Automatic Workarounds: Exploiting the Intrinsic Redundancy of Web Applications

ANTONIO CARZANIGA, University of Lugano, Switzerland
ALESSANDRA GORLA, IMDEA Software Institute, Spain
NICOLÒ PERINO, University of Lugano, Switzerland
MAURO PEZZÈ, University of Lugano, Switzerland and University of Milano-Bicocca, Italy

Despite the best intentions, the competence, and the rigorous methods of designers and developers, software is often delivered and deployed with faults. To cope with imperfect software, researchers have proposed the concept of *self-healing* for software systems. The ambitious goal is to create software systems capable of detecting and responding "autonomically" to functional failures, or perhaps even preempting such failures, to maintain a correct functionality, possibly with acceptable degradation. 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 either explicitly or implicitly. 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 does not always satisfy the time and cost constraints of many software projects.

With this paper we take a different approach. We observe that modern software systems naturally acquire another type of redundancy that is not introduced deliberately but rather arises *intrinsically* as a by-product of modern modular software design. We formulate this notion of intrinsic redundancy and we propose a technique to exploit it 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 *automatic workarounds.* In essence, a workaround amounts to replacing some failing operations with alternative operations that are semantically equivalent in their intended effect, but that execute different code and ultimately avoid the failure. The technique we propose finds such workarounds automatically. We develop this technique in the context of Web applications. In particular, we implement this technique within a browser extension, which we then use in an evaluation with several known faults and failures of three popular Web libraries. The evaluation demonstrates that automatic workarounds are effective: out of the nearly 150 real faults we analyzed, 100 could be overcome with automatic workarounds, and half of these workarounds found automatically were not publicly known before.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries*

General Terms: Reliability, Design

Additional Key Words and Phrases: Automatic Workarounds, Web Applications, Web API

ACM Reference Format:
Carzaniga, A., Gorla, A., Perino, N., and Pezzè, M. 2013. Automatic Workarounds: Exploiting the Intrinsic Redundancy of Web Applications ACM Trans. Softw. Eng. Methodol. 0, 0, Article 00 (0000), 43 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Complete correctness is an elusive objective for many software systems. One reason is that developing correct software is fundamentally difficult, since even the best methods are typically either incomplete or too costly. For example, testing can improve the quality of software at reasonable costs, but cannot guarantee the absence of faults, and conversely, analysis and verification techniques that give absolute guarantees are often intractable. Another reason is that software systems are typically used in ways or in environments that are poorly specified or even never anticipated, and yet the software is expected to somehow behave correctly. In other words, even those systems that could be considered correct, in the sense that they behave according to the specifications, effectively *become* faulty because they are used in unspecified situations or because user expectations change. In summary, whether due to economic pressure or unrealistic expectations, software is often deployed with faults, actual or perceived.

This *status quo* challenges the traditional notion of software maintenance and even the very notion of correctness. Traditional fault-fixing is in fact an off-line and often slow process. Typically, the developers would have to be notified of a failure. They would then have to reproduce it, identify the root causes and develop a corrective change. The corrective change would then be incorporated in a new version of the system that would then be released and deployed. And each of these activities is time consuming, so a failure may reduce the utility of a system for a long period of time. The situation is somewhat different but not less challenging in the case of Web applications. On the positive side, the release process may be simpler and faster, since it involves relatively few application servers independently of the end-users. On the other hand, the execution platform is more complex and diverse, with a multi-tier server side and a client side connected by a more or less fast and reliable network, possibly through a chain of filtering and caching proxy servers, and with each client using a potentially different combination of operating system, browser type, and version. And such complexity and diversity may lead to specific problems that are hard to reproduce and therefore hard to fix.

We consider a different notion of fault-fixing and in general of software correctness. We still assume a traditional development and maintenance process. However, we only assume that such process would produce software that is *almost* correct—that is, software that behaves as expected in most situations, but that may occasionally fail. We then propose to cope with such failures on-line within the deployment environment.

This general idea is not new. In fact, in order to cope with software faults while maintaining the system somewhat functional, researchers have for decades studied techniques to either avoid or mask failures. Such techniques are generally referred to as fault tolerance and, more recently, self-healing. In essence, fault tolerance techniques attempt to mimic reliable hardware architectures in software. In hardware, reliability is typically achieved by replicating components, under the assumption that faults are independent and therefore that multiple components are very unlikely to fail at the same time. For example, RAID is a successful technology that overcomes faults in disk storage (as well as some performance limitations) by bundling and controlling an array of disks, and presenting them as a single and more reliable storage device [Patterson et al. 1988].

The successes of redundant hardware 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 among the most well-known techniques in this line of research [Avizienis 1985; Randell 1975]. In N-version programming, a system consists of multiple independently designed and developed versions of the same program. The N versions execute independently with the same input, and their output is combined, typically through a majority vote, to produce the output of the system. Similarly, recovery blocks are independent blocks of code intended to perform the same function that, instead of running in parallel, may be called sequentially, each one taking over in case of a failure in the previous one.

The essential ingredient of N-version programming and recovery blocks, as well as many other software fault-tolerance techniques, is some form of redundancy [Carzaniga et al. 2009]. In particular, a redundancy that is introduced *deliberately* within the software system. Unfortunately, this form of deliberate redundancy has some fundamental shortcomings in cost and effectiveness, which can be explained by considering some differences between hardware and software. Unlike redundancy in hardware, which multiplies production costs, redundancy in software multiplies design and development costs, which are typically much more onerous. Also, unlike hardware faults, which are mostly independent, software faults exhibit correlations that fundamentally defeat the whole idea of redundancy [Knight and Leveson 1986; Brilliant et al. 1990; Hatton 1997].

In this research we focus on Web applications, and in particular on failures that may depend on the integration of external Web services. Modern "Web 2.0" applications are ever more sophisticated and now increasingly similar to desktop applications in terms of their rich functionalities and user interactions. Such applications often integrate and combine external services through complex Web libraries,[1] which may lead to integration problems often exacerbated by the scale, the heterogeneity, and the decentralized nature of the Web.

Consider the case of popular Web services and APIs used by many Web applications, some of which may also be developed by end-users. These cases pose problems on both sides of the Web API. On the one hand, the applications are likely to use the Web libraries and services in unanticipated ways, pushing them beyond the nominal and even the correct user behavior assumed by the developers. On the other hand, the applications have no control over the release schedule of the APIs and services they use. So regression problems introduced with new releases may lead to application problems that cannot be fixed by the application developer, and that may only be avoided by means of appropriate workarounds. In fact, the use of such workarounds is a common practice among developers of Web applications, as indicated by the many discussions of workarounds found in on-line support groups.

In this paper we propose a new approach to software self-healing for Web applications. We still propose to exploit redundancy to overcome failures. However, our key insight is to exploit a form of redundancy that is *intrinsic* in software systems. This form of redundancy is a by-product of modern software design and development methods, and as such does not incur the extra costs that characterize deliberate redundancy. We first argue that software is indeed inherently redundant, and then develop and exploit this notion of redundancy to obtain effective workarounds for Web applications

Several considerations lend plausibility to the idea of intrinsic redundancy. First, large software systems are known to contain significant quantities of code clones, including semantic clones [Gabel et al. 2008; Jiang and Su 2009]. Admittedly, code clones may be considered a sign of poor design. However, other forms of redundancy are a natural consequence of good design practices, such as modularization and reuse. In fact, modern software is often built out of components, probably the most common way of reusing software modules. Thus a system might incorporate different components, each used for a different functionality, that also provide additional and similar or even

[1]Examples of such external functionalities include photo sharing, mapping, blogging, advertising and many more. The `www.programmableweb.com` site illustrates well the growing popularity of Web APIs. In May 2011 the site referenced more than 3000 Web APIs.

identical and therefore redundant functionalities. Other times the same functionality is intentionally duplicated within a system. This might be done to address different non-functional requirements. For example, the GNU Standard C++ Library implements its basic stable sorting function using the insertion-sort algorithm for small sequences and merge-sort for the general case. Another source of redundancy is design for reusability in libraries or frameworks. Reusable components are often designed, for the convenience of their users, to export the same logical functionality through a wide variety of slightly different but mostly interchangeable interfaces that may also be backed up by partially redundant implementations. Yet another source of redundancy are deprecated or otherwise duplicated functionalities in libraries that are kept to guarantee backward compatibility. Beyond these plausibility arguments, in this paper we present experimental evidence that demonstrates the existence of intrinsic redundancy in software systems.

Not only redundancy is inherent in software systems, but this type of intrinsic redundancy can also be exploited to achieve some level of self-healing through what we call *automatic workarounds*. This is the main technical contribution of this paper. We say that two *different* executions are *redundant* if they present the same observable behavior. We then say that a redundant execution is a *workaround* when it substitutes a failing execution producing the expected behavior. At a high-level, the key idea is to first document the intrinsic redundancy of a system, in particular through the specification of *rewriting rules* that substitute executions with redundant ones, and then to use those rules to automatically find workarounds for failing executions. Thus, a workaround substitutes a code fragment that executes faulty operations with one that is supposedly equivalent but that in reality executes redundant code that somehow avoids the faulty operations.

In this paper we develop the notion of automatic workarounds in the context of Web applications where we also evaluate their effectiveness. We focus specifically on JavaScript libraries, since those are essential components of modern Web applications, and are often the source of their failures [Carzaniga et al. 2010a]. We also make two important assumptions that are reasonable in the context of Web applications and that we validate in part through specific experiments. First, we assume that re-executing code on the client side does not invalidate the state of the application on the server side. Second, we take advantage of the highly interactive nature of Web applications by assuming that users are willing and able to report perceived failures.

We start by analyzing the issue trackers of several popular JavaScript libraries to categorize faults and redundant executions often proposed by developers as temporary workarounds for open problems. We find that workarounds are often proposed to avoid known issues, that such workarounds are used as temporary patches to prevent failures caused by faults that are yet to be fixed, and that sometimes workarounds can also lead to permanent fixes for those faults. We use the information extracted from the issue trackers to identify families of workarounds, which we then generalize and abstract into *rewriting rules*.

We then apply the rewriting rules in response to unanticipated failures in an attempt to find valid workarounds. We implement this technique to find such automatic workarounds within a browser extension. The extension allows users to report failures, reacts by applying rewriting rules that are likely to lead to a workaround, and then allows the user to validate the execution of those workarounds.

We evaluate the proposed technique by selecting more than 150 known issues of three popular JavaScript libraries and by trying to automatically find valid workarounds for them. The results of our experiments are encouraging, since a failure can be automatically avoided for more than 100 of those issues.

This paper extends a previous paper published at SIGSOFT FSE in 2010 [Carzaniga et al. 2010a]. In the FSE paper we presented the core idea of exploiting intrinsic software redundancy to automatically generate workarounds for Web applications, and provided initial experimental evidence of the applicability of the technique. This paper extends the FSE paper in several ways: (i) We discuss the nature and characterize the presence of intrinsic redundancy in Web applications. (ii) We provide detailed examples of rewriting rules, a key element of our approach, as well as a definition of the syntax and semantics of code rewriting rules for JavaScript using the λJS formalism. (iii) We present the detailed design of the enhanced prototype used for our experiments. (iv) We present the results of a new set of experiments conducted on large applications that confirm and significantly extend the experimental evaluation presented in the FSE paper. (v) We experimentally validate our hypothesis that users can detect failures in Web applications.

The remainder of the paper is structured as follows. In Section 2, we discuss the intrinsic redundancy of software libraries: We investigate the existence of redundancy in popular libraries, introduce the concept of rewriting rules, and show how rewriting rules can efficiently capture some redundancy present in Web libraries. In Section 3, we present the main approach to automatically generate workarounds, motivate and formalize the concept of rewriting rules, and illustrate the priority and validator mechanisms that we use to identify valid workarounds among the many redundant executions identified with rewriting rules. In Section 4, we illustrate the main features of the prototype implementation that we used in our experiments. In Section 5, we present the results of our empirical evaluation on popular libraries widely used in Web applications: Google Maps, YouTube and JQuery. In Section 6, we overview relevant work that exploits software redundancy to prevent system failures and improve software reliability, and discuss the relation with the approach presented in this paper. In Section 7, we summarize the main contribution of the paper, and indicate new research directions opened by the results of our research.

2. INTRINSIC REDUNDANCY

Many studies indicate that software is intrinsically redundant. Intuitively, two different executions are redundant when they have the same observable behavior, that is, if a user cannot distinguish the outcome of the two executions. Two software components, for example two methods, are redundant when their executions are redundant, that is, they produce the same expected effect for all relevant inputs. Redundant executions do not necessarily behave in exactly the same way. They may for example produce different internal states or have different performance or usability characteristics. Still, they have the same effect from the perspective of an external observer, that is, any element that interacts with the component. For instance, two implementations of a sorted container may use different sorting algorithms with different performance profiles, and they may store data using different internal structures, but still they may be redundant from an external viewpoint if they both return a sorted set as expected by the external observer, who has no access to the internal state and may or may not see or in any case may well tolerate performance differences.

Intrinsic redundancy can manifest itself at different levels of granularity: it can involve entire components, as in the case of multiple similar reusable libraries, redundant functions or methods, as in the case of reusable components, and only few redundant statements, as in the case of semantic or syntactic clones. Some kinds of redundancy, for instance redundancy at the unit level in the form of syntactic code clones, may derive from poor development processes and may increase development and maintenance costs, and should therefore be avoided when possible [Baxter et al.

