarounds out of potentially many equivalent sequences.

We assume that we can dismiss the second problem (repeated executions) in the case of Web applications because those are typically *designed* to avoid potential side-effects caused by repeated executions. We can therefore execute a workaround by simply re-executing the client-side code of the page that manifests the failure (after applying the workaround). We discuss more about this assumption in the next section.

The remaining problems are the focus of the rest of this chapter. For the first problem (detecting failures and verifying the validity of deployed workarounds) our general approach is to rely on the interactive nature of Web applications, and in practice to assume that users can easily detect a failure and explicitly request a workaround. Similarly, users can reject the execution of an equivalent sequence, or implicitly validate it as a workaround by proceeding with their interactive session. As we rely on users to find workarounds, we must also take special care not to annoy and ultimately alienate them with too many repeated attempts. We address this problem by developing a priority mechanism that ranks equivalent sequences such that the ones that are more likely to work are selected first, and an automatic oracle that can conservatively detect, and therefore discard, some ineffective equivalent sequences.

In the next section we further motivate the need of Automatic Workarounds in Web applications, and we do it by showing how workarounds can solve real issues.

4.1.1 Workarounds for Web Applications

Workarounds are a very popular means to avoid issues in software. Developers use them to code some hacks for avoiding known problems, and users use them to minimize the effects of failures due to unknown faults. Workarounds are effective in dealing with Web library issues, too. We illustrate the notion of workaround, and specifically the nature of failures and workarounds in the context of Web applications, using two examples of failures affecting two widely used Web APIs. The two examples we present here are typical of a large number of problems that we observed in our survey of bug repositories, discussion forums, and interest groups related to popular Web APIs.

Flickr, Photo visibility Flickr is a popular photo-sharing application. The Flickr API allows users to upload and publish photos on the Web. Flickr associates photos with visibility tags that control access to photos on the Web, and tags can be *public*, *family*, or *private*. The Flickr API includes a method setPerms() that allows users to change the visibility of their photos. A message posted by a user on the Flickr forum in March 2007 reported a problem with setPerms(): the method failed to change the visibility of a photo from *private* to *family* for photos that were originally uploaded as *private*.[2] The problem persisted for some time in Flickr, and was originally avoided with a simple workaround posted on the forum: to change the visibility of a photo from *private* to *family*, first change it from *private* to *public*, and then from *public* to *family*. This fault has since been fixed.

Facebook, Dialog windows The popular social networking system Facebook exposes an API for Web applications. The Facebook API includes a JavaScript library that, among other things, allows applications to create and control graphical elements. Report no. 2385 in the Facebook bug repository describes a problem with the setStyle() method of the Facebook JavaScript API, which is intended to set the width and height of a graphical element. The method works well when one of the two dimensions is set individually, but fails when used to set both dimensions at the same time.[3] The report was filed in June 2008 by a developer who also suggested a workaround in which one can set width and height with two separate setStyle() calls. This fault has been fixed in October 2010.

The examples above show that some faults may survive for a long time in Web APIs, even if they are properly reported through issue trackers. We do not intend to investigate the software maintenance processes for Web applications, and we realize that those are complex processes affected by human factors and driven by technical as well as non-technical objectives. Nevertheless, we observe that those kinds of failures have to go through two maintenance steps: they are initially reported, either indirectly by application users (first case) or more directly by application developers (second case) but can be corrected only by the developers of the Web API, who may not be aware of the actual impact of the fault, and in any case may have other priorities.

The two cases also exemplify two different types of workarounds. In the first case, it is very likely that the setPerms() method of the Flickr API be exposed directly to users, allowing them to control access to their photos. Therefore, not only users can more easily notice the failure and report it with an accurate diagnosis, but they can also directly apply the proposed workaround. This is most probably not the case for the setStyle() method of the Facebook API, which is typically used by applications but not exposed to users. So, in this latter case, users may or may not detect the failure, but would certainly not be able to use the proposed workaround. In fact, the failure

[2] http://www.flickr.com/help/forum/36212, —/46985
[3] http://bugs.developers.facebook.com/show_bug.cgi?id=2385

report explicitly characterizes it as a "workaround for developers." The Automatic Workarounds technique can find and apply both types of workarounds, and thus brings advantages both to the Web application users, who may stop experiencing failures, and to the Web application developers, who may take advantage of the workarounds found automatically to implement a permanent fix.

4.1.2 Architecture

The Automatic Workarounds technique, as it has been described in Figure 4.1, makes two strong assumptions. First it requires that a *failure-detection* mechanism be present and available within the system. Second, it requires that the failure detector be coupled with a basic *recovery mechanism* that can bring the application back to a consistent state right after the occurrence of a failure. The failure detector is essential to trigger the search of workarounds only when it is necessary, and the recovery mechanism is required to avoid unexpected side-effects caused by the failing execution and by the non successful attempts to find a workaround.

In the context of Web applications these assumptions can be relaxed, as both failure detection and recovery mechanisms are straightforward. Web applications should be designed (according to the POST/Redirect/Get pattern [4]) to avoid potential side-effects caused by repeated executions, and are highly interactive by nature. The first characteristic allows the execution of equivalent sequences by simply reloading the Web page that manifests the failure (after applying the equivalent sequence). We can then exploit the interactive nature of Web applications by assuming that users can easily detect a failure and explicitly request a workaround by pressing a special button on their browser. For example, if an application that uses the Google Maps API does not display a map as expected, the user can, with minimal effort, detect the failure and request a workaround. We decided to focus on JavaScript Web APIs without considering Ajax requests, such that the whole execution is on the client side, and thus we do not have to deal with rollbacks.

To automatically deploy workarounds at runtime in the context of Web applications we adapt the layer presented in Figure 4.1 such that it mediates the interaction between one or more servers and the browsers that execute the applications. The servers produce the HTML pages and the code associated with a Web application, while the browsers execute the code, render the pages, and manage the interaction with the users. The workaround layer, interposed between servers and browsers, deploys workarounds by modifying the code of the applications that go to the clients. This layer can be implemented either as an HTTP proxy or as a browser extension.

Figure 4.3 adapts the architecture described in Figure 4.1 to Web applications. The interaction starts with a client requesting a specific page (step 1). The layer for-

[4]PRG is a common pattern that avoids duplicate form submissions when user refreshes a Web page containing POST requests.

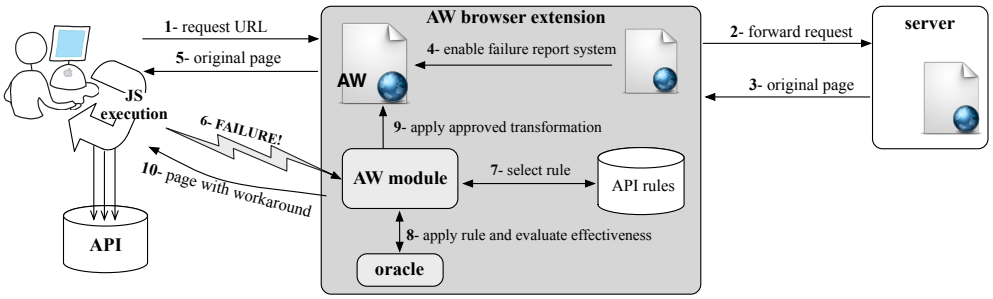


Figure 4.3. High level architecture of the Automatic Workarounds approach for Web applications

wards the initial requests for all the objects that compose the page to the source server (step 2). The layer then examines the objects returned by the server (step 3) looking for references to the API of some known, third-party Web libraries for which a catalog of equivalent sequences exists. If such references are found (step 4) then the layer enables a failure-reporting mechanism for the requested page, and then returns the result to the user browser (step 5). The browser then displays the page and executes the JavaScript code that comes with it, which might in turn retrieve and execute additional code fragments from Web libraries. If no failure occurs, or more precisely is not perceived, the user continues interacting with the application, and the layer is transparent.

If the users perceive a failure, they may report the issue to the layer by using the reporting mechanism when enabled (step 6 in Figure 4.3). When the layer receives a failure report it extracts the JavaScript code from the page, identifies an equivalent sequence (step 7), applies it to the code, and passes the page with the new code to the automated oracle for validation (step 8). If the oracle approves the page, then the extension applies the same equivalent sequence to the original page (step 9) and presents the result to the user (step 10). If the equivalent sequence fails to resolve the problem, either because it is immediately rejected by the oracle (step 8) or because the user reports another failure, then the Automatic Workarounds layer reiterates the same process (steps 7–10) until it has exhausted its repository of equivalent sequences or the users give up reporting problems.

4.2 Equivalent Sequences in Web Applications

In Section 3.2 we have presented three classes of equivalent sequences, namely functionally null, invariant and alternative, which represent intrinsic redundancy in software modules. The same classes of operations can be found in Web APIs. We now want to provide some examples of equivalent sequences that can be easily identified in the Google Maps API, and we show how these equivalent sequences can serve as valid

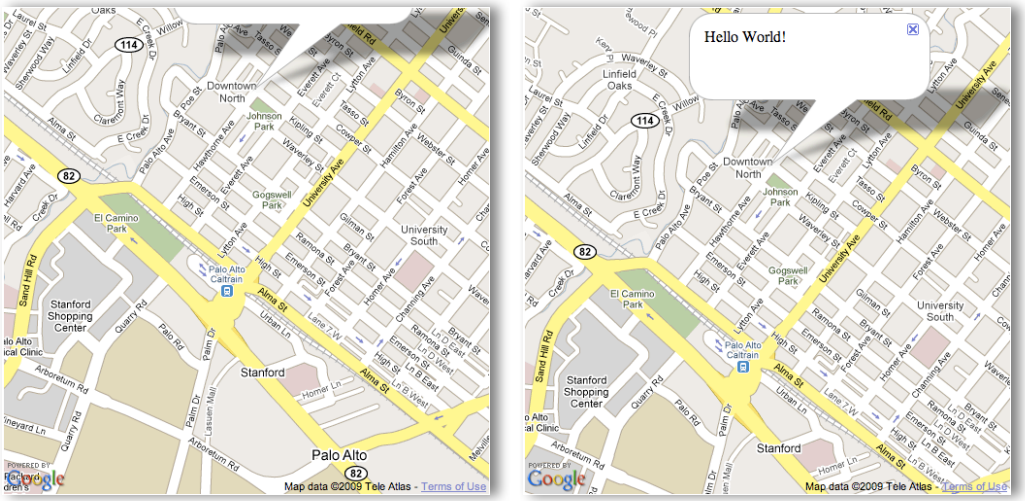


Figure 4.4. Issue 519 of the Google Maps API, fixed with a null operation

workarounds.

Functionally null operations – issue 519. In July 2008 several Web application developers reported a problem about information windows, which can be used to show some information on a map. When the `openInfoWindow` function, which opens the information window, was invoked right after the `setCenter` operation, which sets the center of a map to a given geographical location, most of the times the information window could not be visualized correctly (see Figure 4.4 on the left side):

```
map = new GMap2(document.getElementById("map"));
map.setCenter(new GLatLng(37,-122),15);
map.openInfoWindow(new GLatLng(37.4,-122), "Hello World!");
```

In normal conditions the center of the map would have been changed such that the whole information window is completely visible. This was not happening, though, especially with some browsers. The cause of this failure is probably [5] a subtle timing issue that occurs when a JavaScript file is calling a function defined in another JavaScript file, before the latter is fully loaded in memory.

This failure could be avoided with a functionally null operation, the `setTimeout` function, which delays the execution of an operation of few milliseconds:

```
map = new GMap2(document.getElementById("map"));
setTimeout("map.setCenter(new GLatLng(37,-122),15)",500);
map.openInfoWindow(new GLatLng(37.4,-122), "Hello World!");
```

[5] This is just an hypothesis, since Google developers did not confirm it yet.

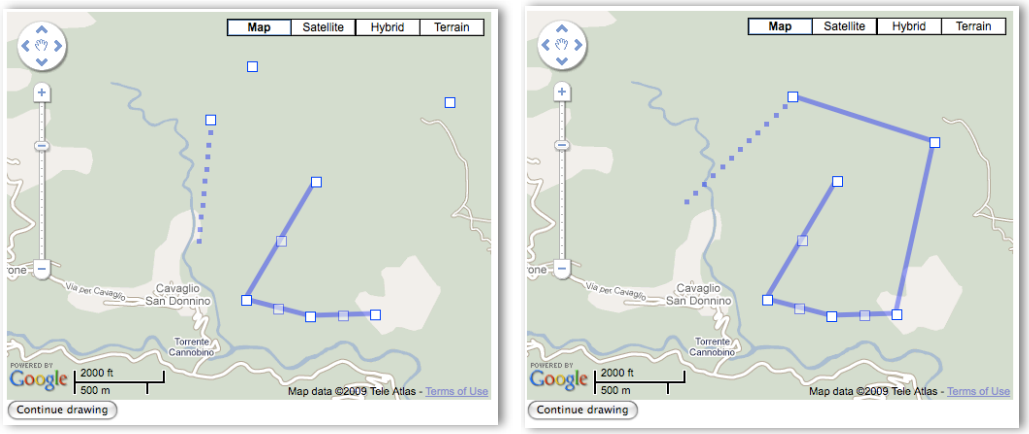


Figure 4.5. Issue 1305 of the Google Maps API, fixed with invariant operations

The delayed execution of the `setCenter` function was solving the problem (see Figure 4.4 on the right side).

Invariant operations – issue 1305. In May 2009 developers reported an issue with the polylines in Google Maps, as the `enabledrawing()` function was not working correctly. In normal conditions, this function should allow users to edit the polyline by adding, removing, and moving vertexes. As it is visible in Figure 4.5 on the left side, users could add vertexes to the existing polyline, but the new vertexes were producing a transparent line.

The invariant operation of removing and reinserting the last vertex of the polyline was solving the problem, though:

```
v = polyline.deleteVertex(polyline.getVertexCount()-1)
polyline.insertVertex(polyline.getVertexCount()-1,v);
polyline.enableDrawing();
```

As it is visible on the right side of Figure 4.5, this workaround could let users correctly draw polylines on a map. Google Maps developers never fixed this issue.

Alternative operations – issue 585. In August 2008 developers reported problems with overlays on a map when the earth view was enabled. In Figure 4.6 the overlay is the red marker in the center of the map. The problem was that after the creation of the overlay, it was impossible to control its visibility. Thus, the following code was not hiding the overlay correctly:

```
map.addOverlay(first);
function showOverlay(){ first.show(); }
function hideOverlay(){ first.hide(); }
```

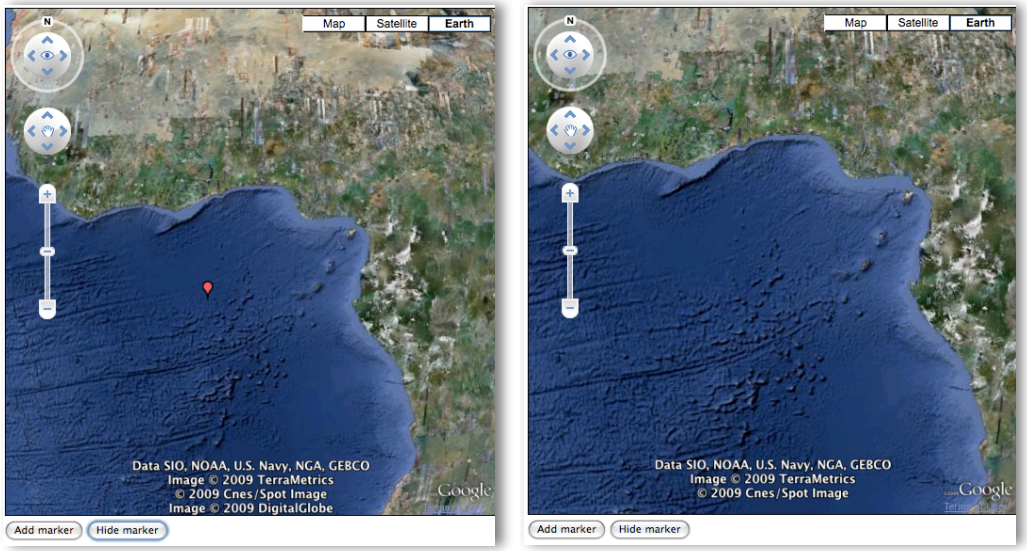


Figure 4.6. Issue 585 of the Google Maps API, fixed with alternative operations

The alternative operations of adding and removing the overlay, instead of hiding and showing it, were producing the expected result, without causing any failure (see right side of Figure 4.6):

```
map.addOverlay(first);
function showOverlay(){
     map.addOverlay(first);
     first.show();
}
function hideOverlay(){ map.removeOverlay(first); }
```

Although adding and removing an overlay does not have the same semantic of showing and hiding the same overlay, the execution of these operations produces the same observable behavior, thus we consider them equivalent sequences.

4.3 Program-rewriting Rules

We call *workarounds* those equivalent sequences that lead to the expected result of the failing operation, but do not cause any failure. To find a workaround automatically we select an equivalent sequence, apply it to the code and re-execute the code to see if it leads to a new failure. However equivalent sequences have to be instantiated correctly in the context of the application, thus considering the actual variables and parameters used in the failing code. We thus propose to create equivalent sequences by applying

program-rewriting rules to the existing JavaScript code. A rule is defined according to the following high-level syntax:

⟨Rule⟩ ::= ⟨Type⟩ ⟨Scope⟩ : ⟨Substitution⟩
⟨Type⟩ ::= null | invariant | alternative
⟨Scope⟩ ::= ANY | ALL | ⟨number⟩
⟨Substitution⟩ ::= (⟨pattern⟩ → ⟨replacement⟩)$^+$

A rewriting rule is defined by three elements: the type of the rule, the scope of the rule, and a substitution expression. The type classifies the rule according to the taxonomy introduced in Sections 3.2 and 4.2 as null, invariant, or alternative. The scope determines where the substitution expression is to be applied within the program code. If the scope is ANY then the substitution is applicable to any one of the occurrences of the substitution pattern; if the scope is ALL then the rule must be applied to all occurrences of the substitution pattern; if the scope is a number *n* then the rule is applied to the *n*-th occurrence of the substitution pattern.

