In-Field Healing of Integration Problems with COTS Components

Hervé Chang and Leonardo Mariani and Mauro Pezzè*
University of Milano Bicocca
Department of Informatics, Systems and Communication
viale Sarca 336, 20126 Milano, Italy
{chang, mariani, pezze}@disco.unimib.it

Abstract

Developers frequently integrate complex COTS frameworks and components in software applications. COTS products are often only partially documented, and developers may misuse technologies and introduce integration faults, as witnessed by the many entries in fault repositories. Once identified, common integration problems and their fixes are usually documented in forums and fault repositories on the Web, but this does not prevent them to occur in the field when COTS products are reused.

In this paper, we propose a methodology and a self-healing technology that can reduce the occurrence of in-field failures caused by common integration problems that are identified and documented by COTS developers. Our methodology supports COTS developers in producing healing connectors for common misuses of COTS products. Our technology produces information that facilitate debugging and patching of applications that use COTS products.

Application developers inject healing connectors into their systems to automatically repair problems caused by misuses of COTS products. Healing takes place at run-time, on-the-fly and in-the-field. The activity of healing connectors is traced in log files, to facilitate debugging and patching of integration problems. Empirical experiences with several applications and COTS products show the feasibility of the approach and the efficiency of the technology.

1 Introduction

To mitigate the development costs, reduce the knowledge required to implement applications, master the complexity of the code and increase the software quality, applications are frequently developed by reusing available COTS[1] frameworks, servers and components [20]. For instance, many Web applications are based on the Struts (http://struts.apache.org/2.x/) and Spring (http://www.springframework.org) frameworks, enterprise systems are frequently integrated with application servers like JBoss (http://www.jboss.org) and Geronimo (http://geronimo.apache.org/), and several desktop applications reuse libraries and execute within virtual environments, like Sun JRE (http://java.sun.com/javase/).

In general, COTS frameworks and components are reliable products, but the complexity of the technology and the incompleteness of the available documentation can result in faulty integration of COTS products.

Common integration problems stem from wrong usage of the interfaces of COTS components, for instance due to incorrect ordering of method invocations or incorrect data values passed as parameters of method calls. A simple example is the interaction with class `MethodInvoker` provided by the Spring framework. Class `MethodInvoker` requires the invocation of method `prepare` before executing method `invoke`. Application developers often forget to invoke the preparation method before method `invoke`, as documented by Spring developers (see bug ID SPR-3386 http://jira.springframework.org/browse/SPR-3386).

More subtle failures can be caused by operations invoked when the target applications are in particular states, by specific environmental conditions, and by unexpected values in the configuration files required by COTS frameworks. These problems are hard to be avoided at development time, because it is hard to foresee all possible uses and misuses of a technology, are difficult to be revealed before the system deployment, because validation can hardly cover all possible usage scenarios, and can cause serious failures, such as application crashes [17, 36].

These problems can be tackled with testing, static analy-

---

*Mauro Pezzè is also professor at the University of Lugano, Faculty of Informatics, via Buffi, 13 6900 Lugano (Switzerland).

[1]Here we use the term COTS (Commercial off-the-shelf) to indicate software products provided and maintained by third-parties, but not necessarily commercial products.

sis and defensive programming [28, 18, 31].

Testing is fundamental to reveal integration problems and reduce the number of faults in deployed software applications. However, testing is not an exhaustive verification technique, and can often fail to reveal all problems related to the interplay of multiple and rare conditions [28]. In section 5, we report data about problems that are related to integration with COTS frameworks, and have been discovered in the released versions of software products, even after extensive pre-release testing.

Static analysis can effectively analyze complex combinations of events that can cause software failures. However, static analysis techniques are limited by scalability and false positive problems. Static analysis techniques are effective in analyzing artifacts of limited size and complexity, but do not scale up well to large programs [23]. Scalability can be obtained at the cost of restricting the scope of the analysis to some classes of problems, for instance, several static analysis techniques can recognize predefined sets of bug patterns [18], but cannot effectively reason about all possible behaviors of large software systems. Static analysis is hindered by the many false positives that it usually generates [18, 32]. The complexity of the analyzed artifacts is typically managed by introducing some degree of imprecision in the analysis, which often results in false alarms that need to be investigated by software analysts. Applications that integrate COTS components are usually quite large. Static analysis can provide useful results, but cannot remove all the integration problems that are frequently reported.

Defensive programming can be used to develop COTS components that are robust with respect to possible misuses by client applications [31]. Defensive programming can reduce the failures observed in the field, but is effective only with the misuses that can be identified at design-time, which cover only a limited subset of possible misuses.

In this paper, we propose a self-healing approach that handles faults that derive from the integration of COTS frameworks and components into software applications. The approach handles problems at run-time, and can successfully heal some of the faults that inevitably escape testing and analysis.

Our technology is based on the idea of transforming the information about possible incompatibilities that is discovered and published by COTS developers, into healing connectors that can be automatically injected into applications to heal integration problems that are not understood by application developers. In this way, we effectively incorporate into applications the information that is available to the COTS developers, and that can avoid integration problems, without involving application developers, who may not understand all details of interactions with complex COTS products. Our approach comes with two benefits: on-

the-fly and in-the-field healing masks failures, and logging eases off-line debugging and fault fixing.