1998]. Furthermore, code clones, including semantic clones, may be limited to fragments of code that may not be usable as replacements in a different context. However, many other forms of redundancy are a natural consequence of good design practices, and as such are present in well designed systems, and can be safely exploited. In particular, we observed that redundancy at the function or method level is quite common in reusable components, precisely because those are designed to offer many variants of their functionality in order to suit specific user needs. This is the type of redundancy we are mainly interested in.

As an example of redundancy induced by a design for reusability, consider *JQuery*, a popular library for Web applications. *JQuery* offers many ways to display elements in a Web page: fadeIn(), show(), fadeTo(), animate(), etc. Although these functions differ in some details, they have the same effect from the user viewpoint, and thus represent a form of functional redundancy. Similarly, many graphic toolkit libraries provide different methods for setting the position and dimensions of graphic elements. For instance, the Java *SWT* library offers the mutually interchangeable methods setLocation(Point) and setLocation(int x, int y), setSize(Point) and setSize(int), etc. Many other examples are easy to find in container libraries. Containers typically offer methods to add a single element (for instance, add()) and methods to add elements from another container (for instance, addAll()), which can become functionally indistinguishable, for example when addAll() is used with a container that contains a single element. Also, the Java containers offer the functions add(Component comp), add(Component comp, int index), add(Component comp, Object constraints), add(Component comp, Object constraints, int index), remove(Component comp), remove(int index), removeAll(), that can be combined in different ways to produce indistinguishable results. Interchangeable interfaces may or may not correspond to radically different implementations, and therefore may be more or less redundant. As we will see, our goal is to exploit this kind of redundancy opportunistically, whenever it is present and whenever it can be effective.

Performance optimization is another source of redundancy. For example, the Apache library *Ant*[2] provides a StringUtils class with a method endsWith() that replicates the functionality of the same method of the standard Java String class, but more efficiently. Similarly, the method frequency() of the CollectionUtils class in *Ant* is an efficient reimplementation of the same method in java.util.Collection, and the method SelectorUtils.tokenizePathAsArray() re-implements tokenizePath(), etc. Another example is the *log4J* library, which is designed to duplicate many of the functionalities offered by the standard Java library *java.util.Logging* but with better performance.

A library might also offer the same service in two variants optimized for different cases. A classic example is sorting. A good implementation of a sorting function is likely to use different sorting algorithms, even to implement a single interface function. For example, the GNU Standard C++ Library implements its basic (stable) sorting function using the insertion-sort algorithm for small sequences, and merge-sort for the general case. Similarly, other functions may be implemented in two or more variants, each one optimized for a specific case, for instance memory vs. time.

Redundancy may also come from the need to guarantee backward compatibility, as in the case of the many deprecated classes and methods available in Java libraries. For example, the Java 6 standard library contains 45 classes and 365 methods that are deprecated and that duplicate exactly or almost exactly the same functionality of newer classes and methods.

All the examples discussed so far show that intrinsic redundancy is plausible. Our recent studies show that such redundancy indeed exists in software systems, and in

[2]`http://ant.apache.org`

particular in reusable components frequently used in Web applications [Carzaniga et al. 2008; Carzaniga et al. 2010a].

2.1. Rewriting Rules

We capture the intrinsic redundancy of software systems at the level of function or method call by means of rewriting rules. A rewriting rule substitutes a code fragment with a different code fragment with the same observable behavior as the original one. We thus abstract away from internal aspects, such as details of the internal state that do not affect the observable behavior of the system, and non-functional properties, such as performance or usability.

Let us consider for example a container that offers the operations add(x) to add an item, addAll(x,y,. . .) to add several items, and remove(x) to remove an item. The executions of add(a); add(b) and of addAll(a,b) produce the same observable result, as expressed by the following rule:

$$\text{addAll(a,b)} \quad \rightarrow \quad \text{add(a); add(b)}$$

Similarly the executions of addAll(a,b); remove(b) and add(a) are equivalent, as expressed by the following rule:

$$\text{add(a)} \quad \rightarrow \quad \text{addAll(a,b); remove(b)}$$

Notice that the executions of add(a); add(b) and addAll(b,a) may also produce the same result if the insertion order and the possibly different internal states are irrelevant or invisible externally.

Our approach can be extended using other forms of transformations, beyond the syntactic replacement of method calls, to better cover other programming constructs. For example, to deal with API migration, Nita and Notkin propose the use of "twinnings" capable also of expressing the replacement of types and exceptions [Nita and Notkin 2010].

2.2. Common Forms of Redundancy in Web Applications

In order to better understand and model the kind of redundancy present in Web applications, we studied the API of several popular Web libraries.[3] As a result, we identified three common forms of redundancy with some corresponding rewriting rules. We classify these as *functionally null*, *invariant* and *alternative* rewriting rules. Although other forms of redundancy may not fall into these three categories, the type of redundancy that belongs to these categories has been effective in automatically generating workarounds. Other forms of redundancy may further extend the effectiveness of our approach. In the following we illustrate these forms of redundancy using concrete examples of rewriting rules that express successful workarounds used to cope with real faults.

Functionally null rewriting rules. These are general rules that express the simplest form of redundancy. As discussed above, we define executions as redundant if they have the same functional behavior, that is, their functional results are indistinguishable from the user viewpoint. Thus, rules that introduce only operations that deal with non functional aspects of the computation produce redundant executions. For example, the Thread.sleep() operation in Java controls the scheduling of the application

[3]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`)

```
SEQ1a: | map = new GMap2(document.getElementById("map"));
       | map.setCenter(new GLatLng(37,-122),15);
       | map.openInfoWindow(new GLatLng(37.4,-122), "Hello World!");
                                  ⇓
SEQ1b: | map = new GMap2(document.getElementById("map"));
       | setTimeout("map.setCenter(new GLatLng(37,-122),15)",500);
       | map.openInfoWindow(new GLatLng(37.4,-122), "Hello World!");
```

Fig. 1. Application of a functionally null rewriting rule: the rewritten code fragment differs from the original only for the functionally null operation setTimeout.

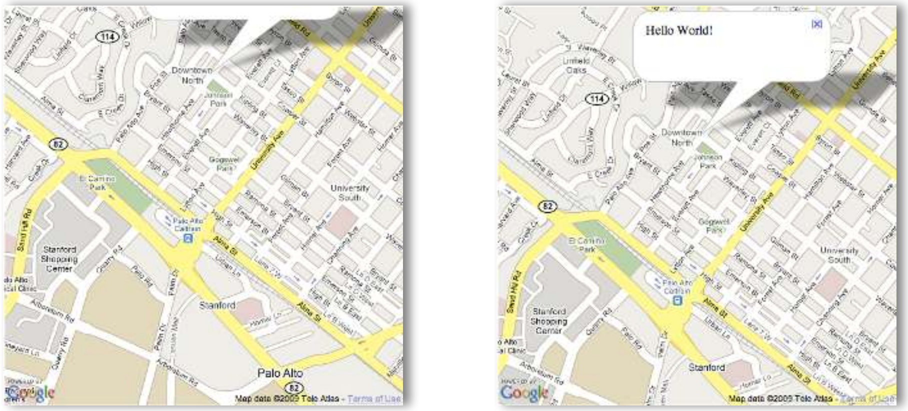


Fig. 2. Issue 519 of the Google Maps API, the faulty behavior on the left hand side is obtained by executing SEQ1a of Figure 1; the correct behavior on the right hand side is obtained by executing SEQ1b of Figure 1.

but does not alter its functional behavior, and similarly the setTimeout operation in JavaScript delays the execution of some operations without altering their functional effects. Although less common, we found functionally null operations implemented also in software components. For example the JQuery JavaScript framework that enhances HTML pages with JavaScript functionalities offers the delay() function that sets a timer to delay the execution of an animation effect.

Figure 1 shows a simple example of application of a functionally null rewriting rule for JavaScript. The rule produces a code fragment that differs from the original only for the functionally null operation setTimeout. The code fragment identified as SEQ1b was suggested as a workaround for code SEQ1a to fix Issue 519 of Google Maps, which was reported in July 2008.[4] Figure 2 illustrates the issue. The original code (SEQ1a) did not show the information window correctly when the openInfoWindow function, which opens the information window, was invoked immediately after setCenter, which sets the center of the map to a given geographical location. The modified code that executes SEQ1b instead of SEQ1a, and thus differs only for the invocation of settimeout, avoids the problem and produces the desired result.

Invariant rewriting rules. These rules describe sequences of operations with no functional effect when executed in some contexts, and therefore are also quite general. Invariant operations are typically sequences of two complementary operations in which the second one reverse the effect of the first. Simple examples of complementary operations are add and remove operations for containers, or zoom-in and zoom-out oper-

[4]The GoogleMaps issues presented in this paper are now closed.

```
SEQ2a: | polyline.enableDrawing();
                        ⇓
       | v = polyline.deleteVertex(polyline.getVertexCount()-1);
SEQ2b: | polyline.insertVertex(polyline.getVertexCount()-1,v);
       | polyline.enableDrawing();
```

Fig. 3. Application of an invariant rewriting rule. The rewritten code adds the invariant sequence of operations v = polyline.deleteVertex(polyline.getVertexCount()-1); polyline.insertVertex(polyline.getVertexCount()-1,v).

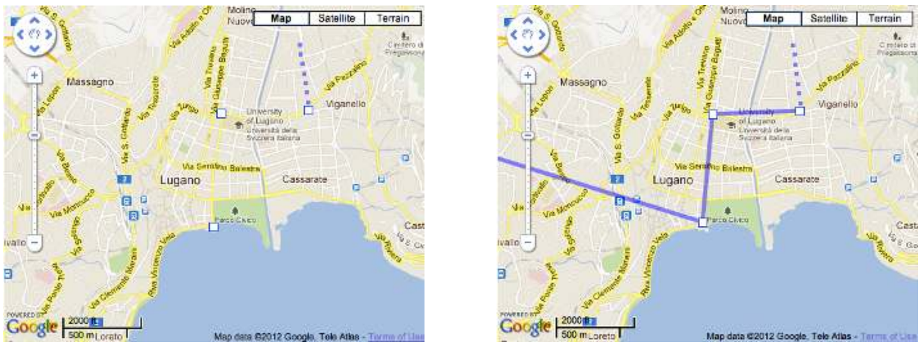


Fig. 4. Issue 1305 of the Google Maps API, the faulty behavior on the left hand side is obtained by executing SEQ2a, the correct behavior of the right hand side is obtained by executing SEQ2b.

```
       | map.addOverlay(first);
SEQ3a: | function showOverlay(){ first.show(); }
       | function hideOverlay(){ first.hide(); }
                        ⇓
       | map.addOverlay(first);
SEQ3b: | function showOverlay() { map.addOverlay(first); first.show(); }
       | function hideOverlay() { map.removeOverlay(first); }
```

Fig. 5. Application of an alternative rewriting rule that produces a different sequence of operations with the same expected result.

ations for graphical elements. When suitably combined, such operations do not alter the functional behavior of the application and can thus generate valid rewriting rules.

Figure 3 shows a simple example of an invariant rewriting rule that differs only for the invariant operations deleteVertex()-insertVertex(), which do not alter a shape when the vertex provided as parameter to both functions already belongs to the shape. Sequence SEQ2b was suggested as a workaround for sequence SEQ2a to fix Issue 1305 of Google Maps, which was reported in 2008. Figure 4 illustrates the issue: the original code (SEQ2a) did not produce the correct polygonal line when a new vertex was added to the line. The modified code (SEQ2b) that differs only for the invocation of a deleteVertex()–insertVertex() sequence, solves the problems and produces the desired result.

Alternative rewriting rules. These rules capture the most application-specific form of redundancy. Alternative operations are different sequences of operations that are designed to produce the same result. The equivalence we presented above between add(a); add(b) and addAll(a,b) on containers is a good example of a rule that captures alternative operations. In general, adding several elements together or one at a time should produce the same result.

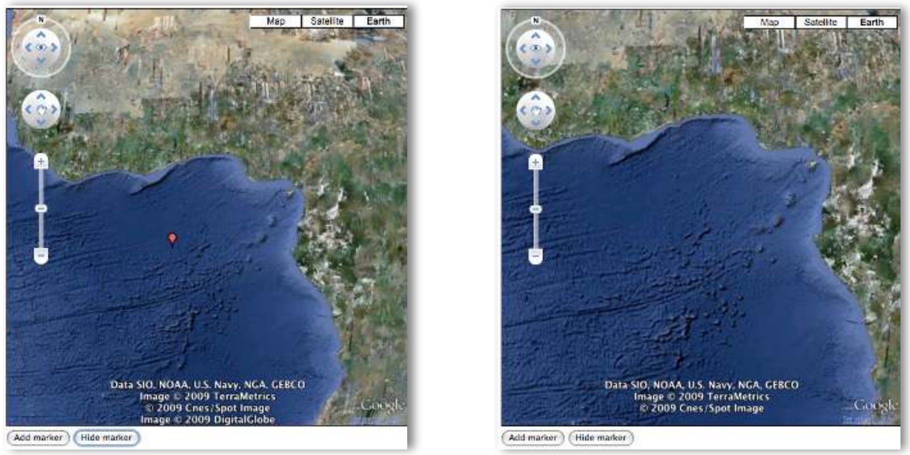

Map
Satellite
Earth
Data SIO, NOAA, U.S. Navy, NGA, GEBCO
Image © 2009 TerraMetrics
© 2009 Cnes/Spot Image
Image © 2009 DigitalGlobe
Add marker
Hide marker

Fig. 6. Issue 1209 of the Google Maps API, fixed with alternative operations.

Figure 5 shows a simple example of the application of an alternative rewriting rule. Sequence SEQ3b was suggested as a workaround for sequence SEQ3a to fix Issue 1209 of Google Maps, which was reported in April 2009. Figure 6 illustrates the issue: the original code (SEQ3a) did not remove the marker as expected; the modified code (SEQ2b) works around the problem and produces the desired result.

