Automatic Workarounds for Web Applications

Antonio Carzaniga, Alessandra Gorla, Nicolò Perino, and Mauro Pezzè*
Faculty of Informatics
University of Lugano
Lugano, Switzerland
{antonio.carzaniga|alessandra.gorla|nicolo.perino|mauro.pezze}@usi.ch

ABSTRACT

We present a technique that finds and executes workarounds for faulty Web applications automatically and at runtime. Automatic workarounds exploit the inherent redundancy of Web applications, whereby a functionality of the application can be obtained through different sequences of invocations of Web APIs. In general, runtime workarounds are applied in response to a failure, and require that the application remain in a consistent state before and after the execution of a workaround. Therefore, they are ideally suited for interactive Web applications, since those allow the user to act as a failure detector with minimal effort, and also either use read-only state or manage their state through a transactional data store. In this paper we focus on faults found in the access libraries of widely used Web applications such as Google Maps. We start by classifying a number of reported faults of the Google Maps and YouTube APIs that have known workarounds. From those we derive a number of general and API-specific program-rewriting rules, which we then apply to other faults for which no workaround is known. Our experiments show that workarounds can be readily deployed within Web applications, through a simple client-side plug-in, and that program-rewriting rules derived from elementary properties of a common library can be effective in finding valid and previously unknown workarounds.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Error handling and recovery*

General Terms

Reliability, Design

Keywords

Automatic Workarounds, Web Applications, Web API

*Mauro Pezzè is also with the University of Milano-Bicocca.

1. INTRODUCTION

Application programming interfaces (APIs) for popular Web applications like Google Maps and Facebook increase the popularity of such applications, but also introduce new problems in assessing the quality of the applications. In fact, third-party developers can use Web APIs in many different ways and for various purposes, and applications can be accessed by many users through different combinations of browsers, operating systems, and connection speeds. This leads to a combinatorial explosion of use cases, and therefore a growing number of potential incompatibilities that can be difficult to test with classic approaches, especially within tight schedules and constrained budgets.

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

In this paper we propose a technique to find and execute workarounds *automatically and at runtime* in response to failures caused by faults in the libraries that the application depends on. Automatic workarounds do not fix the faults in the API code, but rather provide a temporary solution that masks the effects of the faults on applications.

We start from the supposition that libraries are often intrinsically redundant, in the sense that they provide several different ways to achieve the same results, and that this redundancy can lead to effective workarounds. For example, *changing* an item in a shopping list, may be equivalent to *deleting* the item and then *adding* a new one. So, to avoid a failing edit operation, one could replace that edit operation with a suitable sequence of delete and add operations. This assumption, that large software systems contain significant portions of functionally equivalent code, is supported by evidence from a recent study on redundant code in the Linux Kernel [13], and is also confirmed by our study of Web APIs that we report in this paper.

Based on this intrinsic redundancy, we propose a technique to build and execute, at runtime and in response to a failure, alternative sequences of operations whose intended behavior is equivalent to that of the failing sequence. We denote such sequences as *equivalent sequences.* We then call *workarounds* those equivalent sequences that execute successfully, thereby avoiding the failure.

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

We assume that we can dismiss the second problem (repeated executions) in the case of Web applications because those are typically *designed* to avoid potential side-effects caused by repeated executions. (We briefly revisit this assumption in Section 4.5.) We can therefore execute a workaround by simply re-executing the client-side code of the page that manifests the failure (after applying the workaround).

The remaining problems are the focus of this paper. For the first problem (detecting failures and verifying the validity of deployed workarounds) our general approach is to rely on the interactive nature of Web applications, and in practice to assume that users can easily detect a failure and explicitly request a workaround, for example by pressing a special "Fix-it!" button on their browser. For example, if an application that uses the Google Maps API does not display a map as expected, the user can, with minimal effort, detect the failure and push the "Fix-it!" button. Similarly, users can reject the execution of an equivalent sequence, or implicitly validate it as a workaround by proceeding with their interactive session. As we rely on users to find workarounds, we must also take special care not to annoy and ultimately alienate them with too many repeated attempts. We address this problem by developing an automatic oracle that can conservatively detect, and therefore discard, some ineffective equivalent sequences.

As for the third and fourth problems (modeling and exploiting redundancy to find valid workarounds) we continue to explore and evolve our ideas on automatic workarounds [7, 6]. In particular, in our earlier work, we propose to derive workarounds from a formal model of the application (e.g., a finite-state machine or an algebraic specification). However, that initial approach requires that the specification be *complete* in the sense that the exact state of the application must be identifiable from every sequence of API calls. Instead, here we develop a more practical notion of automatic workaround that requires only a much simpler, possibly partial, and therefore more general application model, in which we construct equivalent sequences from explicitly given sequences that are known to be functionally null or identical to others. This paper presents a concrete instantiation of this new approach in the case of Web applications, and specifically makes the following novel contributions:

- We propose an application-independent notation to represent classes of equivalent sequences. This notation consists of source-code rewriting rules that, applied to a JavaScript program, produce an equivalent but possibly failure-free program.
- We propose and evaluate a general architecture, implementable within a Web proxy or a browser plug-in, for the generation, prioritization, and deployment of workarounds for Web applications at runtime. This architecture includes an automated oracle that greatly reduces the need for user intervention.
- We present the study of some popular Web APIs together with several of their failure reports. This study leads to two contributions: (1) We present a set of concrete, API-specific rules that generate valid, known but also previously unknown workarounds for each API; (2) We derive a taxonomy of abstract, generic rules that are relatively simple but surprisingly effective.