The substitution expression, which is the central component of the rule, is defined by one or more pairs of pattern and replacement text. We define each pair with a regular-expression language common to many text processing tools.[6] For the sake of simplicity, here we illustrate our examples with a slightly different notation in which we identify sub-expressions as variables. These variables are assigned by the pattern expression, and then expanded in the replacement text. We use the dollar sign followed by a capital letter (for instance, '$X') to indicate one of those variables. For simplicity, we omit the actual definition of the sub-expression corresponding to a variable, which can be intuitively deduced from the context. For instance the three equivalent sequences that could solve the Google Maps issues described in Section 4.2 can be expressed like this:

```
(1) null ANY:
    $X.setCenter($Y);
    → setTimeout("$X.setCenter($Y)", 500);
(2) invariant ALL:
    $X.enableDrawing();
    → v = $X.deleteVertex($X.getVertexCount()-1);
    $X.insertVertex($X.getVertexCount()-1,v);
    $X.enableDrawing();
(3) alternative ALL:
    $X.addOverlay($Y); $Y.show();
    → $Y.show();
    $X.removeOverlay($Y);
    → $Y.hide();
```

These program rewriting rules can rewrite any JavaScript code such that: (1) one

[6] See the POSIX.2 Regular Expression Notation.

invocation per time of the `setCenter` function would be delayed to avoid potential timing issues, (2) all the invocations to the `enableDrawing` function would come after `deleteVertex` and `insertVertex` invocations to avoid potential problems with polylines, and (3) all the `show` and `hide` invocations would be replaced by `addOverlay` and `removeOverlay` invocations respectively to avoid potential problems with the overlays.

4.4 Priority Schemes for Equivalent Sequences

Even simple programs with only few rewriting rules can result in many equivalent sequences, among which the Automatic Workarounds layer must select the ones that are more likely to serve as workarounds. Many failing attempts may badly affect performance and usability if their evaluation is deemed to the application users. To reduce the amount of attempts, we propose priority schemes. Here we discuss the priority schemes that we investigated:

Distance: Priority can be computed on the basis of the differences with the failing sequence in terms of function calls. The rationale is that the more the executed code differs from the failing one, the more likely it is going to avoid the faulty element. Thus, if for instance the failing sequence amounts to the function calls `a(); b(); c();`, then we would select the equivalent sequence `d(); c();` before `b(); a(); c();`.

History: Priority can also be computed on the basis of the success in previous attempts. The rationale is that if an equivalent sequence has served as a workaround in the past it is likely to be valid in the future as well.

Fault localization: The Automatic Workarounds technique can be coupled with a fault localization technique to get the information on which functions are likely to contain the fault. This information can be used to prioritize the equivalent sequences, as the Automatic Workarounds mechanism would select these equivalent sequences that do not contain the functions that are more likely to contain the fault, first.

Combination of the above: The priority of the equivalent sequences can be computed also as a combination of the above strategies.

Priority schemes based on distance and fault localization require additional information, that is the failing sequence in the first case and the fault localization information in the second case. This information is not easy to get in Web applications, especially when no runtime exception is raised by the JavaScript code. This was the case for most of the failures we examined in our experiments.

Thus, in the Automatic Workarounds technique we implemented the priority scheme based in the history that although rather simple is quite effective in practice (see Chapter 6).

Priority Based on History

The *history* scheme assigns a priority that is defined as a pair of values ⟨*success-rate*, *success*⟩, where

$$\begin{array}{lcl} \textit{successRate} & = & \textit{no. of successful applications/no. of uses} \\ \textit{success} & = & \textit{no. of successful applications} \end{array}$$

The *successRate* is the ratio between the number of times the equivalent sequence has been used successfully as a workaround, and the total number of times it has been used. The *success* is the total number of times the equivalent sequence has been used successfully as a workaround. A sequence with higher *success-rate* than another is given higher priority. When two sequences share the same *success-rate*, the priority is given to the one with higher *success*. In other words, priority $p_1 = (r_1, s_1)$ is greater than priority $p_2 = (r_2, s_2)$ if $r_1 > r_2$ or if $r_1 = r_2$ and $s_1 > s_2$. Both *success-rate* and *success* are initialized to 1 and updated at each application of the sequences.

When two or more equivalent sequences have the same priority, which is the case with the first failure, we heuristically use *alternative* rules first, then *invariant* rules, and finally *null* rules. We prefer alternative operations over invariant or null operations because those replace code in the failing sequence, and therefore are more likely to avoid faults. We then prefer invariant operations over null operations because the former are API-specific and therefore are more likely to mask faults in the API.

In summary, in the absence of any recorded workaround, we start from alternative operations, then continue with invariant and null operations. In the presence of recorded workarounds, we try those in order of success rate and, among the ones with the same success rate, in order of the absolute number of successes.

It is often the case that Web library developers are aware of some potential problems that may arise when the API is deployed, and most of the times they are also aware of some workaround to avoid the problem. In the Google Maps API, for instance, the `setCenter` function, which centers the map into a given geographical location, is critical, since it has to be invoked right after the call to the constructor. However, several issues that users reported could be solved by delaying the execution of this function. Google developers know that this workaround is very successful, and in fact they often suggest it, even without analyzing the problem.

The priority mechanism can improve a lot with this knowledge, especially at the very beginning, when all the values are set to 1. We thus give the possibility to API developers to specify a third priority, which can be a value from 0 to 1 to select which equivalent sequences are more likely to work, according to them. If this value is specified, then it is considered as a third priority.

4.5 Automatic Oracle

Prioritization schemes increase the probability of selecting equivalent sequences that can be turned into valid workarounds, but cannot guarantee success, especially in the early phases when little or no information is available on the effectiveness of the available rules. Ideally, we could build an automatic oracle that is able to discard all the failing attempts, and approve only the valid workarounds. Such an oracle would be able to remove the user from the loop, and would completely automate the Automatic Workarounds process. However, building such oracle is infeasible, given that it would require the complete specification of each Web page to automatically decide whether the Web page is behaving correctly (i.e. the page has been fixed), or whether it still fails (i.e. the equivalent sequence did not work).

We propose to generate a partial oracle, instead, which is based on the comparison between generated and failing pages. This oracle can identify those equivalent sequences that do not change the behavior of the faulty Web page, i.e. those sequences that produce Web pages that fail in a way that has been observed already. This oracle does not present these pages to the user, and discards them automatically.

More specifically, when the user reports a failure, the oracle saves the HTML and JavaScript code of the original (failing) page. The oracle compares the new page generated with the selected program rewriting rule with the page recorded from the failing execution. If the pages show no structural differences, that is, if their DOM representations are identical, then the oracle rejects the proposed solution. Otherwise, if the structure of the pages is different, then the oracle accepts the proposed equivalent sequence and shows it to the user. In fact, if the generated and the failing pages do not differ, the users would not perceive any difference, and thus would reject the page. Otherwise, the page is presented to the user who can accept the page or signal a further failure. In this last case, the page is added to the current set of failing pages, which is cleared when the user decides to proceed with a new page.

Figure 4.7 shows an example of a failing Web page, and three attempts to find a workaround. The screenshot on top shows a Google Map that is affected by an issue regarding draggable markers in non draggable maps [7]. Any click on the draggable marker is ignored, and thus the information window is not visible. This is the original page, and it is what the user sees at the beginning before starting the Automatic Workarounds process reporting that this page is not working correctly. The second screenshot shows an attempt to fix the map by inserting the invariant operation `map.enableDragging(); map.disableDragging()`. This attempt does not change the Web page, and the new page has exactly the same DOM of the original failing page. Therefore, the oracle can automatically discard it. The third screenshot shows the result of the attempt to fix the map by delaying the execution of the creation of the map (`map = new GMap2(...)`). Although this attempt does not produce

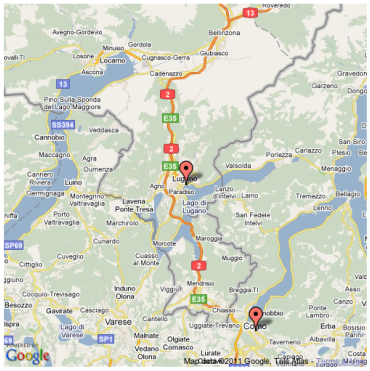
[7] `http://code.google.com/p/gmaps-api-issues/issues/detail?id=33`

a correct page, it fails in a different way, and thus the DOM of the new web page is different. Consequently, the oracle approves the equivalent sequence, and shows the page to the user. The user discards this attempt. The last screenshot shows the final attempt, which produces a valid workaround. The solution is to delay the execution of the `openInfoWindowHtml` function on the faulty marker. The DOM of the page is different from any of the previous discarded attempts, thus the oracle accepts the equivalent sequence and shows the new page to the user, who can approve the workaround.

The DOMs of the web pages have be compared after the the execution of the actions that show the failure (in the example, the DOM should be extracted after the click on the draggable marker). Capture and replay techniques can be used to automate the re-execution of the user actions.

Notice that any valid workaround would change the observable behavior of the page, and consequently its structure. Thus, the oracle acts conservatively, by accepting any change as a potentially valid workaround.

Pages rejected by the oracle are interpreted as pages rejected by the user for the purpose of computing the priorities associated with rules.

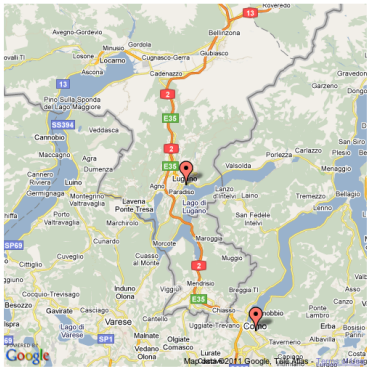


Original JavaScript code:

```
map = new GMap2(document.getElementById("map"));                    1
map.setCenter(new GLatLng(46.003,8.953),10);                        2
map.disableDragging();                                              3
marker = new GMarker(new GLatLng(45.81,9.08),{draggable:true});     4
GEvent.addListener(markerDrag, "click", function(){                 5
  marker.openInfoWindowHtml('This marker can be dragged'),500});    6
map.addOverlay(marker);                                             7
```

DOM of the original failing page:

```
<img style="width: 20px; height: 34px; [...]
position: absolute; left:337px; top: 417px; cursor: pointer;"
src="http://maps.gstatic.com/intl/en_ALL/mapfiles/markerTransparent.png"
class="gmnoprint" id="mtgt_unnamed_1" title="Como">
```



First attempt, invariant operations (line 4):

```
map = new GMap2(document.getElementById("map"));                    1
map.setCenter(new GLatLng(46.003,8.953), 10);                       2
map.disableDragging();                                              3
map.enableDragging(); map.disableDragging();                        4
...                                                                 5
```

DOM of the page with invariant operations. Discarded by oracle:

```
<img style="width: 20px; height: 34px; [...]
position: absolute; left:337px; top: 417px; cursor: pointer;"
src="http://maps.gstatic.com/intl/en_ALL/mapfiles/markerTransparent.png"
class="gmnoprint" id="mtgt_unnamed_1" title="Como">
```

Second attempt, delay creation of the map (line 1):

```
setTimeout('map = new GMap2(document.getElementById("map"))',500);  1
map.setCenter(new GLatLng(46.003,8.953), 10);                       2
map.disableDragging();                                              3
...                                                                 4
```

DOM of the page with delay. Discarded by user:

```
<img style="position: absolute; left: 0px; top: 0px; width: 256px;
height: 256px; -webkit-user-select: none; border-top-width: 0px;
border-right-width: 0px; border-bottom-width: 0px; border-left-width:0px;
border-style: initial; border-color: initial; padding-top: 0px;
padding-right:0px; padding-bottom:0px; padding-left: 0px;margin-top: 0px;
margin-right: 0px; margin-bottom: 0px; margin-left: 0px; "
src="http://maps.gstatic.com/intl/en_ALL/mapfiles/transparent.png">
```

Third successful attempt, delay opening window (line 6-7):

```
marker=new GMarker(new GLatLng(45.81,9.08),{draggable:true});       4
GEvent.addListener(marker, "click", function(){                     5
  setTimeout("marker.openInfoWindowHtml('This marker can be         6
    dragged')",500);                                                7
});                                                                 8
map.addOverlay(marker);                                             9
```

DOM of the page with workaround:

```
<div style="position: absolute; left: 16px; top: 16px;
 width: 217px; height: 58px; z-index: 10; "><div>This marker can be
dragged</div></div>
```

Figure 4.7. The oracle automatically discards the first attempt. Thus, the user evaluates only one failing page (beside the original faulty one) before having the page fixed.

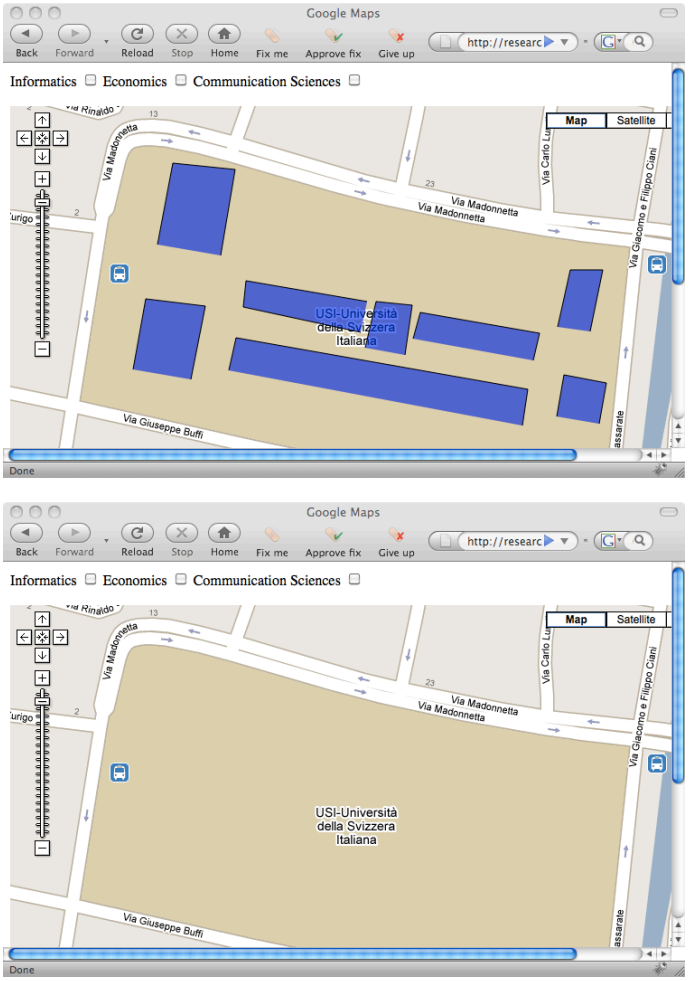
Chapter 5

Prototype Implementation

To evaluate the effectiveness of the Automatic Workarounds technique we developed a browser extension as a prototype. This extension enables a button on the browser toolbar whenever the currently visualized Web page contains references to any of the Web APIs for which we have a set of program rewriting rules. The user can report problems in the current Web page through this button, and can evaluate the attempts of the Automatic Workarounds module to fix the problem. This chapter provides the details of the design and the implementation of the browser extension.

We implemented the Automatic Workarounds technique described in Chapter 4 in RAW (Runtime Automatic Workarounds), which is a browser extension for the Google Chrome and Firefox browsers. The extension does not affect the normal Web browsing activity, since it does not interfere with the browser until the user is willing to activate it in reaction to a page failure.

RAW extends the browser interface by adding three control buttons: *Fix me*, *Approve fix* and *Give up*, as shown in the screenshots in Figure 5.1. The Firefox extension adds all the three buttons to the toolbar. The Chrome extension, instead, has only the *Fix me* button on the toolbar, and only when the user presses this button, it opens a pop-up window containing the other control buttons. The *Fix me* button activates the automatic generation of workarounds. It is active only when the currently loaded Web page uses some Web APIs supported by RAW (e.g., Google Maps and JQuery, for which RAW has a set of equivalent sequences). A user experiencing a failure (actual or perceived) with the displayed page may press this button to report the problem and request a workaround at runtime. The *Approve fix* and the *Give up* buttons become active only after pressing the *Fix me* button, and allow the user to notify the success (*Approve fix*) or failure (*Give up*) of the generated workarounds. By pressing the *Fix me* button the user can request another workaround, implicitly saying that the one that has been proposed is not effective. The *Approve fix* and *Give up* buttons are not essential for the user viewpoint who can ignore them, but, if pressed, they provide useful information



Faulty page

The page shows all the polygons even if checkboxes are not selected

Page automatically corrected by RAW

The page does not show the polygons when checkboxes are not selected

Figure 5.1. Issue n. 1264 in Google Maps

for the priority scheme used by RAW to select which equivalent sequence to apply.

We illustrate the functionality of RAW through an example that refers to a known (and now fixed) problem of the Google Maps API, reported as issue n. 1264 in the Google Maps bug-report system.[1] Figure 5.1 presents two screenshots of a simple Web page that we wrote to show this Google Maps issue. The page shows the map of the University of Lugano campus, and offers a set of checkboxes to display the buildings of the faculties as polygonal overlays. More precisely, the buildings of each faculty should appear on the map only when the checkbox of the corresponding faculty is checked, and they should be hidden otherwise.

Initially, the page should not display the polygons, which should become visible only after selecting the checkboxes on top of the map. Moreover, when visible, the

[1] `http://code.google.com/p/gmaps-api-issues/issues/detail?id=1264`

polygons should scale according to the zoom level. As illustrated by the screenshot on the top of Figure 5.1, the initial page is not displayed correctly, as all polygons are visible with no selected checkbox. Zooming into the page would also show that the polygons do not scale as expected.

With a standard browser interface, users who experience this problem have no way to react and solve the problem. They might attempt to reload the page several times, but once they notice that the problem is deterministic they can only report the problem to the application developers and hope for a fix sometime in the future. With RAW, users experiencing this problem can ask for the help of the tool by pressing the *Fix me* button in the toolbar (screenshot on the top of Figure 5.1). The *Fix me* button activates RAW to look for a workaround. When the user requests a workaround for the first time, RAW extracts the JavaScript code of the current page, applies one of the program rewriting rules and reloads the page with the new JavaScript code. If not satisfied by the reloaded page, the user may do further attempts and request new workarounds by pressing the *Fix me* button multiple times. The screenshot on the bottom of Figure 5.1 shows the correct application behavior fixed by RAW. If satisfied by the reloaded page, the user may report the successful workaround by pressing the *Approve fix* button. The user may press the *Give up* button to stop searching for valid workarounds. RAW prints an information message when there are no program rewriting rules left to apply, thus warning the user that a workaround cannot be found.

In implementing RAW we made the implicit assumption that users are not malicious, and they are thus interested in reporting correct information when approving or rejecting fixes. Although malicious users cannot use RAW to attack the Web applications, they can reduce the effectiveness of the tool by providing incorrect feedback.

Popular Web applications may have thousands or even millions of visitors, and simple problems like the one illustrated above may affect many users for a long period before the application or API developers might be able to provide a fix. In order to handle recurring problems more efficiently, when users notify the system of a successful workaround, RAW keeps track of it by recording the URL, the workaround, and the specific execution conditions (i.e. browser and operating system).

When a user loads a page for which RAW has a known workaround, the *Fix me* button on the toolbar is slightly different to notify the presence of workarounds. Thus, if the Web page does not display correctly, the user would immediately know that other users experienced some problems in the same page, and that workarounds are available for those problems. Although RAW could apply workarounds automatically when a user visits a Web page that is known to fail, we prefer to be conservative and be sure that the same page fails also for the new user. Thus, even if a workaround is known to work for a Web page, RAW applies it only when the user requests it, as not every user may need a workaround in first place, and the known workaround may not work under all specific circumstances.