2.3. Identifying Rewriting Rules

Rewriting rules can be seen as a form of specification, and therefore may be formulated manually by developers. As reported in Section 5, domain experts with some familiarity with a Web library used by the application can identify an equivalent sequence and produce the corresponding rewriting rule in a few minutes. The presence of redundancy captured by means of rewriting rules is the essential ingredient of our technique, as libraries that come with more rewriting rules are more likely to offer an escape in case of failures. Thus, the amount of redundancy can be used as a proxy measure of the effectiveness of the approach. Our experience tells us that Web libraries are often redundant, and that it is worthwhile to model such redundancy in the form of equivalent sequences to make the Web applications that use such libraries more resilient to failures. We recently developed a technique that automatically identifies equivalent sequences, and that could be used to assess the applicability of our approach to a new library [Goffi et al. 2014].

For the experiments reported in this paper, we identified all the rewriting rules manually by examining the library API specifications. Alternatively, rewriting rules can be derived from other forms of specification, including API documentation written in natural language, or they may be synthesized automatically. Semantic clone detection [Jiang and Su 2009] and behavioral model inference [Dallmeier et al. 2006; Lorenzoli et al. 2008; Zhang et al. 2011] may provide useful information to identify equivalent sequences, and our recent work confirms that it is possible to find equivalent sequences automatically (in Java programs) using search-based techniques [Goffi et al. 2014].

3. AUTOMATIC WORKAROUNDS

In the previous section we introduced the idea of intrinsic redundancy and we presented a method to express this redundancy through code rewriting rules. We also showed real faults in Web libraries with corresponding workarounds that exploit the intrinsic redundancy of those libraries as expressed by specific rewriting rules.

As it turns out, workarounds are a common and sometimes effective way to cope with faulty Web applications, as demonstrated by the numerous cases discussed in online fault repositories and support systems (surveyed in Section 5.2). However, such workarounds are identified manually by expert users and developers, and they are communicated in an ad-hoc manner to other users who must still apply them manually. In other words, such workarounds are helpful but can not effectively mask failures.

We now present a technique to mask failures through *automatic workarounds.* The technique focuses on failures caused by incorrect interactions between Web applications and reusable components such as libraries, services, and frameworks. Specifically, the technique responds to a failure by finding and applying workarounds *automatically* and *at runtime.* The technique tries to construct a valid workaround for some failing code by applying relevant rewriting rules to that code. Thus the technique exploits the intrinsic redundancy expressed by those rewriting rules.

3.1. Automatic Workarounds for Web Applications

Automatic workarounds require a preliminary step at design time, namely the identification of a set of meaningful rewriting rules. Rewriting rules apply to a specific Web library and may be written by the developers or the users of that library, possibly with the aid of an automatic search algorithm [Goffi et al. 2014].

At runtime, the technique operates within a layer between the Web application and a Web library. The technique takes advantage of two fundamental properties of Web applications and JavaScript code: the interactive nature of the application and its stateless behavior on the client side. Web applications are typically highly interactive, so we assume that users can detect failures in the displayed pages and therefore invoke a workaround. Users can then also verify the correctness of the proposed workarounds, and therefore accept or reject them. We validated our hypothesis that users can detect failures in Web applications through a qualitative user study discussed in Section 5.3.2. JavaScript code runs mainly on the client side, which we assume to implement a stateless component of the application.[5] The stateless nature of the client side of the application allows one to reload a page without worrying about possible side-effects on the state of the application.

As shown in Figure 7, the layer that implements our technique acts as a proxy between the server that contains the requested Web pages and the client that requests the pages. We refer to the proxy as the *automatic workaround browser extension (AW browser extension).* When a user notifies a failure in the displayed page, the AW extension selects and deploys a workaround by applying relevant rewriting rules and by reexecuting that modified code. The AW extension has two execution modes. The normal execution mode (steps 1–5 in the diagram of Figure 7) corresponds to a client–server interaction without failures. In this case the AW extension simply scans the incoming JavaScript code looking for references to Web libraries for which the extension has applicable rewriting rules. If workarounds are applicable, the AW extension activates its failure reporting mechanism (a simple button in the browser toolbar). When the user reports a failure, the AW extension switches to its workaround mode (steps 6–10) in which it tries to automatically deploy a workaround for the signaled failure.

We now detail each operation of the AW extension. We start from a client requesting a specific page (step 1, request(URL)). The AW extension forwards every request to the server (step 2, forward(URL)). The AW extension then examines the objects returned by the server (step 3, send(page)) looking for references to third-party libraries aug-

[5]In this work we do not consider Ajax requests.

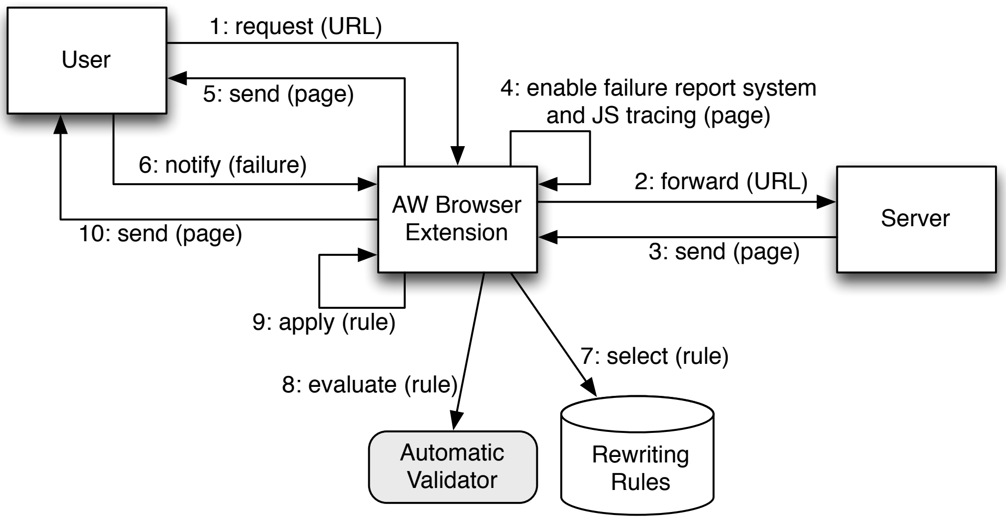


Fig. 7. The high level architecture of the Automatic Workarounds approach for Web applications. The UML communication diagram indicates the main logical components, the essential actions and the core parameters.

mented with rewriting rules. If the page contains references to such libraries, the AW extension activates a failure-reporting mechanism in the browser and instruments the JavaScript code to trace the execution (step 4, enable_failure_report_and_JS_tracing(page)). The execution trace may be used later to select relevant workarounds (as described in Section 3.3). The AW extension then returns the result to the browser (step 5, send(page)) and the browser displays the page and executes the embedded JavaScript code, which might in turn retrieve and execute additional code fragments from Web libraries. If the page displays correctly and therefore the user does not report any failure, the interaction continues in normal mode with the AW extension effectively acting as a transparent proxy.

If the user perceives a failure, then he or she may report it through the previously activated failure-reporting mechanism (step 6 notify(failure)). The failure signal switches the AW extension to workaround mode. The AW extension first extracts the JavaScript code from the page and selects an applicable rewriting rule for that code (step 7, select(rule)). The AW extension then changes the code according to the rewriting rule and forwards the page with the new code to the automated validator (step 8, evaluate(rule)). If the validator confirms that the new code indeed behaves differently from the failing code (see Section 3.4) then the AW extension applies the same rewriting rule to the original page (step 9, apply(rule)) and displays the new page to the user (step 10, send(page)). If either the validator discards the page as equivalent to the failed one (step 8, evaluate(rule)) or the user reports a new failure (step 6), the AW extension reiterates the process (steps 7–10) until a valid workaround is found or until no more applicable rewriting rule are available, or until the user gives up reporting problems.

In the following sections we detail the syntax and the semantics of the rewriting rules used by the AW extension, the process by which the AW extension selects an prioritizes the rewriting rules, and the process by which the AW extension detects and immediately rejects useless rules through its automatic validator.

3.2. Code Rewriting Rules

We capture the intrinsic redundancy of software systems by means of rewriting rules. A rewriting rule expresses the supposed equivalence between two code fragments. In other words, a rule asserts that a certain code fragment *should* have the same observable behavior as another code fragment, and therefore that the two fragments may be used interchangeably at runtime to meet the same goals. Since a failure is an unexpected deviation from the specified behavior, the idea is to recover from the failure by replacing the failing code with some other code that is supposed to be semantically equivalent, but that in reality would avoid the failure.

The notion of a rewriting rule is quite general. However, in practice we must define rules that are specific to the target language. Thus in this section we define a specific syntax and semantics for rules that apply to JavaScript code using the λJS formalism [Guha et al. 2010]. In particular, we formalize JavaScript, the patterns that occur in the code rewriting rules, and the code rewriting rules themselves.

JavaScript. We first define a grammar for the abstract syntax of the essential elements of JavaScript borrowing from Guha et al. [Guha et al. 2010].

$$e ::= x \mid \text{let } x = e \text{ in } e \mid s \mid e.f \mid e.f = e \mid e.m(e^*) \mid \text{new } C(e^*) \tag{1}$$

An expression e can be a local variable x (including *this*), a let expression that introduces new local variables x in an expression e, a string literal s, a field access $e.f$, a field assignment $e.f = e$, a method call with zero or more parameters $e.m(e^*)$ or an instantiation of a new object through a call to a constructor with zero or more parameters $new\ C(e^*)$.

We then define a *rewriting context* E for Grammar 1 inductively. The rewriting context is an "expression with a hole". The hole may then be filled with an expression to obtain another expression. In other words, the hole represents an element to be rewritten, and therefore is the basis for the code rewriting rules.

$$\begin{aligned} E ::= [\bullet] \mid \text{let } x = E \text{ in } e \mid E.f \mid x.E \mid x.E(e^*) \mid e.f = E \mid \\ E.m(e^*) \mid e.m(e^*, E, e^*) \mid \text{new } C(e^*, E, e^*) \end{aligned} \tag{2}$$

Grammar 2 defines the rewriting context E. For each expression there can be only one hole [•]. The hole in the expression can be located in an instantiation of a new local variable ($let\ x = E\ in\ e$), an object that accesses a field ($E.f$), a field of an object ($x.E$), a method called on an object ($x.E(e^*)$), a value stored in a field ($e.f = E$), an object on which a method is called ($E.m(e^*)$), a parameter of a method ($e.m(e^*, E, e^*)$), or a parameter of a constructor (new $C(e^*, E, e^*)$).

We will use rewriting contexts to define which parts of a program may be changed by a code rewriting rule. Let us consider for example a source program fragment P = *oldObject.field* that accesses the field *field* of an object *oldObject*, and a target program P' = *newObject.field*. The transformation between P into P' effectively replaces *oldObject* with *newObject*. This replacement corresponds to the rewriting context [•]*.field*, which is consistent with Grammar 2 and therefore acceptable.

Pattern. Code rewriting rules are pairs of patterns ⟨*pattern*,*pattern′*⟩, where *pattern* is an expression that matches a code fragment of the Web application and *pattern′* is the replacement code. In code rewriting rules, in addition to normal JavaScript expressions, a pattern p may contain meta-variables, which we refer to as $mv(p) = \{\$X_1, \dots, \$X_k\}$. Thus a pattern is a subset of JavaScript expressions augmented with meta-variables and can be defined as:

$$p ::= x \mid p.m(p^*) \mid s \mid \textsf{new}\ C(p^*) \mid \$X \tag{3}$$

A pattern p can be a local variable x, a method call with zero or more parameters $p.m(p^*)$, a string literal s, a parameter in a constructor ($\textsf{new}\ C(p^*)$) or a meta-variable $\$X$.

Code Rewriting Rule. We can now define a code rewriting rule with the grammar:

$$r ::= q\ \textsf{ANY}\ (p \mapsto p') \mid q\ \textsf{ALL}\ (p \mapsto p')^+ \tag{4}$$

The qualifier q indicates the kind of redundancy, and can be one of the keywords null (for functionally null), invariant, or alternative. The keywords ANY and ALL define the scope of the rule within the program code. If the scope is ANY, the transformation is applicable to any one of the occurrences of the substitution pattern. Therefore, an ANY rule can produce several new programs, each differing from the original in exactly one specific application of the transformation. If the scope is ALL, the rule applies to all the occurrences of the substitution pattern, and therefore may produce only one new program. Rules apply to either exactly one occurrence of the left hand side of a code rewriting rule (rules of type ANY) or all occurrences of the left hand side (rules of type ALL), but not a subset of occurrences. Rules of type ALL can apply one or more substitutions (q ALL $(p \mapsto p')^+$), while rules of type ANY can apply only one substitution (q ANY $(p \mapsto p')$). This is because we cannot infer the semantic relation between different code fragments. We can for example modify all occurrences of put and pop in a method of a stack, but to modify a specific pair of put and pop we would need context information that indicates which pop is related to which push in the code.

The mapping $(p \mapsto p')$ defines a transformation from a pattern p, called substitution pattern, to a pattern p', called replacement pattern. The meta-variable in the replacement pattern refers to the corresponding meta-variables in the substitution pattern, thus we require for all rules $p \mapsto p'$ that $mv(p') \subseteq mv(p)$.