The remainder of the paper is structured as follows. In Section 2 we present some examples that show how to exploit the implicit redundancy of Web applications to derive workarounds. In Section 3 we introduce a framework and a concrete architecture to support automatic workarounds for Web applications. In Section 4 we present experimental data obtained with applications that use the Google Maps and YouTube APIs, and that show the effectiveness of the approach proposed in this paper. In Section 5 we discuss related work, highlighting the salient differences and the novelty of our proposal. We then conclude in Section 6 with a summary of our results and some ideas for future research.

2. MOTIVATING EXAMPLES

We illustrate the notion of workaround, and specifically the nature of failures and workarounds in the context of Web applications, using two examples of failures affecting two widely used Web APIs. The two examples we present here are typical of a large number of problems that we observed in our survey of bug repositories, discussion forums, and interest groups related to popular Web APIs. We report more extensively on the results of our survey in Section 4.

Flickr, Photo visibility. Flickr is a popular photo-sharing application. The Flickr API allows users to upload and publish photos on the Web. Flickr associates photos with a visibility tag that controls access to photos on the Web, which can be *public*, *family*, or *private*. The Flickr API includes a method setPerms() that allows users to change the visibility of their photos. A message posted by a user on the Flickr forum in March 2007 reported a failure of setPerms(): the method failed to change the access status of a photo from *private* to *family* for photos that were originally uploaded as *private*.[1] The problem persisted for some time in Flickr, and was originally avoided with a simple workaround posted on the forum: to change the visibility of a photo from *private* to *family*, first change from *private* to *public* and then from *public* to *family*. This fault has since been fixed.

Facebook, Dialog windows. The popular social networking system Facebook exposes an API for Web applications. The Facebook API consists of a JavaScript library that, among other things, allows applications to create and control graphical elements. Report n. 2385 in the Facebook bug repository describes a problem with the setStyle() method of the Facebook API, which is intended to set the width and height of a graphical element. The method works well when one of the two dimensions is set individually, but fails when used to set both dimensions at the same time.[2] The report

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

was filed in June 2008 by a developer who also suggested a workaround in which one can set width and height with two separate setStyle() calls. As of the time of this writing (June 2010) this fault is still unresolved.[3]

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

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

3. AUTOMATIC WORKAROUNDS

We now describe the architecture of a system for the deployment of automatic workarounds. This description refers to a concrete implementation in the form of a browser extension, although the same architecture can also be implemented as a Web proxy. The browser extension mediates the interaction between, on the one hand, one or more servers that produce the HTML pages and the code associated with a Web application, and on the other hand, the browser components that execute the code, render the pages, and manage the interaction with the user. At a high level, the browser extension deploys a workaround by modifying the code of the pages (i.e., the application) rendered to the user.

Figure 1 shows the high-level architecture of the browser extension as well as the general process by which the extension deploys automatic workarounds. This process unfolds in the following steps. The interaction starts with a user requesting a specific page or application (step 1). The browser issues the requests for all the objects (data and code) that compose the page to the origin server (step 2). The browser extension then examines the objects returned by the server (step 3) to decide whether to enable the services of its automatic-workaround module. In particular, the browser extension looks for references to the API of known third-party Web applications for which the extension may be able to generate workarounds. If such references are found (step 4) then the browser extension activates a failure-reporting button on the toolbar (step 5). The user then displays the page and executes the JavaScript code that comes with it, which might in turn retrieve and execute additional code fragments from the API of other common Web applications. In the absence of failures, or more precisely in the absence of any perceived failure, the user continues to interact with the application, and the browser extension acts as a transparent layer.

In the presence of a perceived failure, the user invokes the browser extension using the failure-reporting button (step 6 in Figure 1) which activates the automatic-workaround module in the browser extension (AW); the AW module then examines the JavaScript code of the page in question, identifies a candidate equivalent sequence (step 7), applies it to a copy of the page, and passes the result to the automated oracle for validation (step 8). If the oracle approves the page, then the extension applies the same equivalent sequence to the original page (step 9) and presents the result to the user (step 10). If the equivalent sequence fails to resolve the problem, either because it is immediately rejected by the oracle (step 8) or because the user reports another failure, then the extension reiterates the same process (steps 7–10) until it has exhausted its repository of equivalent sequences. Equivalent sequences are produced by code-rewriting rules as described in Section 3.2; the selection of a likely workaround (step 7) is guided by a prioritization scheme described in Section 3.3; the oracle is described in Section 3.4.