RAW also saves information about URL, workarounds and execution conditions for

the benefit of application developers, who may take advantage of this information to diagnose and remove faults. More precisely, developers can use RAW to express their interest in being notified when users report problems to specific Web pages, and have a list of the workarounds that users report as valid.

The initial prototype of RAW has been implemented as a HTTP proxy. The proxy modifies the Web pages that use any of the supported Web APIs such that the control buttons are added to the Web page as a `<div>` element on top of the page. Although this solution had the advantage of being compatible with all the browsers, we preferred the browser extension implementation over the HTTP proxy solution because the latter may break the design of the Web pages.

The following section provides further details about the implementation of the browser extension components.

Internals of RAW

RAW comprises a client- and a server-side subsystem. The client-side subsystem is the browser extension that implements the user interface described in the previous section, and it is mainly written in JavaScript. The server-side subsystem, instead, implements the main functionality of the Automatic Workarounds technique in Python, and runs on a centralized server.

More precisely, the browser extension checks the header of the Web page requested by the user, and looks for references to JavaScript libraries supported by the tool. If any reference is found, then it activates the control buttons on the toolbar. When the user presses the *Fix me* button, the browser extension extracts the JavaScript code in the Web page together with the information on which Web API is used (i.e. name and version of the API, if multiple versions are available). Then it sends this information to the server that manages the database of the program rewriting rules. The server side selects a program rewriting rule to apply, depending on the priority scheme, applies it to the JavaScript code, and sends the code back to the browser extension. Finally, the browser extension substitutes the JavaScript code in the DOM of the current page with the new code, and reloads the page. The page reload, however, is not an actual reload, as in this case the Web page would be requested again from the remote server, and thus the original faulty JavaScript code would be executed again. To reload the page and execute the new code without retrieving again the content from the remote server, RAW fires a `load` event.

Client and server side communicate thanks to `XMLHttpRequest` objects. Usually the JavaScript code in a Web page is limited by the *same origin policy*, which forces the client to send and receive data only from the server that hosts the Web page itself. Browser extensions do not have this limitation, though, and consequently RAW can send requests to the external server that hosts the server-side components.

The automatic oracle, which automatically discards Web pages that look like the

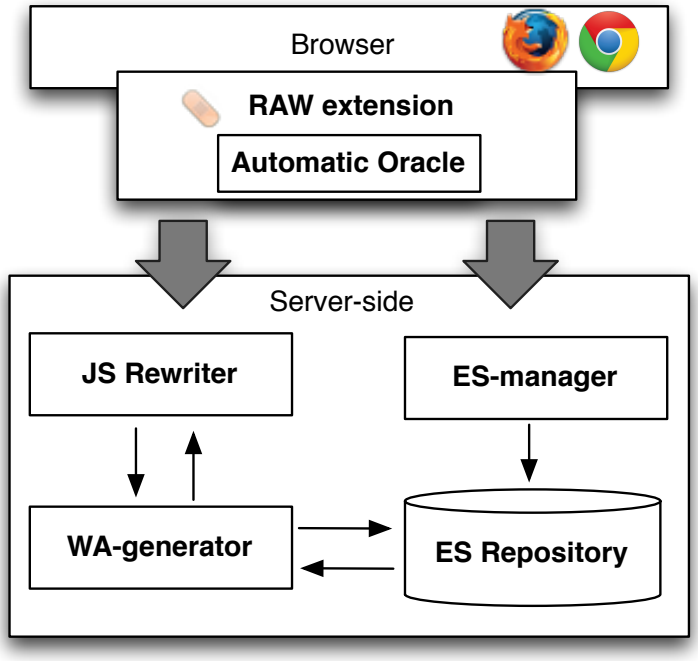


Figure 5.2. Overall architecture of RAW

ones that the user reported as failing, stands on the client-side. Once the new JavaScript code has been executed, the browser extension extracts the DOM of the new Web page and compares it to the previous ones. If it automatically discards the attempt, then it automatically asks for a new attempt, thus sending another request to the server side.

We now separately describe the components shown in Figure 5.2: the *ES Repository*, which contains the program rewriting rules, the *JS Rewriter*, which applies the changes to the JavaScript code, the *WA Generator*, which is responsible of selecting which rule to apply, the *ES Manager*, which is an interface to manage program rewriting rules, and the *Automatic Oracle*, which automatically discards some non valid attempts.

ES Repository. The ES Repository is the core of RAW and contains a set of program-rewriting rules that specify equivalent sequences. Each rule indicates the *type*, *scope*, and *substitution* that characterize the equivalent sequence. The *type* gives the category of the equivalent sequence (functionally null, invariant, or alternative), and is used by the priority mechanism. The *substitution* is defined as a non-empty set of pairs of pattern and replacement specified with regular expressions. The *scope* indicates if the rule must be applied to ANY or ALL occurrences of the substitution pattern in the JavaScript code.

For instance the following program rewriting rule

```
alternative ALL:
```

```
$X.addOverlay($Y) → $X.addOverlay($Y); $Y.show()
$X.removeOverlay($Y) → $Y.hide()
```

specifies that adding an overlay to a map is equivalent to adding the overlay and explicitly showing it, and that removing an overlay from a map is equivalent to hiding it.

Similarly, this program rewriting rule

```
alternative ALL:
$X.addOverlay($Y) → setTimeout('$X.addOverlay($Y)', 500)
```

specifies that it is equivalent to add an overlay and to delay this operation of few milliseconds.

Both these rules can be applied to the same pattern (`$X.addOverlay($Y)`), and both of them can generate valid, and previously unknown, workarounds for the example described in Figure 5.1. Each rule is stored in the database with a unique id, and it is associated with a specific API. Moreover, each rule has three priority values in the database. The first one is the success rate (i.e. the number of successful attempt divided by the number of times the rule has been applied). The second priority value is the number of successful attempts. RAW computes these two values depending on the information reported by the users through the *Approve fix* and *Fix me* and *Give up* buttons. The third value of the priority, instead, is by default set to 1. However developers can change this value (by specifying a number between 0 and 1) to express their experience about the effectiveness of the rule.

ES Manager. The ES Manager is a Python module that provides an interface for Web applications and API developers to update the information stored in the ES Repository. Developers initialize the ES Repository with a set of rules generated from the specifications of the Web API, and they can express their priority by setting a value from 0 to 1 for the third priority value in the database. Even when RAW is running they can update the rules, and they can add new rules by coding workarounds that were manually discovered.

Developers can also use the ES Manager to extract information about successful workarounds and failure conditions to identify and implement permanent fixes for the application.

JS Rewriter. The JS Rewriter is the Python module in charge of applying a program rewriting rule to the JavaScript code. When the user reports a failure in the current web page, the RAW browser extension extracts both the JavaScript code and the information about the API used in the current Web page, and forwards all the data to the JS Rewriter. The JS Rewriter forwards the information about the API to the WA Generator, which returns a rewriting rule complying with the priority scheme. The JS

Rewriter uses sed[2] to implement the substitutions specified by the rule. If the rule that has been selected by the WA Generator does not produce any change in the JavaScript code, the JS Rewriter asks the WA Generator for another rule, otherwise, it returns the generated JavaScript code to the RAW browser extension. The extension replaces the original JavaScript code with the newly generated one, and reloads the page.

WA Generator. The ES Repository contains several rules, and each rule may be applied in many ways to the same JavaScript code, but only a few applications of rules may generate valid workarounds. Applying rules indiscriminately may generate many useless equivalent sequences that will soon annoy the users. The WA Generator is the Python module responsible for selecting the rules that are more likely to generate valid workarounds. To rank rules it uses the priority scheme, described in Section 4.4, that is based on the history of successes and failures of the rule, and is defined as a pair ⟨success-rate, success⟩. The rules applicable to a given API are ranked by success-rate and then success. The priority specified by developers is also used as a third ranking value, if present. Thus, when the JS Rewriter requests a rule, the WA Generator selects the rules associated to the API used, and looks for known workarounds for the current URL. If any rule is known to solve the problems of the Web page that RAW is trying to fix, then this rule is selected and applied first. If no workaround is known already, then the WA Generator selects the next rule with the highest priority. When the JS Rewriter asks for a new rule to apply, it provides the list of the previous attempts as a list of rules IDs. This list of attempts is stored on the client side. Thus, before selecting a new rule from the ES Repository, the WA Generator reads the ID of the latest rule that has been applied, and updates its priority values. More precisely, it is updated positively if the attempt has been successful (i.e. the user pushed the *Approve Fix* button), negatively if the attempt has failed (i.e. the user either requested a new workaround or pushed the *Give up* button), and stays unchanged if the previous rule could not be applied (i.e. the substitution applied by the JS Rewriter produced the same result).

Automatic Oracle. The Automatic Oracle is the client side module that automatically discards some failing attempts, and it is implemented in JavaScript.

Every time the user reports a failure by pushing the *Fix me* button, this module extracts the DOM of the Web page, it computes its hash value and stores it. This is done by having the following JavaScript code injected and running in the context of the Web page:

```
var body = document.getElementById(‘‘content’’).contentDocument.body;
var DOMcurrent = body.innerHTML;
```

Every time RAW applies a new rule to the original JavaScript code, and it re-executes the new JavaScript code, it invokes this module to extract the DOM of the newly gen-

[2]http://www.gnu.org/software/sed

erated page. The hash value of the current DOM is then compared to all the hash values of the DOMs of the failing Web pages. If it is equal to any of them, then RAW automatically discards it, and the JS Rewriter module is invoked again.

To comply with the description of the automatic oracle of Section 4.5, this module should include a capture and replay mechanism, in order to reproduce all the actions done by the user before extracting the DOM of the page. This is required to be sure that the failure is visible when the DOM is extracted. However, the current implementation has one limitation. Instead of actually recording the actions done by the user to reproduce them after reloading the page, RAW expects to have the list of events to reproduce. In the future we plan to integrate RAW with any of the existing open source capture and replay tools (e.g. ecarena [3]), to completely automate this step.

[3] `http://code.google.com/p/ecarena-firefox-extension`

Chapter 6

Evaluation

To evaluate the Automatic Workarounds technique we first studied the prevalence of workarounds to address failures in Web applications, and the possibility of generating them automatically. Then we evaluated the effectiveness of the automatically generated workarounds to deal with issues without known solutions. The results show that workarounds are widely used by both application developers and end users to deal with unexpected failures, and most of the workarounds can be generated automatically thanks to intrinsic redundancy. Finally, we show that our technique can generate valid workarounds to address known issues of three popular Javascript APIs (Google Maps, YouTube and JQuery) for which no workaround was known before.

As stated in the introduction of this dissertation, our research hypothesis is that *"software is intrinsically redundant, and this type of redundancy can be captured, represented and exploited for several purposes, in particular for fault-handling at runtime"*.

Several studies confirm our intuition that software is intrinsically redundant [JS09; GJS08], and the examples that we mentioned throughout the thesis support this hypothesis. In Section 4.3 we showed how intrinsic redundancy can be represented in the form of program rewriting rules. However, so far we have provided little evidence that intrinsic redundancy can be exploited to effectively address failures at runtime. Thus, we focus our studies towards the validation of the last part of our research hypothesis.

To validate our hypothesis that intrinsic redundancy can be used for fault-handling at runtime, in other words to validate the notion of Automatic Workarounds, we need to study the effectiveness of workarounds in dealing with runtime failures, and estimate the amount of workarounds that can be generated thanks to intrinsic redundancy. We also have to evaluate the effectiveness of the technique to deal with faults for which no solution is known.

Since we implemented the notion of Automatic Workarounds in the context of Web applications, we articulate our studies within this scope. Our evaluation can be summarized with the following three research questions:

Q1 Can workarounds cope with failures effectively?

Q2 Can workarounds be generated automatically?

Q3 Can Automatic Workarounds generate valid workarounds?

The first question (Q1) explores the possibility of using workarounds to handle failures, and their effectiveness. In other words, we ask whether workarounds even *exist* and whether they can be used with Web applications. A positive answer to this first basic question then leads directly to the second question (Q2), which asks whether it is possible to generate workarounds *automatically* thanks to intrinsic redundancy or whether that is an inherently creative activity that should be left to human designers. Finally, the third question (Q3) evaluates the ability of the technique presented in Chapter 4 to generate and deploy valid workarounds, in particular to address issues for which no valid workaround is known.

Our general method of evaluation uses a two-pronged experimental approach. To provide an answer to Q1 and in part to Q2, and also to gain a better understanding of typical failures in Web applications, we surveyed the fault repositories and other on-line forums dedicated to popular Web APIs. Then, to provide a constructive answer to Q2 as well as a direct answer to Q3, we relied on RAW, the prototype implementation presented in Chapter 5. We now briefly outline the survey method, and then turn to the specific results we obtained pertaining directly to the three research questions, examining each question in turns.

6.1 Survey of Failure Reports

For our survey of failures in Web applications we proceeded as follows. We first identified a number of sources of failure reports. In some cases (e.g., Google Maps, YouTube and JQuery) these were official and specialized bug-tracking systems. In other cases (e.g., Yahoo! Maps, Microsoft Maps, Flickr, Picasa) we had to rely on on-line discussion forums. Of all the official and unofficial reports, we selected those that we thought might reveal useful information on workarounds, using simple textual searches. We then examined all the failure reports that even superficially indicated the possibility of a workaround to exclude irrelevant ones and to precisely identify failures and workarounds for the relevant ones.

In some cases, we were fortunate enough to be able to replicate failures and workarounds. We could easily do that with JQuery, since it is an open source library, and the whole version history is publicly available. We could also do that with the Google Maps library, since Google used to export the complete version history of their Web APIs. Unfortunately, during the evaluation process Google removed many program versions from its publicly available history, so we were unable to complete all the experiments we had planned. In particular, we could not repeat all our initial experiments with

the automated oracle described in Section 4.5. In the case of YouTube, instead, we could only replicate the failures that have not been fixed, since YouTube does not have versions for its API.

These cases, where we could replicate failures and workarounds, are particularly important because they provide concrete case studies for the evaluation of our prototype. We report extensively on these cases in the following sections. In all other cases, we could only rely on the description of the failure. Nevertheless, all the failure reports we analyzed were useful in characterizing failures and typical workarounds, and are the basis for the taxonomy of equivalent sequences synthesized in Sections 3.2 and 4.2.

6.2 Prevalence of Workarounds in Web Applications

Our study of several repositories and forums dedicated to open as well as fixed faults in popular Web APIs indicates that workarounds exist in significant numbers, and are often effective at avoiding or mitigating the effects of faults for Web applications. To quantify the effectiveness of workarounds to mitigate the effects of faults, we limited our analysis to the Google Maps and the YouTube chromeless player issue trackers. We selected these issue trackers because they contain enough issue reports to get useful information, but the number of issues is not too large, so that it was possible to manually inspect them. The results of our survey for Google Maps and YouTube are summarized in Table 6.1.

API	*reported faults*	*analyzed faults*	*actual workarounds*
Google Maps	411	63	43 (10%)
YouTube	21	21	9 (42%)

Table 6.1. Faults and workarounds for the Google Maps and YouTube API

We studied the Google Maps API issue tracker in July 2009 and we found a total of 411 faults. We excluded the bug reports that were marked as *invalid* by the Google team. We selected the entries potentially related to workarounds by matching the keyword "workaround" in the bug report descriptions and we obtained 63 entries. We then focused on these entries, ignoring other possible workarounds not marked explicitly as such. Upon further examination, we found that 43 of them were proper workarounds. The other 20 bug reports had the keyword "workaround" in their description, but they do not report any valid workaround to avoid the problem. Most of the times the word "workaround" occurs in comments such as "*Does anybody know a workaround for this issue?*" that do not report any valid workaround. In minor cases users reported workarounds that were not working. We verified that for all the 43 Google Maps API issues the workaround that was reported in the description was actually working, and we

did it by reproducing the failure, and by avoiding it with the reported workaround. In one case it was impossible to reproduce the problem, since it was due to transient environment conditions [1].

In total, 43 amounts to about 10% of the reported faults. This result indicates that workarounds can successfully address a good amount of API runtime issues. However, we should note that the 10% prevalence of workarounds for the Google Maps API is a conservative estimate, since we analyzed only 63 out of 411 reports. We probably missed several valid workarounds that were reported without using the keyword "workaround" in the description. Moreover, several issues have a workaround, although not publicly known.

To get additional information on the effectiveness of workarounds to deal with faults in Web APIs, we considered the bug tracker of the YouTube chromeless player, which included 21 known issues at the time of the investigation (July 2009). Given the modest size of the repository, we analyzed all issues without resorting to any prefiltering. Out of the 21 reports, we identified 9 workarounds, corresponding to about 42% of all issues. This second result confirms that workarounds can effectively address many runtime issues. Unfortunately, we could not reproduce the issues there were already fixed at the time of investigation, because YouTube has a single version of the API that is publicly available. Consequently, we could not verify the validity of the proposed workarounds, and we had to rely on the comments of other users who were confirming their effectiveness.

The data collected with the analysis of the Google Maps and YouTube API issue trackers cannot be generalized to conclusive quantitative results, but they nonetheless indicate that it is often possible to overcome Web APIs issues by means of appropriate workarounds.

6.3 Automatic Generation of Workarounds

Having observed that some failures of Web APIs can be fixed with workarounds, we wanted to assess whether at least some of these workarounds can be generated automatically from a set of equivalent sequences. To that end, we further analyzed the 52 workarounds that we found in the bug-tracking repositories of Google Maps and YouTube (43 and 9, respectively). In particular, we tried to distinguish between ad-hoc, hardly generalizable workarounds that require human reasoning to be developed and deployed, from more general workarounds that exploit the intrinsic redundancy of the APIs, and thus are in the form of equivalent sequences.

To illustrate the notion of an ad-hoc workaround, consider issue n. 40 from the bug-tracking system of Google Maps [1]. The report states the following:

> Some previously working KML files are now reporting errors when entered

[1] `http://code.google.com/p/gmaps-api-issues/issues/detail?id=40`

> in Google Maps ... The team confirms this is due to problems with Google fetching servers right now. Moving the file to a new location is a possible temporary workaround.

The KML files mentioned in the report are files that the application must make available to the Google Maps system by posting them onto an accessible Web server. However, due to some problem with its internal servers, Web applications using the Google Maps API could not access the KML files, thereby causing an API and application failure. The proposed workaround amounts to moving the KML files on a different server that the Google servers could access correctly.

