A Self-Healing Technique for Java Applications

Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino
Faculty of Informatics, University of Lugano
Lugano, Switzerland
{*carzania,gorlaa,mattavea,perinon*}*@usi.ch*

***Abstract*—Despite the best design practices and testing techniques, many faults exist and manifest themselves in deployed software. In this paper we propose a self-healing framework that aims to mask fault manifestations at runtime in Java applications by automatically applying workarounds. The framework integrates a checkpoint-recovery mechanism to restore a consistent state after the failure, and a mechanism to replace the Java code at runtime to apply the workaround.**

***Keywords*-Self-healing; Checkpoint-recovery; Failure avoidance; Equivalent sequences**

I. Introduction

Reusable components are hard to test in isolation, as it is usually impossible to imagine all the possible scenarios in which they can be used. It is therefore good practice to test the integration of such components within the application that is using them. However, many integration faults may still escape the testing process, and these faults may manifest themselves when the system has been already deployed.

As in our previous work, we present a technique to augment software systems with self-healing capabilities by exploiting the intrinsic redundancy of those systems [1]. Our intuition is that software systems are to some extent redundant, in the sense that the same functionality may be implemented in different ways. For instance, container classes typically offer redundant interfaces, with operations such as *add(element)* and *addAll(collection)* that can be used interchangeably with minor adaptations (for example, *add(x)* can be rewritten as *addAll(collection(x))*). We refer to those sequences of operations, which are functionally equivalent according to the specifications, as *equivalent sequences.* Sometimes this redundancy extends beyond the interface, and two equivalent sequences might actually execute different code. In this case, when one sequence of operation fails, the execution of a supposedly equivalent one might avoid the failure. We refer to such supposedly equivalent sequences that are in fact correct as *workarounds*.

In prior work, we developed the idea of finding and applying workarounds automatically in response to failures [1]. We did that in the context of Web applications, which allow us to make important simplifying assumptions. One such assumption is that the application is stateless. More specifically, we assume that the client side of the application, which consists of JavaScript code embedded in Web pages, is stateless, so we can apply a workaround by simply changing the failing code and then reloading the page. Second, we rely on the user to report a failure. Third, we assume that the fault is located in the current page. Finally, we select and apply workarounds after the Web page is completely loaded and rendered (i.e., when the JavaScript code has run to completion).

We now extend this technique to more general Java applications, where some of those initial assumptions do not hold. As a first and crucial issue, we must consider the possibility that the failure would corrupt the state of the application. Therefore, before applying a workaround, we must restore the application to a consistent state. The second issue is that failures may not have visible effects, or users might be too slow in noticing the problem, or there might not even be a user to act as a failure detector. In practice, if we want to generalize the application of automatic workaround, we also need an automatic failure detection mechanism. Third, the root cause of a failure may be located far from the failure manifestation. So, in order to know where to apply a workaround, we also need a fault localization mechanism. Finally, we have to apply a workaround at runtime, which means changing the code of classes that are already loaded and instantiated in the JVM, without waiting for the application to terminate and restart.

At a high conceptual level, our technique works as follows: We rely on a failure detection mechanism whereby failures can be specified through assertions or some other specific mechanisms [2], [3], and can be signaled through runtime exceptions. We then rely on a checkpoint-recovery mechanism to handle the state of the application. Finally, we rely on equivalent sequences to provide an alternative execution to possibly avoid the failure.

II. Dealing with State and Runtime Updates

Our technique aims to mask the effect of faults in Java applications at runtime, and in particular it aims to avoid functional failures that are caused by faulty interactions between components. These components can be either single classes or third-party libraries. More precisely, we distinguish between *target components,* for which we have a set of equivalent sequences, and *client components,* which are using such target components. In the example of Figure 1, the client component *OrderedList* creates a list of sorted

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

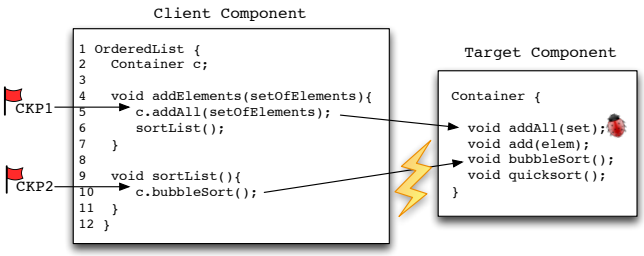

Figure 1. Sample target and client components

items using the target component *Container*, which provides the storing and sorting functionalities.