The repository of equivalent sequences (i.e., rules), which are API-specific, is populated off-line by API developers or users. It is also conceivable that the repository be automatically updated as new workarounds become available, or even that the extension would consult external repositories in the attempt to find a valid workaround. However, we do not explore such technical solutions here, and instead focus on the specific nature and form of equivalent sequences. In particular, we continue in two directions. First, in Section 3.1 we describe some general classes of equivalent sequences and offer some guidelines to derive equivalent sequences from API specifications, language features, and programming experience. Second, in Section 3.2 we present a practical, perhaps simple-minded, but nevertheless effective way of expressing equivalent sequences as code-rewriting rules.

3.1 General Classes of Equivalent Sequences

We propose three general classes of equivalent sequences that are applicable to all APIs, namely *functionally null*, *invariant* and *alternative* operations. More precisely, these are classes of methods or entire code fragments that can be used to obtain concrete equivalent sequences. Here we formulate these classes intuitively, since indeed they can be derived from common sense and programming experience. However, later we also show that they can be derived experimentally by analyzing the fault repositories of such popular Web applications as Yahoo! Maps, Microsoft Maps, Facebook, Picasa, Flickr, YouTube and OpenOffice (see Section 4.1).

Functionally null operations are operations with no functional effects. This is the case, for instance, of operations that affect only timing or scheduling. As a concrete example, the JavaScript setTimeout statement is a functionally null operation that often leads to effective workarounds. Functionally null operations can affect the order of execution of some operations, and therefore can solve failures that derive from concurrency problems such as race conditions or missing barriers. Functionally null operations are easily identifiable, often do not depend on the particular API in use, and

[3]The fault is actually classified as "unconfirmed." However, we were able to replicate the fault, and are not aware of any permanent fix.

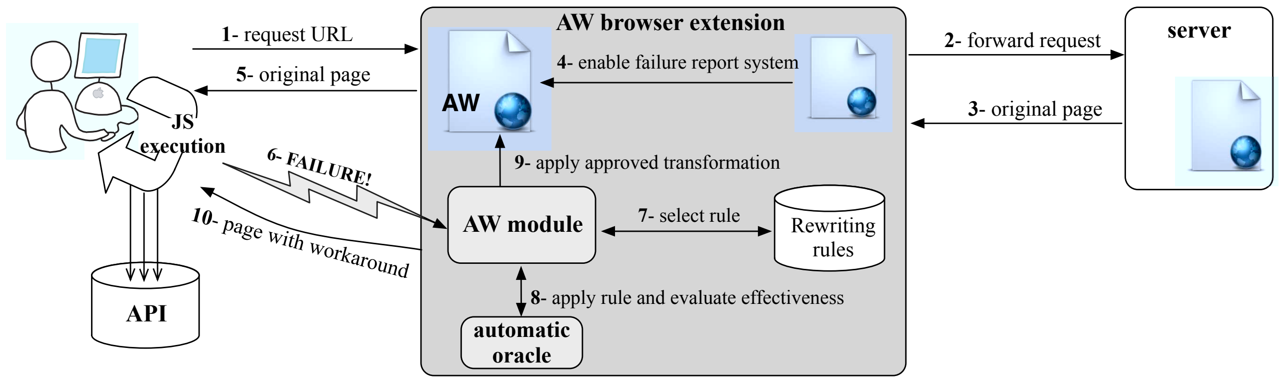

Figure 1: A Framework for Automatic Workarounds

can be simply inserted in the application code to produce equivalent sequences.

Invariant operations are typically pairs of operations in which the second one reverses the effects of the first. Longer and more complex combinations of operations are also possible, as long as the overall effect of the combination is null. Examples of invariant sequences are *add* and *remove* methods for containers, or *zoom-in* and *zoom-out* for graphical elements. Some operations within an invariant sequence may reset the state of the application, and therefore solve initialization problems, as illustrated by the Flickr example reported in Section 2. Like functionally null operations, invariant operations may also affect the scheduling of concurrent operations, and thereby solve concurrency problems. Invariant operations are API-specific but are still easily identifiable, and can also be simply inserted in the application code to generate alternative sequences.

Alternative operations are sets of operations that produce the same results as other (different) sets of operations. A simple example is the workaround for the failure of the setStyle() operation of the Facebook API described in Section 2. In that case, one call of the setStyle() method can be replaced with two calls of the same method (and vice-versa). Another example can be found in the Flickr API, where an application can associate a set of tags to a photo either with a single setTags operation for all tags, or with repeated addTags operations, each adding an individual tag. Alternative operations tend to engage more redundant code in the API, and therefore are generally more likely to lead to effective workarounds. Alternative operations are API-specific and can be identified by gaining a basic understanding of the semantics of the API. Alternative operations can generate equivalent sequences by substitution.

