Self-Healing by Means of Automatic Workarounds

Antonio Carzaniga
University of Lugano
Faculty of Informatics
via Buffi 13
CH-6904 Lugano, Switzerland
antonio.carzaniga@unisi.ch

Alessandra Gorla
University of Lugano
Faculty of Informatics
via Buffi 13
CH-6904 Lugano, Switzerland
gorlaa@lu.unisi.ch

Mauro Pezzè[*]
University of Lugano
Faculty of Informatics
via Buffi 13
CH-6904 Lugano, Switzerland
mauro.pezze@unisi.ch

ABSTRACT

We propose to use *automatic workarounds* to achieve self-healing in software systems. We observe that software systems of significant complexity, especially those made of components, are often redundant, in the sense that the same functionality and the same state-transition can be obtained through multiple sequences of operations. This redundancy is the basis to construct effective workarounds for component failures. In particular, we assume that failures can be detected and intercepted together with a trace of the operations that lead to the failure. Given the failing sequence, the system autonomically executes one or more alternative sequences that are known to have an equivalent behavior. We argue that such workarounds can be derived with reasonable effort from many forms of specifications, that they can be effectively prioritized either statically or dynamically, and that they can be deployed at run time in a completely automated way, and therefore that they amount to a valid self-healing mechanism. We develop this notion of self-healing by detailing a method to represent, derive, and deploy workarounds. We validate our method in two case studies.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Error Handling and Recovery*

General Terms

Reliability

Keywords

Self-healing, autonomic computing, workarounds, equivalent behaviors, system redundancy, fault healing

[*]Mauro Pezzè is also professor at the University of Milano Bicocca, Department of Informatics, Systems and Communication, 20126 Milan, Italy

1. INTRODUCTION

We use the term *self-healing* to refer to the ability of a software system to autonomically detect and overcome a significant class of failures of internal as well as external components. For the purpose of this paper, we consider a two-phase self-healing process whereby a failure is first detected and then dealt with by means of an appropriate corrective action. In particular, we focus on the second phase, and assume the availability of a failure-detection mechanism that, in addition to signaling a failure, provides some contextual information regarding the failure.

In response to a failure, we propose to identify and execute one or more *automatic workarounds*. Like most fault-tolerance techniques, what we propose to do with automatic workarounds amounts to exploiting redundancy. However, the kind of redundancy and the methods to exploit it differ from those of traditional fault-tolerance. We discuss this difference in detail in Section 2. The main intuition behind automatic workarounds is that software systems are inherently redundant by virtue of common modularization principles. This means that the interfaces of software components are often designed so as to admit *specification-equivalent* sequences of operations. That is, sequences that have the same intended effect according to the component specification, and therefore that are interchangeable at run-time. Notice that, while the sequences may be equivalent according to the specification, they may not be equivalent in the actual implementation. In fact, the intuition is precisely that, when one particular sequence fails due to a fault, there might be another specification-equivalent sequence that succeeds because it is not affected by the fault or because it avoids the fault completely.

Thus, an automatic workaround is a non-failing alternate sequence of operations that is equivalent (in the sense of the specification) to a failing sequence, and that can be automatically identified and selected at run-time.

As an example, consider a component of a GUI toolkit that implements a "panel" container. Such a component would most likely have a redundant interface that would admit multiple equivalent sequences of interface method calls. For example, one could add a resizable GUI widget, say a button, through a single call to a specialized *add-resizable* method, or perhaps by first using a more generic *add* method and then by making the object resizable through a *reconfigure* method. Yet another option could be to add two objects and then immediately remove one of them.

In this example, the first sequence (a single *add-resizable*) may fail, and the second or third sequences could serve as automatic workarounds. The specialized *add-resizable* method may execute an optimized but faulty insertion algorithm, while the generic *add* method may be correct in that respect. In turn, the *add* method may contain a faulty initialization procedure in the case of an empty container that would result in an incorrect internal state, which could be corrected by the execution of the *remove* method.