Our approach involves both developers of COTS components and frameworks, hereafter *COTS developers*, and developers of applications that include COTS products, hereafter *application developers*. COTS developers should use the information about integration faults that is routinely collected and published on the Web, to produce healing connectors, that is connectors that fix typical misuses of COTS frameworks that are experienced after their release. Application developers inject healing connectors into applications. Since at the time of the injection both the COTS components and the applications that integrate them are available, healing connectors can heal both integration faults and common misuses of COTS components in client applications.

To minimize run-time overhead, healing connectors are activated only when COTS components raise exceptions. The activated connectors apply the available healing strategies. If the healing fixes the problem, the exception is masked and the application continues run safely. If the healing fails, the exception is propagated to the caller side.

The paper is organized as follow. Section 2 presents how healing connectors work in-the-field. Section 3 illustrates the methodology followed by COTS developers to implement and release healing connectors, and the methodology followed by application developers for injecting connectors into applications. Section 4 shows the feasibility of the approach by describing our early experience with the identification of typical integration problems with COTS components and the development of an initial set of healing connectors. Section 5 presents empirical evidences of the effectiveness and efficiency of healing connectors with well-known COTS components and frameworks. Effectiveness is demonstrated by the successful healing of a number of misuses that we found in popular applications available on the Web. Efficiency is demonstrated by the limited overhead introduced by healing connectors. Section 6 discusses related work. Section 7 summarizes the main contributions of our research and presents on going work.

2 Healing connectors

Healing connectors are software modules that are designed by COTS developers to fix integration problems, and are injected into applications that integrate COTS frameworks and components. Healing connectors are activated by exceptions raised by COTS components, and can be injected both at the application side, by instrumenting the application that uses the COTS frameworks, and at the COTS side, by instrumenting the COTS frameworks, without requiring source code. Connectors can be developed with many different technologies: aspect-oriented technolo-

gies [13], the TPTP probekit [19] and bytecode instrumentation tools [6]. In this section we describe the structure and the behavior of healing connectors referring to the prototype implementation with the AspectWerkz aspect-oriented framework that we used in the experiments [1].

A *healing connector* is composed of three elements: a *connector* that intercepts exceptions and is developed as an aspect, a set of *healing strategies* that are invoked by the connector and are implemented as Java classes, and a *specification of the points where the connector must be injected*, which is defined as a set of pointcuts and corresponding joinpoints that bind the connector to all the specified pointcuts.

Listing 1 shows an excerpt of the aspect class `MethodInvokerAspect` that implements the healing strategy for the problem with the Spring class `MethodInvoker` described in the previous section. Every call to method `invoke()` triggers the execution of the advice method `handle`, which propagates the original invocation by executing `jp.proceed()`. If `jp.proceed()` raises an exception, the exception handler executes the corresponding healing strategy. Listing 2 shows the definitions of the pointcuts and joinpoints necessary to bind the execution of the advice method `handle` to any call to method `invoke` of class `MethodInvoker`.

```
public class MethodInvokerAspect{

 public Object handle(JoinPoint jp) throws
 Throwable{
  try{
      // invocation of the target operation
      return jp.proceed();
 }
 catch(IllegalStateException ise){
 // implementation of the healing steps here.
  ...
 }
}
```

Listing 1. Excerpt of an Aspect class relative to a healing connector.

```
<aspect class="healing.connector.springframework.
              util.MethodInvokerAspect">
 <pointcut name="pc" expression="call(java.lang.
     Object org.springframework.util.
     MethodInvoker.invoke()"/>
 <advice name="handle(JoinPoint jp)"
         type="around" bind-to="pc" />
</aspect>
```

Listing 2. Specification of the binding of aspect class `MethodInvokerAspect`**.**

Healing connectors may require some information about the environment in which applications are executed to work properly. For instance, a healing connector that heals a failing load of configuration files by searching the configuration files in default folders requires a specification of the folders. This specification can only be provided by the application developers, and cannot be hard coded into the healing connectors. To handle these cases, the bundle that includes healing connectors also includes configuration files that can be edited by application developers.

Healing connectors work in four main steps: detecting exceptions, identifying healing strategies, executing strategies and returning to normal. Figure 1 summarizes the behavior of healing connectors.
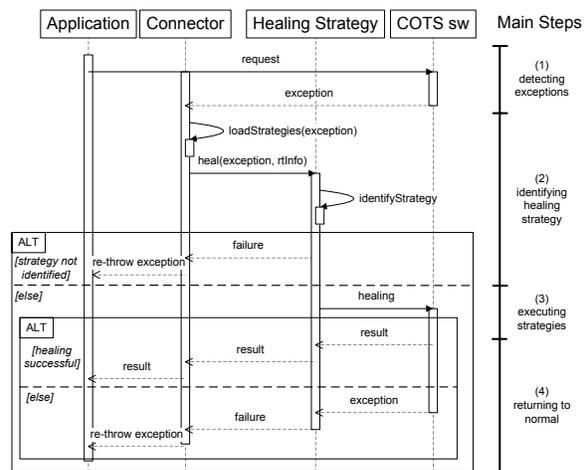

Figure 1. Behavior of a healing connector.