Application of Code Rewriting Rules. In order to define the semantics of a code rewriting rule, consider a general ANY rule

$$r = q\ \textsf{ANY}\ (p \mapsto p')$$

of kind q, scope ANY, and transformation patterns $(p \mapsto p')$, and without loss of generality let $mv(p) = mv(p') = \{\$X_1, \ldots, \$X_k\}$. Given a program P, rule r applies to any fragment $\overline{P}$ within P if, for each meta-variable X_i in p there exists a rewriting context E_i in $\overline{P}$ such that $\$X_i$ matches the hole in E_i, and the rest of $\overline{P}$ matches the rest of pattern p completely, token by token. We indicate this condition as follows:

$$P = P_L \cdot \overline{P} \cdot P_R \quad \text{where} \quad \overline{P} = p[\$X_1 = e_1, \ldots, \$X_n = e_n]$$

where $p[\$X_1 = e_1, \ldots, \$X_n = e_n]$ indicates pattern p in which a concrete expression e_i replaces each meta-variable $\$X_i$.

The rewritten program code is then

$$P' = P_L \cdot \overline{P}' \cdot P_R \quad \text{where} \quad \overline{P}' = p'[\$X_1 = e_1, \ldots, \$X_n = e_n]$$

that is, program fragment $\overline{P}$ in P is rewritten as $\overline{P}'$ corresponding to the replacement pattern p' in which each meta-variables $\$X_i$ is replaced with the concrete code e_i bound to $\$X_i$ in the substitution pattern.

The semantics of a code rewriting rule with scope ANY extends naturally to rules with scope ALL. Intuitively, the substitution applies to all disjoint program fragments

```
(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();
```

Fig. 8. Examples of code rewriting rules.

$\overline{P}$ where it would be applicable in an identical ANY rule. Operationally, the transformation of an ALL rule can be obtained by applying the corresponding ANY transformation to the *leftmost* applicable fragment $\overline{P}$, and then recursively to the rest of program to the right of $\overline{P}$. A bit more formally, let r_{ANY} and r_{ALL} be two identical rules with scope ANY and ALL, respectively, and let $r(P = P_L \cdot \overline{P} \cdot P_R) = P_L \cdot \overline{P}' \cdot P_R$ indicate the application of the r transformation to the leftmost fragment $\overline{P}$ in P, then the ALL transformation can be defined recursively as $r_{\text{ALL}}(P = P_L \cdot \overline{P} \cdot P_R) := P_L \cdot \overline{P}' \cdot r_{\text{ALL}}(P_R)$.

Notice that in any case rewriting rules are applied statically, independent of any dynamic context in which the code executes. However, the replacement code in a rule may contain a conditional expression, which would *effectively* allow a rule to apply a substitution dynamically, depending on the runtime context.

Figure 8 shows three examples of code rewriting rules. Rule 1 applied to code fragment SEQ1a of Figure 1 generates code fragment SEQ1b; rule 2 applied to code fragment SEQ2a of Figure 3 generates fragment SEQ2b, and rule 3 applied to code fragment SEQ3a of Figure 5 generates fragment SEQ3b. Rule 1 changes one invocation at a time of the setCenter function so that it would be delayed to avoid potential timing issues. Rule 2 adds a sequence of deleteVertex and insertVertex right before every invocation of the enableDrawing function so as to avoid potential problems with polylines. Rule 3 replaces all sequences of invocations to show and hide with addOverlay and removeOverlay to avoid potential problems with the visibility of overlays.

3.3. Selecting Code Rewriting Rules

Even only a few code rewriting rules apply to a simple program can produce many alternative variants of that program, and the simplistic application of all the available rules would typically generate too many candidate workarounds, which would greatly reduce performance and usability.

The Automatic Workaround browser extension uses a tracer and a priority scheme to select the rules that are more likely to lead to workarounds. The tracer identifies which parts of the application code were indeed executed during the failing execution and therefore reduces the number and scope of applicable code rewriting rules. The tracer alleviates but typically does not completely eliminate the problem of the many candidate workarounds produced with code rewriting rules.

```
map = new GMap2(document.getElementById(”map”));
map.setCenter(new GLatLng(46.003,8.953),10);
map.disableDragging();
marker = new GMarker(new GLatLng(45.81,9.08),{draggable:true});
GEvent.addListener(markerDrag, ”click”, function(){
  marker.openInfoWindowHtml(’This marker can be dragged’),500});
map.addOverlay(marker);
```

Fig. 9. A fragment of code that fails due to a fault in the first statement.

We complement the tracer with a priority scheme that selects the candidates for workarounds that are most likely to be effective. We use a priority scheme based on the history of usage of workarounds, having observed that workarounds that were successful in the past are often successful also in the future. We define the historical priority value of a particular rule as a pair of values $\langle \textit{success-rate}, \textit{successes} \rangle$, where *successes* is the number of successful applications of the rule (i.e., the number of times a rule has produced a successful workaround), and *success-rate* is the ratio between the number of successful applications of the rule (*successes*) and the total number of times the rule was used:

$$\textit{success-rate} = \frac{\text{number of successful applications}}{\text{total number of uses}}$$

A rule with higher *success-rate* is given priority over one with lower *success-rate*. When two rules have the same *success-rate*, priority is given to the rule with higher absolute *successes*. In other words, priority order is defined first on success rate and then on absolute successes, so $p_1 = (r_1, s_1)$ is greater than priority $p_2 = (r_2, s_2)$ if $r_1 > r_2$ or if $r_1 = r_2$ and $s_1 > s_2$. When no historical information is available, the priority of a rule is assigned an initial *success-rate* $= 1$ and *successes* $= 1$. The priority scores are then updated at each application of the rule.

When two or more code rewriting rules 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.

Let us assume for example that a failure occurs within the code fragment of Figure 9, and that our mechanism automatically applies the code rewriting rules shown in Figure 10 to the first statement of Figure 9.

If the priorities associated to the rules are the ones shown in Figure 10, the AW extension will try to apply rule 2 first, then rule 1, and finally rule 3, since both rules 1 and 2 have higher success rate than rule 3, and rule 2 has a number of absolute successes higher than rule 1. If rule 1 succeeds in generating a valid workaround, the system modifies the priorities as shown in Figure 11. The readers should notice that the failure of rule 2 and the success of rule 1 change both the frequency of successes for both rules and the absolute amount of successes for rule 1.

3.4. Automatic Validator

A priority scheme may increase the probability of selecting code rewriting rules that would lead to valid workarounds, but cannot guarantee an immediate success. The AW extension may still generate several candidate workarounds before finding a valid one, especially in the early phases when little or no information is available on the

	Code Rewriting Rules	*Priority*
(1)	alternative ALL: $X = new GMap2($Y); ↦ $X = new GMap2($Y); $X.zoomIn(); $X.zoomOut();	⟨0.6, 6⟩
(2)	invariant ALL: $X = new GMap2($Y); ↦ $X = new GMap2($Y); $X.enableDragging();	⟨0.6, 9⟩
(3)	alternative ALL: $X = new GMap2($Y); ↦ $X = new GMap2($Y); $X.enableInfoWindow();	⟨0.2, 4⟩

Fig. 10. Some code rewriting rule for Google Maps with their priority.

	Old priority		*New priority*
Rule 1	⟨0.6,6⟩	→	⟨0.67,7⟩
Rule 2	⟨0.6,9⟩	→	⟨0.56,9⟩
Rule 3	⟨0.2,4⟩	→	⟨0.2,4⟩

Fig. 11. Updated priorities after the unsuccessful application of rule (2) and the successful application of rule (1).

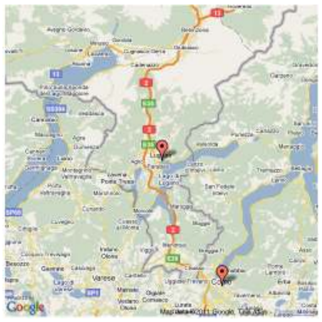
effectiveness of each code rewriting rule. Such numerous repeated attempts to find a workaround, each submitted to the user for approval, are likely to frustrate the user and would therefore limit the usability of the technique.

Ideally, we would like to have an automatic validator capable of discarding all invalid workarounds and of selecting the valid ones. Such an ideal validator would allow us to relieve the user from having to review candidate workarounds, and therefore would completely automate the search for workarounds. It is conceivable that such a validator could be built, perhaps based on a complete specification of the Web application. However, such specifications are very rarely available for deployed applications.

Here we propose to use a partial validator based on a comparison between the failing pages and the pages generated by each candidate workaround. In practice, this validator automatically identifies and discards those candidate workarounds that produce an identical Web page as the failing one.

More specifically, when the user reports a failure, the validator saves the HTML and JavaScript code of the original (failing) page. The validator then compares the saved failing page with the new page generated with the variant code obtained through the rewriting process. If the two pages show no structural differences, that is, if their DOM representations are identical, the validator rejects the new page. Otherwise the validator accepts the page and displays it to the user, in which case the user can either accept the page or reject it by signaling another failure. When the user rejects a page, the AW extension adds that page to the current set of failing pages that is used for further comparisons by the automatic validator. The set of failing pages is cleared when the user decides to proceed with a new page. Although comparing DOMs may lead to spurious results, for example when including variable elements like timestamps that may change when reloading the page, these cases seem quite rare and we have not experienced either false negatives or false positives in our experiments.

Figure 12 shows an example of a failing Web page together with three internal attempts to find a workaround. Only one of these candidates is presented to the user and

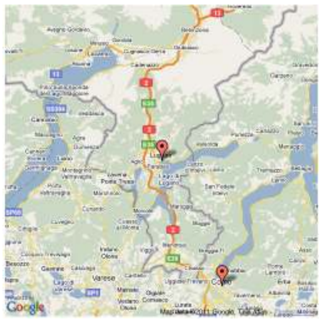


Original JavaScript code

```
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">
```

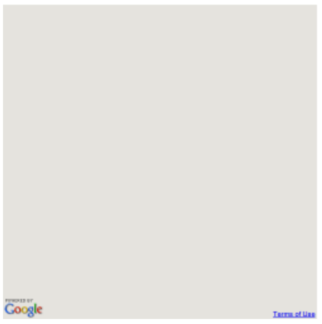


First attempt: invariant operations (line 4)

```
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 with invariant sequence, discarded by the validator

```
<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">
```

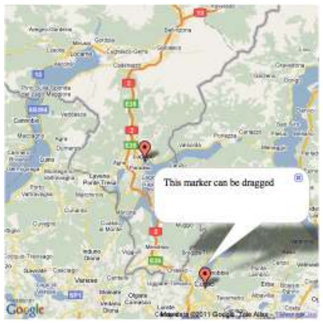


Second attempt: delay creation of the map (line 1)

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

DOM with delay, discarded by the 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 (lines 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 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>
```

Fig. 12. The validator automatically discards the first attempt. Thus, the user evaluates only one failing page (beside the original faulty one) before having the page fixed.

is not valid. The screenshot on top of the figure shows a Google Map with markers that should be draggable in a non draggable maps, but are not.[6] Any click on the draggable markers is ignored, and the attached information window is not displayed. This is the original faulty page. The second screenshot shows the results of a first attempt to fix the map by inserting the invariant sequence map.enableDragging(); map.disableDragging(). This attempt does not change the behavior of the Web page, as evidenced by the same DOM of the two pages that is recognized and automatically rejected by the validator. The third screenshot shows the result of a new attempt to fix the map by delaying the execution of the creation of the map (map = new GMap2(. . .)). This attempt does not produce a correct page either, but the new page fails in a different way, with a different DOM than the one of the failing pages. Consequently, the validator accepts the new page, which is displayed to and rejected by the user. The last screenshot shows the results of the final attempt that amounts to delaying the execution of openInfoWindowHtml on the faulty markers, and that produces a valid workaround. The DOM of the page is different from any of the previous rejected attempts, the validator accepts the page and the user approves the workaround. In this example, the validator identifies one of the two invalid attempts. The results reported in Section 5 indicate that our validator can be very effective in rejecting many invalid pages thus increasing the usability of the approach.

Notice that any valid workaround would change the observable behavior of the page, and consequently its structure. Thus the validator acts conservatively, by accepting any change as a potentially valid workaround. The pages rejected by the validator are considered as pages rejected by the user for the purpose of computing the priorities associated with rules.

4. PROTOTYPE IMPLEMENTATION

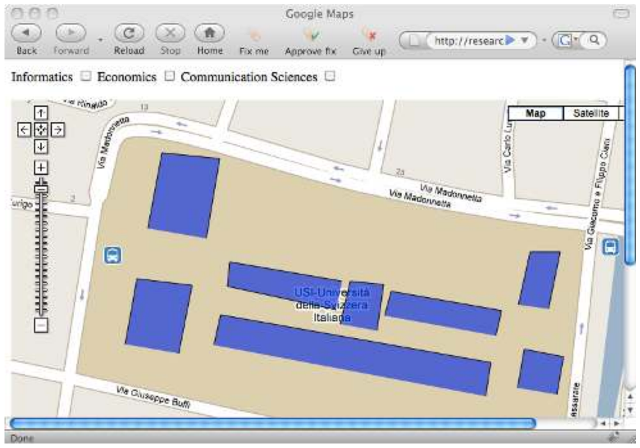
We implemented the Automatic Workaround technique in RAW (Runtime Automatic Workarounds), an extension of the Google Chrome and Firefox browsers [Carzaniga et al. 2010b]. RAW extends the browser interface by adding three control buttons: *Fix me*, *Approve fix* and *Give up*, as shown in the screenshots in Figure 13. The Firefox extension adds the three buttons to the toolbar. The Chrome extension adds a single *Fix me* button to the toolbar, and opens a pop-up window containing the other control buttons when the user presses this button.