In this paper we develop the idea of automatic workarounds. We propose a general architecture for the deployment of automatic workarounds and examine its essential requirements. We then elaborate a method to represent workarounds and to use them at run time, and we review a few methods to derive workarounds from formal specifications. We also discuss possible strategies to prioritize workarounds in order to improve their effectiveness.

We illustrate our ideas through two examples in which we apply automatic workarounds to two popular applications to handle a total of three failures. The results of our study are preliminary in two ways. First, we have used an ad-hoc, manual process to derive equivalent sequences from informal specifications of the components. Second, we did not cover the failing executions through a complete coverage of the component functionality. Instead, we started with *a-priori* knowledge of some specific failures, and then focused only on the specification of equivalent sequences only for those failures. Nevertheless, the study offers an initial positive validation of the idea of automatic workarounds, and opens a promising line of research, which we plan to pursue in the future.

2. CONTEXT AND PRELIMINARIES

The idea of self-healing and the concept of automatic workaround are clearly related to the vast and mature research area of fault-tolerance. Magee and Maibum [11] state that self-healing systems are similar to fault tolerant systems, and they can be modeled using the same techniques. Admittedly, like most fault-tolerance techniques, the idea of self-healing through automatic workarounds amounts to exploiting a form of *redundancy* in a software system [2]. However, there is a fundamental difference between the kind of redundancy used in the automatic workarounds proposed in this paper compared to that one used in classic software fault-tolerance. Also, research in fault-tolerant and self-healing uses similar terms with slightly different meaning. So, before moving forward with the design of automatic workarounds, it is worth pausing to define some fundamental notions, and also to frame automatic workarounds in the context of classic fault-tolerance techniques.

In this paper, we use the terms *fault* and *failure* with their standard software engineering connotation [9]. So, a failure is a behavior of a software component that deviates from the specified, normal behavior (e.g., an infinite loop). When a failure is caused directly or indirectly by the execution of a certain code fragment, we say that the code fragment contains a fault (e.g., wrong end-of-loop condition). A fault, in turn, is caused by a programming *error*. Thus, the term *fault* refers to the software artifact, while the term *error* refers to the process of producing that artifact, and the term *failure* refers to their run-time manifestations.[1]

[1]In the fault-tolerance literature, the term *error* refers to the run-time state of the system that is caused by a fault and that may lead to a failure [12].

A fault-tolerant software system is one that avoids failures despite the presence of faults. Software fault-tolerance techniques can be broadly classified as *N-version programming* or *single-version* techniques. N-version programming exploits redundancy in the code by using multiple, independently designed and implemented versions of a system. The diversity in the designs and implementations should guarantee that fault occurrences do not correlate across the different versions, making it unlikely that two versions would fail on the same input. Therefore faults can be detected and possibly masked by executing the multiple versions of the system on the same input, and by comparing their outputs. The high cost of producing truly independent implementations limit the scope of applicability of N-version programming.

By contrast, single-version fault-tolerance operates on a single design and implementation, and typically exploits redundancy in the time dimension. Specifically, the system can be augmented with "wrappers" to perform additional run-time sanity checks on the parameters and results of some components. A similar use of wrappers has been naturally applied in the area of self-healing systems [4, 5, 6]. Also, the system can be treated with regular preventive actions intended to restore the system or some of its components to a consistent state. This technique is also called *software rejuvenation*, and is implemented, for example, by so-called microreboots [1, 13]. Single-version techniques mask only some classes of faults.

Automatic workarounds can be seen as a hybrid of the two techniques: they operate on a single version of a system, so they do not incur the heavy cost of the development of multiple versions, and yet attempt to exploit redundancy and diversity in the code, which allows them to cover a wider range of faults.

3. AUTOMATIC WORKAROUNDS

We focus on systems made of components, in which components encapsulate state and access-methods, and in which they interact through calls to the methods of their public interfaces. Automatic workarounds are managed within this component-based architecture by a layer that mediates the interactions between components.

