Int J Softw Tools Technol Transfer (2008) 10:493–502
DOI 10.1007/s10009-008-0088-8

SPECIAL SECTION WQVV 07

Healing Web applications through automatic workarounds

Antonio Carzaniga · Alessandra Gorla · Mauro Pezzè

Published online: 21 October 2008
© Springer-Verlag 2008

Abstract We develop the notion of *automatic workaround* in the context of Web applications. A workaround is a sequence of operations, applied to a failing component, that is equivalent to the failing sequence in terms of its intended effect, but that does not result in a failure. We argue that workarounds exist in modular systems because components often offer redundant interfaces and implementations, which in turn admit several equivalent sequences of operations. In this paper, we focus on Web applications because these are good and relevant examples of component-based (or service-oriented) applications. Web applications also have attractive technical properties that make them particularly amenable to the deployment of automatic workarounds. We propose an architecture where a self-healing proxy applies automatic workarounds to a Web application server. We also propose a method to generate equivalent sequences and to represent and select them at run-time as automatic workarounds. We validate the proposed architecture in four case studies in which we deploy automatic workarounds to handle four known failures in to the popular Flickr and Google Maps Web applications.

Keywords Self-healing · Autonomic computing · Equivalent sequences · Automatic workarounds · Fault recovery

This work has been supported by the project PerSeoS funded by the Swiss National Fund.

A. Carzaniga · A. Gorla (✉) · M. Pezzè
Faculty of Informatics, University of Lugano, Via Buffi13, 6904 Lugano, Switzerland
e-mail: alessandra.gorla@lu.unisi.ch

A. Carzaniga
e-mail: antonio.carzaniga@unisi.ch

M. Pezzè
e-mail: mauro.pezze@unisi.ch

M. Pezzè
Dipartimento di Informatica, Sistemistica e Comunicazione, University of Milano-Bicocca,
Via Bicocca degli Arcimboldi 8, 20126 Milan, Italy

1 Introduction

In a previous preliminary paper [5], we have proposed the notion of *automatic workaround* as a basic mechanism to tolerate faults, and therefore to implement self-healing [17, 18], for component-based systems. The idea of automatic workaround amounts to exploiting the redundancy of code in such modularized systems.

In order to define automatic workarounds, we focus on a single server component that exports a well-defined set of interface operations and that is used by one or more other client components. The server component is the one that may contain a fault, and to which we apply automatic workarounds. (Notice that the server/client categorization is not specific to networked components, but instead it simply allows us to identify the target of automatic workarounds.) We make two important assumptions. First, we assume that, when the component fails, the failure can be detected and the component can be brought back to a consistent state. This is usually done explicitly by the programmer, through a variety of mechanisms such as assertions and exception handling. Second, we require a specification of component behavior. The specification can be given in various forms. We prefer a formal model, such as a state-based model, but the technique works with partial models as well, including models generated by the observation of correct behaviors.

Within this context and with the basic assumptions outlined above, we developed the following operational definition of automatic workarounds. When the component fails, we examine the sequence of operations that lead to the failure. In particular, we consider the initial state of the component, the failing sequence, the intended final state, and the fall-back state (possibly but not necessarily the same as the initial state). Then, we select an alternative sequence of operations that, according to the specified behavior, would bring the component from the fall-back to the intended final state. We call these sequences *specification-equivalent sequences*. The idea is to select among the specification-equivalent sequences one that would not incur a failure, and therefore that would serve as a workaround. We call such alternate sequences *automatic workarounds* whenever they can be generated on the basis of the model of the component, and when they can be selected and applied automatically at run time, effectively masking faults.

In this paper, we describe a concrete instantiation of the notion of automatic workarounds in the context of Web applications that offer an interesting domain for a number of reasons. From a purely technical viewpoint, Web applications use communication and implementation mechanisms that are very amenable to the deployment and use of automatic workarounds. In terms of communication, their primary interface (HTTP) admits an almost completely transparent monitoring, interception, and redirection of operation calls. In terms of implementation, Web applications are typically based on multi-tier architectures that clearly separate the application logic from the application state (database) and therefore are inherently capable of maintaining a consistent state in the presence of failures.

The second and more specific contribution of this paper is a method to automatically generate and select equivalent sequences of operations to serve as workarounds. This method assumes a state-based model of the system. Starting from the intended state transformation of the failing run, we perform a constrained exploration of the state space to find alternative sequences of operations that represent an equivalent state transition, and therefore a potential workaround.

We applied this method to four case studies in which we look for automatic workarounds for four faults in the two popular Web applications Flickr and Google Maps. Our experience shows that automatic workarounds are indeed possible with Web applications, and that the generation method we propose coupled with a simple prioritization rule is effective in identifying valid ones.

We continue in Sect. 2 with a detailed presentation of the application of automatic workarounds to Web applications. Then in Sect. 3 we describe the process by which we generate alternative sequences and then select good candidates to use as automatic workarounds. In Sect. 4 we present the four case studies that illustrate and validate our methods. In Sect. 5 we position our work within the context of other self-healing techniques with a particular attention to techniques developed for Web applications. Finally, we conclude in Sect. 6 with a roadmap for future research in the development and refinement of the idea of automatic workarounds.

