Automatic Recovery from Runtime Failures

Antonio Carzaniga* Alessandra Gorla† Andrea Mattavelli* Nicolò Perino* Mauro Pezzè*

*University of Lugano
Faculty of Informatics
Lugano, Switzerland

†Saarland University
Computer Science
Saarbrücken, Germany

***Abstract*—We present a technique to make applications resilient to failures. This technique is intended to maintain a faulty application functional in the field while the developers work on permanent and radical fixes. We target field failures in applications built on reusable components. In particular, the technique exploits the intrinsic redundancy of those components by identifying *workarounds* consisting of alternative uses of the faulty components that avoid the failure. The technique is currently implemented for Java applications but makes little or no assumptions about the nature of the application, and works without interrupting the execution flow of the application and without restarting its components. We demonstrate and evaluate this technique on four mid-size applications and two popular libraries of reusable components affected by real and seeded faults. In these cases the technique is effective, maintaining the application fully functional with between 19% and 48% of the failure-causing faults, depending on the application. The experiments also show that the technique incurs an acceptable runtime overhead in all cases.**

I. Introduction

Software systems are sometimes released and then deployed with faults, and those faults may cause field failures, and this happens despite the best effort and the rigorous methods of developers and testers. Furthermore, even when detected and reported to developers, field failures may take a long time to diagnose and eliminate. As a perhaps extreme but certainly not unique example, consider fault n. 3655 in the Firefox browser, which was reported first in March 1999 and other times over the following ten years, and is yet to be corrected at the time of writing of this paper (summer 2012).[1] The prevalence and longevity of faults in deployed applications may be due to the difficulty of reproducing failures in the development environment or more generally to the difficulty of diagnosing and eliminating faults at a cost and with a schedule compatible with the objectives of developers and users.

At any rate, dealing with faults that escape the testing environment seems to be a necessity for modern software, and in fact, several lines of research have been devoted to avoiding or at least mitigating the effects of faults in deployed software. A primary example is software fault tolerance. Inspired by hardware fault-tolerance techniques such as RAID [1], software fault tolerance is based on the idea of producing and executing different versions of an application (or parts of it) so as to obtain a correct behavior from the majority (or possibly even just one) of the versions [2], [3].

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=3655

The problem with these fault-tolerance techniques is that they are expensive and are also considered ineffective due to correlation between faults. Therefore, more recent techniques attempt to avoid or mask failures without incurring the significant costs of producing fully redundant code. Among them, some address specific problems such as inconsistencies in data structures [4], [5], configuration incompatibilities [6], infinite loops [7], security violations [8], and non-deterministic failures [9], [10], while others are more general but require developers to manually write appropriate patches to address application-specific problems [11], [12].

In this paper we describe a technique intended to incur minimal costs and also to be very general. The technique works opportunistically and therefore can not offer strict reliability guarantees. Still, short of safety-critical systems, our goal is to support a wide range of applications to overcome a large class of failures. Similarly to other techniques, the main ingredient we plan to use is redundancy. In particular, we propose to exploit a form of redundancy that is *intrinsic* in modern component-based software systems. We observe that modern software and especially reusable components are designed to accommodate the needs of several applications and therefore to offer many variants of the same functionality. Such variants may be similar enough semantically, but different enough in their implementation, that a fault in one operation might be avoided by executing an alternative variant of the same operation. The automatic selection and execution of a correct variant (to avoid a failure of a faulty one) is what we refer to as an *automatic workaround*.

In prior work we have developed this notion of automatic workarounds by showing experimentally that such workarounds exist and can be effective in Web applications [13]. We initially focused on Web applications because they allowed us to make some simplifying assumptions regarding the state and execution flow of the application. In particular, Web applications consist of a user interface built and modified by event-driven procedures that always run to completion, and are also essentially stateless (the state of the application is typically held on the server side). With this execution model, it is relatively easy to apply an automatic workaround by changing the code of one or more procedures and simply re-executing them (by reloading the page). Web applications also simplify the failure-detection problem, since the user can be assumed to detect failures and explicitly request workarounds.

In this paper we present a technique to apply automatic workarounds to general-purpose, possibly long-running applications. The technique is implemented for Java applications but makes little or no assumptions about the nature of the application. We make the very reasonable assumption that the application uses components taken from libraries, and for simplicity in the following discussion we consider one such library per application. The only two significant assumptions we make are that the library comes with a specification of the equivalence between the operations it supports, and that failures are somehow detected and reported.

The equivalence specifications may be written by the programmer of the library or by a knowledgeable user of the library. In any case, they need to be written only once for each library. (In Section III we briefly discuss the possibility of deriving these specifications automatically.) As in our prior work, we express those specifications with code-rewriting rules that are supposed to produce semantically equivalent code.

As for failure detection, our technique is almost completely independent of the particular detection and reporting mechanism. In practice we use exceptions as a reporting mechanism, and we experimented with implicit failure detectors such as run-time exceptions, as well as with more sophisticated detectors based on derived application-specific invariants.

At a high-level, the technique works as follows: when a failure is detected, the state of the application is restored to a previous checkpoint, then the code of the application is dynamically changed to replace a chosen code fragment that contains a call to the library with a potential workaround, and then the execution is restarted from the checkpoint with that new code.

A bit more in detail, the technique consists of an off-line preprocessing that also instruments the application with the necessary machinery to apply workarounds at runtime. The preprocessor identifies (through static analysis) those sections of the application code in which the application calls the library in ways that could potentially be rewritten and in which failures may be detected. We refer to those sections of application code as *roll-back areas (RBAs)*. The preprocessor then produces alternative versions of each RBA using all the applicable rewriting rules. These versions are then compiled and stored for possible use at runtime. The preprocessor also instruments the application code by wrapping each RBA in a loop. The loop starts with a checkpoint of the state of the application, then proceeds with the execution of the RBA code, and then iterates in case of failures. The iteration restores the state to the initial checkpoint, replaces the code of the RBA with one of the alternative versions, and resumes the execution with the new version.