In addition to this very general architectural model, we make three fundamental assumptions.

- First, we require that a failure-detection mechanism be present and available within the system. Such a mechanism may be implemented by the components themselves, by component wrappers, or as a general system-wide service. Regardless of the specific architecture of the failure detector, we assume that failures are signaled in the form of exceptions (or an equivalent language device).

- Second, we require that the failure detector be coupled with a basic recovery mechanism that can bring the component back to a consistent state right after the occurrence of a failure.

- Third, we assume that the behavior of components is specified formally so as to allow the synthesis of automatic workarounds. We discuss this requirement in Section 3.1.

Figure 1 illustrates the position of automatic workarounds in a component-based architecture. The figure shows a caller (client) component, at the top of the diagram, calling a method on a called (server) component, at the bottom of the diagram. The self-healing layer monitors the calls and maintains a partial history of calls. As we will see later, this history would be typically represented by a finite model, and is intended to abstract the state of the component. In addition to monitoring calls, the self-healing layer intercepts failure signals, and in response to those, selects and executes workarounds. The occurrence of a failure and the execution of one or more workarounds are invisible to the caller component. However, in case all workarounds fail, the failure is reported up to the caller component.

The selection of workarounds is guided by the combination of the call history maintained by the self-healing layer, and the context of the failure and recovery reported by the failure signal. In particular, the history and the recovery information identify the *failing sequence*, which consists of a sequence of calls of which some have been successful while others (typically the suffix of the sequence) have failed or have been rolled-back.

This execution of a workaround is illustrated in Figure 2. The sequences of letters symbolize sequences of method calls. The figure shows an initial sequence of calls which results in a failure. The automatic recovery executed in response to the failure restores the system by rolling back some calls. The self-healing starts from that point, selects an appropriate workaround that matches the intended semantics of the failing sequence, and executes the workaround to completion.

3.1 Finding Workarounds

Automatic workarounds are sequences of operations that, starting from the state of the component restored by the automatic recovery, have the same effect as the failing sequence. More specifically, workarounds must have the same *intended* effect as the failing sequence, as opposed to the same actual effect (which would be a failure). Thus, the equivalence relation between a failing sequence and a workaround must be verified with respect to the component specification. To avoid ambiguity, we call these *specification-equivalent* sequences.

For the purpose of this paper, we do not formalize the derivation of specification-equivalent sequences for any particular type of specification. Rather, we observe that a number of such derivations are already known for some common specifications, and that it is also simple to generate specification-equivalent sequences by inserting *no-op* operations.

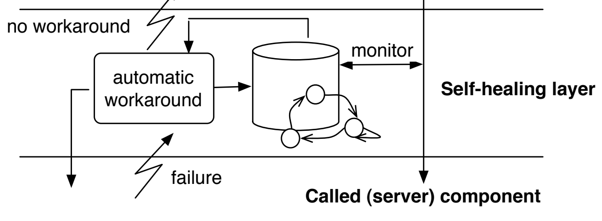


Figure 1: Workarounds in a component-based architecture.

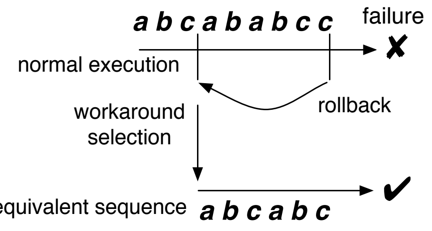


Figure 2: Execution of a workaround.

There are two types of formal specifications that naturally provide a method to derive equivalent sequences. One is algebraic specifications. This kind of specifications is particularly good to model components that implement containers as well as data structures with rich access methods such as a complex query language. Algebraic specifications can easily be used to generate equivalent sequence through term-rewriting. For example, Henkel and Diwan use term-rewriting to simulate algebraic specifications, effectively finding equivalent sequences [8]. Also, Doong and Frankl propose the use of equivalent behavior derived from algebraic specifications to implement test oracles [3]. Their verification technique (for equivalent behaviors) can be used to generate specification-equivalent sequences.