3.2 Program-Rewriting Rules

We propose to create equivalent sequences by applying program-rewriting rules. We now define these rewriting rules in detail. The syntax is as follows:

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

Thus, a rewriting rule is defined by three elements: an indicator of the type of rule, an indication of the scope of the rule, and a substitution expression. The type classifies the rule within one of the three categories presented in Section 3.1. Thus, the type can be null, invariant, or alternative. The scope determines *where* the substitution expression is to be applied within the program code. If the scope is ANY then the substitution is applicable to any one of the occurrences of the substitution pattern; if the scope is ALL then the rule must be applied to all occurrences of the substitution pattern; if the scope is a number n then the rule is applied to the n-th occurrence of the substitution pattern.

The substitution expression, which is the central component of the rule, is defined by one or more pairs of pattern and replacement text. In our implementation, we define each pair with a regular-expression language that is common to many text processing tools.[4] However, for simplicity of exposition, here we illustrate our examples with a slightly different notation in which we identify subexpressions as a sort of variables. These variables are assigned by the pattern expression, and then expanded in the replacement text. We denote such variables with a dollar sign followed by a capital letter (e.g., '$X'). For simplicity, we omit the actual definition of the subexpression corresponding to a variable, which can be intuitively deduced from the context. The semantics of a substitution with two or more pattern–replacement pairs is that *all* pairs must be applied. Depending on the scope, they must all be applied one time (scope ANY), for every respective occurrence (scope ALL), and for the respective n-th occurrence (scope n).

As an example, consider the following rule:

```
alternative ALL:
$X.setStyle(width:$Y,height:$Z)
→ $X.setStyle(width:$Y); $X.setStyle(height:$Z)
```

This rule refers to the example of the Facebook API given in Section 2, and indicates the equivalence between a call to the setStyle() method with two parameters, and two consecutive calls to the same method, each with one of the two parameters. So, for example, this rule would transform the code w.setStyle(width:10, height:20); into the equivalent sequence w.setStyle(width:10); w.setStyle(height:20);.

Consider now a slightly more complicated rule that alludes to the Flickr permission problem described in Section 2:

```
invariant ANY:
getInfo()
→ if(isPrivate()) {setPerms(public); setPerms(private);}
    getInfo()
```

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

This rule represents an equivalent sequence generated by an invariant operation. In particular, setPerms(public); setPerms(private) is an invariant sequence if the photo is already classified as private. Therefore, the rewriting rule prefixes the sequence with the appropriate runtime check. Notice also that, even though such an invariant operation could be inserted everywhere in the program, the rule specifies a pattern that limits the insertion of the invariant to a call to the getInfo() method (in particular right before such a call).

3.3 A Priority Scheme for Rewriting Rules

A short application program with only a few rewriting rules can give rise to many equivalent sequences, among which the browser extension must select the ones to try as workarounds. However, since equivalent sequences are submitted to, and implicitly validated by users, and since users are likely to tolerate only a few repeated failures, it is crucial that equivalent sequences be prioritized so as to increase the chances of finding a valid workaround within the first few selected equivalent sequences. In this section we describe a simple priority scheme intended for this purpose.

The scheme assigns a priority value to each rewriting rule. A priority value consists of a pair (*success-rate*, *success*), where *success-rate* is the ratio between the number of times the equivalent sequence has been used successfully as a workaround, and the number of times it has been used; and *success* is the total number of times the rule has successfully generated a workaround. The priority values of two rules are compared by first comparing their *success-rate* and then *success*. In other words, priority $p_1 = (r_1, s_1)$ is greater than priority $p_2 = (r_2, s_2)$ if $r_1 > r_2$ or if $r_1 = r_2$ and $s_1 > s_2$. If a rule was never used as a workaround, both *success-rate* and *success* are defined to be 1.

When we have two or more rules with the same priority, which happens certainly with the first failure, we apply 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 to null operations because the former are API-specific and therefore are more likely to mask faults in the API.

In the process of selecting equivalent sequences, unsuccessful sequences are counted to determine the priority of the corresponding rule, but are not considered a second time. Also, each one of the equivalent sequences generated by a rule with scope ANY is considered with the priority of that rule, and in the order of occurrence of the substitution pattern in the program.[5]

In summary, in the absence of any recorded workaround, we start from alternative operations, then continue with invariant and null operations. In the presence of recorded workarounds, we try those in order of success rate and, among the ones with the same success rate, in order of the absolute number of successes. This priority scheme is rather simple, but still quite effective in practice (see Section 4).

3.4 Automatic Oracle

An ideal prioritization scheme would rank the rewriting rules such that, if a rule exists that would produce a valid workaround, then that rule should be selected first. However, that is not always the case, especially at the beginning, when little or no information is available on the effectiveness of the various rules. To cope with such cases, and in general to further reduce the reliance on users to act as oracles, we add an *automatic* oracle to the browser extension.