We demonstrate and evaluate our technique by applying it to two popular libraries and four applications that use those libraries. The first library is *Guava*, Google's "core" library for collections, caching, string processing, and more; the second library is *JodaTime*, a utility to process and convert dates and time across multiple and diverse calendar systems. The applications are an e-book converter called *Fb2pdf*, a framework for Java microbenchmarks called *Caliper*, a search results clustering engine called *Carrot2*, and a JavaScript source-to-source compiler called *Closure*. We evaluate the effectiveness of our technique by measuring its ability to handle real as well as injected faults and also by measuring its runtime overhead. In summary, our experiments demonstrate that the technique is effective while incurring an acceptable overhead. In particular, the technique maintains the applications fully functional with between 19% (Fb2pdf) and 48% (Carrot2) of the failure-inducing faults with a run-time overhead of between 2% (Caliper) and 194% (Closure).

II. A Motivating Example

JodaTime is an open-source Java library of utilities to deal with dates and time. It provides full support for several calendars, it allows to easily represent dates and intervals, and to easily parse and format dates.

In May 2011 a developer reported issue n. 3304757 with the JodaTime library.[2] As it turns out, that issue was fixed within a short period of time, but the issue is still interesting because of the nature of the fault and the resulting failure.

Issue n. 3304757 reported a failure resulting in an exception when trying to get the instant corresponding to the beginning of the day on certain dates in countries that observe daylight saving time (DST). This failure could not be easily obtained in testing since it is triggered only under particular conditions, namely in regions where the DST leap occurs over midnight (e.g., America/Sao_Paulo every year, some years in US regions and some Latin America regions).

```
1 DateTimeZone tz = DateTimeZone.forID("America/Sao_Paulo");
2 DateTimeZone.setDefault(tz);
3 // Gets a new instance of datetime with
4 // the current date and time
5 DateTime dt = new DateTime();
6 // Method call causing an exception in specific
7 // days and time zones.
8 DateTime startDay = dt.millisOfDay().withMinimumValue();
```

Listing 1. Code failing on 2011-10-16 due to issue n. 3304757

Due to this issue, the code in Listing 1 would fail with an exception on the day when daylight saving starts (for instance, 2011-10-16). The reason of the failure is that the invocation of dt.millisOfDay().withMinimumValue(), which is a way to get the representation of the beginning of the day, would return 2011-10-16 00:00:00.000 on 2011-10-16, but that instant does not exist on 2011-10-16 because that day begins at 01:00:00.000 due to the DST leap.

Thus issue n. 3304757 would cause any Java application using JodaTime in similar ways to fail, unless the developer had also put in place a proper and specific exception-handling procedure. However, that is unlikely because the failure scenario is somewhat obscure and hard to anticipate and test for. One might argue that cases like this one are rare, but while

[2] http://sourceforge.net/tracker/?func=detail&aid=3304757&group_id=97367&atid=617889

this specific case is in and of itself rare, it is also indicative of many similar "special" cases or features whose implications are not fully understood and therefore not properly handled.

```
1 // failing operation
2 DateTime beginDay = dt.millisOfDay().withMinimumValue();
3 // workaround 1
4 DateTime beginDay = dt.toDateMidnight().toDateTime();
5 // workaround 2
6 DateTime beginDay = dt.withTimeAtStartOfDay();
```

Listing 2. Workarounds for issue n. 3304757 of JodaTime

Fortunately however, JodaTime offers several different ways to get the instant representing the beginning of the day. As shown in Listing 2, there are at least two alternative ways to obtain the same expected result that do not suffer from that failure. Such alternative operations could be easily documented and therefore be used as automatic workarounds at runtime.

III. Intrinsic Redundancy in Reusable Components

The foundational idea of our technique is that failures might be avoided with workarounds consisting of operations or sequences of operations that are equivalent to the failing ones but that do not suffer from the same failures. More specifically, a workaround consists of operations whose effect is equivalent to the *intended* effect of the original failing sequence, but whose implementation does not fail and therefore must be to some extent different. In other words, workarounds require redundancy: different code to do the same thing. Moreover, contrary to classic N-version programming, we expect this redundancy to be available at no cost, since we argue that this redundancy is an intrinsic property of modular software.

It is therefore natural or even necessary to ask whether modern software is indeed intrinsically redundant, and if so to what extent. And if the extent is significant, it would also be interesting to find out why. In this section we provide some qualitative arguments, with references to other studies (including our own), that, together with the direct evidence assembled during the experimentation we carried out for this work, lead us to believe that intrinsic redundancy exists and is significant enough to be exploited. A more thorough study of the prevalence and nature of intrinsic redundancy is the subject of another research we are currently pursuing.

Redundancy in software has been already studied, especially to identify and remove code clones. These are code fragments that are syntactically similar or even identical to other fragments in the same application. Code clones are considered harmful for maintenance [14], [15] and are not exactly the kind of redundancy we expect to find. However, more recent studies in this and related areas also reveal more or less directly that *semantic* equivalence is also present and natural in software. Hindle et al. show that even with today's rich programming languages, developers tend to write simple and repetitive code [16]. More importantly, Jiang and Su found that several code fragments are semantically equivalent even though they are syntactically different [17].