Another class of specifications that lend themselves naturally to the generation of equivalent sequences are various forms of state-based specifications. Examples are finite-state machines and Statecharts [7]. As a general strategy, state-based specifications can generate equivalent sequences whenever they admit different paths from the same starting state to the same end state.

In the absence of a complete specification, it might still be possible to generate automatic workarounds by inserting *no-op* operations into the original sequence. As the name implies, *no-op* operations are sequences that should have no visible effect on the component. The following is a list of general no-op sequences.

- *Do/undo:* these are a pairs of operations in which the second operation reverses the effect of the first. For example, *push/pop* in a stack (assuming enough stack space) or creating and immediately destroying an object (again, assuming enough resources).

- *Timing and scheduling:* these are operations that control the scheduling of the execution of the component, for example by introducing a small delay or by yielding control to the scheduler. Other than potentially delaying the execution of the sequence, these operations should have no visible effect.

- *Environment maintenance:* these are operations on the environment that have absolutely no direct functional effect on the component. Examples are running a garbage collector or clearing a temporary cache.

The idea here is that while a complete or usable specification may not be available, the specification that is available might admit one or more easily identifiable no-op sequences. Notice that the identification of specification-equivalent sequences is a process that takes place off-line, typically at design time or at the time the components are assembled.

3.2 Using Workarounds

Once identified, the sets of equivalent sequences must be somehow packaged and deployed into the self-healing layers of the running system. In our current architecture (see Figure 1) workarounds are triggered by a run-time system that must be able to recognize and select equivalent sequences. The run-time system consists of a wrapper for each component, in which the component is abstracted by a finite-state machine model where each state enables a set of specification-equivalent sequences, and therefore a set of workarounds. State transitions are determined by the calls to the methods of the component interface, and by failure signals. More specifically by the information transmitted by the failure detector along with a failure signal.

So far, we have modeled workarounds in the form of explicit replacement rules. The left side of the rule matches the missing suffix of the failed sequence, and the right side of the rule indicates a possible equivalent sequence. For each state, the same failed sequence could select zero, one, or more workarounds.

This method raises two issues. The first one is how to best represent equivalent sequences within component wrappers for the purpose of using them at run-time. We realize that enumerating explicitly every sequence is impractical and ultimately not scalable. Therefore, we envision a more expressive representation based on a generative model (a grammar) that would provide a compact representation of potentially several equivalent sequences. Although we have not studied this problem in detail, we do not foresee any serious problem in developing an effective grammar for workarounds.

The second question is more fundamental in nature. The issue is that there may be several selectable workarounds for each failure—potentially infinitely many. Therefore, the effectiveness of the whole system depends crucially on a good prioritization of workarounds. This is again an area we have not studied in great detail, but we can nevertheless list a set of heuristic methods that seem worth exploring.

- *Distance:* one way to choose a workaround is to prefer sequences that are as far away as possible from the failing sequence. The intuition is that a good way to avoid failures is to avoid executing the same faulty code. From the perspective of the self-healing layer, the only information available to decide what might execute different code segments, is the sequence of method calls (and their parameters). So, one strategy is to choose a workaround that has the maximum hamming-distance from the failing sequence.

- *Fault classification:* a more informed way to proceed is to rely on a classification of faults. Each sequence could be associated with a distribution of likely classes of faults. The priority of a workaround would then be inversely proportional to the number (or total probability) of faults that are common between the failing sequence and the workaround. Notice that this distance metric, like the previous one, can be computed statically at the time that all the equivalent sequences are identified.

- *History:* another simple idea is to keep track of the history of failures and workarounds, and to give higher priority to those workarounds that were used successfully more often in the past.

- *Fault localization:* yet another approach is to weight method calls by the probability of being faulty, as recently suggested by Lorenzoli et al. [10], and choose workarounds that do not call likely faulty methods.