The *Fix me* button activates the automatic generation of workarounds. The button is active only when the currently loaded page uses one or more Web libraries supported by RAW, that is, libraries for which RAW has a set of rewriting rules. A user experiencing a failure with the displayed page may press the *Fix me* button to report the problem and request a workaround at runtime.

The *Approve fix* and the *Give up* buttons become active only when RAW operates in workaround mode in response to a failure (signaled by the user with the *Fix me* button). The user may press the *Approve fix* button to confirm that a workaround was successful, or the user may reject a workaround by pressing the *Fix me* button once again, or in the end the user may signal a failure and also terminate the search for workarounds with the *Give up* button. Notice that the *Approve fix* and *Give up* buttons are not essential from the user's perspective, since the user may implicitly accept or reject a workaround by continuing to another application page or to some other activities. In our prototype those two buttons also provide useful information for the priority scheme.

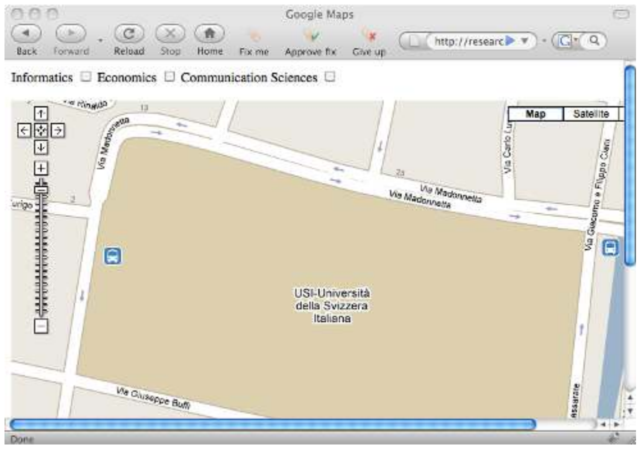
We illustrate the use of RAW through an example that refers to a known (and now fixed) problem of the Google Maps API, reported as Issue 1264 in the Google Maps

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



Faulty page

The page shows all the buildings with blue polygons even if the checkboxes of the faculties on top of the page are not selected



Page automatically fixed by RAW

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

Fig. 13. Issue 1264 in Google Maps automatically fixed by RAW.

bug-reporting system.[7] Figure 13 presents two screenshots of a simple Web page that illustrates this Google Maps issue. The page shows the map of the campus of the University of Lugano, and offers a set of checkboxes to display the various buildings as polygonal overlays. The buildings of each Faculty should appear on the map only when the corresponding checkbox is checked. Initially, the page should not display the polygons, which should become visible only after selecting the checkboxes on top of the map. When visible, the polygons should scale according to the zoom level. As illustrated by the screenshot on the top of Figure 13, the initial page does not display 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 may only attempt to reload the page several times. Once noticed that the problem is deterministic users can report the problem to the application developers and hope for a fix sometime in the future. With RAW, users experiencing this problem can request an immediate workaround by pressing the *Fix me* button in the toolbar (screenshot on the top of Figure 13). The *Fix me* button activates RAW, which extracts the JavaScript code of

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

the current page, applies one of the code rewriting rules, and reloads the page with the new JavaScript code. If not satisfied by the reloaded page, the user may request another workaround by pressing the *Fix me* button once again. The screenshot on the bottom of Figure 13 shows the correct behavior of the application as 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 also press the *Give up* button to stop searching for valid workarounds. When there are no code rewriting rules left to apply, RAW gives up and displays a warning message.

We implemented the RAW prototype under the assumption that users are not malicious, and would therefore act in good faith when reporting failures or when approving workarounds. Malicious users cannot use RAW to attack the applications, but 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 provide a fix. To handle recurring problems more efficiently, RAW keeps track of reported failures and successful workarounds. In particular, RAW records the URL, the workaround, and the specific execution conditions, including the type and versions of the browser and of the operating system.

When a user loads a page for which RAW has known workarounds, the *Fix me* button signals the presence of workarounds. Thus, if the Web page does not display correctly, the user would know immediately that other users may have experienced the some problem, and that they obtained valid workarounds for those problems. Although RAW could proactively apply known workarounds to notoriously faulty pages, we choose to implement a conservative and less intrusive behavior and still wait for a failure report from the user.

RAW is composed of a client side and a server side. The client-side component is the browser extension that implements the user interface described above, and is written primarily in JavaScript. The server-side component implements the main functionality of the Automatic Workarounds technique in Python, and runs on a centralized server.

The browser extension checks the header of the Web page requested by the user and looks for references to supported JavaScript libraries. The extension activates the *Fix me* button when it finds one or more such library references. When the user presses the *Fix me* button, the browser extension extracts the JavaScript code from the Web page together with information about the used Web API, including the name and the version of the API. Then, it sends this information to the centralized server that manages the database of code rewriting rules.

The server component selects a code rewriting rule, depending on the priority scheme it applies the rewriting to the JavaScript code, and sends the new code back to the browser extension. The browser extension then substitutes the JavaScript code in the DOM of the current page with the new code, and executes the new code without retrieving again the content from the remote server.

The client and server components communicate through `XMLHttpRequest` objects. Usually the JavaScript code of a Web page must obey the *same origin policy* that forces the client to send and receive data only to and from the origin server of the current page. However, browser extensions do not have this limitation, and RAW can therefore send requests to the external server that hosts the server-side components.

The automatic validator that automatically discards Web pages that look like the ones that the user reported as failing, is hosted in the client-side extension. Once the new JavaScript code is executed, the browser extension extracts the DOM of the new Web page and compares it to the previous ones. If the DOMs are the same, the validator discards the page and asks for a new one by sending a new request to the server.

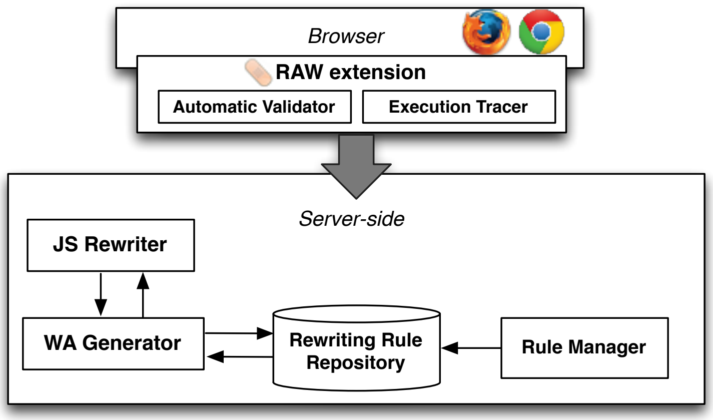


Fig. 14. Overall architecture of RAW.

The execution tracer traces the executed JavaScript code, and is also hosted in the client side. Upon a *load* event, the tracer extracts the JavaScript code of the current Web page and instruments it to trace the execution at the statement level. At runtime, the tracer keeps track of which lines are executed, and sends that information to the server in case of failure.

We now briefly describe the components shown in Figure 14, the *Rewriting Rule Repository* that contains the code rewriting rules, the *JS Rewriter* that applies the changes to the JavaScript code, the *WA Generator* that is responsible of selecting which rule to apply, the *Rule Manager* that is an interface to manage code rewriting rules, and the *Automatic Validator* that discards non valid attempts.

Rewriting Rule Repository. The Rewriting Rule Repository contains the code rewriting rules as described in Section 3.2. Each rule is stored in the repository with a unique id, and is associated with a specific API, and with its priority value.

Rule Manager. The Rule Manager is a Python interface that allows developers to update the information stored in the Rewriting Rule Repository. Developers initialize the repository with a set of rules generated from the specifications of the Web library or service. Developers can later update the existing rules and add new ones, for example by coding workarounds found and reported by other developers or users outside the RAW system. Developers can also use the Rule Manager to extract information about successful workarounds and failure conditions to identify and implement permanent fixes for their applications.

JS Rewriter. The JS Rewriter is the Python module that applies code rewriting rules to the JavaScript code. When the user reports a failure in the current Web page, the RAW browser extension extracts the JavaScript code, the information about the API used in the current Web page and the list of executed JavaScript statements, and forwards all the data to the JS Rewriter. The JS Rewriter forwards the information about the API to the WA Generator that returns a code rewriting rule selected according to the priority scheme described in Section 3.3. The JS Rewriter implements the substitutions specified by the rule using the sed stream text processor.[8]

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

WA Generator. The WA Generator is responsible for selecting the rules that are more likely to generate valid workarounds. To rank rules, the WA Generator uses the history-based priority value of the rule (see Section 3.3). When the JS Rewriter requests a rule, the WA Generator selects the rules associated with the API used, and checks if there are known workarounds for the current URL. If any rule is known to solve the problem of the Web page that RAW is trying to fix, then this rule is selected and applied first. If no workaround is known, the WA Generator selects the next rule with the highest priority.

Automatic Validator. The Automatic Validator discards clearly invalid workaround candidates. Every time the user reports a failure, the Automatic Validator extracts the DOM of the Web page, computes its hash value, and stores it. Every time RAW applies a new rule to the original JavaScript code and re-executes the new JavaScript code, the Automatic Validator extracts the DOM of the newly generated page and compares the hash value of the current DOM with all the hash values of the DOMs of the previously discarded pages. RAW automatically discards all workaround candidates that produce pages with a DOM equal to any of the failing or otherwise discarded pages. Then, in case of failure, RAW invokes the JS Rewriter once again.

5. EVALUATION

The main research hypothesis of this paper is that *software systems, in particular in the form of reusable components such as Web libraries and services, are intrinsically redundant, in the sense that the same functionality can be achieved in different ways, and this redundancy can be exploited to avoid failures at runtime.* Various studies show that software systems contain several semantically equivalent code fragments [Jiang and Su 2009; Gabel et al. 2008], and our studies confirm the presence of redundancy in many software systems [Carzaniga et al. 2010a; Carzaniga et al. 2008]. In this section we demonstrate experimentally that it is possible to exploit the intrinsic redundancy of software, and in particular of Web libraries, to automatically generate effective workarounds for Web applications. In particular, we validate our hypothesis by first studying the effectiveness of workarounds in dealing with runtime failures, and then by studying *automatic* workarounds, and specifically those generated by our technique in the context of Web applications. Thus our evaluation is articulated around the following three research questions:

Q1. Can workarounds cope effectively with failures in Web applications?
Q2. Can workarounds be generated automatically?
Q3. Can the Automatic Workarounds technique generate effective workarounds to avoid failures in Web applications?

The first question (Q1) explores the effectiveness of workarounds to handle failures in general. This includes ad-hoc workarounds developed manually by users or developers. We ask whether such workarounds *exist* and whether they can be used for Web applications. In Section 5.1 we report the results of our survey of fault repositories and other on-line forums dedicated to well known Web APIs to document the use of workarounds to fix failures in popular Web applications. The second question (Q2) asks whether it is possible to generate workarounds *automatically* by exploiting the intrinsic redundancy of software systems, and, in particular, of Web libraries. In Section 5.2 we identify workarounds that can be automatically generated with our approach. The third question (Q3) evaluates the ability of the technique presented in Section 3 to generate and deploy valid workarounds. In Section 5.3 we investigate the amount of failures that can be automatically avoided with our technique.

5.1. Q1: Can workarounds cope effectively with failures in Web applications?

To answer question Q1, we analyzed the fault reports of popular Web applications and we identified the amount of failures addressed with a workaround.

We referred to the official bug-tracking systems when available, as in the case of Google Maps, YouTube, and JQuery, we relied on on-line discussion forums otherwise, as in the case of Yahoo! Maps, Microsoft Maps, Flickr and Picasa.

Of all the available official and unofficial reports, we selected those that we thought might reveal useful information on workarounds. We did that with simple textual searches. We then examined all the failure reports that even superficially indicated the possibility of a workaround. We did that to exclude irrelevant reports and to precisely identify failures and workarounds for the relevant reports.

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 number, and are often effective in avoiding or mitigating the effects of faults in Web applications. To quantify the effectiveness of workarounds in mitigating the effects of faults, we limited our analysis to the issue trackers of Google Maps[9] and the YouTube chromeless player.[10] We selected these issue trackers because they contain a number of issue reports that is large enough to contain useful information, and yet small enough to be manually inspected in a reasonable amount of time. Table I summarizes the results of our survey for Google Maps and YouTube.

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

API	*reported faults*	*analyzed faults*	*proper workarounds*
Google Maps	411	63	43 (68%)
YouTube	21	21	9 (42%)

We studied the Google Maps API issue tracker at the beginning of 2010, 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 searching for the keyword "workaround" in the bug descriptions. We obtained 63 entries and then focused on those entries, ignoring other possible workarounds not marked explicitly as such. Upon further examination, we found that 43 of them were proper workarounds and not spurious uses of the keyword. For all the 43 Google Maps API issues we then verified that the reported workarounds were valid by reproducing the failures and by checking that the workaround could indeed avoid the failure. In one case we could not reproduce the problem, since it was attributable to transient environment conditions.[11]

The 43 valid workarounds we found for Google Maps amount to about 68% of the analyzed faults and 10% of the total reported. The actual overall prevalence of workarounds is likely to be somewhere in between, since on the one hand the selection of the "workaround" keyword makes the analyzed set highly relevant, but on the other hand it might also exclude valid workarounds that were not described as such or simply not known. We discuss some of these cases in our experimental evaluation.

In the case of the YouTube chromeless player we considered the 21 issues reported in the issue tracker at the time of this investigation (beginning of 2010). Given the

[9]`https://code.google.com/p/gmaps-api-issues`
[10]`https://code.google.com/p/gdata-issues/`
[11]`http://code.google.com/p/gmaps-api-issues/issues/detail?id=40`

modest size of the repository, we analyzed all the issues without resorting to any pre-filtering. 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 the investigation, because YouTube provides access only to the current version of the API. Consequently, we could not verify the validity of the proposed workarounds, and we had to rely on the comments of other users who had confirmed their effectiveness.