2 Automatic workarounds and Web applications

We propose a general architecture for implementing automatic workarounds where a self-healing layer mediates the interactions between a client component and a server component. (Again, here the term "server" refers to the target of automatic workarounds.) Figure 1 depicts this general architecture. The self-healing layer observes the interaction between the components, which consists of a sequence of method calls from the client (top of the diagram) to the server (bottom). With this sequence, and having a model of the server component, the self-healing layer maintains an abstraction of the state of the server, effectively using the model to simulate the internal behavior of the server. Whenever a failure is detected, and after the server has been brought back to an internally consistent state, the self-healing layer intercepts the failure signal and intervenes by selecting and executing an equivalent sequence. In case the equivalent sequence fails, the self-healing layer may try to execute other equivalent sequence. If one of the equivalent sequences is successful, then the execution proceeds normally as if the failure never occurred. If none of the selected equivalent sequences is effective, then the self-healing layer reports the failure back to the client.

In order to apply this general architecture to Web applications, we choose a specific design for the self-healing layer. We also make specific assumptions about the availability and nature of the failure detector as well as the recovery mechanism used by the server. In particular, we assume that client components are primarily end-user applications that are controlled directly through an interactive graphical user interface (typically a Web browser). Furthermore, we assume that the server component resides outside of the

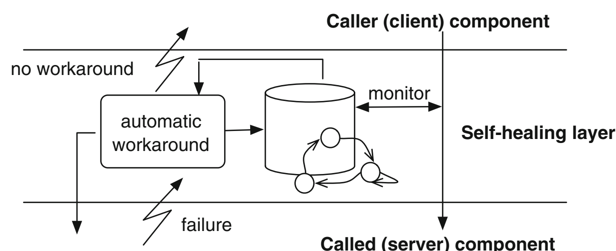


Fig. 1 General architecture for automatic workarounds

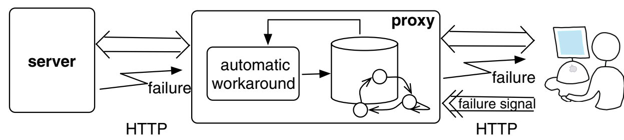


Fig. 2 Architecture for automatic workarounds in Web applications

administrative domain of the user, that it is accessible only through its HTTP interface, and that it is designed to maintain a basic level of internal consistency despite internal or external failures and repeated or erroneous requests. This latter assumption is realistic, since most application servers are implemented using specific frameworks that guarantee a basic form of error handling, and also because application servers that implement the main application logic are typically backed by a database to maintain a stable and consistent state.

Based on these assumptions, we design the self-healing layer as shown in Fig. 2. The diagram highlights two important architectural choices. The first notable difference with the general architecture is that the user is now involved in the failure detection process. Therefore, the self-healing layer no longer masks failures on behalf of the user. This decision is dictated by practical considerations regarding the characteristics of Web applications. Although Web application servers offer some basic form of failure detectors (for instance, internal server errors are signaled through a "500" HTTP response), a better failure detection mechanism is necessary and available. In fact, the interactive and user-driven nature of Web applications suggests that users themselves can be the most effective failure detectors, and also that users would be willing to play that role if that were made accessible to them through a convenient user-interface mechanism.

The second notable design decision is to locate the self-healing layer close to the client. Specifically, we propose to use a self-healing layer implemented as a Web proxy. This architectural model seems very appropriate given the nature of Web applications, and especially given the prominent role of proxies as integral components of the HTTP protocol. Also, implementing the self-healing layer as proxy is the least invasive solution, since it does not affect the code of the application server or the client.

As for its other features, the design of the self-healing layer does not differ substantially from the general design. A Web application (on the right of Fig. 2) invokes the methods of a service (on the left of Fig. 2) while the self-healing proxy (in the center of Fig. 2) monitors the calls and maintains a partial history of calls. As we will see later, this history is represented by a finite model that abstracts the state of the service. In addition to monitoring calls, the self-healing proxy intercepts failure signals coming from both the application server and the user. The server signals failures through the usual error reporting mechanisms provided by HTTP, while the user may signal failures through a GUI element (for instance, a button or a link) inserted by the proxy within the application content.

In response to a failure signal, the proxy selects and executes workarounds. As a first step, the proxy uses information provided by the failure signals, together with its call history and the corresponding model of the internal state of the application server, to establish (1) the *intended state* of the application after the sequence of calls issued by the client, and (2) the *actual state* of the application after the failure. Then the proxy proceeds to identify a workaround. Workarounds are sequences of service invocations that, starting from the actual state, have the same intended effect as the failing sequence, that is, have the same effect according to the specifications, regardless of the actual (failing) behavior. To avoid ambiguity, we call these *specification-equivalent* sequences. We propose to generate workarounds by matching the failing sequence with the specifications, as discussed in detail in the next section.

Notice that, in the particular case of Web applications, service interfaces are often redundant, so that clients can obtain equivalent results through many sequences of method invocations. Therefore Web applications lend themselves to the automated generation of workarounds. For example, the popular Flickr API offers different services for manipulating tags associated to photos, such as *setTags* to associate new sets of tags to photos, and *addTags* to append additional tags to photos. We can use different sequences of invocations of these and other services to attach the same set of tags to a photo.

3 Automatic generation of equivalent sequences

Our approach towards producing automatic workarounds relies on the ability to generate equivalent sequences automatically from specifications. This amounts to identifying sequences of service invocations that produce equivalent effects according to the specifications, and can thus substitute the failing invocation sequences. In this section, we show how to automatically generate specification-equivalent sequences from finite state machines.