This report offers a good example of a workaround that is not amenable to automatic generalization and deployment. This is the case for a number of reasons. First, the workaround is tied to an internal functionality of the Google Maps application. Second, the workaround has almost nothing to do with the code of the application, and in any case can not be implemented by changing the application code. Third, the solution involves components that are most likely outside of the control of the client application or anything in between the client application and the application server. Fourth, the report indicates that the problem exists "right now," and therefore might be due to a temporary glitch, which is unlikely to generalize to a different context at a different time.

By contrast, consider the workaround proposed for issue n. 61 from the same Google Maps bug tracker [2]. The report reads as follows:

> Many times the map comes up grey ... a slight setTimeout before the setCenter ... might work. ... if you change the zoom level manually ... after the map is fully loaded, it will load the images perfectly. So, what I did was add a "load" event ... and had it bump the zoom level by one and then back to its original position after a really short delay.

The report describes a problem with maps that were not visualized correctly when dynamic loading was used. External Javascript files can be loaded by either adding their reference in the HEAD section of a web page, or on demand (dynamic loading). The second strategy was causing problems with maps during the loading phase, as the map was coming up grey. This report contains two workarounds that are examples of equivalent sequences: *add a setTimeout* and *add a zoom-in-zoom-out sequence*, which are good examples of null and invariant operations, respectively, as defined in Section 3.2 and 4.2.

Table 6.2 summarizes the results of this analysis: 32% and 55% of the known workarounds found in the Google Maps and YouTube repositories respectively are instances of equivalent sequences, and therefore are good candidates for automatic generation. This analysis provides some evidence that workarounds can be generated at runtime from equivalent sequences.

[2] `http://code.google.com/p/gmaps-api-issues/issues/detail?id=61`

API	*analyzed workarounds*	*reusable workarounds*
Google Maps	43	14 (32%)
YouTube	9	5 (55%)

Table 6.2. Amount of reusable workarounds

6.4 Effectiveness of Automatic Workarounds

To investigate the effectiveness of the Automatic Workarounds technique we performed various experiments with the Google Maps, YouTube and JQuery APIs. These experiments aimed to evaluate the ability of the technique to find valid workarounds, both for faults for which a workaround was already known, and for faults for which no solution was publicly known.

API developers can write program rewriting rules in two ways, that is as a reaction to issues, and as a proactive attempt to address potential future issues. In the first case, which we call reactive approach, developers can insert a program rewriting rule in the repository when an issue and a respective workaround have been found. In the second case, which we call proactive approach, developers provide a list of program rewriting rules as an additional form of specification for the API. In the first experiment we evaluated the reactive approach, and in particular we aimed to assess the reusability of workarounds, that is the ability of known workarounds to deal with new problems (Section 6.4.1). In the second experiment we evaluated the proactive approach, which is the technique as described in Chapter 4. Therefore before running the experiment we populated the database of program rewriting rules by looking at the API specifications (Section 6.4.2). In the last experiment we selected a subset of the issues used in the previous experiments, and we focused our attention on the effectiveness of the priority mechanism and the automatic oracle in presence of several candidate workarounds (Section 6.4.3).

6.4.1 First Experiment: Reactive Approach

The goal of the first experiment is to assess whether workarounds are reusable in different contexts, in other words whether some workarounds that are known to solve some issues can be a valid solution for other issues.

Therefore we simulated the incremental reporting of failures following their chronological order, starting with an empty repository of rewriting rules, and considering the 14 issues with workarounds identified in the study described in Section 6.3. Moreover, we selected 24 new issues for which no workaround was known. We selected the 24 issues without known workarounds among the issues on the tracker that could be easily reproduced and had already a sample Web page showing the failure.

We considered each issue sequentially in ascending order of index number (in the issue-tracking system), which corresponds to their chronological order. We first tried to solve each issue automatically, using the rules available in the repository at that time. Whenever the issue had a known workaround, and we could not find it automatically (i.e., the workaround was not coded in any of the rules in the repository), we manually added it into the repository, to make it available for subsequent issues.

Table 6.4 shows the results of this experiment. The first and the last columns show the issue number and a reference to the corresponding workaround listed in Table 6.3. The central columns indicate whether the generation of a valid workaround was possible or not. In positive cases we report if the workaround was automatically generated (i.e., thanks to workarounds of previous issues), or if it was generated manually (i.e., the issue was solved thanks to a new workaround that was reported in the issue tracker discussion, and was not coded as a program rewriting rule in the repository yet).

The salient result is that one third of the issues (13 out of 38) could have been solved by workarounds generated automatically on the basis of other workarounds identified while solving previous issues. In particular, half of the issues (7 out of 13) for which a workaround was found automatically did not have any workaround that was known before. These issues are marked in grey in Table 6.4.

These results, although limited to a few issues of a single API, are very positive, because they show that several workarounds can be effective multiple times and in different contexts. We can see, for instance, that rules G2 and G4 in Table 6.3 can solve 8 and 7 issues each respectively. Therefore, if Google developers coded the known workarounds in the form of program rewriting rules, they would have solved automatically several subsequent issues.

6.4.2 Second Experiment: Proactive Approach

To investigate the effectiveness of the technique as it is described in in Chapter 4, we performed various experiments with the Google Maps, YouTube and JQuery APIs.

We first populated the repository of rewriting rules with three sets of rules, one for each of the three selected APIs. We generated these rules by studying the APIs of the three libraries, and by instantiating the three classes of rules introduced in Section 4.2. In total, we wrote 39 rules for Google Maps, 40 rules for YouTube, and 68 for JQuery. A subset of the Google Maps and YouTube rules are listed in Table 6.3, with labels G1–G14 and Y1–Y6 for Google Maps and YouTube, respectively. A subset of the rules for JQuery is listed in Table 6.5.

Google Maps

We started with the 14 problems that had known workarounds for Google Maps. These were the 14 issues that have been identified in the previous study (Section 6.3). We first of all verified that our prototype could automatically generate a workaround for

	Google Maps	
G1	null ALL:	$X.openInfoWindowHtml($Y); → setTimeout("$X.openInfoWindowHtml($Y)", 1000);
G2	alternative ALL:	$X.addOverlay($Y); → $X.addOverlay($Y); $Y.show(); $X.removeOverlay($Y); → $Y.hide();
G3	alternative ALL:	$X.hide(); → $X.remove();
G4	null ANY:	$X.setCenter($Y); → setTimeout("$X.setCenter($Y)", 1000);
G5	invariant ALL:	$X.show(); → $X.show(); $X.show();
G6	alternative ALL:	$X.setCenter($Y); $X.setMapType($Z); → $X.setCenter($Y, $Z);
G7	alternative ALL:	$X.disableEditing(); → setTimeout("$X.disableEditing()", 200); GDraggableObject.setDraggableCursor("default");
G8	alternative ALL:	$X.enableDrawing($Y); → var l=$X.getVertexCount(); var v=$X.getVertex(l-1); $X.deleteVertex(l-1); $X.insertVertex(l-1,v); $X.enableDrawing($Y);
G9	alternative ALL:	$X.getInfoWindow().reset($Y); → $X.getInfoWindow().reset($Y, $X.getInfoWindow().getTabs(), new GSize(0,0));
G10	null ALL:	$X.getVertexCount($Y); → setTimeout("$X.getVertexCount($Y)", 1000); $K.getBounds($Z); → setTimeout("$K.getBounds($Z)", 1000);
G11	alternative ALL:	$X.bindInfoWindowHtml($Y); → GEvent.addListener($X, "click", function()$X.openInfoWindowHtml($Y));
G12	alternative ALL:	var $X = new GDraggableObject($Y); -.setDraggableCursor($K); -.setDraggingCursor($Z); → var $X = new GDraggableObject($Y, draggableCursor:$K, draggingCursor:$Z);
G13	alternative ALL:	GEvent.addDomListener($X), "click", function(){$Y}; → $X.onclick = function(){$Y}
G14	null ALL:	GEvent.trigger($X); → setTimeout("GEvent.trigger($X)", 1000);
	YouTube	
Y1	alternative ANY:	$X.seekTo($Y); → $X.loadVideoUrl($X.getVideoUrl(),$Y);
Y2	alternative ALL:	$X.setSize($Y,$Z); → $X.width=$Y; $X.height=$Z;
Y3	alternative ALL:	$X.seekTo($Y); → $X.cueVideoByUrl($X.getVideoUrl(),$Y);
Y4	invariant ALL:	$X.stopVideo(); → $X.pauseVideo(); $X.stopVideo();
Y5	invariant ANY:	$X.stop() → $X.stop(); $X.stop();

Table 6.3. Some rewriting rules for the Google Maps and YouTube APIs

Google Maps				
issue	*workaround*			*rule*
	none	*manual*	*automatic*	
15	✓			–
29	✓			–
33		✓		G1
49		✓		G2
61		✓		G4
193	✓			–
240	✓			–
271	✓			–
315	✓			–
338			✓	G2
456			✓	G4
519			✓	G4
542	✓			–
585			✓	G2
588			✓	G4
597		✓		G7
715	✓			–
737			✓	G4
754			✓	G2
823	✓			–
826	✓			–
833			✓	G4
881		✓		G14
945			✓	G2
1020			✓	G4
1101	✓			–
1118	✓			–
1200	✓			–
1205	✓			–
1206	✓			–
1209			✓	G2
1234			✓	G2
1264	✓			–
1300	✓			–
1305		✓		G8
1511	✓			G13
1578			✓	G2
1802	✓			–

Table 6.4. Incremental reporting experiment with known Google Maps API issues

	JQuery
J1	alternative ANY: $('#x').set('disabled', false) → $('#x').get(0).disabled = false $('#x').set('disabled', true) → $('#x').get(0).disabled = true
J2	alternative ANY: $('#foo').attr('disabled', 'false'); → document.getElementById('foo').sheet.disabled= false; $('#foo').attr('disabled', 'true'); -> document.getElementById('foo').sheet.disabled=true;
J3	alternative ANY: $('select#Choice :selected').val() -> $('select#Choice :selected').text()
J4	alternative ANY: $("p").removeClass(""); → $('p').toggleClass("" , false);
J5	alternative ANY: $('newTag').hide().appendTo(document.body); -> $('newTag').appendTo(document.body).hide();
J6	alternative ANY: $('<input />').attr('type', 'checkbox') → $('☐')
J7	alternative ANY: $('.target').change() → $('.target').click()
J8	alternative ANY: $("#test").click() → $("#test").trigger("onclick")
J9	invariant ALL: $(this).attr('value', $(this).attr('originalvalue')) -> $(this).removeAttr('value'); $(this).attr('value', $(this).attr('originalvalue'));
J10	alternative ANY: $(#).val() → $(#).attr('value')
J11	alternative ANY: hasClass(#) → .attr('class').match(##)
J12	alternative ANY: :not(:first) → :gt(0)
J13	alternative ANY: $("newTag").wrap(" ").appendTo(document.body); → $("newTag").appendTo(document.body).wrap(" ");
J14	null ANY: ui.draggable.remove() → setTimeout(function() ui.draggable.remove(); , 1);
J15	alternative ANY: $("#resizable").resizable("option", "aspectRatio", .1); → $("#resizable").resizable('destroy').resizable(aspectRatio: .1);
J16	alternative ANY: $('#tag').attr('onclick', 'onclick1();'); → $('#tag')[0].onclick = function() onclick1(); ;
J17	alternative ANY: show → fadeIn, fadeTo, animate
J18	alternative ANY: .submit() → .find('input[type=submit]').click()
J19	alternative ANY: fadeIn(), fadeOut() → fadeTo() or show/hide
J20	alternative ANY: hide(##) → hide() – show(##) → show()
J21	alternative ANY: $(input).hide(); $("#id").after(input); $(input).show(); → $("#id").after(input); $(input).hide(); $(input).show();
J22	alternative ANY: .animate({ opacity: # }) → .fadeTo(0, #)
J23	alternative ANY: ('str').replaceAll('#div') → $('#div').replaceWith('str')
J24	alternative ANY: $("#divID").width(n); → $("#divID").css('width', n);
J25	alternative ANY: $('[autofocus]') → $('[autofocus=""]')
J26	alternative ANY: :visible → :not(:hidden)
J27	alternative ANY: div.animate(anprop, anopt); → div.animate(jQuery.extend(true, {}, anprop), jQuery.extend(true, , anopt));
J28	alternative ANY: $("ul:has(li:gt(1))") → $("li:gt(1)").parent()

Table 6.5. Some rewriting rules for the JQuery API

all of them with the current set of rules. We then added the 24 problems selected for the previous study on incremental reporting (Section 6.4.1). These issues had no workaround known, and they were selected among the ones that could be reproduced with a version of the API available in the Google Maps version history. We then replicated each of the 38 cases with RAW, following the chronological order given by their issue number, initializing RAW with the same priority $\langle 1, 1 \rangle$ for all the 39 rules. In a first run of these experiments, we used a prototype implementation of RAW that did not include the oracle. We then repeated the experiments with the latest prototype implementation of RAW that includes the oracle. However, unfortunately, during this second set of experiments, some versions of the Google Maps API were unavailable, so that we could only reproduce 24 of the original 38 failures.

Table 6.6 reports the results of the experiments just described. The first column (*issue*) indicates the issue number in the issue tracker. The following set of columns (*workaround*) reports the results of using RAW to generate workarounds. Specifically, *none* means that RAW could not generate any valid workaround; *known* means that RAW automatically generated a workaround that was already known; and *new* means that RAW generated a new workaround for an open problem. Out of the 24 issues for which no workaround was known, RAW could automatically find a solution for 15 of them. The fact that RAW can not only generate all known workarounds, but also many additional workarounds for open problems, provides an affirmative answer to our third research question (Q3) and confirms our general research hypothesis.

The *rule* column indicates the rule that generated the valid workaround (find the corresponding rule in Table 6.3). The experiment shows that workarounds are generated from different rules and that some rules can generate more than one workaround, thus confirming the first study.

The last two columns (*attempts*) are intended to measure the effectiveness of the priority scheme and the oracle. In particular, both columns indicate the number of user interventions required to either identify a valid workaround or to conclude that RAW could not generate any such workaround. The two columns labeled *no oracle* and *oracle* report the number of necessary interventions when RAW functions without or with its automatic oracle, respectively. Unfortunately, we could not reproduce all the failures for the *oracle* experiments, so that column is incomplete. The general conclusion we draw from these experiments is that the priority mechanism is quite effective in finding valid workarounds, but can also annoy the user with numbers of iterations ranging between 1 and 15. On the other hand, the oracle seems very effective in improving the situation by discarding many invalid attempts and letting the users focus on few relevant cases. The oracle prunes the set of candidate workarounds and identifies the correct workaround in the first attempt in 16 out of 25 cases. It also always succeeds within the third attempt, either producing a valid workaround or signaling that such a workaround could not be found.

Google Maps						
issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*no oracle*	*oracle*
15	✓			–	10	2
29			✓	G12	1	–
33		✓		G1	6	1
49		✓		G2	2	1
61		✓		G4	9	–
193	✓			–	13	3
240	✓			–	10	2
271		✓		G5	2	–
315		✓		G6	1	1
338			✓	G2	3	1
456			✓	G4	1	–
519		✓		G4	1	–
542			✓	G10	4	–
585			✓	G2	4	–
588		✓		G4	2	–
597		✓		G7	1	–
715	✓			–	10	1
737		✓		G4	3	–
754			✓	G2	13	2
823	✓			–	10	2
826	✓			–	15	3
833		✓		G4	2	–
881		✓		G14	2	1
945		✓		G2	3	–
1020			✓	G4	1	–
1101			✓	G10	1	1
1118			✓	G11	1	1
1200	✓			–	14	3
1205	✓			–	14	2
1206	✓			–	8	1
1209			✓	G2	2	–
1234		✓		G2	2	1
1264			✓	G3	3	2
1300			✓	G3	2	1
1305		✓		G8	1	1
1511			✓	G13	1	1
1578			✓	G2	3	1
1802			✓	G9	1	1

Table 6.6. Google Maps API issues

YouTube

We turned to the issue tracker of YouTube chromeless player API, aiming to reproduce the experiments on another API beside Google Maps. Unfortunately, the issue tracker contained only 21 entries at the time of investigation. Moreover, YouTube does not provide access to the version history of their API, so we could not reproduce any failure that was later fixed. Nevertheless, we first populated the repository with the 40 rules that we derived from the specification (a subset of the rules are listed in Table 6.3), and then applied them manually to the 21 issues. We verified that we could generate valid workarounds for the five problems reported with known workarounds. These were the five issues that were identified in the previous study of the automatic generation of workarounds (Section 6.3). We then selected the only open issue that we could reproduce, for which no workaround was known. Similarly to the Google Maps experiment, we sorted the six issues (five issues with known workaround and 1 open issue) in chronological order, and we used RAW to find valid workarounds.

Table 6.7 shows the results of the experiment. Beside the five known workarounds, RAW could find a new workaround for the only open issue. Moreover, for the only open issues that, the oracle filtered out all the failing attempts, thus proposing the valid workaround as the first attempt.

YouTube						
issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*no oracle*	*oracle*
522		✓		Y1	6	-
981		✓		Y2	8	-
1030		✓		Y3	8	-
1076		✓		Y4	8	-
1180		✓		Y2	1	-
1320			✓	Y5	8	1

Table 6.7. YouTube API issues

JQuery

To confirm the positive results of the studies on the Google Maps and YouTube APIs, we studied the API and the issue tracker of JQuery, which is a largely used open source Javascript library. JQuery uses a publicly available repository to store all the previous versions of the API. Moreover, given the popularity of the API, the development activity is high, and so is the number of the reported issues. Therefore JQuery seemed to be a good candidate to have an experiment with larger and more convincing numbers.

We wrote 68 program rewriting rules from the API specifications (some of which

are listed in Table 6.5), and we selected a total of 102 issues for the experiment. 25 of the 102 issues already had a known workaround listed in the bug tracker, and 77 did not. As for the previous experiments, we sorted the issues in chronological order. Table 6.8 reports the results of the experiment.

The most noticeable result is that RAW could automatically find a valid workaround for more than half of the issues without a publicly known workaround (42 out of 77), beside finding all the workarounds that were already known (25 out of 77).

The second positive result of this experiment comes from the priority mechanism, and the automatic oracle. As it is visible from Table 6.8, most of the times the valid workaround was proposed as the first attempt, even without the oracle. Moreover, the automatic oracle could often discard most of the failing attempts when a workaround could not be found (35 cases out of 102). More precisely, for 14 of the issues that we could not solve, the oracle automatically rejected all the proposed failing attempts. In such cases (that can be spotted in the table looking for zeros on the last column) RAW could immediately warn the users that no solution could be found.