There are also plausible qualitative explanations for the intrinsic redundancy of software, especially in the case of libraries of reusable components. One is that a library might maintain various versions of the same components for backward compatibility. For example, the Java-6 standard library contains 45 classes and 365 methods that are deprecated and that duplicate exactly or almost exactly the same functionality of newer classes and methods. Another one is that modern libraries are designed to offer many flavors of the same or very similar functionalities. For example, the popular library *JQuery* for Web applications offers many methods to display elements in a Web page: *fadeIn()*, *show()*, *fadeTo()*, *animate()*, etc. These differ in some details, but are essentially equivalent in terms of the end result. Another example is when the same functionality is duplicated in different libraries. For example, the logging functionality of the Apache library *log4J* implements the same logging functions of the standard Java library (*java.util.Logging*), and several other similar examples of replicated functionality are found in the Apache *Ant* project. Yet another reason to have different variants of the same functionality is to address different non-functional requirements or different use cases. For example, the GNU Standard C++ Library implements its basic (stable) sorting function using the insertion-sort algorithm for small sequences, and merge-sort for the general case. Similarly, other functions may be implemented in two or more variants, each one optimized for a specific case (for instance, memory vs. time).

In our own prior work we also found that redundancy exists and is usable in some very popular libraries for Web applications [13]. With this paper we add more evidence of that kind for other types of libraries and applications. As shown in the example of Section II, the JodaTime library offers a number of redundant methods to obtain the time of the beginning of a day. However, that is by no means the only case of a redundant functionality in JodaTime. For instance, the *DateTimeFormatter* class provides several methods to parse dates and time (*parseDateTime*, *parseLocalDateTime*, *parseLocalTime*, and *parseLocalDate*). Most of these operations are equivalent only under special conditions (for instance, *parseLocalDateTime* and *parseLocalDate* can be used interchangeably only if the time is not specified) but it is still usually possible to find at least an alternative method to execute in many conditions.

Beyond these exemplary cases, we methodically studied and documented the redundancy of the JodaTime and Guava libraries, which we use as the basis for the experimental evaluation presented in Section V, as well as the widely used SWT graphical user interface library. We identified equivalences between sequences of library calls by first reading the documentation and later by testing each equivalence experimentally. We did not examine the code of the libraries, but instead we manually validated each rule by considering its observable effects.

Notice that this manual derivation and validation can be aided by some form of automation, for example by selecting equivalent operations based on their behavior under the available tests. However, a full automation would be counter to the

TABLE I
EQUIVALENT SEQUENCES FOUND IN REPRESENTATIVE JAVA LIBRARIES

Library	Guava	SWT	JodaTime
Classes considered	116	252	12
Total equivalences found	1715	1494	135
Average per class	14.78	5.93	11.25

notion of workarounds that we seek to support. Recall in fact that we are interested in finding operations that are equivalent in their intended behavior, but not exactly equivalent in their actual observable behavior.

Table I shows the results of our analysis. We report the number of classes analyzed for each library (all SWT classes, a selected set for Guava and JodaTime) and the number of equivalent sequences that we found at the method level, in total and on average per class.

Notice that having many equivalences does not necessarily imply that a system is highly redundant. In fact, several equivalences may correspond to a shallow redundancy that exists only at the interface level, with different calls executing the same code after an initial veneer of interface code. Still, our experience with Web applications [13] and even more so the experiments presented in this paper in Section V show that a significant number of equivalent sequences do indeed correspond to redundant behaviors that can be harnessed and used productively to avoid failures.

IV. IMPLEMENTATION

We now present the architecture of *ARMOR*, a system that implements our automatic-workaround technique.[3] ARMOR works in the following general scenario: a Java application fails because of faults in one of the libraries it uses. Such faults may trigger a failure in the library code or in the application code. ARMOR, which is embedded within the application and is notified of the failure, reacts to the failure by first restoring the state of the application to a previously set checkpoint, and then by selecting and executing an equivalent sequence of operations that might avoid the failure. If multiple equivalent sequences are available, ARMOR selects the one that was most successful for past failures. ARMOR iterates this process until it obtains a valid workaround (i.e., a failure-free execution) or until there are no more equivalent sequences left to try. In the first case the execution of the application proceeds as if no failure occurred. In the second case ARMOR forwards the failure (an exception) to the application code as if ARMOR did not exist.

ARMOR works in two phases and with two main components. In a first off-line phase, a preprocessor component analyzes the application to identify where workarounds might be applied, pre-compiles all alternative sequences, and instruments the application with the necessary code to select those alternative sequences at runtime in response to a failure. At runtime, ARMOR records the state of the application at chosen checkpoints set before potential workarounds, and then reacts to failures by selecting and activating workarounds. We now detail all these preprocessing and runtime activities.

[3]http://star.inf.usi.ch/armor/

A. Preprocessing

The preprocessor starts with the source code of the application, the binary distribution of the libraries, and the specification of the rewriting rules for each library. The preprocessor (1) identifies the units of code to which workarounds might be applied, which we call *roll-back areas (RBAs)*, (2) instruments them with the necessary code to set checkpoints and to react to failures, and (3) compiles and stores the RBA variants to be used as potential workarounds at runtime.

1) Identifying Roll-Back Areas: A roll-back area (RBA) is the primary structural element of the application upon which ARMOR operates. We define a roll-back area as a segment of the application code within which (1) the application calls one or more operations of any one of the libraries, and (2) a failure can be detected and reported. Ideally, a roll-back area should also be *minimal* in the sense that it should be confined to operations that might fail and that could be replaced with a workaround. This is because the execution of any other code, before or after the library calls, might invalidate the checkpoint (for instance, with I/O operations, see Section IV-B1) and in any case would increase its runtime overhead.