We start with a finite state machine specification and a failing sequence of service invocations. The finite state machine specifies the expected behavior of the application, and is produced during the design. Figure 3 shows an example of a finite state machine specification of the sale function in a simplified e-commerce application. Customers can add items that are initially *on sale* either to the *wish list*, to indicate interest in the items, or to the *cart*, to buy the items by proceeding with payment.

A *failing sequence* is identified by both the failure-detection (the user) and the recovery mechanisms that we assume is available for the system. The failure detection

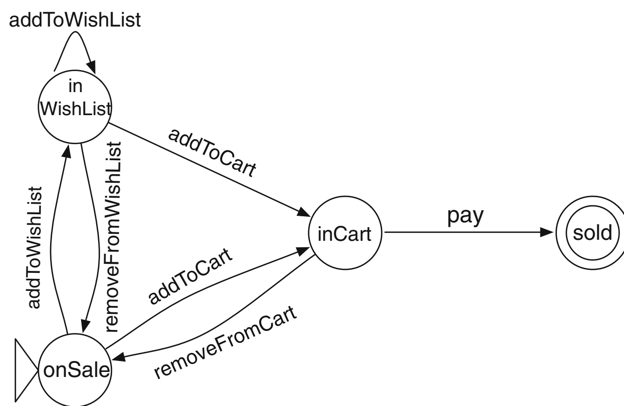


Fig. 3 A finite state machine specification of a simple sale function

mechanism signals the *failing state*, that is the final state of the failing sequence, while the recovery mechanism backtracks to a consistent *fall-back state*, that is the state of the system after recovery.

Once identified a failing sequence, we automatically identify sequences equivalent to a failing one as follows:

1. We produce a new finite state machine from the original one by removing all the invocations of the failing sequence that do not affect the reachability of the intended final state from the fall-back state. We then generate sequences that connect the fall-back state to the intended final state. While generating equivalent sequences, we traverse cycles at most once. We do that because the positive effect of methods that can impact on the final result and fix the problem is usually already visible after the first invocation of the methods. In other words, equivalent sequences containing multiple, cyclic invocations have little or no advantage in leading to an effective workaround.

   For example, if the failing sequence of the sale function of Fig. 3 consists of the only invocation of function *addToCart* from state *onSale*, we can remove the *addToCart* transition from state *onSale* to state *inCart* without affecting the reachability of the intended final state, and then use the new FSM to identify equivalent sequences. Examples of such sequences are:

   addToWL, addToCart
   addToWL, removeFromWL, addToWL, addToCart
   addToWL, addToCart, removeFromCart, addToWL, addToCart

2. if none of the invocations that belong to the failing sequence can be removed from the FSM without disconnecting the intended final state from the fall-back state, we interleave invocations of the failing sequence with *indifferent* invocation sequences.

   Indifferent sequences are sequences that, according to the specification, do not affect either the application behavior or the final result, and that, when invoked in presence of failure, may mask or avoid the problem.

   Such invocations include single functions that alter only the *timing* or *scheduling*, and thus have no functional effect (for example, delay the execution of a function), or *maintenance* actions (for example, clean the browser cache) that have no direct functional effect on the application. They can also involve sequences of two or more services that cancel each other (for example, pairs of add and remove, or load and unload operations).

   Thus, it is possible to generate equivalent sequences by interleaving the failing sequence with indifferent invocations. As in the previous case, we do not generate sequences that include cyclic invocation of the same sub-sequences.

   For example, if the failing sequence of the sale function of Fig. 3 consists of the only invocation of function *pay* from state *inCart*, we cannot remove transition *pay* from state *inCart* to state *Sold*, without affecting the reachability of the intended final state. Thus, we generate sequences equivalent to the failing one by adding indifferent invocations, for instance:

   removeFromCart, addToCart, pay
   removeFromCart, addToWL, removeFromWL, addToCart, pay
   sleep, pay

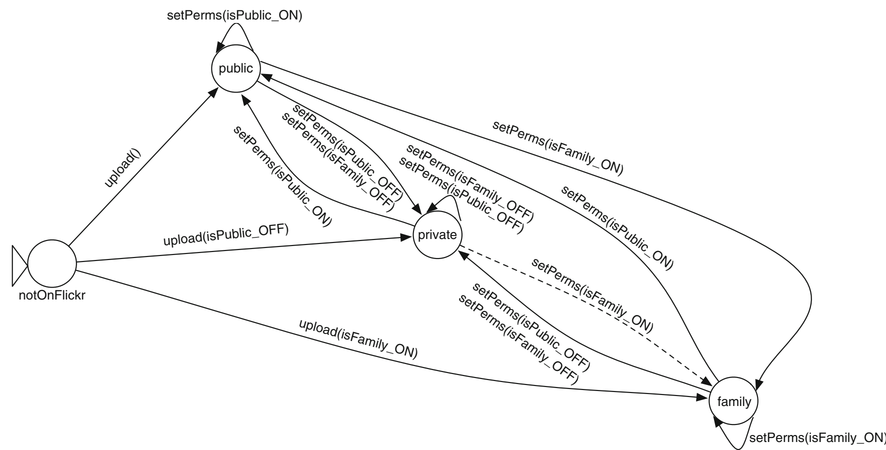
   where the function *sleep* is a function that delays the execution of the following invocations of some time.