The role of the automatic oracle is to reject equivalent sequences that have absolutely no visible effect on the failing page. More specifically, when the extension is invoked for the first time, the oracle is given the HTML and JavaScript code of the original (failing) page as well as the new page containing the selected equivalent sequence. In addition, the oracle is given the recording of the observable events of the failed execution. Then, the oracle executes the new application code and compares the resulting page with the page recorded from the failing execution. If the two pages show no structural difference—that is, if their DOM representations are identical—then the oracle rejects the proposed equivalent sequence. Otherwise, if the two pages differ even minimally, the oracle accepts the proposed equivalent sequence and shows it to the user as a likely valid solution, and then it is up to the user to confirm the validity of the workaround. Notice that valid workarounds would change the observable behavior of the page, and consequently its structure. In other words, the oracle acts conservatively by accepting any change as a potentially valid workaround.

Every rejection by the oracle is interpreted as a rejection by the user for the purpose of adjusting the statistics and priorities associated with rules. For every subsequent rejection by the user, the oracle adds the new failing page to the set of the failing pages it compares with every subsequent candidate workaround. Thus, each equivalent sequence supplied by the AW module is compared with all the pages previously flagged by the user, and the oracle rejects all equivalent sequences that are identical to any one of the failing pages.

4. EXPERIMENTAL EVALUATION

We articulate our evaluation around the following research questions:

Q1 Can workarounds cope with failures of Web applications effectively?

Q2 Can workarounds be generated automatically?

Q3 Can our proposed method generate valid workarounds?

The first question (Q1) explores the possibility of using workarounds to handle failures in Web applications, and their effectiveness. In other words, we ask whether workarounds even *exist* and whether they can be used with Web applications. A positive answer to this first basic question then leads directly to the second question (Q2), which asks whether it is possible to generate workarounds *automatically* or whether that is an inherently creative activity that should be left to human designers. Finally, the third question (Q3) evaluates the ability of the specific technique proposed in this paper to generate and deploy valid workarounds.

Our general method of evaluation uses a two-pronged experimental approach. To provide an answer to Q1 and in part to Q2, and also to gain a better understanding of typical failures in Web applications, we surveyed the fault repositories and other on-line forums dedicated to popular Web APIs. Then, to provide a constructive answer to Q2 as well

[5]The natural ordering was chosen for its simplicity. In the future we plan to study other orders.

as a direct answer to Q3, we implemented and tested a prototype of the architecture described in Section 3. This prototype, called *RAW* [5], is a Firefox browser plug-in, and contains a database of rewriting rules for the Google Maps and YouTube APIs. We now briefly outline both the survey method and the prototype implementation, and then turn to the specific results we obtained pertaining directly to the three research questions, examining each question in turns.

4.1 Survey of Failure Reports

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

In some cases, we were fortunate enough to be able to replicate failures and workarounds. We could do that with the Google Maps library, since Google used to export the complete version history of their Web APIs. Unfortunately, recently Google removed many program versions from its publicly available history, so we were unable to complete all the experiments we had planned. In particular, we could not repeat all our initial experiments with the automated oracle, which we introduced only recently. In another case (YouTube) we could only replicate the failures that have not been fixed.

These cases, where we could replicate failures and workarounds, are particularly important because they offer direct evidence and also because they provide concrete case studies for the evaluation of our prototype. We report extensively on these cases in the following sections. In all other cases, we could only rely on the more or less formal description of the failure. Nevertheless, all the failure reports we analyzed were useful in characterizing failures and typical workarounds, and are the basis for the taxonomy of workarounds synthesized in Section 3.1.

4.2 Workarounds for Web Applications

Our study of several repositories and forums dedicated to open as well as fixed faults in popular Web APIs indicates that workarounds exist in significant numbers and are often effective at avoiding or mitigating the effects of faults for Web applications.

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

Table 1: Faults and workarounds for Google Maps and YouTube

The high-level results of our survey for Google Maps and YouTube are summarized in Table 1. We studied the Google Maps API bug tracker in July 2009 and we found a total of 411 faults (excluding invalid bug reports). We selected the entries potentially related to workarounds by matching the keyword "workaround" in the bug report descriptions and we obtained 63 entries. We then focused on these entries, ignoring other possible workarounds not marked explicitly as such. Upon further examination, we found 43 proper workarounds (the word "workaround" occurs in comments such as "Does anybody know a workarounds for this issue?" that do not report any valid workaround). In total, this amounts to about 10% of the reported faults. This result indicates that workarounds can successfully address a good amount of API runtime issues. We should note that the 10% prevalence of workarounds for the Google Maps API is a conservative estimate, since we analyzed only 63 out of 411 reports.

To get additional information on the effectiveness of workarounds for fixing Web APIs, we considered the bug tracker of the YouTube chromeless player, which included 21 known issues at the time of the investigation. Given the modest size of the repository, we analyzed all 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. These results have been further confirmed by the sampling of the bug repositories of other popular Web APIs.