When *detecting exceptions*, healing connectors listen to the types of exceptions related to the class of problems they address. For instance, connectors that heal the problem related to the incorrect invocation order of methods `prepare` and `invoke` of the Spring class `MethodInvoker` described in Section 1, listens for exceptions of type `IllegalStateException`.

In this paper, we focus only on problems that can be detected by monitoring exceptions, to avoid run-time overhead when applications execute successfully, and to minimize overhead in case of failures (see Section 5 for empirical data). Even focusing only on the limited class of problems that raise exceptions, we address a large number of important faults that frequently cause failures, as studied and reported in [24]. However, the technique is not bounded by this choice, and can be extended to other classes of problems.

When connectors catch exceptions, they load the healing strategies available for the observed exception to see if any of them may match a problem behind the caught exception.

Figure 1 illustrates the loading process as a single method invocation. In general, since multiple strategies can be tried to repair the problem, the process loads a *chain of responsibility* of healing strategies [16] that are sequentially executed until one of them fixes the problem or all the strategies have been tried.

To *identify healing strategies* that may solve the problems behind the caught exception, healing connectors inspect the current status of the system to see if the exception has been raised as a consequence of a problem that can be healed by some available healing strategies. If no matching strategy is found, the exception is re-thrown, otherwise, the connector attempts to heal the problem by executing the identified strategies. Identifying the strategies can be as easy as checking information reported within the caught exceptions, or may require invoking inspector methods to investigate the state of the components or the environment. For instance, when the connector detects exceptions that may be caused by the incorrect invocation order of methods `prepare` and `invoke` of class `MethodInvoker`, the connector may invoke method `isPrepared` to check if the called object of type `MethodInvoker`, is prepared or not. If method `isPrepared` returns `false`, the connector has successfully identified a healing strategy that may solve the problem, and can attempt to apply it. Figure 1 shows the simplest case that does not require additional method invocations.

To attempt healing the fault, healing connectors *execute the identified strategies*. The healing strategies are executed on-the-fly and in-the-field. If successful, the application returns to normal execution without failing. Healing strategies may include different types of actions, depending on the problem to be healed. Section 4 presents a preliminary taxonomy of strategies that we derived from our early experience. Figure 1 shows the simple case of a healing strategy that invokes a method.

If the healing strategies succeed, the *control is normally returned* to the client, by propagating return values, if any. If the healing strategies fail, the exception is re-thrown.

The activities of the healing connectors are logged into log files, independently from the result of the healing process. Log files can be inspected off-line by application developers to debug and permanently fix the problems healed on-the-fly by the healing connectors. In this way, healing connectors fix problems on-the-fly when they first occur, and developers can fix them permanently in new releases. Permanent fixing can be quick, thanks to the detailed log files produced by healing connectors.

3 Methodology

In this section, we discuss the impact of healing connectors on the development of both COTS products and COTS based applications by overviewing the methodology that COTS and application developers should follow to release healing connectors, and to integrate them into component-based applications, respectively.

The designing of healing connectors does not impact on the development before the release of COTS products. Novel activities are required only during maintenance, when problems of integrating COTS products into applications are reported.

Currently, when application developers report problems with COTS products, *COTS developers* either release patches, when problems can be solved at the level of single COTS products, or document product misuses into official forums on the Web, when integration problems can be solved more effectively by modifying the application that uses COTS frameworks. Unfortunately, application developers are not always up-to-date about the latest documented problems, and COTS based applications often latently include many problems that are reported on forums and that eventually cause in-field failures.

Healing connectors provide an effective alternative to posting problems on forums, and reduce the impact of known integration problems latently included into applications. When integration problems can be solved more effectively by modifying the application than by patching COTS products, COTS developers produce healing connectors that are automatically interponed between applications and COTS products to solve the integration problem. COTS developers follow a three step process: check for failure types, identify healing conditions, and develop healing connectors.

To check for failure types, developers simply verify the type of exceptions raised by the integration problems. For example, the missing initialization of the Spring class `MethodInvoker` raises an `IllegalStateException`. We already observed that focusing only on problems that raise exceptions minimizes the run time overhead, while still solving a large amount of problems [24]. Extending the approach to other classes of problems would change this step.

To identify healing conditions, COTS developers investigate the status of the COTS product when the exception is raised, and identify the conditions that hold at run-time and characterize the detected problem. For example, when the missing initialization of an object of type `MethodInvoker` raises an `IllegalStateException`, method `isPrepared` invoked on that object returns false.

To develop healing connectors, COTS developers design solutions of the problems triggered by the healing conditions. For instance, a healing connector for the missing initialization of the Spring class `MethodInvoker` invokes

methods `prepare` and `invoke`, thus initializing the object before invoking the method.