4. PRELIMINARY EXPERIENCE

We are validating our approach by identifying equivalent sequences for representative applications, and by checking the effectiveness of the identified sequences to avoid failures. In this section we present preliminary experience with Tomcat,[2] a popular open source servlet container, and Flickr[3] a web application to manage and share photos on line. In this preliminary experience, we started with known failures and responsible faults, we isolated the specifications corresponding to the faulty components, we derived equivalent sequences, and we looked for sequences that can serve as workaround for the faults.

In all the cases considered so far and reported in this section, we found many equivalent sequences, some of which provided workarounds that can effectively avoid known failures.

4.1 Tomcat

Versions 6.0.x of Tomcat fail when attempting to load web applications that contain JSP files, and are deployed before starting the Tomcat server.[4] The failure produces the error message `JspFactory.getDefaultFactory()= null`, and is caused by a missing variable initialization fault. When pointing the browser to any of these web applications, Tomcat returns a "503" HTTP error ("the application is not currently available"). However, the exception does not affect the behavior of the servlet container: Tomcat runs correctly and succeeds in loading the other deployed applications. The failure occurs when invoking the sequence of operations

```
deployApp(appWithJSP),
startTomcat()
```

that does not correctly load the `appWithJSP` application.

Figure 3 shows the Statechart specification of the Tomcat Web application loader, which is responsible for loading web applications. Web applications can be deployed either before or after the Tomcat startup. Under default loading options, applications deployed before the startup are loaded during startup, while applications deployed after startup must be loaded with an explicit load command. Web applications can be stopped and restarted while Tomcat is running without affecting the server behavior (*no-op* sequences).

As discussed on Section 3.1, sequences of operations from the same starting state to the same end state are equivalent from the viewpoint of their intended effect. We can easily derive many equivalent sequences from Figure 3. Figure 4 shows some examples.

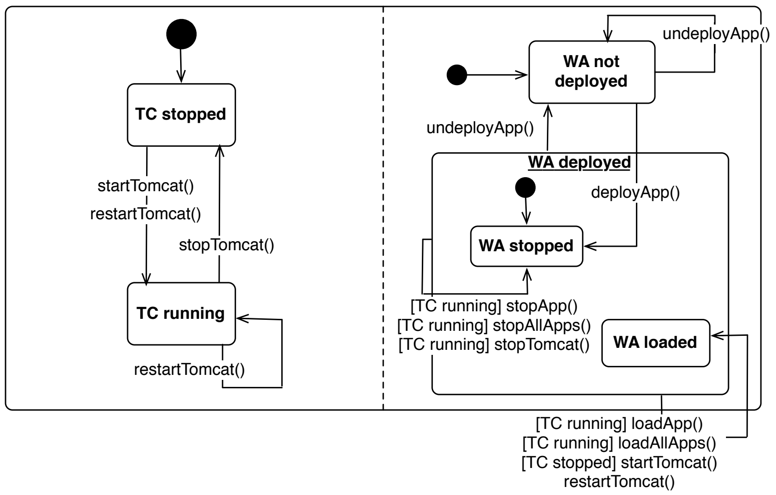
The equivalences in Figure 4 provide effective workarounds that can avoid the null-pointer exception failure of versions 6.0.x described above. The equivalence

```
    startTomcat()
≡
    startTomcat(), loadApp()
```

[2]http://tomcat.apache.org/

[3]http://www.flickr.com/services/api/

[4]Failure reported in October 2006, see http://issues.apache.org/bugzilla/show_bug.cgi?id=40820



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

Figure 3: Statechart specification of the Tomcat Web application loader.