3. In both cases, we produce many sequences. We prioritize sequences by length from the shortest to the longest, without a specific order within sequences of the same length, and we generate and try sequences in order of priority. The experience reported in the next section provides some initial evidence of the validity of this choice.

4 Experience results

We evaluated the approach proposed in this paper on some representative Web applications. Here we describe our experience with *Flickr*, a popular Web application that manages photos, and *Google Maps*, the well known Web application to access world maps. In the cases reported in this paper, we proceeded as follows:

- we considered some known and documented failures,
- we produced finite state machine specifications from the (informal) documentation of the failing services,
- we identified the failing sequences,
- we derived equivalent sequences from the finite state machine specifications, and we sorted equivalent sequences according to their length, as suggested in Sect. 3,



Method	Method description
upload()	upload a photo as *public*.
upload(isPublic_OFF)	upload a photo as *private*.
upload(isFamily_ON)	upload a photo as visible to *family* contacts only.
setPerms(isPublic_ON)	set the visibility of a photo to allow anybody to see it.
setPerms(isPublic_OFF)	set the visibility of a a photo to allow nobody to see it.
setPerms(isFamily_ON)	set the visibility of a photo to allow the family contacts to see it

Sequences equivalent to *setPerms(isFamily_ON)* up to length 4			
setPerms(isPublic_ON)	setPerms(isFamily_ON)		
setPerms(isFamily_OFF)	setPerms(isPublic_ON)	setPerms(isFamily_ON)	
setPerms(isPublic_ON)	setPerms(isPublic_ON)	setPerms(isFamily_ON)	
setPerms(isPublic_ON)	setPerms(isFamily_ON)	setPerms(isFamily_ON)	
setPerms(isPublic_OFF)	**setPerms(isPublic_ON)**	**setPerms(isFamily_ON)**	
setPerms(isFamily_OFF)	setPerms(isPublic_ON)	setPerms(isPublic_ON)	setPerms(isFamily_ON)
setPerms(isFamily_OFF)	setPerms(isPublic_ON)	setPerms(isFamily_ON)	setPerms(isFamily_ON)
setPerms(isPublic_ON)	setPerms(isPublic_ON)	setPerms(isFamily_ON)	setPerms(isFamily_ON)
setPerms(isPublic_OFF)	setPerms(isPublic_ON)	setPerms(isPublic_ON)	setPerms(isFamily_ON)
setPerms(isPublic_OFF)	setPerms(isPublic_ON)	setPerms(isFamily_ON)	setPerms(isFamily_ON)
setPerms(isPublic_ON)	setPerms(isFamily_OFF)	setPerms(isPublic_ON)	setPerms(isFamily_ON)
setPerms(isPublic_ON)	setPerms(isPublic_OFF)	setPerms(isPublic_ON)	setPerms(isFamily_ON)
setPerms(isPublic_ON)	setPerms(isFamily_ON)	setPerms(isPublic_ON)	setPerms(isFamily_ON)

Fig. 4 Photo visibility handling in Flickr

– we tried the equivalent sequences in the order of priority, and verified the presence of an effective workaround in the first set of equivalent sequences (within the first eight sequences for the cases presented in this paper).

The experience reported in this paper illustrates both the case of failing sequences that can be deleted from the FSM specifications, and the case of failing sequences that cannot be removed from the FSM specifications without affecting the reachability of the intended final state.

4.1 Flickr visibility

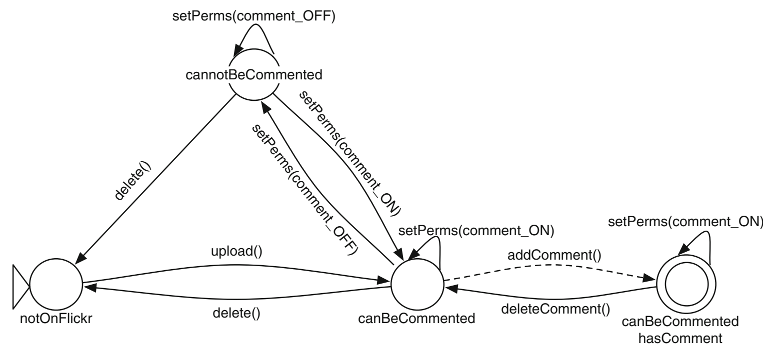
Flickr allows users to upload and share photos on the Web. Photos can have a *public*, *family* or *private* visibility, and according to the description of the Flickr application, users can change the visibility of their photos through the *set-Perms()* function (see Fig. 4).

Here we focus on a problem with function *setPerms()* reported in March 2007. In this case, the *setPerms()* function failed to change the status of a photo from *private* to *family*.[1]

In particular, the failure reportedly occurred with a failing sequence consisting of a single call to the function *setPerms(isFamily_ON)*, which is supposed to change the visibility to *family*, immediately following a successful call to *upload(isPublic_OFF)*, which uploads a photo with initial *private* visibility.