JQuery						
issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*no oracle*	*oracle*
8		✓		J1	1	1
118			✓	J2	1	1
151			✓	J2	1	1
1008		✓		J3	1	1
1167			✓	J4	1	1
1239		✓		J5	2	1
1359		✓		J6	2	2
1360			✓	J7	1	1
1414	✓			–	1	0
1439	✓			–	2	0
1733	✓			–	1	0
2233			✓	J5	1	1
2352		✓		J8	1	1
2416		✓		J9	1	1
2551			✓	J10	3	2
2636		✓		J8	1	1
3255		✓		J6	1	1
3343		✓		J5	1	1
3380			✓	J11	1	1
3395			✓	J5	1	1
3745		✓		J12	2	1
3814		✓		J12	1	1

Table 6.8. (continued)

issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*no oracle*	*oracle*
3828		✓		J13	1	1
3891		✓		J12	1	1
3940	✓			–	1	0
3972	✓			–	3	0
4028			✓	J12	1	1
4088		✓		J14	1	1
4130	✓			–	2	0
4161	✓			–	1	0
4174	✓			–	2	0
4186		✓		J15	1	1
4281		✓		J16	1	1
4468			✓	J12	1	1
4472		✓		J16	1	1
4512	✓			–	1	0
4535			✓	J17	2	2
4649			✓	J18	1	1
4652			✓	J18	1	1
4681			✓	J19	1	1
4687	✓			–	2	1
4691	✓			–	2	1
4761			✓	J11	1	1
4817			✓	J11	1	1
4965			✓	J11	1	1
4984			✓	J10	2	2
5010			✓	J20	2	1
5018	✓			–	1	0
5130		✓		J21	2	2
5163	✓			–	2	1
5177			✓	J22	1	1
5316			✓	J23	1	1
5388			✓	J24	1	1
5414			✓	J12	1	1
5452			✓	J10	1	1
5505			✓	J11	1	1
5555			✓	J10	1	1
5637		✓		J25	1	1
5658		✓		J7	1	1

Table 6.8. (continued)

issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*no oracle*	*oracle*
5700			✓	J12	1	1
5708a	✓			–	1	0
5708b			✓	J26	1	1
5724	✓			–	1	0
5806	✓			–	1	0
5829	✓			–	8	2
5867	✓			–	1	0
5873	✓			–	2	1
5889	✓			–	2	0
5916	✓			–	1	0
5917			✓	J23	1	1
5986	✓			–	1	0
6035	✓			–	2	2
6038			✓	J7	1	1
6050	✓			–	2	1
6056	✓			–	1	0
6088			✓	J24	1	1
6158			✓	J23	1	1
6159			✓	J7	1	1
6160	✓			–	3	2
6264			✓	J18	1	1
6309		✓		J27	1	1
6330	✓			–	3	2
6476	✓			–	3	2
6496		✓		J19	1	1
6576	✓			–	3	0
6581	✓			–	2	2
6585		✓		J20	2	1
6610	✓			–	1	0
6643		✓		J2	1	1
6723		✓		J28	1	1
6731	✓			–	5	1
6774	✓			–	1	0
6835	✓			–	3	1
6837			✓	J10	1	1
6838			✓	J12	1	1
6895			✓	J19	5	2

Table 6.8. (continued)

issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*no oracle*	*oracle*
6945			✓	J18	1	1
6982			✓	J24	1	1
6999			✓	J17	2	2
7007	✓			–	1	1
7141			✓	J19	1	1
7151			✓	J19	1	1

Table 6.8. JQuery API issues

Despite the good results of the experiment on JQuery, a careful analysis of the data forced us to admit that the effectiveness of the priority mechanism was biased by the length of the Javascript code that we used to reproduce the failure. In fact, most of the web pages that we used to reproduce the JQuery issues had a few lines of Javascript code. As a consequence, most of the times only a few of the 68 program rewriting rules could be applied on each page, thus minimizing the number of attempts. Therefore, we decided to perform the last experiment, described in the next section, using a set of Web pages that were using a large portion of Javascript code.

Nonetheless, the three experiments described in this section show the ability of the Automatic Workarounds technique to find valid workarounds, even for issues for which no workaround was known before. The bar chart in Figure 6.1 summarizes the results of the three experiments, reporting the number of workarounds that have been found that were already known, the number of workarounds that were found and were not known, and the number of issues that are left without a solution.

6.4.3 Third Experiment: Priority Mechanism and Automatic Oracle

To further assess the effectiveness of the technique, we ran our last experiment on a set of Web pages that were including a large portion of Javascript code, and we used JQuery for this purpose. The goal of this last experiment was to study the ability of both the priority mechanism and the oracle to minimize the failing attempts in situations were the program rewriting rules could generate a large number of candidate workarounds.

We selected three Web pages from the JQuery website that were describing how to use the JQuery API [3]. We selected these pages because they include a lot of Javascript code (over 900 lines of code for each page), and heavily use JQuery itself. We changed the version of JQuery used by these web pages, such that we could reproduce failures that were no longer manifesting with the current version of the library. In this way we

[3] `http://docs.jquery.com/Tutorials:Live_Examples_of_jQuery`

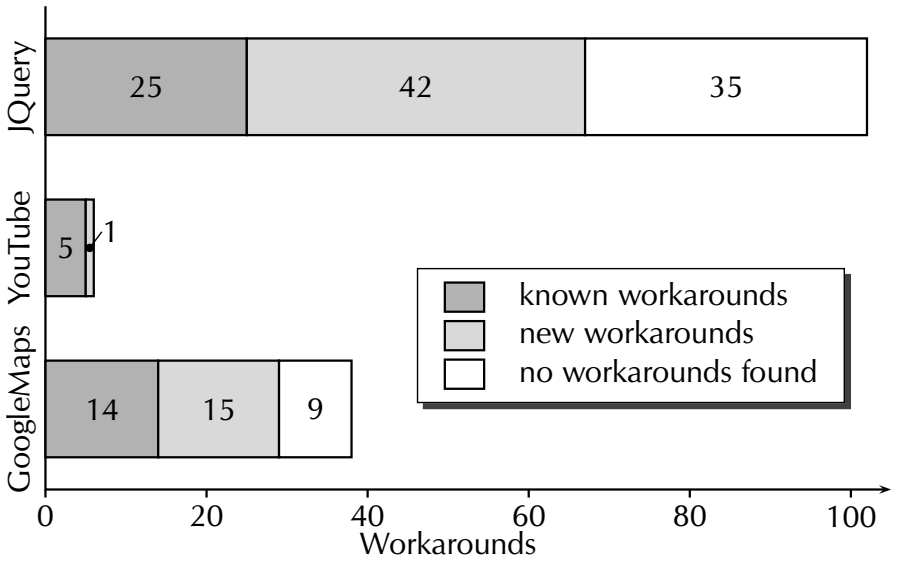


Figure 6.1. Summary report of the three experiments on Google Maps, YouTube and JQuery

were able to reproduce 14 issues using three older versions of the API. The details on the issues we could reproduce are reported in the Appendix B.

Although each page had more than one issue, we set up the experiment such that only one issue at a time was activated. We did this to avoid unexpected side-effects due to the manifestation of more than one failure in the same page. We then simulated 100 different users visiting the same three pages, and at each visit the user was experiencing one of the 14 issues, which was randomly selected. For each simulated user we then counted the number of attempts needed to have a valid workaround, when it could be found, both with and without the oracle.

Table 6.9 reports a summary of the results of this last experiment.

Each column reports the maximum (max) number of attempts and the average (avg) number of attempts for 100 page visits. We ran the experiment three times, and the results of each run are reported in the first three columns. The last column reports the average values of the three runs. We computed the maximum and the average number of attempts considering all the 14 issues, only the issues for which a workaround was found (11 cases out of 14), and the issues for which a workaround was found in the first attempt thanks to the automatic oracle.

The salient result is that even if the number of failing attempts was quite high, the oracle could filter most of them. In fact, in the worst case the user had to click three times to either find a valid workaround or to be warned that no solution could be found. However, on average the number of attempts needed is very close to 1 (1.07) for the cases for which a workaround could be found. Moreover, the oracle could very

JQuery																	
	1st run				*2nd run*				*3rd run*				*Total*				
	No oracle		*oracle*		*No oracle*		*oracle*		*No oracle*		*oracle*		*No oracle*		*oracle*		
	Max	avg	Max	avg	Max	avg	Max	avg	Max	avg	Max	avg	Max	avg	Max	avg	
All issues	23	4.81	2	0.94	23	5.88	3	0.90	23	6.51	3	0.89	23	5.73	3	0.91	
Fixed issues	13	2.52	2	1.05	14	2.44	3	1.08	13	2.40	3	1.09	14	2.46	3	1.07	
Fixed 1st attempt	4	1.97	–	–	4	1.79	–	–	4	1.67	–	–	4	1.81	–	–	

Table 6.9. User simulation

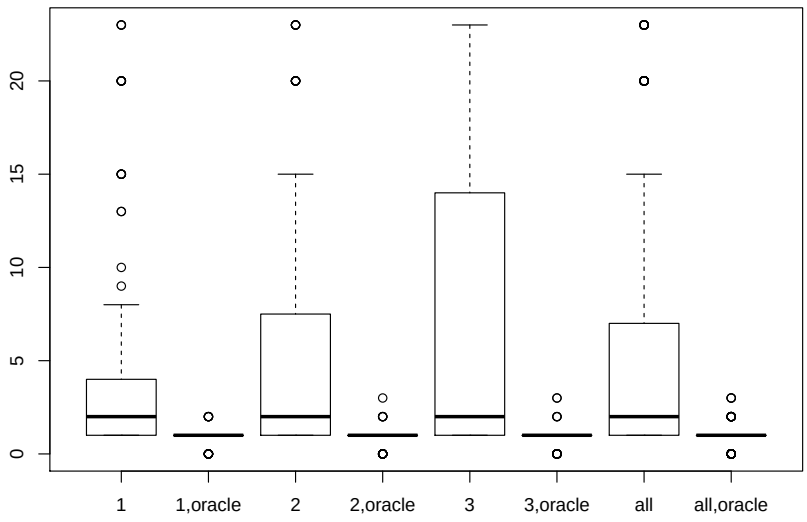


Figure 6.2. Boxplot representing the study on the effectiveness of the priority mechanism and the automatic oracle.

often filter all the attempts for the cases for which no solution could be found, thus bringing the average number of attempts below 1.

The boxplot in Figure 6.2 provides an alternative representation of the data listed in Table 6.9. It represents on the Y axis the number of attempts needed for all the issues in each run. Excluding few outliers, which are the cases in which no workaround could be found, the number of attempts is pretty low, and it is very close to 1 when the oracle was used. The median of the number of attempts of the executions without oracle is close to 2, and it is close to 1 for the executions with the automated oracle, proving that the priority mechanism and the automated oracle are effective.

6.5 Discussion

The evaluation presented in this chapter covered different aspects, which span from the ability of workarounds to address runtime failures, to the effectiveness of our technique to generate valid workarounds in few attempts. The study presented in Section 6.2 shows that workarounds are popular temporary solutions to address problems. In Google Maps at least 10% of the known issues can be effectively tolerated thanks to workarounds, and numbers rise till over 40% for the YouTube API. We also showed, thanks to the manual study presented in Section 6.3, that a large portion of the known

workarounds are instances of equivalent sequences, and thus could have been found automatically. Precisely, the number amounts to more than one third of the Google Maps workarounds, and more than half of the YouTube ones.

Finally, in Section 6.4 we demonstrated the effectiveness of the Automatic Workarounds technique presented in Chapter 4. In particular, we showed that the reactive approach, which amounts to inserting program rewriting rules coding workarounds that are known to work, is effective because some workarounds are reusable, and they can thus be a valid solution for several subsequent issues (Section 6.4.1). The proactive approach, which we present in Section 6.4.2, is even more effective, since it leads to the identification of a valid workaround for most of the issues that we considered for Google Maps, YouTube and JQuery. We could also verify that our technique can deal with high numbers of candidate workarounds, as the priority mechanism and the automatic oracle can filter most of the failing attempts, thus minimizing the number of times that the user has to manually judge the result (Section 6.4.3).

6.6 Limitations and Threats to Validity

The Automatic Workarounds technique is limited primarily by the assumed nature of failures in Web applications. We assume that failures are visible to the user, and the user can correctly report them. In fact, since we rely on the user to train the priority mechanism, we assume that he or she can always report failures, and can correctly signal when a workaround has been found.

The lack of a specific user study is one of the major threats to the validity of our experimental evaluation. Our hypothesis that users would be able to correctly detect failures and identify valid workarounds, could not be confirmed with an appropriate experiment.

The Automatic Workarounds technique is also limited by the assumption that multiple executions of the Javascript code do not have any side-effects, and we can thus ignore the rollback problem. However, we are aware that the execution may be not only at the client side, and some requests can be sent to the server side. Even if we assumed that all the Web applications were built according to the Post-Redirect-Get pattern, which we described in Section 4.1, we would still need to assume that the data that are sent to the server are not affected by the issue.

Another limitation of the technique is about the number of failures, as it implicitly assumes that there is at most one issue causing a failure in each Web page. In presence of two issues, in fact, the technique should apply several rules at the same time, bringing to a combinatorial explosion of rules to apply.

Another main threats to validity of the experiments are related to the amount of data collected, and to the number of Web APIs studied so far. Even if we used a relevant number of issues for the experiments, we focused our attention on only three APIs. The results obtained from the analysis of these libraries may not represent well

the wider domain of Web APIs. We investigated other APIs, but most of the times the failures that we found were not reproducible, because there was no access to previous versions of the API with the described faults. JQuery is the only exception, this is why most data reported here come from JQuery. Google Maps is a partial exception, since when we started our experiments we had access to all the previous versions of the library. However, as we already said, during experiments some of the old versions were removed, and consequently we had to abandon several issues that we could no longer reproduce. This is the reason why most of the initial experiments come from Google Maps, and we switched to JQuery towards the end of the experiments.

Chapter 7

Conclusions

This thesis explores a particular type of redundancy that is *intrinsic* in software. There are different reasons why software may be intrinsically redundant, but in essence the common cause of such redundancy is software reuse. Reusable components are typically designed to be highly configurable and versatile, and therefore to offer redundant interfaces to meet different application needs. Thus, a component might expose the same functionality through multiple semantically-equivalent sequences of methods, and this redundancy at the interface level may then lead to redundancy at the implementation level, which means that the component might be able to implement the same functionality through the execution of different combinations of code fragments.

This intrinsic redundancy, expressed in the form of equivalent sequences, can be exploited for several purposes, such as software testing, fault localization and fault handling. This thesis introduced a technique that exploits intrinsic redundancy to find workarounds automatically and at runtime. The idea behind Automatic Workarounds is to have a self-healing layer that monitors the execution of the application components, and upon a failure occurrence selects and executes a new sequence of operations that should be semantically equivalent, in its original intent, to the operations that led to the failure.

We designed and implemented Automatic Workarounds in the context of Web applications, with the intent of making Web pages resilient to faults in the Javascript libraries used by Web applications. A browser extension, which we built as a prototype to demonstrate the Automatic Workaround technique, extracts the Javascript code from the page when the user reports a failure, replaces existing code with equivalent sequences in an attempt to find a valid workaround, and proposes the newly generated page to the user. We demonstrated the effectiveness of the technique on almost 150 documented issues of Google Maps, YouTube, and JQuery, three popular Web libraries with public issue trackers. Our prototype could always find a workaround for the issues for which a workaround was already known, and could also find a valid solution for many other issues. Moreover, the prototype could automatically discard most of

the equivalent sequences that do not lead to an effective workaround, thereby either quickly finding the valid workaround, or quickly determining that no such workaround can be found.

7.1 Contributions

The first major contribution of this thesis is the notion of intrinsic redundancy. Although this type of redundancy has been used previously by other researchers, mainly for software testing, this thesis provided a thorough study on the nature of intrinsic redundancy and on the possible ways to exploit it. The second major contribution is the Automatic Workarounds technique, which exploits intrinsic redundancy to mitigate the effect of faults in software. We now summarize in more detail some aspects of these two major contributions.

Analysis and classification of equivalent sequences. The intrinsic redundancy that is present in modular software can be expressed in the form of equivalent sequences. In this thesis we analyzed several issue trackers and library specifications looking for equivalent sequences. As a result, we uncovered several specific cases of equivalent sequences that then lead us to formulate the general classification in null, invariant, and alternative operations presented in Chapter 3.

Program rewriting rules. To be used correctly and at runtime, equivalent sequences must be instantiated within the context of the application. This means that the correct parameters in the original sequence must be referred in the corresponding places in the equivalent sequence, and also that the applicability of the equivalence must be checked in the current application state. In Chapter 4, we introduced the program rewriting rules used by the Automatic Workarounds mechanism to substitute original code with equivalent sequences of operations.

Automatic Workarounds architecture. Together with the definition of the Automatic Workarounds technique, we proposed a general architecture for building self-healing component-based software systems, and a more detailed one for Web applications. The architecture enables the automatic applications of workarounds to heal from runtime failure, assuming the availability of a failure detector and a rollback mechanism. The architecture for Web applications relies on the user to report failures and to validate workarounds.

Priority mechanism and automated oracle. Program-rewriting rules can produce a lot of candidate workarounds, many of which may not lead to valid solutions. Asking the user to evaluate the correctness of each candidate workaround would seriously reduce the practical applicability of the technique. To minimize the number of ineffective rewritings of the application code, we implemented a

priority mechanism and an automated oracle. The priority scheme ranks the program-rewriting rules based on their success rate such that the ones that are more likely to work are applied first. The oracle complements the priority mechanism by immediately discarding the rules that do not change the faulty behavior of the Web page at all.

Prototype implementation. We built the Automatic Workarounds prototype of the self-healing architecture for Web applications as a browser extension for the user interaction. The prototype sends requests to a centralized server that contains the repository of the program rewriting rules. We used this prototype to test the effectiveness of the Automatic Workarounds technique experimentally.

Study on existing workarounds. For the purpose of understanding how workarounds are used in practice, we studied the issue trackers of two popular Web APIs (Google Maps and YouTube) and we manually analyzed a large number of issues reported there. We found that workarounds are a popular way to address open issues, and a significant portion of the workarounds suggested on issue trackers are instances of equivalent sequences.

Evaluation of the technique. We evaluated the Automatic Workarounds technique by selecting almost 150 issues that we could reproduce from the issue trackers of Google Maps, YouTube, and JQuery. Our prototype could find a valid workaround for all the issues that already had a known workaround described in the issue tracker, and could find a valid solution for many other issues for which no workaround was known. This study shows that our technique can effectively mask the presence of faults in Web applications, and that even in the presence of a large number of candidate workarounds, the automated oracle can discard most of the ineffective ones.

7.2 Future Directions

The work presented in this dissertation uncovered or touched upon problems and ideas that remain open for future research. We start by presenting some of these ideas that we plan to explore in the near future, namely the automatic identification of equivalent sequences and the improvement of the priority mechanism, and conclude with possible long-term developments, such as exploiting intrinsic redundancy for different purposes other than fault-handling, and applying the Automatic Workarounds technique in other contexts beside Web applications.