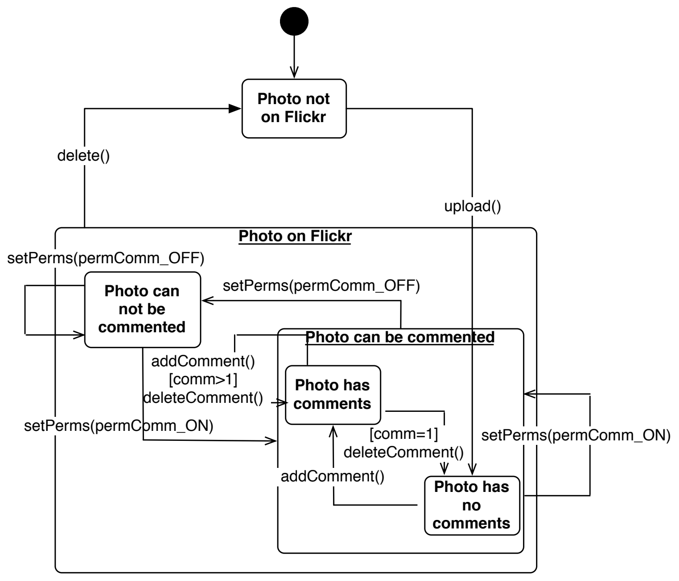
Tomcat stopped and Web Application not deployed	
startTomcat()	≡ restartTomcat()
startTomcat()	≡ startTomcat(), stopTomcat(), startTomcat()
startTomcat()	≡ startTomcat(), restartTomcat()
startTomcat()	≡ startTomcat(), restartTomcat(), stopTomcat(), startTomcat()
restartTomcat()	≡ startTomcat()
restartTomcat()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

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

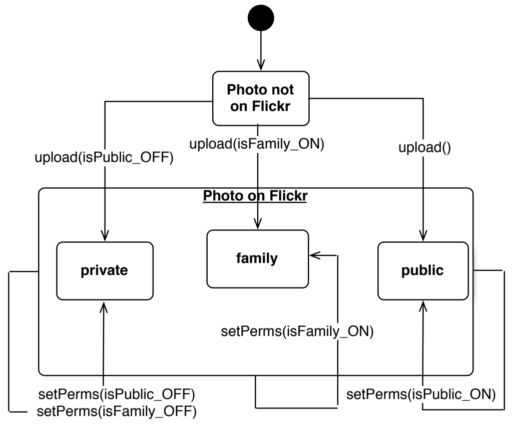
Tomcat stopped and Web Application deployed	
startTomcat()	≡ restartTomcat()
startTomcat()	≡ startTomcat(), stopTomcat(), startTomcat()
startTomcat()	≡ startTomcat(), restartTomcat()
startTomcat()	≡ startTomcat(), restartTomcat(), stopTomcat(), startTomcat()
startTomcat()	**≡ startTomcat(), loadApp()**
startTomcat()	≡ startTomcat(), loadAllApps()
. . .	. . .

Tomcat running and Web Application deployed	
restartTomcat()	≡ stopTomcat(), startTomcat()
restartTomcat()	≡ stopTomcat(), restartTomcat()
restartTomcat()	≡ restartTomcat(), stopTomcat(), startTomcat()
loadApp()	≡ stopApp(), loadApp()
stopApp()	≡ stopAllApp()
stopApp()	≡ loadAllApp(), stopAllApp()

Figure 4: Some equivalent sequences derived from the Statechart in Figure 3.



(a) Comment Manager



(b) Permission Manager

`upload()`	Upload a photo to Flickr. Several optional parameters can be passed to this method. Among others, `is_Public` (ON to make it public, OFF for private), and `is_Family` (ON to make it visible to family)
`delete()`	Delete a photo from Flickr
`setPerms()`	Set permissions for a photo. Several permissions can be set. Among others, `isPublic`, `isFamily` and `permComm` (ON to allow comments on a photo)
`addComment()`	Add comment to a photo
`deleteComment()`	Delete comment from a photo

Figure 5: Settings in Flickr.