Figure 4 shows the relevant subset of the FSM specifications that we derived from the informal description of the Flickr visibility manager interface, together with the set of top-priority equivalent sequences derived following the approach described in Sect. 3. Given the failing sequence *setPerms(isFamily_ON)*, from initial state *private* to intended final state *family*, we generated equivalent sequences by removing the *setPerms(isFamily_ON)* transition connecting state *private* to *family* from the FSM, and by finding alternative paths from *private* to *family*. We generated 412

[1] http://www.flickr.com/help/forum/36212 and http://www.flickr.com/help/forum/46985



Method	Method description
upload()	upload a photo to Flickr
delete()	delete a photo from Flickr
setPerms(comment_ON)	allow any Flickr user to add a comment to a photo
setPerms(comment_OFF)	do not allow users to comment a photo
addComment()	add a comment to a photo
deleteComment()	delete a comment of a photo

Sequences equivalent to *addComment()* up to length 4			
setPerms(comment_ON)	addComment()		
addComment()	setPerms(comment_ON)		
addComment()	deleteComment()	addComment()	
setPerms(comment_ON)	setPerms(comment_ON)	addComment()	
setPerms(comment_ON)	addComment()	setPerms(comment_ON)	
addComment()	setPerms(comment_ON)	setPerms(comment_ON)	
delete()	upload()	addComment()	
setPerms(comment_OFF)	**setPerms(comment_ON)**	**addComment()**	
setPerms(comment_ON)	addComment()	deleteComment()	addComment()
addComment()	setPerms(comment_ON)	deleteComment()	addComment()
addComment()	deleteComment()	setPerms(comment_ON)	addComment()
addComment()	deleteComment()	addComment()	setPerms(comment_ON)
setPerms(comment_ON)	setPerms(comment_ON)	addComment()	setPerms(comment_ON)
setPerms(comment_ON)	addComment()	setPerms(comment_ON)	setPerms(comment_ON)
delete()	upload()	setPerms(comment_ON)	addComment()
delete()	upload()	addComment()	setPerms(comment_ON)
setPerms(comment_OFF)	setPerms(comment_ON)	setPerms(comment_ON)	addComment()
setPerms(comment_OFF)	setPerms(comment_OFF)	setPerms(comment_ON)	addComment()
setPerms(comment_OFF)	setPerms(comment_ON)	addComment()	setPerms(comment_ON)
setPerms(comment_ON)	delete()	upload()	addComment()
setPerms(comment_ON)	setPerms(comment_OFF)	setPerms(comment_ON)	addComment()
setPerms(comment_OFF)	delete()	upload()	addComment()

Fig. 5 Comment permission handling of public photos in Flickr

equivalent sequences of increasing length and up to 8 calls. The sequence highlighted in bold in Fig. 4 represents a valid workaround, and corresponds to the workaround suggested on the Flickr forum for this problem. The workaround is the fifth sequence according to the length-based prioritization order.

Once the failure is identified and the system is brought back to a consistent fall-back state, which in this case does not require any special operation, the time necessary to generate and try the first five sequences is almost imperceptible from the user viewpoint.

4.2 Flickr comments

In Flickr, only authorized users may add tags, comments, and descriptions to private or family photos, while by default all users can add comments to public photos.

The second problem we consider in our experiments was reported in December 2005, and has to do with Flickr's comment permissions.[2] According to the report, users could not add comments to photos uploaded as public. Following the report, we identified the failing sequence as a single call to *addComment()*, issued immediately after a successful *upload()*.

Figure 5 shows the relevant subset of the FSM specifications that we derived from the informal description of the Flickr comment manager interface, as well as the top-priority, automatically derived equivalent sequences.

In this case, we cannot remove the *addComment()* transition that goes from state *canBeCommented* to *canBeCommented&hasComment* without affecting the reachability of the intended final state *canBeCommented&hasComment*.

[2] http://www.flickr.com/help/forum/15259

setTimeout()	**openInfoWindow()**		
openInfoWindow()	setTimeout()		
setTimeout()	openInfoWindow()	setTimeout()	
GMarker.disableDragging()	GMarker.enableDragging()	openInfoWindow()	
GMap.enableDragging()	GMap.disableDragging()	openInfoWindow()	
disableInfoWindow()	enableInfoWindow()	openInfoWindow()	
GMarker.disableDragging()	GMarker.enableDragging()	openInfoWindow()	setTimeout()
GMap.enableDragging()	GMap.disableDragging()	openInfoWindow()	setTimeout()
disableInfoWindow()	enableInfoWindow()	openInfoWindow()	setTimeout()
disableInfoWindow()	setTimeout()	enableInfoWindow()	openInfoWindow()
GMarker.disableDragging()	setTimeout()	GMarker.enableDragging()	openInfoWindow()
GMap.enableDragging()	setTimeout()	GMap.disableDragging()	openInfoWindow()

Fig. 6 Sequences equivalent to *openInfoWindow()*

Therefore, we start to generate equivalent sequences without changing the FSM. Specifically, we obtain them by inserting indifferent sub-sequences into the failing sequence. We generated 484 sequences up to length 8, and prioritized them by length.

The equivalent sequence shown in bold is *setPerms (comment_OFF) setPerms(comment_ON) addComment()*, and is a valid workaround, which also corresponds to the workaround proposed in the Flickr forum for this problem. The workaround is the eighth sequence according to chosen priority ordering. Also in this case the workaround can be found in negligible time.

4.3 Google Maps draggable markers

Google Maps serves on-line maps that users can explore by dragging their images on the screen with the pointer. Users can also annotate maps with markers, to indicate points of interest, can click on markers to show additional information, and can drag markers to move them on the map. Both maps and markers can be either fixed or draggable.

Here we consider a problem reported in November 2007.[3] Clicking on draggable markers in non-draggable maps does not display the information attached to the marker, as expected.

