Healing Web applications through automatic workarounds

Antonio Carzaniga · Alessandra Gorla · Mauro Pezzè

Abstract We develop the notion of \textit{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 the popular Flickr and Google Maps Web applications.

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

1 Introduction

In a previous preliminary paper [5], we have proposed the notion of \textit{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 fallback 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