In principle, a roll-back area may extend over sections of the application code at any level of granularity, from a single statement to a basic block to an entire method, and conceivably also across methods. However, our current implementation of ARMOR supports two types of extents for roll-back areas: a whole method body and a single initialization expression for a field (static or not). The extent of a roll-back area is constrained by the mechanism that ARMOR implements to dynamically replace the code of the RBA with one of its variants. This mechanism that we describe in detail in Section IV-B2 is capable of replacing only entire methods and thus requires every RBA to be encapsulated as a method. Thus the ARMOR preprocessor identifies RBAs consisting of a method body, which do not need additional encapsulation, and field initialization expressions, which need to be encapsulated through an ad-hoc additional method.

It is also conceivable to encapsulate RBAs consisting of blocks of instructions. However, the encapsulation of such RBAs poses a number of technical problems that we ultimately decided to avoid. Among these problems, the most significant one is the handling of local variables along with the application state that those variables might refer to.

Listing 3 shows a simple application intended to illustrate the preprocessing performed by ARMOR. This application uses the JodaTime library to get the instant corresponding to midnight of the current date, and would fail on specific dates and time zones because of the issue described in Section II. In this example, in the first step of preprocessing, ARMOR would identify three roll-back areas. These are the initialization of field tz on line 2, the initDayAndZone method on line 5, and the setMidnight method on line 12, since they all contain at least one invocation to the JodaTime library.

```
1  class CurrentMidnight {
2    DateTimeZone tz = DateTimeZone.forID("America/Sao_Paulo");
3    DateTime midnight;

5    public void initDayAndZone(){
6      DateTimeZone.setDefault(tz);
7      DateTime dt = new DateTime();
8       ...
9      setMidnight(dt);
10   }

12   private void setMidnight(DateTime dt){
13     midnight = dt.millisOfDay().withMinimumValue();
14   }

16   public DateTime getMidnight(){
17     return midnight;
18   }
19 }

21 class Main {
22   public static void main(String args[]){
23     ...
24     CurrentMidnight cm = new CurrentMidnight();
25     cm.initDayAndZone();
26     ...
27     cm.getMidnight();
28     ...
29   }
30 }
```

Listing 3. Example application code

Notice that ARMOR supports nested RBAs, that is, RBAs that invoke other RBAs. Method initDayAndZone is an example of a nested RBA, since it uses JodaTime directly, and invokes setMidnight, which is itself an RBA.

2) RBA Encapsulation and Proxy Methods: Once the RBAs have been identified, the preprocessor encapsulates and instruments them to allow them to be dynamically replaced with alternative variants at runtime. The encapsulation applies only to RBAs consisting of initialization expressions, which must be rewritten as methods. Once each RBA is encapsulated as a method, the preprocessor creates a proxy method for each RBA method. The role of the proxy is to set the checkpoint and then call the original RBA method and to respond to potential failures.

Listing 4 shows the encapsulation of initialization expressions (method tz_init_original on line 3) and the proxy methods created for two of the three RBAs identified in the example of Listing 3. For the sale of readability, we omit some details and show a simpler code than what is produced by ARMOR.

Let us consider the RBA that consists of the original method setMidnight. ARMOR renames this method to setMidnight_original and then creates a proxy method called setMidnight with the same signature as the original method. In particular, the proxy method must also declare and handle the same exceptions declared and handled by the original method. In this respect, ARMOR distinguishes between checked and unchecked exceptions. (In Java, exceptions can be either checked or unchecked.) Checked exceptions are invalid or simply special conditions that are explicitly declared as potential

```
1  class CurrentMidnight {
2    DateTimeZone tz = tz_init();
3    public DateTimeZone tz_init_original() {
4      return DateTimeZone.forID("America/Sao_Paulo");
5    }
6    public DateTimeZone tz_init() {
7      try {
8        create_checkpoint();
9        return tz_init_original ();
10     } catch (Exception ex) {
11       while (more_rba_variants_available) {
12         try {
13           restore_checkpoint();
14           load_new_rba_variant();
15           return tz_init_original ();
16         } catch (Exception ex1) {
17           // record variant failure and adjust priorities
18           ...
19         }
20       }
21       throw ex;
22     } finally {
23       discard_checkpoint();
24     }
25   }
26   DateTime midnight;
27   ...
28   // initDayAndZone proxy method not shown
29   ...
30   public void setMidnight_original(DateTime dt) {
31     midnight = dt.millisOfDay().withMinimumValue();
32   }
33   public void setMidnight(DateTime dt) {
34     try {
35       create_checkpoint();
36       setMidnight_original(dt);
37     } catch (Exception ex) {
38       boolean success = false;
39       while (!success && more_rba_variants_available) {
40         try {
41           restore_checkpoint();
42           load_new_rba_variant();
43           setMidnight_original(dt);
44           success = true;
45         } catch (Exception ex1) {
46           // record variant failure and adjust priorities
47           ...
48         }
49       }
50       if (!success) throw ex;
51     } finally {
52       discard_checkpoint();
53     }
54   }
55   ...
56 }
```

Listing 4. Result of preprocessing (simplified)

outcomes of the calls to library functions. These are exceptions that the application code must deal with explicitly, either by handling them or by passing them up the stack. By contrast, unchecked (or runtime) exceptions are unexpected conditions that may or may not be handled explicitly by the application.

ARMOR ignores checked exceptions, since it is the responsibility of the programmer to handle those, and in many cases those may well represent a normal path of execution for the application. Therefore, masking those exceptions may interfere with the correct behavior of the application. On the

other hand, unchecked exceptions typically represent failures, and therefore ARMOR catches them and responds to them. In practice, the proxy method catches all exceptions with a generic catch statement, but must also explicitly catch all the specific exceptions thrown by the original method only to immediately re-throw them to the application.

In addition to handling the exceptions thrown by the original RBA method, the proxy handles the state of the application (and the library) in the execution of the original RBA and of potential workarounds. In particular, the proxy sets a checkpoint for the state of the application immediately before the execution of the RBA (line 35). Then, in case of failure, the proxy restores the state to that checkpoint (line 41) before trying an alternative variant. Before terminating, the proxy discards the checkpoint.