Even if COTS developers know the COTS products very well, and fully understand the problems to be healed, healing connectors can introduce undesired side-effects into applications. To verify the correctness of the connectors in the field, COTS developers design test cases that are executed when connectors are integrated into applications. Test cases must check that the misuse to be healed generates the expected exception indeed, the identification condition characterizes the misuse, and the connector fixes the problems on the fly. Listing 3 shows an example test case for the healing connector associated with the initialization problem of class `MethodInvoker`.

```
@Test
public void testMethodInvokerConnector()
throws IllegalAccessException ,
       InvocationTargetException{
 // deploy the healing connector
 MethodInvokerConnectorDeployer.enableMonitoring(
     " call(java.lang.Object org.springframework.
     util.MethodInvoker.invoke()) ");
 try {
 // call invoke without prepared
 minvoker.invoke();
 }
 catch(IllegalStateException ise){
 // connector should have healed the fault
 Assert.fail("Should not have raised
             IllegalStateException");
 }
 finally {
 // undeploy the healing connector
 MethodInvokerConnectorDeployer.
     disableMonitoring();
 }
}
```

Listing 3. JUnit test cases for the healing connector that fixes the missing initialization of the Spring class `MethodInvoker`**.**

Application developers can integrate healing connectors transparently without knowing either about the potential problems or the healing strategies. Application developers download connectors made available by COTS developers. Connectors are automatically injected into applications with procedures that depend on the technology. For example, using aspect-oriented techniques, the injection of connectors consists in instrumenting the application at the specified joinpoints to insert calls to the precompiled aspects of the healing connectors. The injection can be done automatically offline by post-processing the application classes, or online by redefining the classes bytecode at runtime.

Connectors are activated only when exceptions are raised, thus developers can inject all available connectors without impacting on the applications.

Finally connectors are validated by first executing the application system tests, to check that the healing connectors do not introduce new problems, even when inactive, then by executing the test cases associated to the healing connectors to validate that healing connectors do not introduce unexpected side effects when activated.

4 Developing Healing Connectors

In this section, we discuss the feasibility of developing healing connectors by investigating faults of COTS products documented on forums. In the next section, we investigate the effectiveness of the connectors to heal integration problems. In our study, we proceeded as follows: we browsed fault repositories and we collected information about common misuses and integration problems, we investigated the problems to identify classes of faults that can be handle with common strategies, and we developed healing connectors for the identified faults.

We browsed four bug repositories: the bug repository for the Sun standard JDK and its libraries [34], the Spring framework bug repository [3], the JBoss bug repository [2] and bug repositories for various systems hosted by the Apache software foundation [5]. We identified four classes of common faults that correspond to four healing strategies, and we implemented 31 healing connectors for the considered COTS frameworks [12].

The four classes of common misuses and integration problems that we identified in the repositories, and for which we developed healing connectors are invalid parameter, incorrect usage of interfaces, faulty method and environmental fault. These four healing strategies aim at healing the execution of the failing operation.

In the following, we survey the classes of common misuses, we exemplify them with cases taken from our experience, and we suggest healing strategies that we successfully used in the cases considered so far, as discussed in the next section.

An *invalid parameter* problem occurs when an application invokes a method of a COTS component with a parameter value that does not meet the method requirements, and raises an exception. For example, constructing a JavaNet `URI` object with underscore characters raises a `URISyntaxException`, see for instance the ActiveMQ bug ID AMQ-1188[2]; while constructing an HTTPClient `GetMethod` object with invalid characters results in an `IllegalArgumentException`, see for instance bug ID HTTPCLIENT-678[3]. The responsible for the failure is

---

[2]http://issues.apache.org/activemq/browse/AMQ-1188
[3]http://issues.apache.org/jira/browse/HTTPCLIENT-678

mainly the application that invokes the method with a wrong value for a parameter. The fault can be often fixed easier by correcting the application than by patching the COTS component.

Invalid parameter problems can be healed with a *change parameter and retry* strategy that automatically substitutes illegal parameter values with legal ones before invoking again the method that raised the exception. For example, we built a connector that fixes the ActiveMQ fault by replacing the string parameter hostname with its IP address and then re-invoking the original operation, and we built a connector for the HTTPClient `GetMethod` fault by escaping the invalid characters of the parameter and then re-invoking the original operation.

An *incorrect usage of interfaces* occurs when an application invokes a method of a COTS component and the component cannot serve it thus raising an exception. Common problems of this type are missing initializations or incidental usages of dead connections. Typical examples of this class of problems are the missing invocation of method `prepare` when using object `MethodInvoker` that we use as a working example in this paper, and sending a message using a disconnected javamail `Transport` that both raise an `IllegalStateException`, see for instance bug ID GERONIMO-1669[4]. As in the former case, the responsible for the failure is mainly the application, and the fault can be fixed easier by correcting the application than by patching the COTS component.

Incorrect usage of interface problems can be healed with a *call operations and retry* strategy that automatically invokes suitable methods to fix the component before invoking again the method that raised the exception. The invocation of method `prepare` before the method `invoke` implemented by class `MethodInvoker` that we use as a working example in the former sections is an example implementation of this strategy. A connector to heal the javamail Transport problem can invoke method `connect` before re-invoking the `send` operation.

A *faulty method* problem occurs when an application invokes a faulty method of a COTS component. Although in many cases faulty method problems can be fixed by patching the COTS component, in some cases developing healing connectors turns to be convenient. For example, invoking method `classLoader.loadClass` with an array syntax parameter under Java 6 results in a `ClassNotFoundException`, see for instance the Geronimo bug ID GERONIMO-3142[5].