We reproduced the failure, and we identified the failing sequence as a single call to *openInfoWindow()*. Removing *openInfoWindow()* from the FSM that specifies the application behavior changes the reachability of the intended final state from the fall-back one. So, we generated equivalent sequences by adding indifferent sub-sequences, as in the case of the problem with the Flickr comment manager.

In this case, indifferent sequences can be obtained by composing functions that enable and disable the same map attributes, or by setting timeouts. In the experiment we considered the following functions:

GMap.enableDragging()	enable map dragging
GMap.disableDragging()	disable map dragging
GMap.enableInfoWindow()	enable the display of the information window on the map
GMap.disableInfoWindow()	disable the display of the information window on the map
GMarker.enableDragging()	enable marker dragging
GMarker.disableDragging()	disable maker dragging
setTimeout(time)	insert a delay

By adding combinations of pairs of enable/disable services and timeouts, we generated 140 sequences equivalent to *openInfoWindow()*. Figure 6 shows the fourteen top priority equivalent sequences up to length 4.

In this case the highest priority sequence is a valid workaround, as confirmed by the Google Maps bug-report website.

4.4 Google Maps dynamic loading

The last problem that we describe in this paper is related to dynamic loading of Javascript code in Google Maps. The Javascript code associated with a Web page can be loaded either together with the Web page, or dynamically just before the code is executed. This form of "lazy" loading is highly recommended to reduce the execution overhead for web pages with a lot of Javascript functions, many of which are used only rarely.

Dynamic loading of Javascript code in Google Maps can lead to failures due to the bad initialization of map images within the Safari web browser, which are themselves loaded dynamically on-demand. In particular, Safari shows dynamically loaded maps as grey rectangles, and correctly visualizes the map images only after some user-interface actions such as zooming or dragging.[4]

The failure depends on the invocation of method *setCenter()* that centers the map around a given point on the map. The method is called right after the creation of the map, while Safari has not started checking for the effects of method invocations yet. Thus, Safari does not see the results of this invocation and does not display the map as expected.

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

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

setTimeout()	**setCenter()**		
setCenter()	setTimeout()		
setTimeout()	setCenter()	setTimeout()	
disableDoubleClickZoom()	enableDoubleClickZoom()	setCenter()	
enableContinuousZoom()	disableContinuousZoom()	setCenter()	
mapDragDIS	mapDragEN	setCenter()	
disableDoubleClickZoom()	enableDoubleClickZoom()	setCenter()	setTimeout()
enableContinuousZoom()	disableContinuousZoom()	setCenter()	setTimeout()
mapDragDIS	mapDragEN	setCenter()	setTimeout()
mapDragDIS	setTimeout()	mapDragEN	setCenter()
disableDoubleClickZoom()	setTimeout()	enableDoubleClickZoom()	setCenter()
enableContinuousZoom()	setTimeout()	disableContinuousZoom()	setCenter()

Fig. 7 Sequences equivalent to *setCenter()*

We replicated the failure and identified the failing sequence as a single call to method *setCenter()*. As in the previous case, removing the method from the FSM specification affects the reachability of the final state, and thus we generated equivalent sequences by adding indifferent sequences chosen from the following functions:

GMap.enableDragging()	enable map dragging
GMap.disableDragging()	disable map dragging
GMap.enableContinuousZoom()	Enable continuous smooth zooming
GMap.disableContinuousZoom()	Disable continuous smooth zooming
GMap.enableDoubleClickZoom()	Enable double click to zoom in and out
GMap.disableDoubleClickZoom()	Disable double click to zoom in and out
setTimeout(time)	insert a delay

By adding combinations of pairs of enable/disable services and timeouts, we generated 100 sequences equivalent to *setCenter()*. Figure 7 shows the twelve top-priority equivalent sequences up to length 4.

As in the previous case, the top priority sequence turns out to be a valid workaround.

5 Related work

Automatic fault recovery mechanisms have been historically investigated in the context of fault tolerant systems [19,25], and more recently also in the context of Web applications [26].

Classic fault recovery approaches proposed in the context of fault tolerant systems rely on *redundant components*, *wrappers* or *rejuvenation*.

Redundant components are additional components designed by developers and exploited at run-time. For example, Diaconescu et al. [9] address performance problems by adapting the application to the environment evolution: They assume the presence of several components that offer equivalent services, and automatically select at run-time components with characteristics suitable to meet the required quality of service.

These approaches rely on additional components developed to provide a suitable redundancy to support fault tolerance, while our approach does not require the development of ad-hoc redundant components, but relies on redundancy implicit in the applications, and thus does not incur in extra costs.

Wrappers perform run-time sanity checks on the components I/O to identify possible mismatches and avoid consequent failures. For example, Fuad et al. [11,12] add wrappers to components to ensure consistency among components, and thus prevent possible failures. Similarly to the case of redundant components, the approach is effective, but requires extra development costs.

Rejuvenation or reboot consists of enabling the periodic total or partial reboot of the system to clean up the run-time state. Rejuvenation protects against failures caused by the corruption of the application state (e.g., failures caused by memory leaks). For example, Candea et al. [3,4] propose a technique for enabling micro-reboot, that is the reboot of selected components to benefit from state reinitialization without the overhead of rebooting the whole application. Rebooting can be very effective for some classes of faults, but does not apply well to the faults addressed in this paper.