3) RBA Variants: For each identified roll-back area, the ARMOR preprocessor produces a series of alternative variants of the application code by applying the rewriting rules of each library used within that roll-back area. These are variants of the original application methods as well as of the ad-hoc methods produced by ARMOR to encapsulate initialization expressions. In practice, referring to the example of Listing 4, these are all the methods with the _original name suffix. The preprocessor produces one variant for each application of a single rewriting rule. The preprocessor then pre-compiles all the RBA variants and stores the bytecode in a database for potential retrieval and use at runtime.

B. Runtime Operations

After the preprocessing phase, the application can be compiled and deployed. ARMOR does not run any special component alongside the application, so the execution of the instrumented application differs from the original one only in the execution of the proxy methods. ARMOR assumes the existence of a failure detector, which may be implemented as a separate autonomous component. However, for the purpose of this paper we assume a typical lightweight failure detection based on assertions or at a minimum on runtime exceptions.

In practice, the most significant difference in the execution of the instrumented application is the checkpointing of the application state performed within the proxy methods. In the case of successful execution, this is also the only difference.

1) Checkpointing and Restoring Application State: ARMOR implements an ad-hoc mechanism to checkpoint and restore the state of the application during execution. The high-level semantics of this mechanism is that of a classic checkpointing mechanism: a checkpoint can be set during the execution of the application, then later the checkpoint may be restored, in which case the state of the application is brought back to what it was at the time the checkpoint was set. The same checkpoint may be restored multiple times.

Since RBAs may be nested and several workarounds might be tried at different levels of the execution of the application, ARMOR maintains a thread-local stack of active checkpoints. Every time a proxy method sets a checkpoint, for instance line 35 in Listing 4, ARMOR pushes a new checkpoint handle on the stack.

ARMOR implements two interchangeable types of checkpoints, one based on a snapshot taken before the execution of the RBA code, and one based on a lazy change-log recorded during the execution of the RBA code.

The first mechanism takes a snapshot of the portion of the application state that might be modified by the execution of a roll-back area at the time the checkpoint is set. In its basic form, the snapshot consists of the transitive closure of all the objects reachable from the object on which the RBA method is called (i.e., this object) plus the parameters to the RBA method and all the static fields that are accessible by the RBA method. The second mechanism uses a change-log whereby the first time a field is written (static or not, primitive values as well as references) the previous value of that field is recorded in the change-log, so that it can be restored later.

The two mechanisms have complementary advantages and disadvantages. The snapshot saves every value that is part of the application state and that is accessible by the RBA at the checkpoint, regardless of whether it is actually modified by the execution of the RBA. The change-log saves every value that is actually modified, regardless of whether that value is part of the application state at the checkpoint. So, the snapshot incurs a potentially high cost at the time the checkpoint is set, but then incurs no cost during the execution of the RBA. Conversely, the change-log incurs no initial cost when the checkpoint is set but may incur a high cost during the execution of the RBA. Both mechanism may be improved through static analysis, although in our current implementation of ARMOR we only applied such analysis to the snapshot method, in order to exclude from the transitive closure those objects that are for sure never modified in the execution of the RBA. In Section V we analyze the performance of both mechanism in all our experiments.

2) Replacing Code: When a failure occurs within an RBA, the proxy method replaces the code of the RBA with a variant of that RBA. ARMOR implements this dynamic code replacement by substituting the code of the whole class that contains the RBA, since this is the only way that code can be dynamically redefined in Java. This is done using the *redefineClasses* method of the *java.lang.instrument* package.

ARMOR selects one of the pre-compiled classes produced by the preprocessor for that RBA. Each pre-compiled class is derived from the original instrumented class (with proxy and original methods) with only the original method changed. The effect of reloading such a class is to change the original method of the present RBA, and in particular all future calls to that method. This redefinition does not affect the execution of methods of the class that are active (on the current stack). In particular, it does not affect the execution of the proxy method, which is in fact the one that initiates the class redefinition. In practice, referring to the example of Listing 4, after the replacement of the RBA code (line 42), the call to the RBA method (line 43) will execute the new RBA variant, different from that executed previously (line 36 or line 43).

V. Experimental Evaluation

We now present the results of the experimental evaluation of the ARMOR system and the technique it embodies. The objective of this evaluation is to determine whether the technique is effective in making applications more resilient to faults, and efficient enough to be practically usable. In answering the first question, we also indirectly provide evidence that modular software is to some significant extent intrinsically redundant.

We conducted this evaluation with four non-trivial applications and two non-trivial libraries affected by real and seeded faults. The two libraries are

- *JodaTime:*[4] a library of utility functions to represent and manipulate dates and time.
- *Guava:*[5] the Google "core" library for collections, I/O, caching, concurrency, string processing, etc.

The four applications are

- *Fb2pdf:*[6] a command-line utility to convert files from the FB2 e-book format into PDF. Fb2pdf uses the Java date/time library but we changed it to use the fully compatible JodaTime library.
- *Carrot2:*[7] a search results clustering engine. Carrot2 uses the Guava library.
- *Caliper:*[8] a framework for writing, running and viewing the results of Java microbenchmarks. Caliper uses the Guava library.
- *Closure:*[9] a source-to-source optimizing JavaScript compiler. Closure uses the Guava library.