Photo not on Flickr	
upload(isPublic_OFF)	≡ upload(), setPerms(isPublic_OFF)
upload(isFamily_ON)	≡ upload(), setPerms(isFamily_ON)
upload()	≡ upload(isPublic_OFF), setPerms(isPublic_ON)
upload()	≡ upload(), setPerms(isPublic_ON)
upload()	≡ upload(), setPerms(permComm_ON)
upload()	≡ upload(), setPerms(permComm_OFF), setPerms(permComm_ON)
. . .	. . .

Photo on Flickr	
setPerms(isPublic_OFF)	≡ setPerms(isFamily_OFF)
setPerms(isFamily_ON)	≡ setPerms(isPublic_OFF), setPerms(isFamily_ON)
setPerms(isFamily_ON)	**≡ setPerms(isPublic_ON), setPerms(isPublic_OFF), setPerms(isFamily_ON)**
setPerms(isPublic_ON)	≡ setPerms(isPublic_OFF), setPerms(isPublic_ON)
setPerms(permComm_ON)	≡ setPerms(permComm_OFF), setPerms(permComm_ON)
setPerms(permComm_OFF)	≡ setPerms(permComm_ON), setPerms(permComm_OFF)
. . .	. . .

Photo can not be commented	
setPerms(permComm_ON)	≡ setPerms(permComm_OFF), setPerms(permComm_ON)

Photo can be commented	
setPerms(permComm_OFF)	≡ setPerms(permComm_ON), setPerms(permComm_OFF)
addComment()	**≡ setPerms(permComm_OFF), setPerms(permComm_ON), addComment()**
addComment()	≡ setPerms(permComm_ON), addComment()
addComment()	≡ addComment(), deleteComment(), addComment()
. . .	. . .

Figure 6: Some equivalent sequences derived from the Statecharts in Figures 5a and 5b.

in state "Tomcat stopped and Web Application deployed" suggests a first alternative to the faulty sequence: The sequence

```
deployApp(appWithJSP),
startTomcat(),
loadApp(appWithJSP)
```

obtained by substituting `startTomcat()` with the equivalent sequence `startTomcat(), loadApp()` provides a valid workaround that avoids the failure.

The equivalence

```
    deployApp(), startTomcat()
≡
    startTomcat(), deployApp(), loadApp()
```

in state "Tomcat stopped and Web Application not deployed" produces the sequence

```
startTomcat(),
deployApp(appWithJSP),
loadApp(appWithJSP)
```

which is another valid workaround that avoids the failure.

This test indicates that we can easily identify many equivalent sequences from state-based specifications (Statecharts in this case), and that equivalent sequences can help avoid some failures.

4.2 Flickr

Flickr is a popular web application to share photos. Within Flickr, photos can be *public*, if visible to all users, *private*, if visible only to the user who uploaded them, or *family*, if visible to the family group of the user who uploaded them. Photos can be associated with descriptions, comments and tags to simplify indexing and searching. Photos can be organized in sets, which can be associated with descriptions and comments.

The Flickr forum reports some failures in early versions of the software distributions. Here, we consider two two of them. The first failure was reported in December 2005, and occurred while uploading photos.[5] The sequence of operations

```
upload(photo),
addComment(photo_id,comment)
```

would correctly upload `photo` as *public*, but with no comment box that generic users could use to add comments, differently from what expected for public photos. The failure was probably caused by the incorrect management of comments permission in the default setting.[6]

The second failure was reported in March 2007, and occurred when trying to change the status of photos uploaded as *private* to *family*.[7] The sequence of operations

```
upload(photo,isPublic_OFF),
setPerms(photo_id, isFamily_ON)
```

would correctly upload `photo` as *private*, but would not modify the status to *family*.

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

[6]Since Flickr does not have a public fault report system, it is difficult to obtain detailed information about faults. Developers usually report only fault presence and fixing, but do not provide further information.