Faulty method problems can be healed with a *replace calls* strategy that automatically substitute the failing invocation with invocations of methods that produce results equivalent to the method that raised the exception. In

the previous example, replacing the invocation of method `classLoader.loadClass` with the invocation of operation `Class.forName` successfully returns the expected result. This healing connector turns out to be particularly convenient since neither the developers producing Sun Java 6 are willing to patch their code, nor the developers integrating this COTS product are willing to implement this workaround in their code[6].

An *environmental fault* occurs when the interaction between an application and a COTS component is badly affected by environmental conditions, and raises exceptions. Typical examples are interactions between applications and components that require deployment descriptors or files that are missing in the environment. Environment problems can result in the generation of several exceptions. In our experience, we frequently noticed `IOException` and `ClassNotFoundException`. For example, when reusing the Xalan XSLT processor, failing to deploy the required jar files leads to a `TransformerConfigurationError`, see for instance the Magnolia CMS bug ID MAGNOLIA-1958[7]. Environment faults often depend on system administrators, who deploy the systems, but can sometimes be fixed with healing connectors if suitable extra-information about the execution environment is available.

Environment faults can be healed with a *change environment and retry* strategy that modifies the environment to enable the operations that failed before invoking again the method that raised the exception. For example, we implemented a healing connector that solves the previous problem by dynamically loading the jar files and deleting the corrupted directories before re-invoking the original operation.

5 Empirical Validation

In this section, we discuss the suitability of our approach to heal COTS based applications. We report data from our experience with COTS based applications downloaded from the Internet. The suitability of our approach depends on the possibility of instantiating the generic strategies presented in the previous sections to generate effective healing connectors, and on the run-time overhead of the connectors. We studied the effectiveness of the generic strategies by applying them to solve problems reported on the Internet, and we measured the overhead of the generated connectors.

Effectiveness of Healing Connectors Here we report the experience on six popular COTS based applications with known incompatibilities with COTS components.

---

[4]https://issues.apache.org/jira/browse/GERONIMO-1669
[5]https://issues.apache.org/jira/browse/GERONIMO-3142

[6]http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4976356
[7]http://jira.magnolia.info/browse/MAGNOLIA-1958

Application			Failure			Healing Connector		
Name	Category	Size (LOC)	COTS component	Description	Bug information (ID, Priority, Status)	Strategy	Description	# Activations
Apache Geronimo (v.2.0-M3)	J2EE Application server	179,875	Sun JRE 1.6 (Class-Loader component)	At startup, a faulty implementation of classloader.loadClass raises an exception when used to load an array with name specified with array syntax	GERONIMO-3142, Major, Resolved	S3	Substitute the invocation of class-Loader.loadClass() with Class.forName()	2
JBoss Application platform (v.5.0.0.Beta2)	J2EE Application server	685,767	Sun JRE 1.6 (Class-Loader component)	At startup, a faulty implementation of classloader.loadClass raises an exception when used to load an array with name specified with array syntax	JBAS-4491, Major, Closed	S3	Substitute the invocation of class-Loader.loadClass() with Class.forName()	7
Developer application reproduced from bug report GERONIMO-1669	J2EE Web application	168	Apache Geronimo (JavaMail component)	Disconnected smtp transport raises an exception when sending a mail Message	GERONIMO-1669, Major, Closed	S2	Call the transport connect() operation before re-invoking the send the message	1
Apache ActiveMQ (v.4.1.0)	Messaging broker	120,433	Sun JRE (JavaNet component)	Starting ActiveMQ raises exceptions when hostname contains underscore characters	AMQ-1188, Minor, Resolved	S1	Replace the hostname string parameter by its IP address, and re-invoke the original operation	3
Apache ServiceMix (v.3.2.1)	Enterprise service bus	110,713	Sun JRE (JavaNet component)	Starting ServiceMix raises exceptions when the hostname contains underscore characters	SM-492, Major, Closed	S1	Replace the hostname string parameter by its IP address, and re-invoke the original operation	1
Magnolia CMS (v.3.5.1)	Enterprise content management system	58,499	Xalan XSLT	Magnolia cannot run and raises exceptions when initializing its content repositories	MAGNOLIA-1958, Blocker, Closed	S4	Dynamically load the jar files, delete the corrupted repository directories, and re-invoke the original operation	1

Table 1. Healing of faults in different applications.

Healing connector	COTS component	Application	# Measurements	Execution Time (ms)		
				without connectors	w/ inactive healing	w/ active healing
Change parameter and retry (S1)	Sun JRE JavaNet	ActiveMQ	300	0.06606	0.07066	0.16724
		ServiceMix	100	0.09184	0.10647	0.29835
Call operation and retry (S2)	Geronimo/ Java-Mail	J2EE Web app.	100	370.24	373.68	394.53
Replace call (S3)	Sun JRE 1.6 Classloader	Geronimo	200	0.02591	0.02997	6.51212
		JBoss AS	700	0.00824	0.00917	0.14336
Change environment and retry (S4)	Xalan	Magnolia	100	48.67019	53.25770	20.78151

Table 2. Time overhead of the different healing strategies.