As a first preliminary step in our experiments, we wrote the equivalence specifications for Guava and JodaTime based on their respective API documentation. Of the 1000+ classes of the Guava v.12 library, we analyzed the 116 classes in the collections package that also come with an API documentation, including all the classes used in Carrot2, Caliper, and Closure. Of the 149 classes of the JodaTime v.2.1 library, we analyzed 12 classes, including all the ones used in Fb2pdf. For both libraries we identified several equivalences between sequences of calls (see Table I). Then, focusing on the relevant equivalences for our experiments, we abstracted and formalized those equivalences through code-rewriting rules. In total, we compiled 63 rewriting rules for Guava, and 100 for JodaTime. With these rewriting rules, we ran the ARMOR preprocessor to identify the roll-back areas and to produce their variants.

TABLE II
Results of the preprocessing on the selected applications

Application	*Caliper*	*Carrot2*	*Closure*	*Fb2pdf*
Total RBAs	130	139	2099	17
RBAs with variants	60	106	687	17

[4]http://joda-time.sourceforge.net/
[5]http://code.google.com/p/guava-libraries/
[6]http://code.google.com/p/fb2pdf/
[7]http://project.carrot2.org/
[8]http://code.google.com/p/caliper/
[9]http://code.google.com/p/closure-compiler/

The results of the preprocessing are reported in Table II. Notice that not every identified RBA (for example, a method that uses the library) would have multiple valid variants. This is because there might be no applicable equivalences, or because the application of the rewriting rules, which are simple textual pattern-substitution rules, sometimes produces invalid code.

We first conducted an evaluation of ARMOR on a limited number of real faults affecting the JodaTime library and the Fb2pdf application, and then we conducted a more extensive evaluation using seeded faults with both libraries and all four applications.

A. Real Faults in JodaTime

We analyzed three real faults reported for JodaTime.

- Issue n. 1375249 reports that if a YearMonthDate (JodaTime class) is created with a Calendar (java.util.Calendar class) as a parameter, the method plusDays() (part of the class YearMonthDate) throws an IllegalArgumentException when the resulting date is in the next year. A workaround exists for this fault: YearMonthDay must be constructed from a Calendar by explicitly specifying an ISOChronology.
- Issue n. 3072758 reports that the method parseDateTime of the class DateTimeFormatter (JodaTime class) fails to parse a DST leaping date even if the LenientChronology is specified (which means it should be tolerant to DST leaps). A workaround exists: by using parseLocalDateTime instead of parseDateTime.
- Issue n. 3304757: see Section II.

For all three of these faults, ARMOR was able to find and execute a valid workaround.

B. Mutation Analysis

In order to obtain a more extensive coverage of the features of the libraries and of ARMOR itself, and also to obtain more statistically significant results, we turned to a systematic mutation analysis. We proceeded as follows:

1) We used the *Major* mutation analysis framework [18] to inject faults in the two libraries. We did that by activating all types of mutations supported by Major.
2) We then ran all the applications with the mutated versions of their respective libraries, and we traced those executions. For each application we obtained an input that we deemed representative for the application. For Fb2pdf we used a third-party e-book file, for Carrot2 and Caliper we used inputs provided by the developers for demonstration purposes, for Closure we used a large and popular JavaScript library (jQuery). Based on the execution traces, we then discarded the mutants that were never executed.
3) We activated each remaining mutant individually and executed all applications in the presence of each mutant. For each application and mutant, we observed and categorized the outcome of the execution as *error*, *loop*, and *success*, when the execution led to an error or exception,

an infinite loop, or a normal termination, respectively. Of the mutants in the *success* categories we further analyzed and distinguished mutants whose execution produced the expected output, which we classified as *equivalent* and that we discarded, and mutants whose execution failed to produce the expected result, which we classified as *non-equivalent.*

4) We then executed all applications instrumented with ARMOR on all *error*, *loop*, and *non-equivalent success* mutants. For the *error* mutants we simply relied on the implicit failure detection (i.e., exceptions). For the mutants in the *loop* and *non-equivalent success* categories, we augmented the application with specific failure detectors that we obtained as follows:
   - We used Daikon [19] to derive invariants from repeated executions of the original program (without mutations).
   - We used Daikon on each mutant (same application, same input) and selected those invariants found within roll-back areas that were valid for the original program but not for the mutant program.
   - We inserted those invariants as assertions in the application code, within the RBA where they were found.

C. Effectiveness

We first evaluate the effectiveness of ARMOR. We measure that by counting the cases in which ARMOR could recover from one or more failures caused by a mutant and allow the application to run to completion with a correct output.

These results are displayed in Table III. The top part of the table summarizes the selection and classification of mutants for each application. The last row shows the effectiveness of ARMOR. These results are very encouraging, since they demonstrate that ARMOR is successful with between 19% and 48% of the mutants. These are cases in which ARMOR is *completely* successful, meaning that the application terminates successfully and with the correct output despite the presence of a failure-inducing fault.

D. Runtime Overhead

We then measure the runtime overhead of ARMOR to verify that the execution of an instrumented application would not suffer an unreasonable penalty due to the instrumentation. We measure the overhead of ARMOR in terms of total execution time and in terms of allocated memory in a normal (non-failing) run, and we compare those measurements with the execution of the original application code.

Table IV summarizes the results of this analysis. At a high-level, the results demonstrate that ARMOR (with the change-log-based checkpoint) incurs a noticeable but also seemingly reasonable overhead in all cases. In particular, the running time overhead ranges from 2% to 194%. Interestingly, we initially assumed that the runtime overhead would be attributable to the checkpoint mechanism, since that is essentially the only active code executed by ARMOR in normal (non-failing) runs. However, a further analysis shows that a significant portion of the total overhead is instead due to the instrumentation alone, which in practice consists of the time needed to execute a try-block in the proxy method.

The somewhat extreme case of the Closure compiler in which RBAs are executed in very hot loops, as evidenced by the high number of recorded checkpoints, also shows that the checkpoint mechanism is quite efficient, since the execution of over 1.2 million checkpoints incurs only a relatively low 99% overhead, corresponding to a bit more than 5 seconds of execution time (on a 2.53GHz Intel Xeon E5630 CPU).