We assume that a list of equivalent sequences is available for the target components. We thus expect developers (or others) to specify that, for example, adding several elements with the *addAll* method is equivalent to adding each element one at a time with the *add* method, and that *bubbleSort* and *quickSort* are functionally equivalent.

Our technique comes into play whenever the interaction between a client component and a target component leads to a failure. We currently rely on runtime exceptions to detect a failure. To avoid interferences with the intentions of the developer, we ignore exceptions that are already handled within the original program. During the healing phase, a fault-localization mechanism (not yet integrated in our prototype) guides the selection of the equivalent sequences to be tried as workarounds. Then, a checkpoint-recovery mechanism brings the state of the application back to a consistent state, and the healing mechanism changes the code in the client component by replacing some calls to the target component with an equivalent sequence. Finally, the execution restarts with the new code from the recovery point. If the chosen equivalent sequence avoids the failure, the execution goes on without further interruptions. Otherwise, new equivalent sequences can be tried until there are no equivalent sequences left.

As an example, imagine that a set of items are added to the ordered list with *addElements* and sorted with *sortList*, and that a failure occurs during the execution of *sortList*. Imagine the failure is caused by the *addAll* method that creates a wrongly terminated list. However, since we currently do not use any fault localization technique, we have no indication on where the fault is. Our technique first restores the state of the container to the point before the execution of the *sortList* method, and then tries to sort the elements in the container using *quickSort* instead of *bubbleSort*. The execution leads to another exception, and since there are no equivalent sequences left to try for the sorting methods, the state is restored to the point before the execution of the *addElements* method. Then, the *addAll* invocation is substituted with a sequence of repeated *add*, and this time the modified code executes successfully, creating the ordered list and avoiding the failure. With a precise fault localization technique, it would have been possible to immediately apply and execute the second equivalent sequence.

We now present in more detail the checkpoint-recovery and equivalent-sequence substitution mechanisms.

Checkpoint-recovery: It is one of the core components of our technique, as it allows to save the state of the application in chosen points, and then to restore it to those points when a failure occurs. The crucial difficulty is to choose *where* to create checkpoints. Checkpoints should not be too frequent in order to limit runtime overhead and so that they would cover potential equivalent sequences. On the other hand, they should not be too sporadic either, since they may also cover operations such as input/output operations that may be difficult or impossible to roll back. Our technique identifies those points by looking for methods that invoke operations of the target components. We call such methods *Roll-Back Areas* (RBAs), and we create a new checkpoint at the entry point of each RBA. Moreover, to limit the memory overhead caused by checkpoints, it is crucial to include in the checkpoint only those parts of the application that may be affected by the failure. To do this, we use static data-flow analysis to instrument the byte-code such that only the variables that are actually redefined will be saved before their redefinition. Then, a recovery amounts to simply restoring all the variables whose values were saved in the checkpoint.

Applying equivalent sequences: After restoring the state of the application to the entry point of the RBA, our technique replaces part of the code in this area with an equivalent sequence. Unlike dynamic programming languages such as Python, Smalltalk, and JavaScript, that offer the possibility to change the behavior of objects at runtime, Java does not provide such a native capability. To implement it ourselves, we apply the equivalent sequences to the original source code, and then recompile the newly transformed class. Finally, we take advantage of the Java Virtual Machine Tool Interface facilities to replace the body of the RBA method, such that the RBA can be re-executed with a new code and without stopping the application.

References

[1] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds for web applications," in *FSE'10: Proceedings of the 2010 Foundations of Software Engineering conference*. New York, NY, USA: ACM, 2010, pp. 237–246.

[2] L. Mariani, F. Pastore, and M. Pezzè, "Dynamic analysis for diagnosing integration faults," *IEEE Transactions on Software Engineering*, vol. 37, no. 4, pp. 486–508, 2011.

[3] M. Pezzè and J. Wuttke, "Automatic generation of runtime failure detectors from property templates," in *Software Engineering for Self-Adaptive Systems*, ser. Lecture Notes in Computer Science, B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, Eds. Springer Verlag, 2009, vol. 5525, pp. 223–240.