The data collected with the analysis of the Google Maps and YouTube API issue trackers indicate that it is often possible to overcome Web APIs issues by means of appropriate workarounds.

5.2. Q2: Can workarounds be generated automatically?

Having observed that a significant number of failures in Web applications can be avoided with workarounds, we asked ourselves whether some of the workarounds could be found *automatically*, perhaps exploiting the intrinsic redundancy of Web applications and libraries. To answer question Q2 we further analyzed the 52 workarounds selected for Google Maps and YouTube (43 and 9, respectively). In particular, we intended to exclude workarounds that are ad-hoc and difficult to generalize, or that were developed through a complex manual process or a particularly clever intuition. In other words, we wanted to exclude workarounds that could not conceivably be obtained through an automated process at runtime using code rewriting rules.

To illustrate the notion of an ad-hoc workaround, consider Issue 40 from the bug-tracking system of Google Maps:[11]

> Some previously working KML files are now reporting errors when entered 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 Google Maps by posting them to an accessible Web server. Due to some problems with the Google servers, Web applications using the Google Maps API could not access the KML files, thereby causing a 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 workaround 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 cannot 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 61 from the same Google Maps bug tracker:[12]

> Many times the map comes up grey ... a slight setTimeout before the set-Center ... 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

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

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 are not visualized correctly when using dynamic loading. External JavaScript files can be loaded either statically by adding their reference in the HEAD section of a Web page, or dynamically on demand by the application. The dynamic loading was causing problems with the maps displayed during the loading phase (the maps would come up grey). This report indicates two workarounds that exploit redundancy: the first is to add a setTimeout, the second is to add a zoom-in-zoom-out sequence, which are good examples of null and invariant operations, respectively, as defined in Section 2.1 and 3.2.

Table II. Amount of reusable workarounds

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

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

5.3. Q3: Can the Automatic Workarounds technique generate effective workarounds to avoid failures in Web applications?

To answer question Q3 we performed various experiments with the Google Maps, YouTube, and JQuery APIs. These experiments aim to evaluate the ability of the technique to find valid workarounds, both for faults for which a workaround had already been reported, and for faults for which no workaround was publicly known.

We conducted experiments to investigate (1) the reusability of code rewriting rules, that is, the ability of rules derived from valid workarounds to solve new problems, thus simulating the expected usage pattern of developers populating the repository of code rewriting rules incrementally as more and more workarounds are found; (2) the effectiveness of code rewriting rules derived directly from the documented redundancy of Web libraries, thus validating the efficacy of the rules themselves; and (3) the effectiveness of the priority and validation schemes, thus evaluating the usability of the automatic workaround approach.

5.3.1. Reusability of Code Rewriting Rules. This first experiment is intended to assess whether workarounds are reusable in different contexts, in other words whether some workarounds that are known to solve some issues can also serve as workarounds for other issues.

To assess the reusability of workarounds, we started with an empty repository of rules to which we incrementally added rules corresponding to the workarounds reported in the issue tracker.

We considered the 14 issues with known workarounds identified in the study described in Section 5.2 plus 24 additional issues for which no workaround was known at the time. We selected the 24 additional issues without known workarounds among the issues on the bug tracker that had already a sample Web page reproducing the failure, to minimize the efforts of reproducing the issues.

We considered each issue in ascending order of index number in the issue-tracking system, assuming that this order corresponds to the chronological order of appearance. We also repeated the experiment with random ordering without noticing any substantial difference. We then tried to solve each issue automatically using the rules available in the repository at that time. Whenever we could not find a workaround at all we recorded the workaround as *none*. Otherwise we recorded the workarounds as *automatic* if we could solve the issue with available rules, or as *manual* whenever the issue had a known workaround that we could not find automatically using the rules available in the repository at that time. For each *manual* workaround, we immediately added a corresponding rule in the repository.

Table III. Reusability of Code Rewriting Rules for Google Maps

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	✓			–
TOTAL	19	6	13	-

Table III shows the results of this experiment. The first and the last columns show the issue number and a reference to the corresponding code rewriting rule listed in Table IV, respectively. The central columns indicates the classification of the corresponding workaround (none, manual, automatic). We highlight two very positive results of this experiment. First, one third of the issues (13 out of 38) could have been solved by workarounds generated automatically on the basis of previously identified workarounds. Second, in half of the cases in which a workaround was found automatically (7 out of 13) the workaround was not previously known (not publicly at least). These issues for which we found *new* workarounds are marked in grey in Table III.

The experiment also shows that several workarounds can be effective multiple times and in different contexts. We can see for instance that rules G2 and G4 in Table IV can each solve 8 and 7 issues, respectively. Therefore, if Google developers had coded the workarounds that they found incrementally in the form of code rewriting rules, they would have solved automatically several subsequent different issues, including many that were not solved manually.

5.3.2. Effectiveness of Users as Failure Detectors. We considered our group Web site, which uses the JQuery library, and we substituted the current correct version of the JQuery library with an obsolete faulty version. In particular, we included a version affected by some of the faults that we use in our evaluation in Section 5. We then asked potential users to perform a set of eleven tasks as part of a study about the usability of the Web site. Five tasks could be executed correctly, while six could not, due to faults in the JQuery library. We performed the study with a set of twenty users of different gender, age, education and occupation, excluding people with a specific computer science background to avoid bias.

All twenty users reported a problem for three of the faulty tasks. Eighteen out of twenty reported a problem for one other task, and only two did not perceive the behavior as faulty. The remaining two faulty tasks were perceived as such by eleven and thirteen users, respectively. In one case, the users who did not report a failure obtained the expected behavior with a workaround (using the browser back button instead of the faulty return feature offered by the application). In the other case, the users who did not report a failure simply did not realize that the results displayed on the page were wrong due to a lack of familiarity with the academic organization described by that page.

We repeated the experiment with seven master students in computer science, and we confirmed both the results and our evaluation. The computer science students reported the problems with the same frequency, except for the two border-line cases: most computer science students used the obvious workaround of the back button, and only one did not report the fault that requires some familiarity with the academic organization. Although limited and of qualitative nature, these results confirm our hypothesis that users can perceive failing behaviors as such.[13]

5.3.3. Effectiveness of Code Rewriting Rules. This second set of experiments conducted with Google Maps, YouTube, and JQuery aims to assess the effectiveness of code rewriting rules derived from API specifications.

We selected these libraries because of the possibility of replicating failures and workarounds. We could easily replicate workarounds with JQuery, since it is an open source library and its entire version history is publicly available. We could also replicate workarounds with Google Maps, since Google used to export the complete version history of their Web APIs. Unfortunately, during our evaluation process, Google removed many versions from its publicly available history, and we were unable to complete all the experiments we had planned. In particular, we could not repeat all our initial experiments with the automatic validator described in Section 3.4. In the case of YouTube

[13]The details of the user study are available at **http://star.inf.usi.ch/star/software/awa/awa.htm**.

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

	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();

we could only reproduce the failures caused by open issues, since YouTube does not expose older versions for its API.

We first populated the repository of code rewriting rules with three sets of rules, one for each of the three selected libraries. We derived the rules by referring only to the API specifications. We wrote these rules following the classification of rules introduced in Section 2.2, expressing the semantic equivalences that we could evince by reading the documentation of the three libraries. 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 IV with labels G1–G14 and Y1–Y6, respectively. A subset of the rules for JQuery is listed in Table V.

To estimate the cost of manually deriving rewriting rules from APIs, we conducted an additional study using JQuery. Independently of the 68 rules already found, we examined 131 interface elements out of 307 of a much more recent release of the library, and we derived 222 rewriting rules in eight working hours, with an average of four minutes per element. Thus we expect that a domain expert can identify rewriting rules rather quickly. The 222 rewriting rules refer to 86 interface elements (that is, we found at least one rewriting rule for 86 out of 131 examined elements) and therefore we observe that redundancy is widely present in the library.

Google Maps. We considered 38 issues from the issue tracker of Google Maps, including the 24 issues considered in the reusability evaluation (see Section 5.3.1) plus 14 other issues with a known workaround (see Section 5.2). We reproduced each of the 38 issues using RAW, following the chronological order given by their issue-tracker number, and initializing RAW with the same priority $\langle 1, 1\rangle$ for all the 39 rules.

In an initial run of these experiments, we used a prototype implementation of RAW that did not include the automatic validator. We later repeated the experiments with the latest prototype implementation of RAW that includes the automatic validator. Unfortunately, some versions of the Google Maps API became unavailable before we could conduct the second set of experiments, and consequently we could only reproduce 24 of the original 38 issues in the final runs of the experiment.

Table VI reports the results of the experiment. 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: *none* indicates cases in which RAW could not generate any valid workaround, *known* indicates that RAW automatically generated a workaround that was already known, and *new* indicates that RAW generated a new valid 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. Moreover, it was interesting to find out that for issues 338 and 1264, for which a workaround was already known, RAW could find additional previously unknown workarounds.

The fact that RAW could not only generate all the 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 reports the rule that generated the valid workaround (see Table IV for the corresponding rules). The experiment shows that workarounds are generated from different rules, and that some rules can generate more than one workaround, thus confirming the observations about rule reusability discussed above.

The last two columns (*attempts*) measure the effectiveness of the priority scheme and the automatic validator. Both columns, labeled *without validator* and *with validator*, indicate the number of user interventions required to either identify a workaround or to conclude that RAW could not generate a valid one. The two columns report the number of necessary interventions when RAW functions with or without the automatic validator. As discussed above, we could not reproduce all the failures for the

Table V. Some code rewriting rules for the JQuery API

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') → $('<input 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("<li/>").appendTo(document.body); → $("newTag").appendTo(document.body).wrap("<li/>");
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 VI. Experiments with Google Maps API issues

Google Maps						
issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*without validator*	*with validator*
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
TOTAL	9	14	15	-	-	-

experiments *with validator*, so that column is incomplete. The results confirm that the priority mechanism is quite effective in finding valid workarounds, but can still involve the user in a fairly high number of iterations (up to 15). On the other hand, the automatic validator seems very effective in discarding many invalid attempts and letting the users focus on a few relevant cases. The validator prunes the set of candidate workarounds and allows RAW to identify a correct workaround in the first attempt for 16 out of 25 issues. It also always succeeds within the third attempt, either producing a valid workaround or signaling that such a workaround could not be found.

The 38 issues that we considered in this experiment represent field bugs of old versions of Google Maps, 25 of which have been fixed, 4 are still open at the time of writing, and 9 are *WontFix* bugs, that is, bugs that the developers closed after a while without a fix. We computed the time the bugs stayed open by referring to the closing time for the closed and the WontFix bugs, and the day of our analysis for the bugs that are still open. These bugs stayed open for a variable amount of time, ranging from a few weeks up to 68 months, with an average of 16 months. The 29 bugs that RAW can mask with a workaround stayed open for a similar amount of time, from a few weeks up to 63 months, with an average time of 12 months. Each of these bugs potentially affects all Google Maps users.

YouTube. We repeated the same effectiveness experiment with the YouTube chromeless player. The issue tracker of the YouTube chromeless player contained only 21 entries at the time of the investigation, and YouTube does not provide access to the version history of their API, so we could not reproduce any issue that was later fixed. We first populated the repository with the 40 rules that we derived from the specification (a subset of the rules are listed in Table IV) and we 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 5.2). We then selected the only open issue that we could reproduce, for which no workaround was known. 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 VII 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 we could consider, the validator filtered out all the failing attempts, thus proposing the valid workaround as the first attempt.

Table VII. Experiments with YouTube API issues

YouTube						
issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*without validator*	*with validator*
522		✓		Y1	6	-
981		✓		Y2	8	-
1030		✓		Y3	8	-
1076		✓		Y4	8	-
1180		✓		Y2	1	-
1320			✓	Y5	8	1
TOTAL	0	5	1	-	-	-

JQuery. To confirm the positive results of the studies on the Google Maps and YouTube libraries and services, we studied the API and the issue tracker of JQuery, which is a widely used open-source JavaScript library. JQuery uses a publicly available repository to store the entire version history of the API. Moreover, given the popularity of the API, the development activity is high, and so is the number of reported issues. Therefore, JQuery seemed to be a good system to experiment with.

We populated the RAW rule repository with 68 code rewriting rules that we derived from the API specifications (some of which are listed in Table V), and we selected a total of 102 issues, 25 of which already had a known workaround, and 77 without known workarounds. We considered the issues in chronological order. Table VIII 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 25).

The second positive result of this experiment is about the priority mechanism and the automatic validator. Most of the times, RAW proposes a valid workaround at the

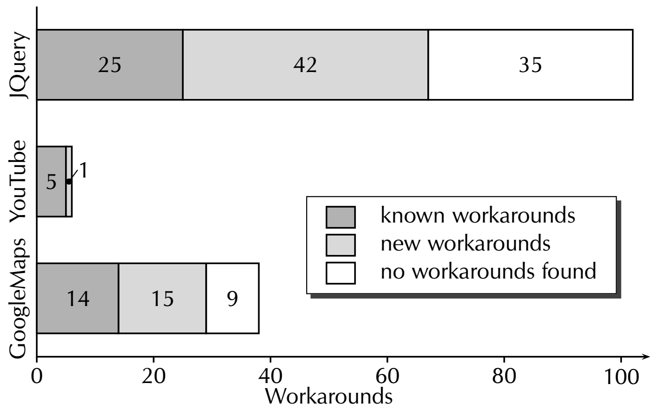


Fig. 15. Summary report of the three experiments on Google Maps, YouTube, and JQuery.

first attempt, even without the validator. Moreover, the automatic validator could discard most failing attempts in the cases where a workaround could not be found (35 cases out of 102). More precisely, for 14 of the issues that we could not solve, the validator automatically rejected all the proposed failing attempts. In such cases, which are identified in the table with a value of zeros in the last column, RAW could immediately warn the users that no solution could be found. To decide the success of a workaround we carefully checked each execution, making sure that it not exhibit visually detectable failure. Comparing our generated workarounds with patches written by developers would provide further evidence of their efficacy. However, only a few of these faults (i.e., a subset of the JQuery faults) have publicly available patches.