We studied the potential incompatibilities and the documented workarounds, we checked the compatibility with the generic strategies presented in the previous sections, and we developed healing connectors by specializing strategies for known fixes. For the experiments reported in this section, we used four of our connectors to heal the six reported problems, since two of them can heal integration problems of two different applications with the same COTS frameworks. We verified the effectiveness of the connectors by testing the applications with and without connectors to verify that the failures observed when executing test cases on the original applications are successfully healed when executing the same test cases with the healed connectors. Table 1 summarizes the empirical results.

The table lists the *Application*s, the considered *Failure*s and the designed *Healing Connector*s. For each application, we report the *Name* that identifies the application on the Web, the *Category* that indicates the technology addressed in the experiments, and the *Size*. For each failure, we indicate the third-party *COTS component* that causes problems when integrated in the application, we provide a short informal *Description* of the integration problem and some *Bug information* that points to the bug report, gives the bug priority (Blocker, Critical, Major, Minor, Trivial) and indicates the bug status at the time of writing (New, Open, In Progress, Resolved, Closed, Reopened). All faults are solved by activating a suitable connector. For each healing connector we indicate the *Strategy* implemented by the connector to heal the fault referring to the ones discussed in the previous section ($S1$ stands for *change parameter and retry*, $S2$ stands for *call an operation and retry*, $S3$ stands for *replace call* and $S4$ stands for *change environment and retry*), we provide a short *Description* of the specific healing strategy implemented by the connector, and we indicate the number of faults that activated the connector, which corresponds to the number of times faults have been successfully healed (*# Activations*).

The healing connectors that we developed so far successfully healed all integration faults that we found on the Web without introducing side effects, and the strategies that we designed so far demonstrated to be effective to heal all the integration faults that we found on the Web.

The preliminary experience reported in this section confirms our hypotheses. The connectors that we developed so far have been obtained by instantiating general strategies starting from fault reports. COTS developers can develop the connectors without knowing the failing applications, and application developers can inject the connectors into their applications without knowing the connector details and with almost no overhead on the application performance, as discussed in details below.

Overhead   As discussed in the previous section, healing connectors are triggered by exceptions, thus we do not expect a significant impact on performances when applications execute correctly. We do expect a small overhead due to the new checks that are introduced in the application by AspectWerkz and that verify the eventual need of activating the aspects that implement the connectors. Healing connectors impact on performance when applications raise exceptions. Usually connectors try to heal the problem locally, and thus we do expect different performance when connectors are activated. The changes in execution time depends on the difference between the execution time of the healing connectors and the execution time of either the original functionality, if applications run correctly, or the exception handlers, if applications fail. In some cases, healing connectors may even improve performances.

We investigated the overhead that healing connectors introduce into the applications by executing the set of case studies presented in Table 1 with and without connectors, and by comparing the execution time. The applications presented in Table 1 may execute for several seconds for processing user requests. For instance, we experienced with integration failures that occur during the (long) start-up phase of the Geronimo and the JBoss AS servers. The execution time of the failing operations are usually a small fraction of the processing time of a user request, and we did not reveal any significant change in performances when measuring the processing time of user requests with and without healing connectors. Thus confirming our hypothesis of the locality of performance changes. To study the execution overhead of healing connectors over single operations, we measured the execution time of single method invocations with and without connectors. Table 2 summarizes the results by listing the execution time measured without connector, with inactive and active connectors for the different connectors.

Table 2 names the connectors according to the strategy they implement (*Healing Connector*), and, for each connector, indicates the component responsible for the integration fault (*COTS component*), the applications involved in the failure (*Application*), the number of measurements (*# Measurements*) and execution time without connectors, and with inactive and active connectors (*Execution time (ms)*).

We obtained time figures by averaging values measured over 100 executions for each connector, and thus $100 * n$ measurements for executions that activate connectors n times. To obtain a precision below the millisecond that is required to measure the extremely limited overhead in the case of inactive healing connectors, we use the Java Native Interface [33] that allows to programmatically query the number and frequency of CPU clock cycles. The empirical investigation about overhead has been ran on an Intel Core2 2.4GHz box with the JRE 1.6.0_04 (except when the use of

JRE 1.5 is specifically required by the investigated configuration). On the technical viewpoint, time measurements are collected by aspects injected around the monitored methods with offline weaving.

Table 2 shows a small overhead of inactive connectors over the execution of the single faulty method (less than 1% in the best case, no more than 16% in the worst case). The overhead of active connectors is more relevant, but still limited to few milliseconds (less than 26 milliseconds in the worst case). In one case the performance of active connectors is even better than without connectors (28 milliseconds faster). The big variance of performance change in presence of active connectors confirms our hypothesis that changes depend on the operations executed to heal the problem, and may even improve with respect to the performance of the original system. For instance, the implemented healing connector for the integration problem between Xalan and Magnolia uses a class loading strategy that runs faster than the one natively implemented in the system. The values of the order of milliseconds confirm the hypothesis that performance changes can hardly be perceived at user level and are limited to local changes.