The data collected so far cannot be generalized to conclusive quantitative results, but they nonetheless indicate that it is often possible to overcome Web APIs issues by means of appropriate workarounds.

4.3 Automatic Generation of Workarounds

Having observed that some failures of Web APIs can be fixed with workarounds, we want to show that at least some of these workarounds can be generated automatically from a set of reasonable and reasonably general rules. To that end, we further analyzed the 52 workarounds that we found in the bug-tracking repositories of Google Maps and YouTube (43 and 9, respectively). In particular, we tried to distinguish between ad-hoc, hardly generalizable workarounds or workarounds that are simply not deployable at runtime, from more general workarounds that are amenable to a runtime deployment.

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

> 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.[6]

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

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

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

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

> Many times the map comes up grey ... a slight setTimeout before the setCenter ... might work. ... if you change the zoom level manually ... after the map is fully loaded, it will load the images perfectly. So, what I did was add a "load" event ... and had it bump the zoom level by one and then back to its original position after a really short delay.[7]

This report contains an example of two workarounds that are easy to generalize and deploy at runtime: *add a setTimeout* and *add a zoom-in-zoom-out sequence*, which are good examples of null and invariant operations, respectively, as defined in Section 3.1.

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

Table 2: Amount of reusable workarounds

Table 2 summarizes the results of this analysis: 32% of the known workarounds found in the Google Maps repository and 55% of the YouTube ones follow general patterns, can be coded and reused, and therefore are good candidates for automatic generation. This analysis provides encouraging although perhaps still preliminary evidence that workarounds can be generated at runtime from general rules.

4.4 Effectiveness of Automatic Workarounds

To investigate the effectiveness of the approach proposed in this paper, we performed various experiments with Google Maps and YouTube.

We first populated the repository of rewriting rules with two sets of rules, one for each of the two selected APIs. We generated these rules by studying the APIs of the two libraries and by instantiating the three classes of rules introduced in Section 3.1. In total, we wrote 40 rules for Google Maps and 38 rules for YouTube, a subset of which are listed in Table 3, with labels G1–G14 and Y1–Y6 for Google Maps and YouTube, respectively.

We started with the 14 problems that had known workarounds for Google Maps, and verified that RAW (our prototype) can automatically generate a workaround for all of them with the current set of rules. We then added 24 problems selected among the ones that were reported without known workarounds and that could be reproduced with a version of the API available in the Google Maps version history. We then replicated each of the 38 cases with RAW, following the chronological order given by their issue number, initializing RAW with the same priority $\langle 1, 1\rangle$ for all the rules. In a first run of these experiments, we used a prototype implementation of RAW that did not include the oracle. We then repeated the experiments with the latest prototype implementation of RAW that includes the oracle. However, unfortunately, during this second set of experiments, some versions of the Google Maps API were unavailable, so that we could only reproduce 24 of the original 38 failures.

We then turned to the 21 faults reported in the YouTube failure repository. Unfortunately, YouTube does not provide access to the version history of their API, so we could not reproduce any failure that was later fixed. Nevertheless, we first applied the rules manually (i.e., off-line) to the 21 failures, and verified that we could generate valid workarounds for the five problems reported with known workarounds. We also found a workaround for an open problem, to which we could then apply RAW, including its oracle.

Table 4 reports the results of the experiments just described. The first column (*issue*) indicates the issue number in the failure repositories. The following set of columns (*workaround*) reports the results of using RAW to generate workarounds. Specifically, *none* means that RAW could not generate any valid workaround; *known* means that RAW automatically generated a workaround that was already known; and *new* means that RAW generated a new workaround for an open problem. The fact that RAW can not only generate all known workarounds, but also many additional workarounds for open problems, provides an affirmative answer to our third research question (Q3) and confirms our general research hypothesis.

The *rule* column indicates the rule that generated the valid workaround (find the corresponding rule in Table 3). The experiment shows that workarounds are generated from different rules and that some rules can generate more than one workaround. The set of experiments is not large enough to generalize on the effectiveness of some rules over others, but we notice that generic rules such as timeout-insertion seem to be more effective than other, more specific rules.

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

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

	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	alternative ALL:	$X.setSize($Y,$Z) → $X.width=$Y; $X.height=$Z;
Y6	invariant ANY:	$X.stop() → $X.stop(); $X.stop();

Table 3: Some Rewriting Rules for the Google Maps and YouTube APIs

attempt, either producing a valid workaround or signaling that such a workaround could not be found.

4.5 Limitations and Threats to Validity

The technique presented in this paper is limited primarily by the assumed nature of failures in Web applications. We assume that failures are visible to the user, and furthermore that the relevant application state consists of the structure (DOM) of the application pages plus possibly server-side data. Although this is consistent with the recommended architecture and the general nature of Web applications, modern Web applications rely increasingly on client-side computations and state that do not affect the structure of the application pages. As future work, we plan to explore methods to fully address failures of this kind of applications.