Much of the recent research on failure-recovery mechanisms for autonomic Web applications exploits techniques developed for fault tolerant systems. Many researchers propose techniques that rely on service brokers to find alternative services equivalent to failing ones. For example, Sadjadi et al. [26] propose to duplicate the development of web services to produce fault-tolerant web applications. Naccache et al. focus on performance problems, and follow the research line indicated by Diaconescu et al. They propose a framework for selecting alternative services to deal with unexpected traffic loads that slow down web services. They rely on multiple services with equivalent behavior but different response time depending on the traffic load, and they change services to

satisfy the required performance. They exploit techniques developed for web based portals [24] and for Ajax-based web applications [23]. Zhang [27] applies the idea of micro-reboot in the context of Web services. These approaches have similar advantages and limitations of classic fault tolerance approaches.

Other researchers are investigating new ideas. Denaro et al. [7,8] focus on integration failures, and propose a self-adaptive approach to automatically detect mismatches between requested and provided web-services, and execute suitable adapters to solve the problems. Gurguis and Zeid [15] define autonomic services, which can be invoked to add self-healing capabilities to classic web services. Liao et al. [20] propose a similar solution based on a federated multi-agent system for autonomically organize and control web services.

Yet other approaches code reactions to failure occurrences in the form of rules with the following format ⟨ *failure, recovery action* ⟩.

Baresi et al.[1] propose *Dynamo*, a framework to augment BPEL specifications with self-healing capabilities. Dynamo monitors the system and evaluates assertions that code functional and non-functional requirements. When assertions are violated, Dynamo executes recovery actions defined by the developers at design time. Recently, Dynamo has been implemented on top of the JBoss rule engine [2].

Fugini et al. [13] propose an architecture for self-healing web applications. Their architecture contains modules for detecting, diagnosing and repairing faults. Similarly to Dynamo, they use a repair rule registry that proposes rules that match type of faults with recovery actions, such as replacing services with similar ones by comparing their WSDL interfaces, changing parameters to adapt services, etc. Similarly, Modafferi et al. [22] propose an approach based on a recovery registry, but augmented with specifications of alternative paths to be followed after a failure occurrence.

These approaches rely on registry provided by the developers to identify recovery actions, while our approach identifies recovery actions at run time without relying on special information provided by the developers.

Equivalent sequences that we exploit as failure workarounds have been proposed in the early nineties by Doong and Frankl [10] to automatically generate test oracles from algebraic specifications, and have been identified by Henkel and Diwan in their approach to reconstruct algebraic specifications from Java code [16].

6 Conclusions and future research directions

We presented an application of the notion of automatic workarounds to Web applications. Automatic workarounds exploit the redundancy of code present in modularized systems to automatically avoid failures and therefore realize a form of self-healing. Specifically, in this paper we present an architecture for automatic workarounds that is particularly suitable to Web applications, together with a method to generate and select workarounds. We also present the validation of these ideas by means of four case studies based on two very popular Web applications. In all these case studies, we found that the automatic generation of equivalent sequences always leads to an alternative sequence that does not incur a failure, which amounts to a valid workaround. Furthermore, we found that this valid workaround is always among the first few sequences selected by our method.

We plan to continue to study and develop the notion of automatic workaround. In particular, we plan to pursue this study in the following three general directions: prioritizing workarounds, using other types of specifications, and expanding the scope of our evaluation and validation in breadth and depth.

In terms of prioritization, so far we have used a simplistic scheme based only on the length of the equivalent sequences (i.e., we prefer shorter sequences). In the future, we plan to study other prioritization criteria used individually or in combinations. Examples of such measures include (1) maximizing the distance between the failed sequence and the workaround, under the assumption that a very different sequence is more likely to avoid the fault that caused the failure in the original sequence; (2) using histories of workarounds or executions, preferring workarounds that were successful in past executions; and (3) using heuristics based on fault classifications, which would select sequences of operations that are typically more effective with the most common faults for the application at hand.

The work described in this paper focuses exclusively on state-based specifications. In addition, we have assumed that such specifications be written by a developer. However, the idea of automatic workaround is not limited to these types of specifications. In the future we plan to develop workarounds for other types of specifications, including, for example, algebraic specifications, which seem to lend themselves to the generation of equivalent input sequences. Also, we plan to experiment with models that are not produced by a developer, but that are instead synthesized from the system behavior. Such automatic-generation techniques exist for various types of models (e.g., state-based [6,21] and algebraic specifications [14,16]).

One crucial limitation of our current evaluation is that we chose the case studies knowing which workarounds would be effective for which failure. In other words, we found automatically what we could already create by hand. We did that to prove that automatic workarounds are indeed possible. In the future, we plan to prove that automatic workarounds are a viable solution more generally, especially for unknown faults. In order to do that, we plan to broaden the range of applications and case studies, and also to develop more

comprehensive experiments. In particular, we plan to experiment with automatic workarounds in the presence of seeded faults, and also in the presence of known (original) faults for which workaround is known.