E. Analysis and Discussion

For each mutant and each application, we manually analyzed the results of the execution with ARMOR to identify the causes of the successes and failures of our technique. Here we report some highlights of this analysis.

An interesting case is Fb2pdf. Fb2pdf is the application with the fewest number of RBAs but also with the largest set of mutants affecting the execution, and it is also the application with the lowest success rate. What we found is in fact that Fb2pdf uses the library quite extensively, but it does that through a few access points and, therefore, at a greater depth than other applications. We draw two conclusions from this analysis: first, there is little hope to avoid the effects of a fault whenever a few calls use a large portion of the library code, since that would require a large amount of redundancy. Second, workarounds are likely to be more effective when the fault (in the library) is somewhat *closer* to the application code, and therefore when the alternative use of the library would have a more direct control in steering the execution away from the fault.

Another interesting and related case is that of a failure in Carrot2 which ARMOR could not avoid. This failure turned out to be caused by a use of the (mutated) Guava library from within another library used by Carrot2, which means that ARMOR was not even involved in that particular use. We do not know whether ARMOR could have prevented the failure, but once again we observe that faults at a greater depth in the call stack have less chances of being avoided through workarounds at the interface between application and library.

The case of Carrot2 is also interesting because it is characterized by several active and also nested RBAs. One of the RBAs of Carrot2 is in fact in its *main* method, and several others are nested up to a depth of 7. Nested RBAs are expensive because they involve more checkpoints and also because they might induce several nested iterations to look for a valid workaround, especially when no workarounds are found for lower-level RBAs. This complexity might explain the overhead incurred by ARMOR with Carrot2.

VI. Related Work

The idea of relying on some form of redundancy to make applications more robust to faults is not new. Some of the most well known (but also disputed) software fault tolerance techniques require the availability of several variants of

TABLE III
MUTATION ANALYSIS AND EFFECTIVENESS OF ARMOR

				Caliper	*Carrot2*	*Closure*	*Fb2pdf*
Total mutants				21297	21297	21297	16858
Relevant mutants				309	187	344	2200
execution	*success*	*equivalent*		210	120	177	1805
		non-equivalent	*detected*	**0**	**2**	**0**	**0**
			not detected	0	8	3	1
	loop	*detected*		**0**	**1**	**0**	**0**
		not detected		12	9	15	47
	error			**87**	**47**	**149**	**347**
Total mutants run with ARMOR				**87**	**50**	**149**	**347**
Mutants where ARMOR is successful				(28%) 24	(48%) 24	(47%) 70	(19%) 67

TABLE IV
OVERHEAD INCURRED BY ARMOR IN NORMAL NON-FAILING EXECUTIONS (MEDIAN OVER 10 RUNS)

		Caliper	*Carrot2*	*Closure*	*Fb2pdf*
Time (seconds)	Original total running time	30.13	2.43	5.40	2.26
	Exception-handling only (no checkpoints)	(1%) 30.41	(69%) 4.15	(95%) 10.53	(68%) 3.79
	Snapshot-based checkpoints	(5%) 31.78	(117%) 5.32	>1h	(121%) 4.99
	Change-log-based checkpoints	(2%) 30.87	(94%) 4.75	(194%) 15.90	(114%) 4.70
Memory (MB)	Original total memory allocated	1.40	8.87	30.56	17.90
	Snapshot-based checkpoints	12.30	23.78	—	90.94
	Change-log-based checkpoints	10.18	11.37	120.58	25.93
Number of recorded checkpoints (approx.)		30	2,350	1,255,000	4
Values saved in change-log-based checkpoints (approx.)		26,000	270,000	1,880,000	9,000

the same components developed independently by different teams [2], [3], [20]. Diversity in the design and development limits the number of correlated faults and therefore increases fault tolerance, but it also increases the development costs. Our technique is similar in that it exploits redundancy, but it is also different in that it exploits redundancy that is already present in libraries and therefore does not incur additional design costs.

Similarly, data diversity exploits redundancy to tolerate faults, but without incurring higher development costs [21]. Instead of executing several variants of the same component, the execution involves the original component with several re-expressions of the input. Other classic software fault tolerance techniques, such as wrappers and exception handling, require the development of additional code to deal with specific failures [22], [23]. Our technique automatically creates proxy methods, which are equivalent to wrappers, to execute alternative code, and it relies on exception handling to automatically deal with failures.

Software fault tolerance also comprises several techniques that specifically deal with non-deterministic failures. Such techniques, including software rejuvenation, microreboots, and checkpoint-and-retry, are complementary to our technique, since they would all fail in avoiding deterministic failures that our technique can instead avoid [24], [9], [25]. Rx extends these techniques by dealing with some deterministic failures. However, it is effective only on failures that are caused by problematic interactions with the environment [10].

Most of the recent techniques to avoid or mitigate the effects of field failures address specific problems, such as repairing inconsistencies in data structures [4], [5], [26], avoiding infinite loops [7], fixing configuration issues [6], fixing invalid HTML code generated by PHP applications [27], and making applications more robust to malicious inputs [28] and malicious attacks in general [8]. Our technique does not target specific faults and instead is intended as a general-purpose solution to avoid failures at runtime.

Other general techniques like ours require a more intense manual intervention. For instance, Chang et al. can automatically deploy patches for integration problems, but they require library developers to manually write the patches first [11]. Moreover, they assume the applications to be stateless, since they do not offer any mechanism to handle the state after a failure. Cabral et al. instead can automatically deploy applications with exception handlers that can deal with general problems (e.g., I/O exceptions) but they still require developers to write the exception handlers for application-specific exceptions [12]. Finally, Harmanci et al. propose new Java constructs to execute alternative blocks of code in case of exceptions [29]. However, it is up to the developers to code the alternative blocks. Our technique requires some manual effort too, since we assume to have a list of rewriting rules for each library. However, this information can be reused with different applications.