The experiments described so far demonstrate that the Automatic Workarounds technique implemented by the RAW prototype achieves excellent results in finding valid workarounds. The bar chart in Figure 15 summarizes the results of the experiments across the three libraries. The chart reports the number of workarounds found by RAW that were already known, the number of workarounds that were not previously known, and the number of issues that are left without a solution.

5.3.4. Effectiveness of the Priority Mechanism. To avoid any bias, in the former experiments we used the Web pages that developers attached to the issue tracker to reproduce the failures, and most of the times for JQuery such Web pages were intentionally simplified to a few lines of JavaScript code to highlight the issue. As a consequence, most of the times only few of the 68 code rewriting rules could be applied on each page, thus reducing the number of attempts. This represents a bias for the validity of the effectiveness of the priority mechanism shown in Table VIII. We therefore decided to perform one last experiment with a set of Web pages that use large amounts of JavaScript code.

We selected two Web applications that use JQuery quite heavily (up to more than 900 lines of JavaScript code per page): *iPatter*, a marketing and communication Web platform, and *UseJQuery*, a user manual Website that shows how to use the JQuery framework through a set of examples.[14] In this experiment we used some old versions of JQuery to reproduce failures that no longer exist with the newer versions of the library. This allowed us to reproduce 7 issues using three older versions of the API.

[14]`http://ipatter.com` and `http://usejquery.com`

Table VIII. Experiments with JQuery API issues

JQuery						
issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*without validator*	*with validator*
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
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
JQuery (continued)						
issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*without validator*	*with validator*
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
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
6945			✓	J18	1	1
6982			✓	J24	1	1
6999			✓	J17	2	2
7007	✓			–	1	1
7141			✓	J19	1	1
7151			✓	J19	1	1
TOTAL	35	25	42	-	-	-

Table IX. Effectiveness of the Priority Mechanism

iPatter

	no prioritization				*with prioritization*			
issue	*attempts*	*tracing*	*tracing and validator*	*saving*	*attempts*	*Tracing*	*tracing and validator*	*saving*
3828	6	4	1.67	72%	5.3	5.3	3	43.3%
5316	10	7	5.3	47%	5.3	5	3.3	37.7%
6158	10	7.3	3.67	63%	8.3	7.3	4.3	48.2%
6264	12.67	9	5	60%	2.3	2.3	1.3	43.5%
6723	8.3	6	2.67	68%	9.67	8.3	4.3	55.5%
6982	14.3	10	5	65%	10	8.3	4.3	57%
6999	13.67	10.3	6.3	53%	12	10	5	58.3%
AVG	10.71	7.67	4.24	60.4%	7.57	6.67	**3.67**	51.5%

UseJQuery

	no prioritization				*with prioritization*			
issue	*attempts*	*tracing*	*tracing and validator*	*saving*	*attempts*	*tracing*	*tracing and validator*	*saving*
3828	5	3	1.3	74%	5.3	4.3	1.67	68.5%
5316	9.3	6	4.67	49.8%	6	4.67	2	66.6%
6158	9.67	6.67	3.67	62%	8.3	6.3	3	63.8%
6264	11.3	7.67	4.3	61.9%	3.3	3.3	1.3	60.6%
6723	7.3	5.3	2.67	63.4%	8.67	7.3	3	15.8%
6982	13.67	9.3	4.3	68.5%	9.3	7.67	3.67	17.5
6999	12.67	9.67	5.3	58.1%	12	9	4.67	25%
AVG	9.85	6.81	3.76	61.8%	7.57	6.09	**2.76**	63.5%

We set up the experiment such that only one issue at a time could manifest itself. We then simulated 100 different users visiting these Web pages such that, at each visit, a user would experience one of the 7 issues selected at random. For each simulated user we then counted the number of attempts needed to identify a valid workaround both with and without the tracer and the automatic validator. We ran this experiment twice, once considering and once ignoring the priority mechanism. Table IX reports a summary of the results of this experiment.

Columns *attempts*, *tracing* and *tracing and validator* report the number of attempts needed on average to find a valid workaround, without the tracer and the automatic validator (*attempts*), considering only the tracer (*tracing*), and considering both the tracer and the automatic validator (*tracing and validator*), respectively. We can see that the tracing information can already significantly reduce the number of attempts that should be executed and evaluated by the automated validator. The latter then can filter even more attempts, leaving only a few to be manually evaluated by the user. Columns labeled as *saving* report the percentage of attempts that do not have to be evaluated by the user when the tracer and the automatic validator are used together. Table IX reports the results of the experiments with (left-hand side) and without (right-hand side) the prioritization mechanism. The salient result is that the prioritization can substantially reduce the amount of attempts needed to find a valid workaround also in presence of large amounts of JavaScript code. On average, the maximum number of attempts is 3.67 for iPatter and 2.76 for JQuery.

5.4. 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 users who report them correctly. We thus assume both an interactive nature of Web applications and a virtuous behavior of the users. In fact, since we rely on the users to reveal failures and train the priority mechanism, we assume that users would report failures whenever they notice them, and that they would also correctly accept valid workarounds.

The Automatic Workarounds technique is also limited by the assumption that multiple executions of the JavaScript code should not have side-effects. We are aware that the execution may be not only at the client side, and some requests can be sent to the server side with potential side effects. Even if we restrict our technique to Web applications built according to the Post-Redirect-Get pattern that we described in Section 3.1, we still need to assume that the data that are sent to the server are not affected by the issue at hand.

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 the presence of two or more issues, the technique should apply several rules at the same time, leading to a combinatorial explosion of applicable code transformations.

Our experiments indicate that many popular Web applications are interactive and side-effect free, and that popular applications contains a few faults rarely appearing more than once in page.

Other threats to the validity of the experiments are related to the amount of data collected and to the number of Web APIs studied so far. Although we used a relevant number of real issues for the experiments, we still 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 of the lack of access to previous versions of the API with the described faults. JQuery is a notable exception, this is why most of the data reported here come from this library. Google Maps is a partial exception, since when we started our experiments we had access to all the previous versions of the library. However, during experiments Google changed their policy and removed some old versions from the public repository, and consequently we had to abandon several issues that we could no longer reproduce. This is the reason why most of the initial experiments that come from Google Maps could not be used through the whole work, and that prompted us to switch to JQuery.

6. RELATED WORK ON SELF-HEALING SYSTEMS AND SOFTWARE REDUNDANCY

In this paper we present the Automatic Workarounds technique, which adds self-healing capabilities to Web applications by means of intrinsic redundancy. The idea of relying on some form of redundancy to mask failures is quite old, and finds its roots in the early work on fault tolerance. The classic fault tolerance approaches are based on common techniques for hardware and system fault tolerance, which in essence amounts to adding some form of redundancy to the systems to overcome production failures and other forms of runtime failures. Examples of these techniques include RAID storage systems [Patterson et al. 1988] and replicated databases [El Abbadi et al. 1985].

Simply replicating software components, however, does not help in tolerating software faults, since all replicas would contain the same faults, and consequently would fail on the same inputs. Instead, software fault tolerance techniques use *design diversity*, that is, the independent design and implementation of multiple variants of the same software components to achieve reliability. The redundant variants can run in parallel and tolerate faults through a voting mechanism, as in the seminal work on N-version programming [Avizienis 1985]. Alternatively, the redundant code can run sequentially, with one block taking over only in case of failures of another block. This is in essence how recovery blocks work [Randell 1975]. The two approaches have also been combined to obtain self-checking components [Yau and Cheung 1975]. More recently Hosek et al. proposed to keep two consecutive versions of the same software components and switch to the old versions to recover from failures in the new ones [Hosek and Cadar 2013]. Samimi et al., instead, refer to formal specifications to compute the correct output to recover from failures [Samimi et al. 2010].

In a similar fashion, Web systems may use redundancy in the form of design diversity to achieve higher reliability. In fact, several techniques propose to integrate multiple variants, designed and maintained independently, of core components such as Web services and databases, and to use them interchangeably in case of failures or in parallel as in the N-version programming paradigm [Nascimento et al. 2013; Looker et al. 2005; Gashi et al. 2004; Dobson 2006]. A significant amount of research has studied how to dynamically replace redundant versions of Web services in case of failures, specifically to increase the transparency of the replacement process [Subramanian et al. 2008; Taher et al. 2006; Sadjadi and McKinley 2005; Mosincat and Binder 2008].

Our work does not require to deliberately add redundant components, as is the case of all the aforementioned approaches. Instead, we propose to exploit a form of redundancy that is intrinsically present in software. Thus our approach is more opportunistic in nature: on the one hand it is not guaranteed to be effective in all cases, but on the other hand it is also significantly less expensive, as it does not incur the multiplicative cost of design diversity [Carzaniga et al. 2009].

Other techniques increase software reliability by making a single variant of a system more resilient to failures using some intrinsic redundancy. Data diversity, for instance, "re-expresses" data inputs to generate logically equivalent inputs that may avoid failures [Ammann and Knight 1988; Nguyen-Tuong et al. 2008; Long et al. 2012]. Our Automatic Workarounds technique is based on a similar idea, in the sense that both techniques rely on supposedly equivalent executions that may be used interchangeably. However, while data diversity exploits equivalences given by symmetries or other algebraic properties of the the data space, our technique exploits equivalences in the code, which are arguably more common, especially in Web applications.

Exception handlers may also be seen as redundant code. Exceptions are linguistic constructs that allow developers to specify how the application should deal with exceptional and typically erroneous situations [Goodenough 1975; Cristian 1982]. Exception handlers are usually very much application specific. However, several techniques extend basic exception handling mechanisms towards general handler that can work for multiple applications [Cabral and Marques 2011; Chang et al. 2013; Harmanci et al. 2011]. A similar approach, whereby a rule-based response mechanisms encodes alternative operations executed in response to a failure, has been applied to Web systems [Baresi and Guinea 2011; Modafferi et al. 2006; Friedrich et al. 2010]. Also related to rule-based exception handling is the notion of wrappers, which embody general transformation rules that filter or adapt inputs and outputs of specific components to somehow avoid anomalous interactions between components, and therefore prevent failures [Popov et al. 2001]. Similar techniques have also been proposed for Web systems, mainly to deal with possible incompatibilities between Web services [Denaro et al. 2013]. Both exception handlers and wrappers can effectively handle failures that are at least partially predictable at design time. In contrast, our technique may also be effective with unpredictable failing conditions at runtime.

Several solutions have been proposed to either avoid or handle non-deterministic failures. These solutions range from simply re-executing a failing operation [Elnozahy et al. 2002], to changing the execution environment before re-executing the failing operation, typically by cleaning or restructuring the system memory [Candea et al. 2003; Zhang 2007; Qin et al. 2007; Garg et al. 1996; Huang et al. 1995]. These approaches are related to what we propose in this paper with null operations. Also related to null operations, some techniques introduce variations in the scheduling of multi-threaded applications (using `sleep()`) to avoid race conditions and deadlocks in Java applications [Nir-Buchbinder et al. 2008; Křena et al. 2009]. While these techniques target solely non-deterministic failures, Automatic Workarounds can deal with deterministic failures as well.

Recently, several techniques were proposed to automatically generate patches for software faults [Wei et al. 2010; Dallmeier et al. 2009; Arcuri and Yao 2008; Weimer et al. 2009; Debroy and Wong 2010; Nguyen et al. 2013; Kim et al. 2013; Liu et al. 2013; Garvin et al. 2011; Perkins et al. 2009]. Although these automatic-fixing techniques share the same fundamental objective as our Automatic Workarounds technique, that is to avoid failures, our technique focuses on specific and temporary fixes for particular failure conditions, while automatic fixing aims to be more general and produce fixes that remain valid over a longer time horizon. Therefore our technique is designed to be effective for short-term fixes at runtime, while automatic fixing aims for more general fixes and thus requires a more sophisticated analysis that is suitable for the development environment but not at runtime.

7. CONCLUSIONS

In this paper we present a technique to automatically generate workarounds for Web applications. The technique exploits the intrinsic redundancy of software systems that can be found at the method level.

Software is intrinsically redundant in the sense that it offers many methods that differ in the executed code, but provide the same functionality. The technique proposed in this paper augments Web applications with a simple mechanism that allows the users to signal possible failures, usually pages that do not correspond to the users' expectations, and that responds to such failures by automatically finding and deploying suitable workarounds. The experimental results reported in the paper show that the technique is efficient and effective in many cases.

The technique described in this paper relies on some strong assumptions that are reasonable in the domain of Web applications, namely that the client-side component of the application is stateless, and that the interactive nature of the application allows the user to easily identify and report failures.