Improving the priority mechanism. Although the evaluation results show that the automated oracle can discard most of the failing attempts, i.e., equivalent sequences that do not lead to effective workarounds, the priority mechanism can be improved to

reduce the amount of results proposed to the oracle for the evaluation, thus improving the efficiency of the technique. In the current prototype, we apply substitutions to Javascript code without considering runtime information. A relatively simple and effective way to improve the priority scheme would be to record some traces of the execution and to use that to select and prioritize the application of the rewriting rules. For example, a record of the statement covered by the failing execution could be used to apply substitutions only to those statements. Similarly, a richer trace such as a path condition could be used to select alternative sequences that would induce different executions.

Automatically identify intrinsic redundancy. In Chapter 3 we argued that application developers and expert users can manually identify intrinsic redundancy and provide a list of equivalent sequences for each module or library with a reasonable effort, and this is what we did to run the experiments. This task can be tedious and error-prone, and would benefit from some automation to at least guide the developer in the identification and formulation of equivalent sequences. Our plan is to rely on dynamic analysis to identify potential equivalent sequences, and propose them to the developers for approval.

Several techniques rely on dynamic analysis to infer different types of specification such as finite state machines, algebraic specifications and class invariants [ECGN01; DLWZ06; MPPar; GMM09; HRD07], but none of these techniques can be used as-is to identify intrinsic redundancy and equivalent sequences. The technique by Henkel et al. seems the most closely related to our problem, since it can identify algebraic specification axioms of Java classes that implement containers [HRD07]. To identify axioms they first generate terms, which are sequences of Java method invocations, and they later compare the effect of the execution of each term on class instances. This process can be adapted to identify equivalent sequences in Javascript libraries, and it amounts to these three steps:

- *Generating sequences of function calls*: Similarly to the technique of Henkel et al., we need to generate sequences of function calls of different lengths. It is possible to either derive them from the execution of existing test suites, or generate them from the in site use of the libraries. In absence of publicly available test suites, as in the case of many Javascript libraries, function calls and parameters would have to be generated as well.

- *Checking for equivalence*: Once we have a set of sequences of function calls, we have to execute them and find which ones are equivalent, that is, which ones lead to the same results. We consider using the notion of observational equivalence to determine which sequences of function invocations have the same effects. To identify equivalent functions in Javascript libraries, we can include the generated function calls in existing Web pages, and we then plan to execute them. The

comparison of the generated DOMs can be used to determine which sequences of function invocations are equivalent.

- *Generalizing the equivalence*: The equivalent sequences identified in the previous step will be bound to the specific parameter values used in the chosen execution. However, to be useful for the generation of workarounds, equivalent sequences must be more general. This means that specific equivalent sequences containing specific values must be abstracted into more general and parametric rewriting rules, and also that these more general rules must be valid, in the sense that they must always produce equivalent sequences.

Approximate equivalent sequences. In Chapter 3 we introduced the concept of approximate equivalent sequences. However, in this thesis we considered only exact equivalent sequences. We believe that in many cases it would be acceptable to find a workaround that improves the behavior of a software system even if the workaround does not completely avoid the problem or if the workaround does not achieve exactly the same (intended) functionality of the failing sequence. Therefore, we believe that approximate equivalent sequences can be used with the Automatic Workarounds technique just as well as exact equivalent sequences.

Automatic Workarounds in other contexts. This thesis developed the concept of Automatic Workarounds for Web applications. We think that the technique is not bound to this domain, and our long-term plan is to adapt and generalize the technique to a wider class of applications, such general purpose applications. To generalize to general purpose applications, we would have to relax some of the initial assumptions that are reasonable in the context of Web applications, but do not apply in general.

Intrinsic redundancy for different purposes. In Chapter 3 we already mentioned possible uses of intrinsic redundancy beside fault-handling. We plan to investigate the effectiveness of equivalent sequences other domains, like testing, fault diagnosis and debugging.

Appendices

Appendix A

Program-Rewriting Rules

In this Appendix, we present the complete set of program rewriting rules for the three Web libraries that we used in our evaluation. In total, we designed 39 rules for the Google Maps API, 40 for the YouTube chromeless player API, and 68 for the JQuery library.

For each rule, we report the type (null, invariant and alternative) and the substitution pattern as accepted by sed [1], the tool we used to automatically apply the substitutions in Javascript code. Substitutions in sed are written as `'s/⟨patternToBeReplaced⟩/⟨replacement⟩/'` commands, where the ⟨patternToBeReplaced⟩ indicates the pattern that has to be replaced by the application of the rule, and ⟨replacement⟩ indicates the substitution. For example, the sed command `s/aaa/bbb` substitutes all the occurrences of the string `aaa` with the string `bbb`. Multiple substitutions can be specified by prepending the `-e` command. For example, `-e 's/aa/bb' 's/cc/dd'` substitutes `aa` with `bb` and `cc` with `dd`.

Here we present the rules in the same order as they appear in the database that we used for the experiments.

A.1 Google Maps Rules

1 **NULL** 's/\([A-Za-z0-9_]\).openInfoWindowHtml(\([[:graph:][:space:]]\));/setTimeout("\1.openInfoWindowHtml(\2)", 1000);/'

2 **ALTERNATIVE** 's/\([A-Za-z0-9_]\).addOverlay(\([[:graph:][:space:]]\));/\1.addOverlay(\2); \2.show();/;s/\([A-Za-z0-9_]\).removeOverlay(\([[:graph:][:space:]]\));/\2.hide();/'

3 **ALTERNATIVE** 's/\([A-Za-z0-9_]\).hide();/\1.remove();/'

4 **NULL** 's/\([A-Za-z0-9_]\).setCenter(\([[:graph:][:space:]]\));/setTimeout("\1.setCenter(\2)", 1000);/'

5 **INVARIANT** 's/\([A-Za-z0-9_]\).show();/\1.show(); \1.show();/'

6 **ALTERNATIVE** '/\(.\).setCenter(\(.\));/ N; s/\(.\).setCenter(\(.\));[[:blank:].]\(.\).setMapType(\(.\));/\1.setCenter(\2, \4);/'

[1] `http://www.gnu.org/software/sed/`

```
7  ALTERNATIVE 's/\([A-Za-z0-9_]\).disableEditing();/setTimeout("\1.disableEditing()",
       200); GDraggableObject.setDraggableCursor("default");/'
8  ALTERNATIVE 's/\([A-Za-z0-9_]\).enableDrawing(\([[:graph:][:space:]]\));/var l=\1.
       getVertexCount(); var v=\1.getVertex(l-1); \1.deleteVertex(l-1); \1.insertVertex(
       l-1,v); \1.enableDrawing(\2);/'
9  ALTERNATIVE 's/\([A-Za-z0-9_]\).getInfoWindow().reset(\([[:graph:][:space:]]\));/
       \1.getInfoWindow().reset(\2, \1.getInfoWindow().getTabs() , new GSize(0,0));/'
10 ALTERNATIVE 's/\([A-Za-z0-9_= ]\).getVertexCount(\([[:graph:][:space:]]\));
       /setTimeout("\1.getVertexCount(\2)", 1000);/;s/\([A-Za-z0-9_= ]\).getBounds
       (\([[:graph:][:space:]]\));/setTimeout("\1.getBounds(\2)", 1000);/'
11 ALTERNATIVE 's/\([A-Za-z0-9_]\).bindInfoWindowHtml(\([[:graph:][:space:]]\));
       /GEvent.addListener(\1, "click", function(){\1.openInfoWindowHtml(\2)});/'
12 ALTERNATIVE 's/. new GDraggableObject(.);/ N; s/var \([[:alnum:]_-]\) = new
       GDraggableObject(\(.\));[[:blank:].]..setDraggableCursor(\(.\));[:blank:].
       setDraggingCursor(\(.\));/var \1 = new GDraggableObject(\2, {draggableCursor:\3,
       draggingCursor:\4});/'
13 ALTERNATIVE 's/GEvent.addDomListener(\([A-Za-z0-9_]\), "click", function() { \(.\)
       });/\1.onclick = function() { \2 }/'
14 ALTERNATIVE 's/\([[:alnum:]_-]\) = new GMap2(document.getElementById(\(.\)),?
       {\(.\)zoom: \([:digit:]\)\(.\)});/\1 = new GMap2(document.getElementById(\2),
       {\3 \5}); \1.setZoom(\4);/'
15 ALTERNATIVE 's/\([[:alnum:]_-]\) = new GMap2(document.getElementById(\(.\)),?
       {\(.\)mapTypes: \([[:upper:]_]\)\(.\)});/\1 = new GMap2(document.
       getElementById(\2), {\3 \5}); \1.setMapType(\4);/'
16 ALTERNATIVE 's/\([[:alnum:]_-]\) = new GMap2(document.getElementById(\(.\)),?
       {\(.\)draggableCursor: \([[:alpha:]_-]\)\(.\)});/\1 = new GMap2(document.
       getElementById(\2), {\3 \5}); GDraggableObject.setDraggableCursor(\4);/'
17 ALTERNATIVE 's/\([[:alnum:]_-]\) = new GMap2(document.getElementById(\(.\)),?
       {\(.\)draggingCursor: \([[:alpha:]_-]\)\(.\)});/\1 = new GMap2(document.
       getElementById(\2), {\3 \5}); GDraggableObject.setDraggingCursor(\4);/'
18 ALTERNATIVE '/new GMap2(document.getElementById(.), {.zoom: [:digit:].});/ N; s/
       \([[:alnum:]_-]\) = new GMap2(document.getElementById(\(.\)), {\(.\)zoom: \([:
       digit:]\)\(.\)});\([.[:blank:]\n]\)\([[:alnum:]_-]\).setCenter(new LatLng
       (\(.\)));/\1 = new GMap2(document.getElementById(\2),{\3 \5}); \6 \1.setCenter
       (\7, zoom: \4);/'
19 INVARIANT 's/\([[:alnum:]_-]\) = new GMap2(\(.\));/\1 = new GMap2(\2); \1.
       enableDragging();/'
20 INVARIANT 's/\([[:alnum:]_-]\) = new GMap2(\(.\));/\1 = new GMap2(\2); \1.
       enableInfoWindow();/'
21 INVARIANT 's/\([[:alnum:]_-]\) = new GMap2(\(.\));/\1 = new GMap2(\2); \1.
       enableDoubleClickZoom();/'
22 INVARIANT 's/\([[:alnum:]_-]\) = new GMap2(\(.\));/\1 = new GMap2(\2); \1.
       disableContinuousZoom();/'
23 INVARIANT 's/\([[:alnum:]_-]\) = new GMap2(\(.\));/\1 = new GMap2(\2); \1.
       disableGoogleBar();/'
24 INVARIANT 's/\([[:alnum:]_-]\) = new GMap2(\(.\));/\1 = new GMap2(\2); \1.
       disableScrollWheelZoom();/'
25 INVARIANT 's/\([[:alnum:]_-]\).disableDragging();/ \1.enableDragging(); \1.
       disableDragging();/'
26 INVARIANT 's/\([[:alnum:]_-]\).disableInfoWindow();/ \1.enableInfoWindow(); \1.
       disableInfoWindow();/'
27 INVARIANT 's/\([[:alnum:]_-]\).disableDoubleClickZoom();/ \1.enableDoubleClickZoom
       (); \1.disableDoubleClickZoom();/'
28 INVARIANT 's/\([[:alnum:]_-]\).enableContinuousZoom();/ \1.disableContinuousZoom();
       \1.enableContinuousZoom();/'
29 INVARIANT 's/\([[:alnum:]_-]\).enableGoogleBar();/ \1.disableGoogleBar(); \1.
       enableGoogleBar();/'
30 INVARIANT 's/\([[:alnum:]_-]\).enableScrollWheelZoom();/ \0.disableScrollWheelZoom
       (); \1.enableScrollWheelZoom();/'
```

```
31 ALTERNATIVE 's/ GEvent.addListener(\([A-Za-z0-9_]\), "click", function(){ [A-Za-z0
       -9].openInfoWindowHtml(\([[:graph:][:space:]]\)) });/\1.bindInfoWindowHtml(\2);
       /', 'gmap',1,1,1,1,1);
32 ALTERNATIVE 's/\([A-Za-z0-9_]\).removeOverlay(\([[:graph:][:space:]]\));/\2.remove
       ();/'
33 NULL 's/\([A-Za-z0-9_]\).addOverlay(\([[:graph:][:space:]]\));/setTimeout("\1.
       addOverlay(\2)", 1000);/'
34 INVARIANT 's/\([[:alnum:]_-]\) = new GMarker(\(.\){draggable: true\(.\));/ \1 =
       new GMarker(\2 {draggable: true \3); \1.enableDragging();/'
35 INVARIANT 's/\([[:alnum:]_-]\) = new GMarker(\(.\){draggable: true\(.\));/ \1 =
       new GMarker(\2 {draggable: true \3); \1.disableDragging(); \1.enableDragging();/'
36 INVARIANT 's/\([[:alnum:]_-]\) = new GMarker(\(.\));/ \1 = new GMarker(\2); \1.
       show();/'
37 INVARIANT 's/\([[:alnum:]_-]\) = new GMarker(\(.\));/ \1 = new GMarker(\2); \1.
       hide(); \1.show();/'
38 INVARIANT 's/\([[:alnum:]_-]\) = new GMap2(\(.\));/\1 = new GMap2(\2); \1.zoomIn()
       ; \1.zoomOut(); /'
39 NULL 's/GEvent.trigger(\([[:graph:][:space:]]\));/setTimeout("GEvent.trigger(\1)",
       1000);/'
```

A.2 List of YouTube Rules

```
1 ALTERNATIVE 's/\([A-Za-z0-9_]\).setSize(\([A-Za-z0-9_"]\), \([A-Za-z0-9_"]\));/
      \1.width=\2; \1.height=\3;/'
2 ALTERNATIVE 's/\([A-Za-z0-9_]\).seekTo(\([A-Za-z0-9_""]\), \([A-Za-z0-9_""]\));/
      \1.cueVideoByUrl(\1.getVideoUrl(), \2); \1.playVideo();/'
3 ALTERNATIVE 's/\([A-Za-z0-9_]\).seekTo(\([A-Za-z0-9_""]\), \([A-Za-z0-9_""]\));/
      \1.loadVideoByUrl(\1.getVideoUrl(), \2);/'
4 INVARIANT 's/\([A-Za-z0-9_]\).stopVideo();/\1.pauseVideo(); \1.stopVideo();/'
5 INVARIANT 's/\([A-Za-z0-9_]\).stopVideo();/\1.stopVideo(); \1.stopVideo();/'
6 INVARIANT 's/\([A-Za-z0-9_]\).playVideo();/\1.playVideo(); \1.playVideo();/'
7 ALTERNATIVE 's/\([A-Za-z0-9_]\).playVideo();/\1.loadVideoByUrl(\1.getVideoUrl(),
      \1.getCurrentTime());/'
8 INVARIANT 's/\([A-Za-z0-9_]\).pauseVideo();/\1.pauseVideo(); \1.pauseVideo();/'
9 ALTERNATIVE 's/\([A-Za-z0-9_]\).loadVideoById(\([A-Za-z0-9_""]\), \([A-Za-z0-9_
      ""]\);/\1.loadVideoById(\2, 0); \1.seekTo(\3, true);/'
10 INVARIANT 's/\([A-Za-z0-9_]\).playVideo();/if (!\1.isMuted()) { \1.mute(); \1.
      unMute(); } \1.playVideo();/'
11 INVARIANT 's/\([A-Za-z0-9_]\).stopVideo();/if (!\1.isMuted()) { \1.mute(); \1.
      unMute(); } \1.stopVideo();/'
12 INVARIANT 's/\([A-Za-z0-9_]\).pauseVideo();/if (!\1.isMuted()) { \1.mute(); \1.
      unMute(); } \1.pauseVideo();/'
13 INVARIANT 's/\([A-Za-z0-9_]\).playVideo();/if (\1.getPlayerState() == 2) { \1.
      playVideo(); \1.pauseVideo(); } \1.playVideo();/'
14 INVARIANT 's/\([A-Za-z0-9_]\).pauseVideo();/if (\1.getPlayerState() == 1) { \1.
      pauseVideo(); \1.playVideo(); } \1.pauseVideo();/'
15 INVARIANT 's/\([A-Za-z0-9_]\).stopVideo();/if (\1.getPlayerState() == 1) { \1.
      pauseVideo(); \1.playVideo(); } \1.stopVideo();/'
16 INVARIANT 's/\([A-Za-z0-9_]\).stopVideo();/if (\1.getPlayerState() == 2) { \1.
      playVideo(); \1.pauseVideo(); } \1.stopVideo();/'
17 NULL 's/\([A-Za-z0-9_]\).playVideo();/setTimeout(""\1.playVideo()"",500);/'
18 NULL 's/\([A-Za-z0-9_]\).pauseVideo();/setTimeout(""\1.pauseVideo()"",500);/'
19 NULL 's/\([A-Za-z0-9_]\).stopVideo();/setTimeout(""\1.stopVideo()"",500);/'
20 NULL 's/\([A-Za-z0-9_]\).clearVideo();/setTimeout(""\1.clearVideo()"",500);/'
21 NULL 's/\([A-Za-z0-9_]\).getVideoBytesLoaded();/setTimeout(""\1.getVideoBytesLoaded
      ()"",500);/'
22 NULL 's/\([A-Za-z0-9_]\).getVideoBytesTotal();/setTimeout(""\1.getVideoBytesTotal()
      "",500);/'
```

```
23 NULL 's/\([A-Za-z0-9_]\).getVideoStartBytes();/setTimeout(""\1.getVideoStartBytes()
     "",500);/'
24 NULL 's/\([A-Za-z0-9_]\).mute();/setTimeout(""\1.mute()"",500);/'
25 NULL 's/\([A-Za-z0-9_]\).unMute();/setTimeout(""\1.unMute()"",500);/'
26 NULL 's/\([A-Za-z0-9_]\).isMuted();/setTimeout(""\1.isMuted()"",500);/'
27 NULL 's/\([A-Za-z0-9_]\).setVolume(\([A-Za-z0-9_""]\));/setTimeout(""\1.setVolume
     (\2)"",500);/'
28 NULL 's/\([A-Za-z0-9_]\).getVolume();/setTimeout(""\1.getVolume()"",500);/'
29 NULL 's/\([A-Za-z0-9_]\).seekTo(\([A-Za-z0-9_""]\), \([A-Za-z0-9_""]\));
     /setTimeout(""\1.seekTo(\2, \3)"",500);/'
30 NULL 's/\([A-Za-z0-9_]\).getPlayerState();/setTimeout(""\1.getPlayerState()"",500);
     /'
31 NULL 's/\([A-Za-z0-9_]\).getCurrentTime();/setTimeout(""\1.getCurrentTime()"",500);
     /'
32 NULL 's/\([A-Za-z0-9_]\).getDuration();/setTimeout(""\1.getDuration()"",500);/'
33 NULL 's/\([A-Za-z0-9_]\).addEventListener(\([A-Za-z0-9_""]\), \([A-Za-z0-9_""]\))
     ;/setTimeout(""\1.addEventListener(\2, \3)"",500);/'
34 NULL 's/\([A-Za-z0-9_]\).getVideoUrl();/setTimeout(""\1.getVideoUrl()"",500);/'
35 NULL 's/\([A-Za-z0-9_]\).getVideoEmbedCode();/setTimeout(""\1.getVideoEmbedCode()
     "",500);/'
36 NULL 's/\([A-Za-z0-9_]\).cueVideoById(\([A-Za-z0-9_""]\), \([A-Za-z0-9_""]\));
     /setTimeout(""\1.cueVideoById(\2, \3)"",500);/'
37 NULL 's/\([A-Za-z0-9_]\).loadVideoById(\([A-Za-z0-9_""]\), \([A-Za-z0-9_""]\));
     /setTimeout(""\1.loadVideoById(\2, \3)"",500);/'
38 NULL 's/\([A-Za-z0-9_]\).cueVideoByUrl(\([A-Za-z0-9_""]\), \([A-Za-z0-9_""]\));
     /setTimeout(""\1.cueVideoByUrl(\2, \3)"",500);/'
39 NULL 's/\([A-Za-z0-9_]\).loadVideoByUrl(\([A-Za-z0-9_""]\), \([A-Za-z0-9_""]\));
     /setTimeout(""\1.loadVideoByUrl(\2, \3)"",500);/'
40 NULL 's/\([A-Za-z0-9_]\).setSize(\([A-Za-z0-9_""]\), \([A-Za-z0-9_""]\));
     /setTimeout(""\1.setSize(\2, \3)"",500);/'
```