The experiments reported in this section provide encouraging results, but are not yet complete. We now discuss the main threats to the validity of the experiments, and discuss how we mitigated their effects. The main threats to validity are related to the amount of data collected, and to the number of Web APIs studied so far.

The data reported in the experiments are limited. We report experiments with a total of 78 rules (40 for Google Maps and 38 for You Tube) and 44 issues (14 faults with known workaround plus 24 additional faults for Google Maps, and 5 with known workaround plus 1 without for YouTube). The quantitative data cannot be generalized with confidence, but the qualitative results are extremely encouraging. We limited our experiments on issues reported in public bug repositories of popular applications. This choice allows us to rely on the results qualitatively already with the available data, and quantitatively when we will have more data points. The limited amount of data is due to the difficulty of finding reproducible failures. Many of the issues reported in bug repositories are published with a solution, and most applications do not support access to previous versions with faults. Google Maps is a partial exception. This is why most data reported here come from Google Maps.

The results are obtained only on two Web APIs, which may not represent well the wider domain of Web APIs. We investigated several APIs, but here we report only on Google Maps and YouTube, which we could study systematically. The data obtained on other applications confirm the results reported in this paper. The chosen applications are two of the most popular applications, and even though they may not represent all Web APIs, they certainly represent a relevant subset, thus even if the results would not generalize to all APIs, they apply to a relevant set of applications.

The lack of a specific user study is another threat to the validity of our experimental evaluation. We hypothesize that users would be able to correctly detect failures and identify valid workarounds, but we did not confirm these hypotheses with an appropriate experiment.

5. RELATED WORK

Avoiding failures by automatically preventing or recovering from the occurrence of faults is the main goal of fault tolerant systems, and more recently of autonomic and self-healing systems. Recent work has taken inspiration from classic fault tolerant and modern self-healing techniques for the automatic recovery from failures of Web applications. In this section, we briefly survey the approaches related to our work, highlighting similarities and differences, and also pointing to the underlying classic methods.

Closely related to our research is the *Browser JavaScript* plug-in for the *Opera* browser that automatically applies some fixes to solve well known problems of interactions between Web applications and the browser. This plug-in implements a given set of fixes that are automatically deployed when the user is visiting a web page that is incompatible

issue	*workaround*			*rule*	*attempts*	
	none	*known*	*new*		*no oracle*	*oracle*
Google Maps						
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
754			✓	G2	13	2
737		✓		G4	3	–
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
YouTube						
522		✓		Y1	6	-
981		✓		Y2	8	-
1030		✓		Y3	8	-
1076		✓		Y4	8	-
1180		✓		Y5	1	-
1320			✓	Y6	8	1

Table 4: Google Maps API and YouTube API issues

with the browser.[8] Differently from this plug-in, we propose a general mechanism to derive and execute workarounds that can be automatically deployed in the presence of *unexpected* failures. Still, the Opera plug-in confirms both the industrial interest and the applicability of the methods and ideas proposed in this paper.

Classic approaches to tolerate faults rely on redundancy and design diversity, that is, the use of independently designed software components to reduce correlated faults. Redundancy and design diversity are exploited, for example, in N-version programming [2], in recovery blocks [19] and in self-checking programming [14]. The increased reliability gained with design diversity comes with high development costs. Conversely, the method proposed here relies on the *intrinsic* redundancy of many Web APIs, and therefore does not incur additional development costs. Also, the program-rewriting rules, which express the intrinsic redundancy of the Web APIs, can be provided incrementally and on demand. We can for example assume a beta-testing period during which failures and workarounds are reported to Web applications and APIs designers, who can then incrementally populate the repository of program-rewriting rules.

Both Baresi et al. and Modafferi et al. propose "rule engines" that rely on registries to cope with failures [3, 15]. Similar to exception handlers, registries are provided by developers at design time, and contain lists of failures with corresponding recovery actions to be executed at run time. This method requires designers to predict failures and recovery actions at design time, and it can hardly deal with unexpected runtime failures. Contrariwise, our approach relies on a set of program-rewriting rules that can be adapted to cope with unexpected runtime failures and can be added incrementally without requiring expensive redeployment.

Denaro et al. propose wrappers to automatically adapt Web services and overcome interface incompatibilities [10], thus extending the work on wrappers that are classically applied to mitigate the problems that may derive from integrating software components [17, 8]. As for rule engines, wrappers have to be provided at design time, and thus have limited success with unexpected failures.

Micro-reboot, checkpoint and recovery, and software rejuvenation address non-deterministic failures by rebooting the components that caused the failure (micro-reboot), by rolling back to the latest consistent state and re-executing the failing operations (checkpoint and recovery), and by regularly restarting the system to prevent failures due to software age (rejuvenation) [4, 11, 12]. Similar to these approaches, Qin et al. propose the Rx method that partially re-executes the failing program under modified environment conditions [18]. All these techniques deal with non-deterministic failures and incompatibilities with the environment that are mostly outside the scope of this paper.