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

Figures 5a and 5b show the Statechart specifications of the Flickr modules responsible for the faults: comment and permission managers, respectively. Figure 6 shows a subset of equivalences for the comment and the permission managers. The equivalence

```
    addComment()
≡
    setPerms(permComm_OFF), setPerms(permComm_ON),
    addComment()
```

in state "Photo can be commented" provides an effective workaround for the first failure, by replacing the sequence

```
upload(photo),
addComment(photo_id, comment)
```

with

```
upload(photo),
setPerms(photo_id, perm_commentOFF),
setPerms(photo_id, perm_commentON),
addComment(photo_id, comment)
```

The equivalence

```
    setPerms(isFamily_ON)
≡
    setPerms(isPublic_ON), setPerms(isPublic_OFF),
    setPerms(isFamily_ON)
```

in state "Photo on Flickr" provides an effective workaround for the second failure by substituting the failing sequence

```
upload(photo,isPublic_OFF),
setPerms(photo_id, isFamily_ON)
```

with

```
upload(photo,isPublic_OFF),
setPerms(photo_id, isPublic_ON),
setPerms(photo_id, isPublic_OFF),
setPerms(photo_id, isFamily_ON)
```

These test cases confirm the possibility of deriving many equivalent sequences from Statechart specifications, and their effectiveness in providing workarounds that avoid known failures.

5. CONCLUSIONS

In this paper we introduced the idea of *automatic workarounds* as a form of self-healing for component-based systems. The main intuition behind automatic workarounds is to exploit code redundancy present in component-based systems, where components export many inter-related functions as well as many specializations and variants of the same function. These versatile component interfaces admit several functionally equivalent sequences of method invocations, which in turn are the basis of automatic workarounds, since a failing sequence could be replaced by an equivalent non-failing one.

Following this original intuition, we developed the idea of automatic workarounds by formulating their general architecture and requirements, and by applying them in a few examples. These examples gave us an initial evidence that automatic workarounds exist and can be effective for some real systems. However, more work needs to be done to validate the idea completely and to refine its implementation.

There are two main directions in which we intend to continue our research. First, we will conduct more extensive case-studies to prove that workarounds are feasible and effective for a significant class of systems and failures. This amounts to studying common failures in component-based systems to evaluate the applicability of automatic workarounds, and can be done by examining some of the many databases of failure reports that are fortunately available in the open-source community. This study starts from the failures and then attempts to derive and model effective workarounds. So, while it can be good to prove the existence of effective workarounds, it would not necessarily prove that such workarounds can be found without prior knowledge of any failure. To prove that, we must show that (1) workarounds can be indeed derived with a reasonable effort and with as much automation as possible from some form of specifications, and that (2) such workarounds can be deployed at run-time, and that (3) failures are indeed avoided by a systematic use of automatic workarounds. We plan to research these questions by developing a complete method and perhaps a tool to support it, and by applying the method to systems of a significant complexity.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

[1] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, Berkeley, CA, USA, 2004. USENIX Association.

[2] R. de Lemos, C. Gacek, and A. Romanovsky. Architectural mismatch tolerance. In *Architecting Dependable Systems*, volume 2677 of *Lecture Notes in Computer Science*, pages 175–194. Springer, 2003.

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

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

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

[6] R. Griffith and G. Kaiser. Manipulating managed execution runtimes to support self-healing systems. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[7] D. Harel and E. Gery. Executable object modeling with statecharts. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 246–257, Washington, DC, USA, 1996. IEEE Computer Society.

[8] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 449–458, Edinburgh, Scotland, May 2004.

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

[10] D. Lorenzoli, L. Mariani, and M. Pezzè. Towards self-protecting enterprise applications. In *ISSRE '07: Proceedings of the 18th IEEE International Symposium on Software Reliability Engineering*, pages 39–48, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[11] J. Magee and T. Maibaum. Towards specification, modelling and analysis of fault tolerance in self managed systems. In *SEAMS '06: Proceedings of the 2006 International Workshop on Self-adaptation and Self-managing Systems*, pages 30–36, New York, NY, USA, 2006. ACM Press.

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

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