Our work is also very related to some recent state-of-the-art automated debugging techniques. These techniques can automatically find patches to fix faults identified either by failures in test suites or by contract violations [30], [31], [32], [33]. Differently from these techniques, we do not aim to *fix* the faulty statements in the application, but instead we only aim to avoid their effects at runtime. In fact, our technique is designed to work on deployed applications, whereas automated debugging techniques are still costly and at the moment are best applied only to an off-line debugging process.

VII. Conclusions

We presented a technique to improve the reliability of applications. This technique is based on and extends the notion of automatic workarounds that we developed in prior work. In particular, we now support general-purpose (Java) applications with workarounds that can be deployed at runtime without interrupting the execution flow of the application and without restarting its components. We demonstrate the effectiveness and usability of this technique through a concrete implementation called ARMOR, with which we obtain very positive results. As part of our evaluation we also evidence and document the intrinsic redundancy of some software systems, which is an interesting result of independent value.

Our future research plans include the further development and refinement of ARMOR on the basis of the experience gained with this study. We plan to work on new methods to deal with faults and failures that are deep in the library code and therefore that are unlikely to be solved by a workaround at the application level. We also plan to conduct more and more thorough experiments with ARMOR by applying it to yet more libraries and applications.

Departing from ARMOR and automatic workarounds, and moving forward towards a more broad and somewhat ambitious research objective, we also intend to formalize the notion of intrinsic redundancy and to study its prevalence and its origin in modern software.

Acknowledgments

This work was supported in part by the Swiss National Science Foundation with project *SHADE* (n. 200021-138006) and by the European Research Coucil with ERC Advanced Grant *SPECMATE* (n. 290914).

References

[1] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *SIGMOD Record*, vol. 17, no. 3, 1988.

[2] A. Avizienis, "The N-version approach to fault-tolerant software," *IEEE Transactions on Software Engineering*, vol. 11, no. 12, 1985.

[3] B. Randell, "System structure for software fault tolerance," in *Proceedings of the International Conference on Reliable software*, 1975.

[4] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *Proceedings of the 18th Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2003.

[5] I. Hussain and C. Csallner, "DSDSR: a tool that uses dynamic symbolic execution for data structure repair," in *Proceedings of the 8th International Workshop on Dynamic Analysis*, 2010.

[6] B. J. Garvin, M. B. Cohen, and M. B. Dwyer, "Using feature locality: can we leverage history to avoid failures during reconfiguration?" in *Proceedings of the 8th Workshop on Assurances for Self-Adaptive Systems*, 2011.

[7] M. Carbin, S. Misailovic, and M. Kling, "Detecting and escaping infinite loops with Jolt," in *Proceedings of the 25th European Conference on Object-Oriented Programming*, 2011.

[8] J. H. Perkins, G. Sullivan, W.-f. Wong, Y. Zibin, M. D. Ernst, M. Rinard, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, and S. Sidiroglou, "Automatically patching errors in deployed software," in *Proceedings of the 22nd International Symposium on Operating Systems Principles*, 2009.

[9] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot—a technique for cheap recovery," in *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, 2004.

[10] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan, "Rx: Treating bugs as allergies—a safe method to survive software failures," *ACM Transactions on Computer Systems*, vol. 25, no. 3, 2007.

[11] H. Chang, L. Mariani, and M. Pezzè, "In-field healing of integration problems with COTS components," in *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[12] B. Cabral and P. Marques, "A transactional model for automatic exception handling," *Computer Languages, Systems and Structures*, vol. 37, no. 1, 2011.

[13] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè, "Automatic workarounds for Web applications," in *Proceedings of the 18th International Symposium on the Foundations of Software Engineering*, 2010.

[14] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th Conference on the Foundations of Software Engineering*, 2005.

[15] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, 2002.

[16] A. Hindle, E. Barr, Z. Su, P. Devanbu, and M. Gabel, "On the "naturalness" of software," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[17] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proceedings of the 18th International Symposium on Software testing and analysis*, 2009.

[18] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a java compiler," in *2011 26th International Conference on Automated Software Engineering*, 2011.

[19] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, 2001.

[20] J.-C. Laprie, C. Béounes, and K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architectures," *Computer*, vol. 23, no. 7, 1990.

[21] P. E. Ammann and J. C. Knight, "Data diversity: An approach to software fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, 1988.

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

[23] F. Cristian, "Exception handling and software fault tolerance," *IEEE Transactions on Computers*, vol. 31, no. 6, 1982.

[24] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: analysis, module and applications," in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, 1995.

[25] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, 2002.

[26] B. Elkarablieh, I. Garcia, Y. L. Suen, and S. Khurshid, "Assertion-based repair of complex data structures," in *Proceedings of the 22th IEEE Conference on Automated Software Engineering*, 2007.

[27] H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, and L. Hendren, "Automated repair of HTML generation errors in PHP applications using string constraint solving," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[28] F. Long, V. Ganesh, M. Carbin, S. Sidiroglou, and M. Rinard, "Automatic input rectification," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[29] D. Harmanci, V. Gramoli, and P. Felber, "Atomic boxes: coordinated exception handling with transactional memory," in *Proceedings of the 25th European Conference on Object-Oriented Programming*, 2011.

[30] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proceedings of the 24th International Conference on Automated Software Engineering*, 2009.

[31] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[32] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proceedings of 11th IEEE Congress on Evolutionary Computation*, 2008.

[33] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010.