We would like to conclude this section by mentioning a few approaches that attempt to automatically fix faults at runtime with different strategies. Weimer et al. and Arcuri et al. investigated genetic programming as a way to automatically fix software faults [20, 1]. Both approaches assume the availability of a set of test cases to be used as oracles. When the software system fails, the runtime framework automatically generates a population of variants of the original faulty program that are subsequently evolved through test-based selection. These approaches are designed to work offline, since the evolution of the population and the execution of the test suite are time consuming activities. Dallmeier et al. proposed a technique that can generate fault fixes by comparing the models inferred from correct and failing executions [9]. Similarly to the approach proposed by Dallmeier et al., ClearView relies on dynamically inferred properties of correct executions to detect and automatically fix incorrect behavior due to malicious attacks [16]. Although this approach could be extended to functional faults, so far it has been proposed only to address security issues.

6. CONCLUSIONS

With this paper we develop the idea of automatic workarounds for Web applications. A workaround either avoids or masks a failure in an application. Workarounds are typically suggested by developers or users, and deployed statically through an ad-hoc process. We propose a systematic

[8]http://www.opera.com/docs/browserjs

method to generate and deploy workarounds dynamically, at runtime, in the particular and increasingly important domain of Web applications. We detail this architecture together with a method to generate workarounds using code-rewriting rules. In our experimental evaluation, we test the validity of the notion of automatic workaround as well as the effectiveness of the proposed architecture. Our experimental evaluation examines several common Web APIs, and focuses on two (Google Maps and YouTube) that are probably the most widely used ones. The results of this evaluation are very encouraging, and lead us to believe that automatic workarounds can be very effective with Web applications.

The notion of automatic workaround as well as the specific methods developed in this paper can and should be explored further. First, we would like to widen the scope of our experiments to more applications and APIs. Part of this effort will require the development of our browser extension prototype beyond its initial capabilities. Furthermore, a serious experimental evaluation would benefit greatly from a more direct access to the version history and the knowledge available from internal issue-tracking systems or even directly from developers.

Beyond improving the specific techniques and strengthening the experimental evaluation, this work raises more general and exciting research questions. One of these is motivated by the wide-scale and social nature of Web application themselves: it is only natural to see the formulation and deployment of workarounds as yet another Web application. Therefore, it is natural to study how automatic workarounds could be developed, enhanced, and shared more effectively.

Acknowledgments

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

7. REFERENCES

[1] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *Proceeding of IEEE Congress on Evolutionary Computation*, pages 162–168, 2008.

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

[3] L. Baresi, S. Guinea, and L. Pasquale. Self-healing BPEL processes with Dynamo and the JBoss rule engine. In *Proceedings of the International Workshop on Engineering of Software Services for Pervasive Environments*, pages 11–20, 2007.

[4] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox. JAGR: An autonomous self-recovering application server. In *Active Middleware Services*, pages 168–178. IEEE Computer Society, 2003.

[5] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. RAW: Runitime automatic workarounds. In *Proceedings of the 32nd International Conference on Software Engineering*, pages 321–322. ACM, 2010.

[6] A. Carzaniga, A. Gorla, and M. Pezzè. Healing Web applications through automatic workarounds. *International Journal on Software Tools for Technology Transfer*, 10(6):493–502, December 2008.

[7] A. Carzaniga, A. Gorla, and M. Pezzè. Self-healing by means of automatic workarounds. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 17–24, 2008.

[8] H. Chang, L. Mariani, and M. Pezzè. In-field healing of integration problems with COTS components. In *Proceeding of the 31st International Conference on Software Engineering*, pages 166–176, 2009.

[9] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 550–554. IEEE Computer Society, 2009.

[10] G. Denaro, M. Pezzè, and D. Tosi. Ensuring interoperable service-oriented systems through engineered self-healing. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 253–262. ACM, 2009.

[11] M. Elnozahy, L. Alvisi, Y. min Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[12] S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. *SIGMETRICS Performance Evaluation Review*, 24(1):252–261, 1996.

[13] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 81–92, 2009.

[14] J.-C. Laprie, C. Béounes, and K. Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, 1990.

[15] S. Modafferi, E. Mussi, and B. Pernici. SH-BPEL: a self-healing plug-in for WS-BPEL engines. In *Proceedings of the 1st workshop on Middleware for Service Oriented Computing*, pages 48–53, 2006.

[16] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd symposium on Operating systems principles*, pages 87–102, 2009.

[17] P. Popov, S. Riddle, A. Romanovsky, and L. Strigini. On systematic design of protectors for employing OTS items. In *Proceedings of the 27th Euromicro Conference*, pages 22–29, 2001.

[18] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Transactions on Computer Systems*, 25(3):7, 2007.

[19] B. Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, 1975.

[20] W. Weimer, T. V. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceeding of the 31st International Conference on Software Engineering*, pages 364–374, 2009.