A.3 List of JQuery Rules

```
1 ALTERNATIVE '"s/:not(:first)/:gt(0)/g"'
2 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).change(\([[:graph:][:space:]]\))/\$
     (\1).click(\2)/g"'
3 ALTERNATIVE '-e "s/\$(''#\([[:graph:][:space:]]\)'')\.attr(''disabled'', ''true'')
     /document.getElementById(''\1'').sheet.disabled=true/g" -e "s/\$(''\.\([[:graph
     :][:space:]]\)'')\.attr(''disabled'', ''true'')/document.getElementById(''\1'').
     sheet.disabled=true/g" -e "s/\$(\([[:graph:][:space:]]\))\.attr(''disabled'', ''
     true'')/document.getElementById(\1).sheet.disabled=true/g" -e "s/\$(''#\([[:graph
     :][:space:]]\)'')\.attr(''disabled'', ''false'')/document.getElementById(''\1'')
     .sheet.disabled=false/g" -e "s/\$(''\.\([[:graph:][:space:]]\)'')\.attr(''
     disabled'', ''false'')/document.getElementById(''\1'').sheet.disabled=false/g" -e
      "s/\$(\([[:graph:][:space:]]\))\.attr(''disabled'', ''false'')/document.
     getElementById(\1).sheet.disabled=false/g" '
4 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).hide().appendTo(\([[:graph:][:space
     :]]\))/\$(\1).appendTo(\2).hide()/g"'
5 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\))\.attr(''onclick'', ''\([[:graph:][:
     space:]]\)'')/\$(\1)[0].onclick = function() { \2 }/g"'
6 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\))\.click()/\$(\1).trigger("onclick")/g"'
7 ALTERNATIVE '"s/\$(''<\([[:alpha:][:space:]]\)\/>'')\.attr(''type'', ''\([[:alpha
     :]]\)'')/\$(''<\1 type=\"\2\" \/>'')/g"'
8 ALTERNATIVE '"s/\$(''\[autofocus]'')/\$(''\[autofocus=\"\"]'')/g"'
9 ALTERNATIVE '-e "s/\$(\([[:graph:][:space:]]\))\.set(''disabled'', false)/\$(\1).
     get(0).disabled = false/g" -e "s/\$(\([[:graph:][:space:]]\))\.set(''disabled'',
      true)/\$(\1).get(0).disabled = true/g"'
10 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\))\.val()/\$(\1).text()/g"'
```

```
11 INVARIANT '"s/\$(\([[:graph:][:space:]]\))\.resizable( \"option\", \"aspectRatio\",
       \([[:digit:][:punct:]]\) )/\$(\1).resizable(''destroy'').resizable({
     aspectRatio: \2 })/g"'
12 INVARIANT '"s/\$(\([[:graph:][:space:]]\))\.attr(''value'', \$(\([[:graph:][:space
     :]]\))\.attr(\([[:graph:][:space:]]\)))/\$(\1).removeAttr(''value''); \$(\1).
     attr(''value'', \$(\1).attr(\3))/g"'
13 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\))\.wrap(\([[:graph:][:space:]]\))\.
     appendTo(\([[:graph:][:space:]]\))/\$(\1).appendTo(\3).wrap(\2)/g"'
14 ALTERNATIVE '''s/\$("ul:has(li:gt(\([[:digit:]]\)))")/\$("li:gt(\1)").parent()/g'''
15 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\))\.hide(); \$(\([[:graph:][:space:]]\))
     \.after(\([[:graph:][:space:]]\)); \$(\([[:graph:][:space:]]\))\.show();/\$(\2)
     .after(\1); \$(\1).hide();\$(\2).show();/g"'
16 ALTERNATIVE '"s/\([[:graph:][:space:]]\).animate(\([[:graph:][:space:]]\), \([[:
     graph:][:space:]]\))/\1.animate(jQuery.extend(true, {}, \2), jQuery.extend(true,
      {}, \3))/g"'
17 NULL '"s/\([[:graph:][:space:]]\)\.draggable.remove()/setTimeout(function() { \1.
     draggable.remove(); }, 10)/g"'
18 ALTERNATIVE '-e "s/\([[:graph:][:space:]]\)\.removeClass(\([[:graph:][:space:]]\))
     /\1.toggleClass(\2, false)/g" -e "s/\([[:graph:][:space:]]\)\.addClass(\([[:
     graph:][:space:]]\))/\1.toggleClass(\2, true)/g"'
19 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).remove()/\$(\1).empty().remove()/g"'
20 ALTERNATIVE '"s/(\([[:graph:][:space:]]\)).appendTo(\([[:graph:][:space:]]\))/\$
     (\2).append(\1)/g"'
21 ALTERNATIVE '"s/\([[:graph:][:space:]]\).detach()/\1.clone(true); \1.remove()/g"'
22 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).replaceWith(\([[:graph:][:space:]]\))
     /\$(\2).replaceAll(\1)/g"'
23 ALTERNATIVE '-e "s/.height()/.css(''height'')/g" -e "s/.height(\([[:graph:][:space
     :]]\))/.css(''height'', \1)/g" -e "s/.width()/.css(''width'')/g" -e "s/.width
     (\([[:graph:][:space:]]\))/.css(''width'', \1)/g"'
24 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).after(\([[:graph:][:space:]]\))/\$
     (\2).insertAfter(\1)/g"'
25 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).insertAfter(\([[:graph:][:space:]]\))
     /\$(\2).after(\1)/g"'
26 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).before(\([[:graph:][:space:]]\))/\$
     (\2).insertBefore(\1)/g"'
27 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).insertBefore(\([[:graph:][:space:]]\)
     )/\$(\2).before(\1)/g"'
28 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).append(\([[:graph:][:space:]]\))/\$
     (\2).appendTo(\1)/g"'
29 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).prepend(\([[:graph:][:space:]]\))/\$
     (\2).prependTo(\1)/g"'
30 ALTERNATIVE '"s/(\([[:graph:][:space:]]\)).prependTo(\([[:graph:][:space:]]\))/\$
     (\2).prepend(\1)/g"'
31 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).replaceAll(\([[:graph:][:space:]]\))/
     \$(\2).replaceWith(\1)/g"'
32 ALTERNATIVE '-e "s/\$(\([[:graph:][:space:]]\)).val()/\$(\1).attr(''value'')/g" -e
     "s/\$(\([[:graph:][:space:]]\)).val(\([[:graph:][:space:]]\))/\$(\1).attr(''
     value'', \2)/g"'
33 ALTERNATIVE '"s/\$(\([[:graph:][:space:]]\)).hide()/\$(\1).css(''display'', ''none
     '')/g"'
34 ALTERNATIVE '-e "s/\$(\([[:graph:][:space:]]\)).hide(\([[:graph:][:space:]]\))/\$
     (\1).hide()/g" -e "s/\$(\([[:graph:][:space:]]\)).show(\([[:graph:][:space:]]\)
     )/\$(\1).show()/g"'
35 ALTERNATIVE '-e "s/.fadeIn()/.show()/g" -e "s/.fadeIn(\([[:graph:][:space:]]\))/.
     show(\1)/g" -e "s/.fadeOut()/.hide()/g" -e "s/.fadeOut(\([[:graph:][:space:]]\))
     /.hide(\1)/g"'
36 ALTERNATIVE '-e "s/.fadeIn()/.fadeTo(0, 1)/g" -e "s/.fadeIn(\([[:graph:][:space
     :]]\))/.fadeTo(\1, 1)/g" -e "s/.fadeOut()/.fadeTo(0, 0)/g" -e "s/.fadeOut(\([[:
     graph:][:space:]]\))/.fadeTo(\1, 0)/g"'
```

```
37 ALTERNATIVE '"s/.clearQueue()/.stop(true)/g"'
38 ALTERNATIVE '"s/.hasClass(\([[:graph:][:space:]]\))/.attr(''class'').match(\1)/g"'
39 ALTERNATIVE '"s/.submit()/.find(''input[type=submit]'').click()/g"'
40 ALTERNATIVE '"s/.trigger(''submit'')/.find(''input[type=submit]'').click()/g"'
41 ALTERNATIVE '"s/:visible/:not(:hidden)/g"'
42 ALTERNATIVE '"s/:gt(0)/:not(:first)/g"'
43 ALTERNATIVE '"s/.animate({ opacity: \([[:digit:][:punct:]]\) })/.fadeTo(0, \1)/g"'
44 ALTERNATIVE '-e "s/.animate({ opacity: 0 })/.fadeOut()/g" -e "s/.animate({ opacity:
       1 })/.fadeIn()/g"'
45 ALTERNATIVE '"s/.fadeTo(0, \([[:digit:][:punct:]]\))/.animate({ opacity: \1 })/g"'
46 ALTERNATIVE '-e "s/.animate({ opacity: 0 })/.hide()/g" -e "s/.animate({ opacity: 1
       })/.show()/g"'
47 ALTERNATIVE '"s/.animate({ opacity: 0 })/.css(''display'': ''none'')/g"'
48 ALTERNATIVE '-e "s/.fadeOut()/.animate({ opacity: 0 })/g" -e "s/.fadeIn()/.animate({
        opacity: 1 })/g"'
49 ALTERNATIVE '"s/.fadeOut()/.hide()/g"'
50 ALTERNATIVE '"s/.fadeOut()/.css(''display'': ''none'')/g"'
51 ALTERNATIVE '-e "s/.fadeTo(0, 1)/.fadeIn()/g" -e "s/.fadeTo(\([[:graph:][:space
       :]]\), 1)/.fadeIn(\1)/g" -e "s/.fadeTo(0, 0)/.fadeOut()/g" -e "s/.fadeTo(\([[:
       graph:][:space:]]\), 0)/.fadeOut(\1)/g"'
52 ALTERNATIVE '-e "s/.fadeTo(0, 1)/.show()/g" -e "s/.fadeTo(\([[:graph:][:space:]]\),
        1)/.show(\1)/g" -e "s/.fadeTo(0, 0)/.hide()/g" -e "s/.fadeTo(\([[:graph:][:space
       :]]\), 0)/.hide(\1)/g"'
53 ALTERNATIVE '"s/.fadeTo(0, 0)/.css(''display'': ''none'')/g"'
54 ALTERNATIVE '-e "s/.hide()/.animate({ opacity: 0 })/g" -e "s/.show()/.animate({
       opacity: 1 })/g"'
55 ALTERNATIVE '-e "s/.hide()/.fadeOut()/g" -e "s/.hide(\([[:graph:][:space:]]\))/.
       fadeOut(\1)/g" -e "s/.show()/.fadeIn()/g" -e "s/.show(\([[:graph:][:space:]]\))/
       .fadeIn(\1)/g"'
56 ALTERNATIVE '-e "s/.hide()/.fadeTo(0, 0)/g" -e "s/.hide(\([[:graph:][:space:]]\))/.
       fadeTo(\1, 0)/g" -e "s/.show()/.fadeTo(0, 1)/g" -e "s/.show(\([[:graph:][:space
       :]]\))/.fadeTo(\1, 1)/g"'
57 ALTERNATIVE '"s/.hide()/.css(''display'': ''none'')/g"'
58 ALTERNATIVE '"s/.css(''display'': ''none'')/.animate({ opacity: 0 })/g"'
59 ALTERNATIVE '"s/.css(''display'': ''none'')/.fadeOut()/g"'
60 ALTERNATIVE '"s/.css(''display'': ''none'')/.hide()/g"'
61 ALTERNATIVE '"s/.css(''display'': ''none'')/.fadeTo(0, 0)/g"'
62 ALTERNATIVE '-e "s/\([[:graph:][:space:]]\)\.toggleClass(\([[:graph:][:space:]]\),
        true)/\1.addClass(\2)/g" -e "s/\([[:graph:][:space:]]\)\.toggleClass(\([[:graph
       :][:space:]]\), false)/\1.removeClass(\2)/g"'
63 ALTERNATIVE '-e "s/\$(\([[:graph:][:space:]]\)).attr(''value'')/\1.val()/g" -e "s/\
       $(\([[:graph:][:space:]]\)).attr(''value'', \([[:graph:][:space:]]\))/\1.val
       (\2)/g"'
64 ALTERNATIVE '-e "s/.css(''height'')/.height()/g" -e "s/.css(''''height'''', \([[:
       graph:][:space:]]\))/.height(\1)/g" -e "s/.css(''width'')/.width()/g" -e "s/.css
       (''''width'''', \([[:graph:][:space:]]\))/.width(\1)/g"'
65 ALTERNATIVE '"s/:not(:hidden)/:visible/g"'
66 ALTERNATIVE '-e "s/:hidden/:not(:visible)/g" -e "s/:not(:visible)/:hidden/g"'
67 ALTERNATIVE '-e "s/:even/:not(:odd)/g" -e "s/:odd/:not(:even)/g"'
68 ALTERNATIVE '-e "s/:not(:odd)/:even/g" -e "s/:not(:even)/:odd/g"'
```

Appendix B

Experiments

In this Appendix, we provide additional details about the experiment presented in Section 6.4.3, where we discuss the effectiveness of the priority mechanism and the automated oracle.

In the experiments, we used three Web pages that rely on three old versions of the JQuery library (versions 1.2.6, 1.3.2 and 1.4.2). Each library has different issues that we indicate hereafter with their identifier in the JQuery issue tracker [1]:

- JQuery version 1.2.6 suffers from issues: 3380, 3395, **1414**
- JQuery version 1.3.2 suffers from issues: 5010, 4681, 4965, 4761, 4817, **5018**
- JQuery version 1.4.2 suffers from issues: 6999, 5316, 6158, 3828, **6610**

Each experiment consisted of a total of 100 visits of the three pages. For each visit we randomly chose one issue, and the corresponding page. We found workarounds automatically for 11 out of 14 issues. The three issues for which no workaround was found are shown in bold in the above lists.

The detailed results of the experiments are reported in Figures B.1, B.2 and B.3. Each figure corresponds to two executions, each with 100 accesses to faulty pages. The two executions used RAW with and without oracle, as marked in the figures. The three figures correspond to the same experiment repeated three times with different selections of issues, to check for the impact of the order in which issues occur, as explained in details in Section 6.4.3.

Each diagram reports the number of attempts needed to either find the valid workarounds, or to signal that a workaround does not exist (Y axis). The number of attempts are grouped according the causing issues. The issues for which no workaround could be found are labeled with *. The set of attempts per issue varies and indicates the number of times the respective issues were chosen in that execution. As discussed in Section 6.4.3 the oracle reduces the number of attempts both when a valid workaround is found and when not.