References

1. Baresi, L., Guinea, S.: Dynamo and self-healing BPEL compositions. In: ICSE COMPANION '07: Companion to the Proceedings of the 29th International Conference on Software Engineering, pp. 69–70. IEEE Computer Society, Washington, DC (2007). doi:10.1109/ICSECOMPANION.2007.31
2. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with dynamo and the jboss rule engine. In: ESSPE '07: International Workshop on Engineering of Software Services for Pervasive Environments, pp. 11–20. ACM, New York (2007). doi:10.1145/1294904.1294906
3. Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., Fox, A.: Microreboot - a technique for cheap recovery. In: OSDI'04: Proceedings of the Sixth Conference on Symposium on Operating Systems Design & Implementation. USENIX Association, Berkeley (2004)
4. Candea, G., Kiciman, E., Zhang, S., Keyani, P., Fox, A.: JAGR: An autonomous self-recovering application server. In: Active Middleware Services, pp. 168–178. IEEE Computer Society, Washington, DC (2003)
5. Carzaniga, A., Gorla, A., Pezzè, M.: Self-healing by means of automatic workarounds. In: SEAMS '08: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 17–24. ACM, New York (2008)
6. Dallmeier, V., Lindig, C., Wasylkowski, A., Zeller, A.: Mining object behavior with ADABU. In: WODA '06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, pp. 17–24. ACM, New York (2006). doi:10.1145/1138912.1138918
7. Denaro, G., Pezzè, M., Tosi, D.: Adaptive integration of third-party web services. In: DEAS '05: Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software, pp. 1–6. ACM, New York (2005). doi:10.1145/1083063.1083088
8. Denaro, G., Pezzè, M., Tosi, D.: SHIWS: A self-healing integrator for web services. In: ICSE COMPANION '07: Companion to the Proceedings of the 29th International Conference on Software Engineering, pp. 55–56. IEEE Computer Society, Washington, DC (2007). doi:10.1109/ICSECOMPANION.2007.66
9. Diaconescu, A., Murphy, J.: A framework for using component redundancy for self-optimising and self-healing component based systems. In: WADS '03: Proceedings of the Workshop on Software Architectures for Dependable Systems. Portland, Oregon (2003)
10. Doong, R.K., Frankl, P.G.: The ASTOOT approach to testing object-oriented programs. ACM Trans. Softw. Eng. Methodol. **3**(2), 101–130 (1994). doi:10.1145/192218.192221
11. Fuad, M.M., Deb, D., Oudshoorn, M.J.: Adding self-healing capabilities into legacy object oriented application. In: ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems. IEEE Computer Society, Washington, DC (2006). doi:10.1109/ICAS.2006.10
12. Fuad, M.M., Oudshoorn, M.J.: Transformation of existing programs into autonomic and self-healing entities. In: ECBS '07: Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, pp. 133–144. IEEE Computer Society, Washington, DC (2007). doi:10.1109/ECBS.2007.74
13. Fugini, M.G., Mussi, E.: Recovery of Faulty Web Applications through Service Discovery. In: SMR '06: First International Workshop on Semantic Matchmaking and Resource Retrieval: Issues and Perspectives. Seoul, Korea (2006)
14. Ghezzi, C., Mocci, A., Monga, M.: Efficient recovery of algebraic specifications for stateful components. In: IWPSE '07: Ninth International Workshop on Principles of Software Evolution, pp. 98–105. ACM, New York (2007). doi:10.1145/1294948.1294972
15. Gurguis, S.A., Zeid, A.: Towards autonomic web services: achieving self-healing using web services. In: DEAS '05: Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software, pp. 1–5. ACM, New York (2005). doi:10.1145/1082983.1083069
16. Henkel, J., Diwan, A.: A tool for writing and debugging algebraic specifications. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, pp. 449–458. Edinburgh, Scotland (2004)
17. Horn, P.: Autonomic computing: IBM perspective on the state of information technology. In: AGENDA 01. Scottsdale, AR (2001). http://www.research.ibm.com/autonomic
18. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003). doi:10.1109/MC.2003.1160055
19. Koren, I., Krishna, C.M.: Fault Tolerant Systems. Morgan Kaufmann Publishers Inc., San Francisco (2007)
20. Liao, B.S., Gao, J., Hu, J., Chen, J.J.: A federated multi-agent system: autonomic control of web services. In: Proceedings of the 2004 International Conference on Machine Learning and Cybernetics (2004)
21. Lorenzoli, D., Mariani, L., Pezzè, M.: Inferring state-based behavior models. In: WODA '06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis, pp. 25–32. ACM, New York (2006). doi:10.1145/1138912.1138919
22. Modafferi, S., Mussi, E., Pernici, B.: SH-BPEL: a self-healing plug-in for ws-bpel engines. In: MW4SOC '06: Proceedings of the First Workshop on Middleware for Service Oriented Computing, pp. 48–53. ACM, New York (2006). doi:10.1145/1169091.1169099
23. Naccache, H., Gannod, G.: A self-healing framework for web services. In: ICWS '07: Proceedings of the 2007 IEEE International Conference on Web Services, pp. 398–345 (2007). doi:10.1109/ICWS.2007.16
24. Naccache, H., Gannod, G.C., Gary, K.A.: A self-healing web server using differentiated services. In: Dan A., Lamersdorf W. (eds.) ICSOC '06: Proceedings of the Fourth International Conference on Service Oriented Computing, Lecture Notes in Computer Science, vol. 4294, pp. 203–214. Springer, Heidelberg (2006)
25. Pullum, L.L.: Software Fault Tolerance Techniques and Implementation. Artech House Inc., Norwood (2001)
26. Sadjadi, S.M., McKinley, P.K.: 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, pp. 76–87. IEEE Computer Society, Washington, DC (2005). doi:10.1109/ICAC.2005.64
27. Zhang, R.: Modeling autonomic recovery in web services with multi-tier reboots. In: ICWS'07: Proceedings of the IEEE International Conference on Web Services (2007). doi:10.1109/ICWS.2007.127