6 Related Work

The problems of faults or incompatibilities that derive from integration between applications and COTS products have been investigated from different viewpoints: patching infrastructures, software connectors, runtime failure detection, fault tolerance and self-healing. In this section we summarize the main approaches for each category, and we discuss relations, commonalities and complementarities between these techniques and the one presented in this paper.

Patching Infrastructures Releasing patches to be installed to fix faults in field is common engineering practice. Updating and upgrading software systems is often facilitated by infrastructures for managing upgrades, like the Windows update system [27].

State-of-art patching infrastructures handle effectively software updates at user sites, as it happens when software providers release patches to all users through their infrastructure. State-of-art patching infrastructures can cope with end-user applications, but are not designed for fixing integration faults and misuses between applications and COTS frameworks. Patches that deal with integration faults between applications and COTS frameworks should be released by COTS developers, and applied either at the integration or at the application level.

Software Connectors Software connectors have been recently suggested as first-class architectural elements to mediate the interactions and bridge incompatibilities between components. State-of-art connectors address a large variety of component interactions and several works have focused on the definition of taxonomies and classification frameworks for software connectors. For instance, Mehta et al. [26] propose a general taxonomy for software connectors and Becker et al. [8] define a selection of adaptation patterns to handle functional and non-functional mismatches. Our healing connectors share the overall goal of handling component incompatibilities. For example, they can be seen as implementing the adaptor type and the interceptor pattern as described respectively in [26] and [8]. However, our healing connectors specifically address the needs of COTS and application developers, implement the on-the-fly recovery of component integration problems and are activated locally only when known problems are identified.

Runtime Failure Detection Runtime failure detection is the first step of self-healing approaches. The most common approaches to detect failure at run time are based on exception handling and assertions.

Exception handlers manage unexpected events directly in the field with a good degree of flexibility, by means of special procedures embedded into programs at design time. Unfortunately, programmers take advantage of the opportunities offered by exception handling only partially: a large amount of exception handlers coded into programs execute simple and generic procedures that often only propagate the exception or terminate the program, as reported in a study by Cabral and Marques [9]. Poor exception handling practice may even produce additional program failures thus decreasing software reliability. In a recent paper, Li et al. show that between 12% and 16% of the failures reported in J2EE application servers are caused by poor exception handling [24]. The limitations of exception handling are not intrinsic in the mechanism, but depend on design and programming practice. In general exception handling is a powerful mechanism for supporting failure detection at run time. In this paper, we exploit exception handlers to capture unexpected events, and trigger the problem identification and healing steps.

Assertions embedded in the code indicate conditions that must hold during program execution [31]. Runtime violations of assertions indicate anomalous execution condition that may lead to program failures. Although not widely experimented within self-healing solutions yet, assertions can well complement exceptions, and can be used to include a rich set of verification points into programs. Some assertion frameworks raise exceptions when conditions are violated [22], thus in principle they can be easily integrated within our solution.

Fault Tolerance Research on fault tolerance has widely investigated techniques to handle and reduce the impact of

failures, focusing in particular on failures with severe effects [4, 7]. Approaches to fault-tolerant systems share the overall goal with self-healing approaches, since both classes of approaches aim to handle failures in the field at runtime, to minimize their impact. Fault tolerance approaches aim primarily to avoid catastrophic consequences, and do not focus on healing per se, while self-healing approaches aim to heal faults or prevent fault occurrence without focusing on their consequences. As a result the two fields are investigating different although overlapping solutions that often complement each other, but sometime satisfy different requirements.

Self-Healing Self-healing approaches aim to augment software systems with capabilities to automatically heal software at runtime. Some self-healing techniques focus on non-functional problems, for instance Vaidyanathan and Trivedi propose a technique to address aging problems [35], Candea et al. address transient problems [10], Krena et al. focus on concurrency problems [21]. We focus on functional faults that cannot be handled with approaches for non-functional problems.

Self-healing techniques that address functional faults can be grouped in four classes: functional redundancy, checkpoint-recovery, rollback, and ad-hoc healing.

Healing based on functional redundancy consists of identifying and replacing failing sequence of actions with equivalent, but correct, sequence of actions. For example, Carzaniga et al. propose a technique to automatically derive workarounds from specifications [11]. Lack of complete specifications, as often happen with COTS components, limits the applicability of this technique.

Healing techniques based on the use of checkpoints and recovery mechanisms consist of embedding into software systems an infrastructure that records the status of the system at specific program points, and recovers the application from those states if failures occur [15, 29].

Healing solutions based on rollback mechanisms are similar to solutions based on checkpoints. These techniques allow to partially rollback failing executions, and automatically add extra behaviors to increase the probability to succeed in future executions, for instance Quin et al. propose re-executing the application in a safe environment [30], while Lorenzoli et al. define a technique to identify the likely faulty operations and thus identify the rollback point dynamically [25].

Solutions based on checkpoints and rollbacks are quite resource consuming and do not heal faults, but try to overcome them. Our solution based on healing connectors is lightweight, thus widely applicable, and provides capabilities for healing the faulty executions through different kinds of strategies.