[1] `http://bugs.jquery.com/`

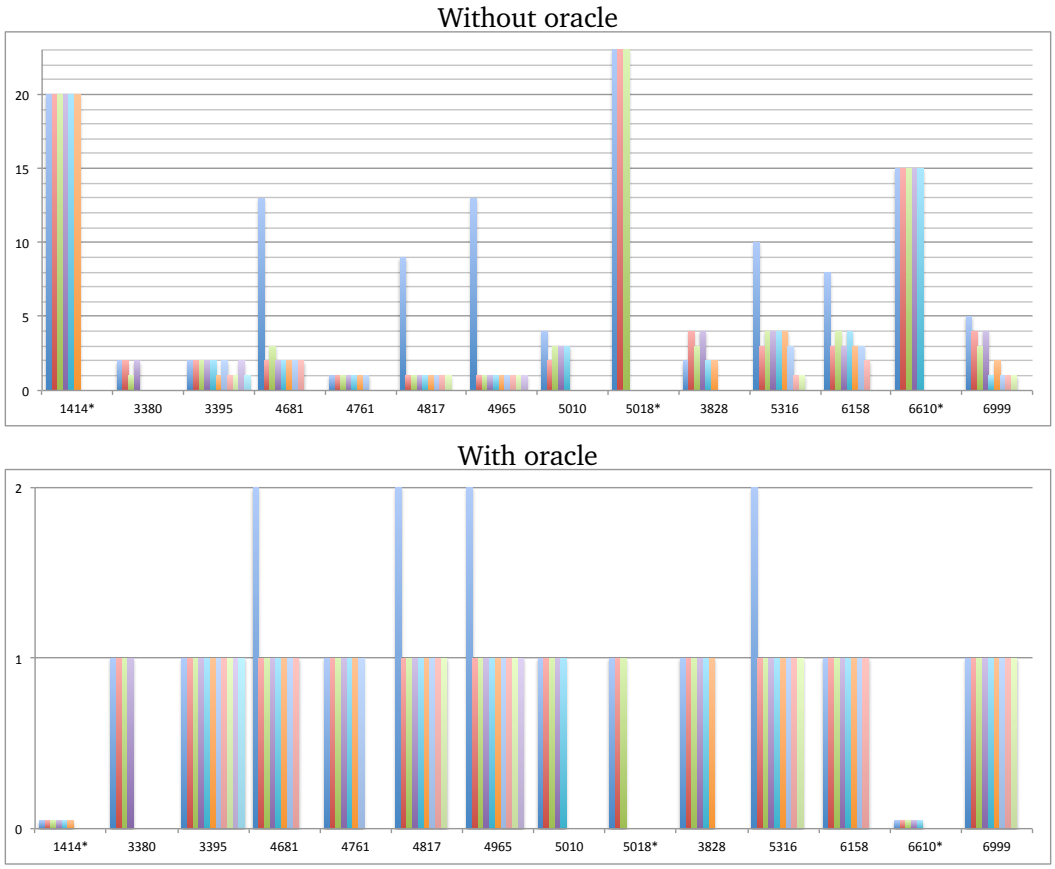


Figure B.1. First run

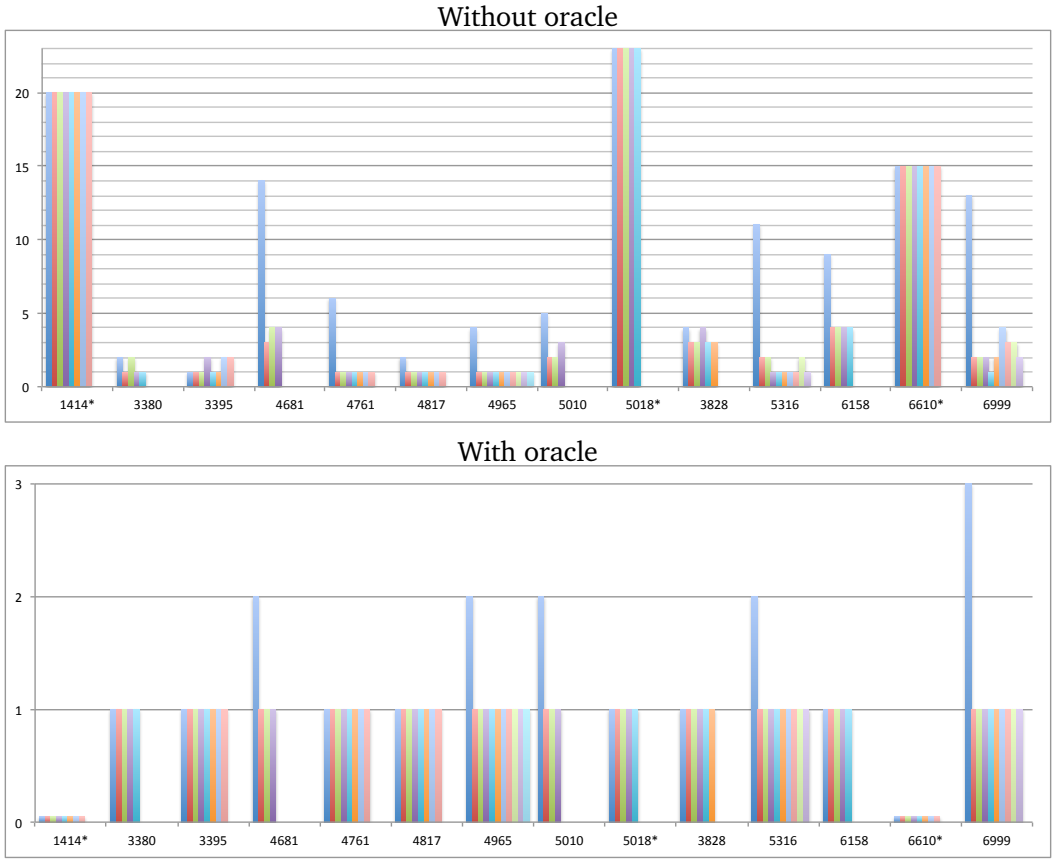


Figure B.2. Second run

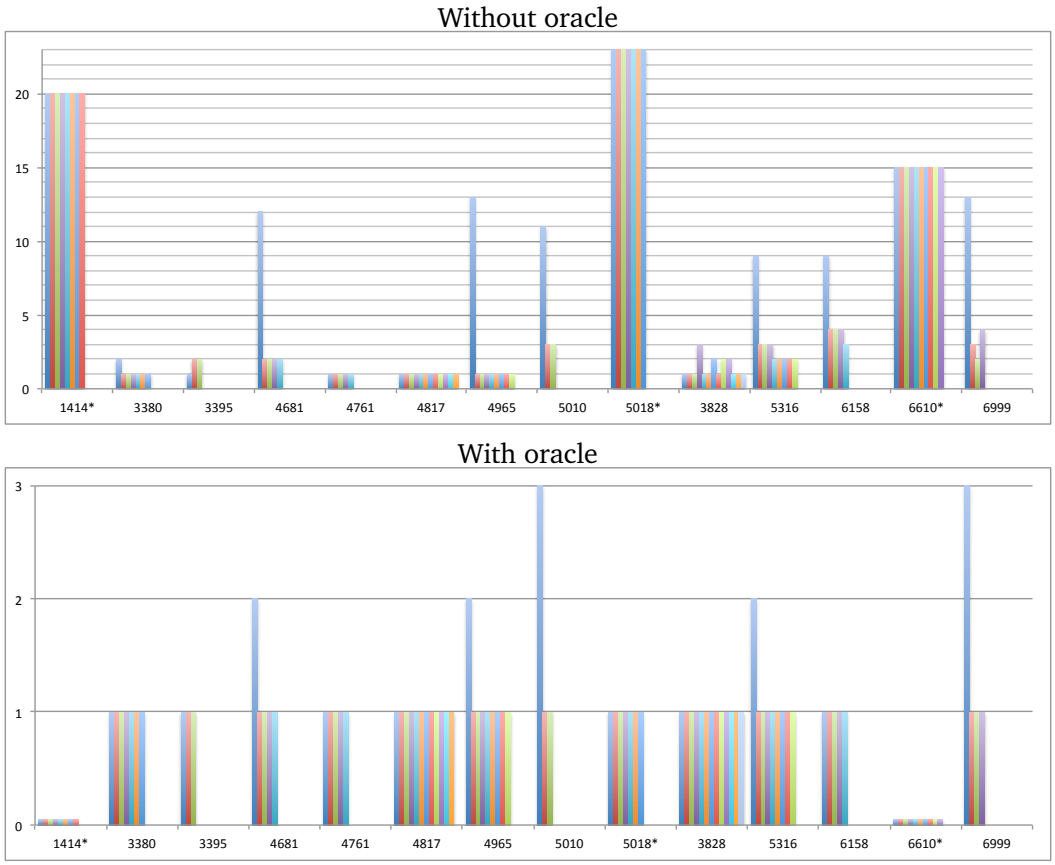


Figure B.3. Third run

Bibliography

[ACMF00] Hany H. Ammar, Bojan Cukic, Ali Mili, and Cris Fuhrman. A comparative analysis of hardware and software fault tolerance: Impact on software reliability engineering. *Annals Software Engineering*, 10(1-4):103–150, 2000.

[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.

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

[AY08] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In *CEC '08: Proceeding of IEEE Congress on Evolutionary Computation*, 2008.

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

[Bak97] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26:1343–1362, October 1997.

[BCL07] Danilo Bruschi, Lorenzo Cavallaro, and Andrea Lanzi. Diversified process replicae for defeating memory error exploits. In *WIA'07:3rd International Workshop on Information Assurance*. IEEE Computer Society, 2007.

[BGP07] Luciano Baresi, Sam Guinea, and Liliana Pasquale. Self-healing BPEL processes with dynamo and the JBoss rule engine. In *ESSPE '07: International workshop on Engineering of software services for pervasive environments*, pages 11–20, New York, NY, USA, 2007. ACM.

[BIPT09] Antonia Bertolino, Paola Inverardi, Patrizio Pelliccione, and Massimo Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 141–150, New York, NY, USA, 2009. ACM.

[BKL90] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson. Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering*, 16(2):238–247, 1990.

[BLR08] Patrick H. Brito, Rogério Lemos, and Cecília M. Rubira. Development of fault-tolerant software systems based on architectural abstractions. In *Proceedings of the 2nd European conference on Software Architecture*, ECSA '08, pages 131–147, Berlin, Heidelberg, 2008. Springer-Verlag.

[BY01] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001.

[BYM+98] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 368–, Washington, DC, USA, 1998. IEEE Computer Society.

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

[CCY98] Tsong Y. Chen, Shing C. Cheung, and Siu Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Dept. of Computer Science, Hong Kong Univ. of Science and Technology, 1998.

[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 *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[CKTZ03] Tsong Y. Chen, F.-C. Kuo, T. H. Tse, and Zhi Quan Zhou. Metamorphic testing and beyond. In *Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice*, pages 94–100, Washington, DC, USA, 2003. IEEE Computer Society.

[CKZ+03] George Candea, Emre Kiciman, Steve Zhang, Pedram Keyani, and Armando Fox. JAGR: An autonomous self-recovering application server. In *Active Middleware Services*, pages 168–178. IEEE Computer Society, 2003.

[CL02] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 231–255, London, UK, 2002. Springer-Verlag.

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

[CMP08] Herve Chang, Leonardo Mariani, and Mauro Pezzè. Self-healing strategies for component integration faults. In *ARAMIS'08: Proceedings of the first International Workshop on Automated engineeRing of Autonomic and run-tiMe evolvIng Systems*, pages 25 –32, 2008.

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

[CPW72] John R. Connet, Edward J. Pasternak, and Bruce D. Wagner. Software defenses in real-time control systems. In *in Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 94–99, 1972.

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

[CTZ03] Tsong Y. Chen, T. H. Tse, and Zhi Quan Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45(1):1–9, 2003.

[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 2006 international symposium on Software testing and analysis*, ISSTA '06, pages 233–244, New York, NY, USA, 2006. ACM.

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

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*, Berkeley, CA, USA, 2004.

[DKM+10] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 85–96, New York, NY, USA, 2010. ACM.

[DLWZ06] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with adabu. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*, pages 17–24, New York, NY, USA, 2006. ACM.

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

[Dob06] Glen Dobson. Using WS-BPEL to implement software fault tolerance for web services. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 126–133, Washington, DC, USA, 2006. IEEE Computer Society.

[DPT09] Giovanni Denaro, Mauro Pezzè, and Davide Tosi. Ensuring interoperable service-oriented systems through engineered self-healing. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 253–262, New York, NY, USA, 2009. ACM.

[DR03] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 78–95, New York, NY, USA, 2003. ACM.

[DR05] Brian Demsky and Martin Rinard. Data structure repair using goal-directed reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 176–185, New York, NY, USA, 2005. ACM.

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

[DZM09] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, 2009.

[EAmWJ02] Mootaz Elnozahy, Lorenzo Alvisi, Yi min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[EASC85] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *PODS '85: Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 215–229, New York, NY, USA, 1985. ACM.

[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 Systems Journal*, 41:111–125, January 2002.

[EGSK07] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of complex data structures. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 64–73, New York, NY, USA, 2007. ACM.

[EK08] Bassem Elkarablieh and Sarfraz Khurshid. Juzi: a tool for repairing complex data structures. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 855–858, New York, NY, USA, 2008. ACM.

[FB08] Vincenzo De Florio and Chris Blondia. A survey of linguistic structures for application-level fault tolerance. *ACM Computing Surveys*, 40(2):1–37, 2008.

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

[FM06] Maria Grazia Fugini and Enrico Mussi. Recovery of Faulty Web Applications through Service Discovery. In *SMR '06: 1st International Workshop on Semantic Matchmaking and Resource Retrieval: Issues and Perspectives*, Seoul, Korea, September 2006.

[FNWLG09] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *GECCO'09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954, New York, NY, USA, 2009. ACM.

[FO07] M. Muztaba Fuad and Michael J. Oudshoorn. Transformation of existing programs into autonomic and self-healing entities. In *ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, pages 133–144, Washington, DC, USA, 2007. IEEE Computer Society.

[FW07] Gordon Fraser and Franz Wotawa. Redundancy based test-suite reduction. In *Proceedings of the 10th international conference on Fundamental approaches to software engineering*, FASE'07, pages 291–305, Berlin, Heidelberg, 2007. Springer-Verlag.

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

[GHKT96] Sachin Garg, Yennun Huang, Chandra Kintala, and Kishor S. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. *SIGMETRICS Performance Evaluation Review*, 24(1):252–261, 1996.

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

[GMM09] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. Synthesizing intensional behavior models by graph transformation. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 430–440, Washington, DC, USA, 2009. IEEE Computer Society.

[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 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 365–, Washington, DC, USA, 2003. IEEE Computer Society.

[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.

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

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

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

[HC10] Ishtiaque Hussain and Christoph Csallner. Dynamic symbolic data structure repair. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 215–218, New York, NY, USA, 2010. ACM.

[HKKF95] Yennun Huang, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, page 381, Washington, DC, USA, 1995. IEEE Computer Society.

[HM08] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys*, 40(3):1–28, August 2008.

[Hor01] Paul Horn. Autonomic computing: IBM perspective on the state of information technology. In *AGENDA 01*, Scottsdale, AR, 2001.

[HRD07] Johannes Henkel, Christoph Reichenbach, and Amer Diwan. Discovering documentation for java container classes. *IEEE Transactions on Software Engineering*, 33(8):526–543, 2007.

[IEE90] IEEE Computer Society. *IEEE Std. 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology*, sep 1990.

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

[JMSG07] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[JS09] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th international symposium on Software testing and analysis*, pages 81–92, 2009.

[JUn11] Junit testing framework, 2011. http://www.junit.org.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KDM+96] Kostas Kontogiannis, Renato DeMori, Ettore Merlo, M. Galler, and M. Bernstein. *Pattern matching for clone and concept detection*, pages 77–108. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[KK07] Israel Koren and C. Mani Krishna. *Fault Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[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, July 2002.

[KL86] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, 12:96–109, January 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 Saddek Bensalem and Doron A. Peled, editors, *Runtime Verification*, pages 101–114. Springer-Verlag, Berlin, Heidelberg, 2009.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[KSNM05] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

[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.

[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 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 20–20, Berkeley, CA, USA, 2004. USENIX Association.

[LMX05] Nik Looker, Malcolm Munro, and Jie Xu. Increasing web service dependability through consensus voting. In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 2*, pages 66–69, Washington, DC, USA, 2005. IEEE Computer Society.

[LS04] Bev Littlewood and Lorenzo Strigini. Redundancy and diversity in security. *Computer Security âĂŞ ESORICS 2004*, pages 423–438, 2004.

[LV62] R. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, 6(2):200–209, 1962.

[MB08] Adina Mosincat and Walter Binder. Transparent runtime adaptability for BPEL processes. In Athman Bouguettaya, Ingolf KrÃijger, and Tiziana Margaria, editors, *ICSOC '08: Proceedings of the 6th International Conference on Service Oriented Computing*, volume 5364 of *Lecture Notes in Computer Science*, pages 241–255, 2008.

[MCLL07] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. Automatic testing of object-oriented software. In *Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '07, pages 114–129, Berlin, Heidelberg, 2007. Springer-Verlag.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[MFC+09] Bertrand Meyer, Arno Fiva, Ilinca Ciupa, Andreas Leitner, Yi Wei, and Emmanuel Stapf. Programs that test themselves. *IEEE Software*, pages 22–24, 2009.

[MLM96] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 244–, Washington, DC, USA, 1996. IEEE Computer Society.

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

[MP08] Leonardo Mariani and Fabrizio Pastore. Automated identification of failure causes in system logs. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 117–126, Washington, DC, USA, 2008. IEEE Computer Society.

[MPPar] Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. Dynamic analysis for diagnosing integration faults. *IEEE Transactions on Software Engineering*, –(PrePrints), to appear.

[MSK09] Christian Murphy, Kuang Shen, and Gail Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, ISSTA '09, pages 189–200, New York, NY, USA, 2009. ACM.

[NBTU08] Yarden Nir-Buchbinder, Rachel Tzoref, and Shmuel Ur. Deadlocks: From exhibiting to healing. In Martin Leucker, editor, *Runtime Verification*, pages 104–118. Springer-Verlag, Berlin, Heidelberg, 2008.

[NBZ07] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: automatically correcting memory errors with high probability. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 1–11, New York, NY, USA, 2007. ACM.

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

[NTEK+08] Anh Nguyen-Tuong, David Evans, John C. Knight, Benjamin Cox, and Jack W. Davidson. Security through redundant data diversity. In *DSN'08: IEEE International Conference on Dependable Systems and Networks*, pages 187–196, 2008.

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

[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.

[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 22nd symposium on Operating systems principles*, 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 29th international conference on Software Engineering*, ICSE '07, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

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

[Pul01] Laura L. Pullum. *Software Fault Tolerance Techniques and Implementation*. Artech House, Inc., Norwood, MA, USA, 2001.

[QTSZ05] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 235–248, New York, NY, USA, 2005. ACM Press.

[QTZS07] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3):7, 2007.

[Ran75] Brian Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[Ros95] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21:19–31, January 1995.

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

[SRFA99] Frederic Salles, Manuel Rodrguez, Jean-Charles Fabre, and Jean Arlat. Metakernels and fault containment wrappers. In *International Symposium on Fault-Tolerant Computing*, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[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 *SAINT '08: Proceedings of the 2008 International Symposium on Applications and the Internet*, pages 33–39, Washington, DC, USA, 2008. IEEE Computer Society.

[TBFM06] Yehia Taher, Djamal Benslimane, Marie-Christine Fauvet, and Zakaria Maamar. Towards an approach for web services substitution. In *IDEAS '06: Proceedings of the 10th International Database Engineering and Applications Symposium*, pages 166–173, Washington, DC, USA, 2006. IEEE Computer Society.

[TMB80] David J. Taylor, David E. Morgan, and James P. Black. Redundancy in data structures: Improving software fault tolerance. *IEEE Transactions on Software Engineering*, 6(6):585–594, 1980.

[Wey82] Elaine J. Weyuker. On testing Non-Testable programs. *The Computer Journal*, 25(4):465–470, November 1982.

[WHV+95] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and C. Kintala. Checkpointing and its applications. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 22–31, Washington, DC, USA, 1995. IEEE Computer Society.

[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 *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72, New York, NY, USA, 2010. ACM.

[WSGF04] Vera Wahler, Dietmar Seipel, Jurgen Wolff v. Gudenberg, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In

Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop, pages 128–135, Washington, DC, USA, 2004. IEEE Computer Society.

[WTNF09] Westley Weimer, and Claire Le Goues ThanVu Nguyen, and Stephanie Forrest. Automatically finding patches using genetic programming. In *ICSE'09: Proceeding of the 31st International Conference on Software Engineering*, pages 364–374, 2009.

[YC75] S. S. Yau and R. C. Cheung. Design of self-checking software. In *Proceedings of the International Conference on Reliable software*, pages 450–455, New York, NY, USA, 1975. ACM.

[Zel99] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-7, pages 253–267, London, UK, 1999. Springer-Verlag.

[ZH02] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28:183–200, February 2002.

[Zha07] Rui Zhang. Modeling autonomic recovery in web services with multi-tier reboots. In *ICWS'07: Proceedings of the IEEE International Conference on Web Services*, 2007.