However, the underlying idea of exploiting the intrinsic redundancy of software to automatically deploy workarounds in response to runtime failures is more general. The same abstract idea of an automatic workaround can also be applied to stateful components and without direct assistance from the user in detecting failures. In fact, as an evolution of the work described in this paper, we have been researching a new technique to generate and deploy automatic workarounds at runtime within general Java programs [Carzaniga et al. 2013]. The results we obtained with this new technique are very encouraging and in many ways confirm the basic notions we developed in the context of Web applications. We can thus conclude that the technique proposed in this paper for Web applications can be applied to more general software systems, if properly extended to deal with stateful behavior and lack of interactive notification of failures.

So far, we have focused on functional failures, but we envisage the possibility of extending the approach to some classes of non functional failures as well. For example, we may address performance problems by replacing the execution of a problematic component with a redundant one that achieves better response times. This challenge is part of our future research plans.

We successfully generated workarounds for many different kinds of faults, but we did not investigate the relation between the faults that can be addressed with our approach and the generated workarounds and also the final fixes of the developers, due to the difficulty of retrieving enough data for a meaningful comparison. Investigating the relations between faults, automated workarounds, and final fixes is part of our future research plans.

In this work we aim to exploit the redundancy intrinsically present in Web applications and more generally in software systems. Our investigation indicates that redundancy is indeed largely present in modern software systems, but we have no detailed data about the pervasiveness of intrinsic redundancy. We may consider extending the work by defining a set of design guidelines to produce strongly redundant software systems to enable self-healing capabilities.

Progressing further, we would like to develop the notion of intrinsic redundancy more broadly, beyond the specific techniques to deploy automatic workarounds. We see intrinsic redundancy as a resource to be developed and exploited to improve the reliability of software but also for a variety of other software engineering activities. Therefore we are now working on other uses of redundancy, for example in testing, and more generally we are trying to develop a principled and systematic characterization of intrinsic redundancy in and of itself.

ACKNOWLEDGMENTS

This work was supported in part by the Swiss National Science Foundation, with projects n. 200021-116287 (“Perseos”), n. 200020-124918 (“WASH”), and n. 200021-138006 (“SHADE”).

REFERENCES

AMMANN, P. E. AND KNIGHT, J. C. 1988. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers 37,* 4, 418–425.

ARCURI, A. AND YAO, X. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *CEC '08: Proceeding of IEEE Congress on Evolutionary Computation*.

AVIZIENIS, A. 1985. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering 11,* 12, 1491–1501.

BARESI, L. AND GUINEA, S. 2011. Self-supervising BPEL processes. *IEEE Transactions on Software Engineering 37,* 2, 247–263.

BAXTER, I., YAHIN, A., MOURA, L., SANT'ANNA, M., AND BIER, L. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance*. 368 –377.

BRILLIANT, S. S., KNIGHT, J. C., AND LEVESON, N. G. 1990. Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering 16,* 2, 238–247.

CABRAL, B. AND MARQUES, P. 2011. A transactional model for automatic exception handling. *Computer Languages, Systems and Structures 37*, 43–61.

CANDEA, G., KICIMAN, E., ZHANG, S., KEYANI, P., AND FOX, A. 2003. JAGR: An autonomous self-recovering application server. In *Active Middleware Services*. IEEE Computer Society, 168–178.

CARZANIGA, A., GORLA, A., MATTAVELLI, A., PERINO, N., AND PEZZÉ, M. 2013. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 782–791.

CARZANIGA, A., GORLA, A., PERINO, N., AND PEZZÈ, M. 2010a. Automatic workarounds for web applications. In *FSE'10: Proceedings of the 2010 Foundations of Software Engineering conference*. ACM, New York, NY, USA, 237–246.

CARZANIGA, A., GORLA, A., PERINO, N., AND PEZZÈ, M. 2010b. RAW: runtime automatic workarounds. In *ICSE'10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (Tool Demo)*. ACM, New York, NY, USA, 321–322.

CARZANIGA, A., GORLA, A., AND PEZZÈ, M. 2008. Healing web applications through automatic workarounds. *International Journal on Software Tools for Technology Transfer 10,* 6, 493–502.

CARZANIGA, A., GORLA, A., AND PEZZÈ, M. 2009. Handling software faults with redundancy. In *Architecting Dependable Systems VI*, R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M. H. ter Beek, Eds. LNCS. Springer, 148–171.

CHANG, H., MARIANI, L., AND PEZZÈ, M. 2013. Exception handlers for healing component-based systems. *ACM Transactions on Software Engineering and Methodology 22,* 4, 30:1–30:40.

CRISTIAN, F. 1982. Exception handling and software fault tolerance. *IEEE Transactions on Computers 31*, 531–540.

DALLMEIER, V., LINDIG, C., WASYLKOWSKI, A., AND ZELLER, A. 2006. Mining object behavior with adabu. In *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis*. ACM, New York, NY, USA, 17–24.

DALLMEIER, V., ZELLER, A., AND MEYER, B. 2009. Generating fixes from object behavior anomalies. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*.

DEBROY, V. AND WONG, W. E. 2010. 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. IEEE Computer Society, Washington, DC, USA, 65–74.

DENARO, G., PEZZÈ, M., AND TOSI, D. 2013. Test-and-adapt: An approach for improving service interchangeability. *ACM Trans. Softw. Eng. Methodol. 22,* 4, 28:1–28:43.

DOBSON, G. 2006. 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*. IEEE Computer Society, Washington, DC, USA, 126–133.

EL ABBADI, A., SKEEN, D., AND CRISTIAN, F. 1985. 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*. ACM, New York, NY, USA, 215–229.

ELNOZAHY, M., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34,* 3, 375–408.

FRIEDRICH, G., FUGINI, M., MUSSI, E., PERNICI, B., AND TAGNI, G. 2010. Exception handling for repair in service-based processes. *IEEE Transactions on Software Engineering 36,* 2, 198–215.

GABEL, M., JIANG, L., AND SU, Z. 2008. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*. ICSE '08. ACM, New York, NY, USA, 321–330.

GARG, S., HUANG, Y., KINTALA, C., AND TRIVEDI, K. S. 1996. Minimizing completion time of a program by checkpointing and rejuvenation. *SIGMETRICS Performance Evaluation Review 24,* 1, 252–261.

GARVIN, B. J., COHEN, M. B., AND DWYER, M. B. 2011. Using feature locality: Can we leverage history to avoid failures during reconfiguration? In *Proceedings of the 8th Workshop on Assurances for Self-adaptive Systems*. ASAS '11. ACM, New York, NY, USA, 24–33.

GASHI, I., POPOV, P., STANKOVIC, V., AND STRIGINI, L. 2004. On designing dependable services with diverse off-the-shelf SQL servers. In *Architecting Dependable Systems II*. Lecture Notes in Computer Science Series, vol. 3069. Springer, 191–214.

GOFFI, A., GORLA, A., MATTAVELLI, A., PEZZÈ, M., AND TONELLA, P. 2014. Search-based synthesis of equivalent method sequences. In *FSE'14: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 366–376.

GOODENOUGH, J. B. 1975. Exception handling: issues and a proposed notation. *Communications of the ACM 18,* 12, 683–696.

GUHA, A., SAFTOIU, C., AND KRISHNAMURTHI, S. 2010. The essence of javascript. In *Proceedings of the 24th European conference on Object-oriented programming*. ECOOP'10. Springer-Verlag, Berlin, Heidelberg, 126–150.

HARMANCI, D., GRAMOLI, V., AND FELBER, P. 2011. Atomic boxes: Coordinated exception handling with transactional memory. In *Proceedings of the 25th European Conference on Object-oriented Programming*. ECOOP'11. Springer-Verlag, Berlin, Heidelberg, 634–657.

HATTON, L. 1997. N-version design versus one good version. *IEEE Software 14,* 6, 71–76.

HOSEK, P. AND CADAR, C. 2013. Safe software updates via multi-version execution. In *Proceedings of the 35th International Conference on Software Engineering*. ICSE '13. 612–621.

HUANG, Y., KINTALA, C., KOLETTIS, N., AND FULTON, N. D. 1995. Software rejuvenation: Analysis, module and applications. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*. IEEE Computer Society, Washington, DC, USA, 381.

JIANG, L. AND SU, Z. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th international symposium on Software testing and analysis*. 81–92.

KIM, D., NAM, J., SONG, J., AND KIM, S. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. IEEE Press, Piscataway, NJ, USA, 802–811.

KNIGHT, J. C. AND LEVESON, N. G. 1986. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering 12*, 96–109.

KŘENA, B., LETKO, Z., NIR-BUCHBINDER, Y., TZOREF-BRILL, R., UR, S., AND VOJNAR, T. 2009. A concurrency testing tool and its plug-ins for dynamic analysis and runtime healing. In *Runtime Verification*, S. Bensalem and D. A. Peled, Eds. Springer-Verlag, Berlin, Heidelberg, 101–114.

LIU, C., YANG, J., TAN, L., AND HAFIZ, M. 2013. R2fix: Automatically generating bug fixes from bug reports. In *ICST'13: Proceedings of the 2013 International Conference on Software Testing, Verification and Validation*. IEEE, 282–291.

LONG, F., GANESH, V., CARBIN, M., SIDIROGLOU, S., AND RINARD, M. 2012. Automatic input rectification. In *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. 80–90.

LOOKER, N., MUNRO, M., AND XU, J. 2005. 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*. IEEE Computer Society, Washington, DC, USA, 66–69.

LORENZOLI, D., MARIANI, L., AND PEZZÈ, M. 2008. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. ACM, New York, NY, USA, 501–510.

MODAFFERI, S., MUSSI, E., AND PERNICI, B. 2006. 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*. ACM, New York, NY, USA, 48–53.

MOSINCAT, A. AND BINDER, W. 2008. Transparent runtime adaptability for BPEL processes. In *ICSOC '08: Proceedings of the 6th International Conference on Service Oriented Computing*, A. Bouguettaya, I. Krüger, and T. Margaria, Eds. Lecture Notes in Computer Science Series, vol. 5364. 241–255.

NASCIMENTO, A. S., RUBIRA, C. M. F., BURROWS, R., AND CASTOR, F. 2013. A systematic review of design diversity-based solutions for fault-tolerant soas. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. EASE '13. ACM, New York, NY, USA, 107–118.

NGUYEN, H. D. T., QI, D., ROYCHOUDHURY, A., AND CHANDRA, S. 2013. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. IEEE Press, Piscataway, NJ, USA, 772–781.

NGUYEN-TUONG, A., EVANS, D., KNIGHT, J. C., COX, B., AND DAVIDSON, J. W. 2008. Security through redundant data diversity. In *DSN'08: IEEE International Conference on Dependable Systems and Networks*. 187–196.

NIR-BUCHBINDER, Y., TZOREF, R., AND UR, S. 2008. Deadlocks: From exhibiting to healing. In *Runtime Verification*, M. Leucker, Ed. Springer-Verlag, Berlin, Heidelberg, 104–118.

NITA, M. AND NOTKIN, D. 2010. Using twinning to adapt programs to alternative apis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ICSE '10. 205–214.

PATTERSON, DAVID, A., GIBSON, G., AND KATZ, RANDY, H. 1988. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Record 17,* 3, 109–116.

PERKINS, J. H., KIM, S., LARSEN, S., AMARASINGHE, S., BACHRACH, J., CARBIN, M., PACHECO, C., SHERWOOD, F., SIDIROGLOU, S., SULLIVAN, G., WONG, W.-F., ZIBIN, Y., ERNST, M. D., AND RINARD, M. 2009. Automatically patching errors in deployed software. In *Proceedings of the 22nd Symposium on Operating Systems Principles*. 87–102.

POPOV, P., RIDDLE, S., ROMANOVSKY, A., AND STRIGINI, L. 2001. On systematic design of protectors for employing OTS items. In *Euromicro'01: in Proceedings of the 27th Euromicro Conference*. 22–29.

QIN, F., TUCEK, J., ZHOU, Y., AND SUNDARESAN, J. 2007. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Transactions on Computer Systems 25,* 3, 7.

RANDELL, B. 1975. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*. ACM, New York, NY, USA, 437–449.

SADJADI, S. M. AND MCKINLEY, P. K. 2005. 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*. IEEE Computer Society, Washington, DC, USA, 76–87.

SAMIMI, H., AUNG, E. D., AND MILLSTEIN, T. 2010. Falling back on executable specifications. In *Proceedings of the 24th European Conference on Object-Oriented Programming*. ECOOP '10. 552–576.

SUBRAMANIAN, S., THIRAN, P., NARENDRA, N. C., MOSTEFAOUI, G. K., AND MAAMAR, Z. 2008. 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*. IEEE Computer Society, Washington, DC, USA, 33–39.

TAHER, Y., BENSLIMANE, D., FAUVET, M.-C., AND MAAMAR, Z. 2006. Towards an approach for web services substitution. In *IDEAS '06: Proceedings of the 10th International Database Engineering and Applications Symposium*. IEEE Computer Society, Washington, DC, USA, 166–173.

WEI, Y., PEI, Y., FURIA, C. A., SILVA, L. S., BUCHHOLZ, S., MEYER, B., AND ZELLER, A. 2010. Automated fixing of programs with contracts. In *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*. ACM, New York, NY, USA, 61–72.

WEIMER, W., NGUYEN, T., GOUES, C. L., AND FORREST, S. 2009. Automatically finding patches using genetic programming. In *ICSE'09: Proceeding of the 31st International Conference on Software Engineering*. 364–374.

YAU, S. S. AND CHEUNG, R. C. 1975. Design of self-checking software. In *Proceedings of the International Conference on Reliable software*. ACM, New York, NY, USA, 450–455.

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

ZHANG, S., SAFF, D., BU, Y., AND ERNST, M. D. 2011. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. ACM, New York, NY, USA, 353–363.

Received 00; revised 00; accepted 00