Other researchers propose ad-hoc healing mechanisms to be embedded into software systems, for instance, Demsky and Rinard propose techniques to handle consistency problems with data structures in an ad-hoc fashion [14]. Ad-hoc mechanisms can be used to automatically heal specific unit faults within COTS products, but do not replace our healing connector mechanism defined to handle misuses and integration problems.

7 Conclusions

Modern software systems extensively reuse COTS components and frameworks. COTS products reduce development costs and speed up development, but can introduce integration faults that may be harder to address than classic integration faults, since the information required to address these faults is often distributed between COTS developers and application developers. COTS developers own the knowledge necessary to fix problems with COTS components, since they know their products, they gain knowledge about common misuses from official and unofficial forums, and they know how to fix misuses, but they do not own the application code to be fixed. On the other hand, application developers own the code to be fixed, but they own only a fraction of the knowledge available to COTS developers.

In this paper, we proposed a practical solution that can both increase reliability of software working in the field and ease debugging. In our approach, COTS developers release solutions to common misuses in the form of healing connectors that can be automatically integrated into software systems. Application developers simply download and deploy these connectors.

The empirical experience reported in this paper shows that healing connectors can effectively heal faults in the field, with very low cost and execution overhead. COTS developers can produce general healing connectors applicable in many different contexts, and application developers can integrate connectors at practically no development cost. Execution overhead is almost non perceivable by application users when software works correctly. The performance of the applications when healing connectors fix faults may be perceivable but not dramatic and can increase system availability by down time due to unrecovered faults.

We are currently working on the definition of a wider set of healing strategies to address other kinds of COTS integration problems, implementing a wider number of healing connectors for different COTS products and also studying the degree of reuse of the healing connectors across different COTS products. We are also defining a reliable infrastructure that facilitates discovery, configuration and deployment of healing connectors to enable extended empirical investigation to confirm the data presented in this paper.

References

[1] Aspectwerkz. http://aspectwerkz.codehaus.org/, visited in 2008.

[2] JBoss issue tracker. http://jira.jboss.org/, visited in 2008.

[3] Spring issue tracker. http://jira.springframework.org/, visited in 2008.

[4] R. Abbott. Resourceful systems for fault tolerance, reliability, and safety. *ACM Computing Surveys*, 22(1), 1990.

[5] Apache Software Foundation. Apache issue tracker. https://issues.apache.org/, visited in 2008.

[6] Apache Software Foundation. BCEL. http://jakarta.apache.org/bcel/, visited in 2008.

[7] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[8] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an engineering approach to component adaptation. In *Architecting Systems with Trustworthy Components*, volume 3938 of *LNCS*. Springer, 2006.

[9] B. Cabral and P. Marques. Exception handling: a field study in Java and .NET. In *proceedings of the 21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*. Springer, 2007.

[10] 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 and Implementation*, 2004.

[11] A. Carzaniga, A. Gorla, and M. Pezzè. Self-healing by means of automatic workarounds. In *proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. ACM, 2008.

[12] H. Chang, L. Mariani, and M. Pezzè. Self-healing strategies for component integration faults. In *proceedings of the 1st IEEE International Workshop on Automated Engineering of Autonomous and Run-Time Evolving Systems*, 2008.

[13] C. A. Constantinides, A. Bader, T. H. Elrad, P. Netinant, and M. E. Fayad. Designing an aspect-oriented framework in an object-oriented environment. *ACM Computing Surveys*, 1, 2000.

[14] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. *SIGPLAN Notices*, 38, 2003.

[15] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[17] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why its hard to build systems out of existing parts. In *proceedings of the 17th International Conference on Software Engineering*. ACM, 1995.

[18] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[19] IBM. Eclipse test & performance tools platform. http://www.eclipse.org/tptp/, visited in 2008.

[20] R. Johnson. J2EE development frameworks. *IEEE Computer*, 38(1):107–110, 2005.

[21] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, 2007.

[22] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[23] T. Lev-Ami, T. Reps, S. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. *proceedings of the International Symposium on Software Testing and Analysis*, 2000.

[24] J. Li, G. Huang, J. Zou, and H. Mei. Failure analysis of open source J2EE application servers. In *proceedings of the 7th International Conference on Quality Software*. IEEE Computer Society, 2007.

[25] D. Lorenzoli, L. Mariani, and M. Pezzè. Towards self-protecting enterprise applications. In *proceedings of the 18th IEEE International Symposium on Software Reliability Engineering*, 2007.

[26] N. Mehta, N. Medvidovic, and S. Phadke. Towards a taxonomy of software connectors. In *proceedings of the 22nd International Conference on Software Engineering*, 2000.

[27] Microsoft. Windows update. http://www.windowsupdate.com/, visited in 2008.

[28] M. Pezzè and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[29] D. K. Pradhan and N. H. Vaidya. Roll-forward checkpointing scheme: A novel fault-tolerant architecture. *IEEE Transactions on Computers*, 43:1163–1174, 1994.

[30] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[31] D. S. Rosenblum. Towards a method of programming with assertions. In *proceedings of the 14th International Conference on Software Engineering*. ACM, 1992.

[32] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *proceedings of the International Symposium on Software Reliability Engineering*, 2004.

[33] Sun. Java native interface specification 1.1. Technical report, Sun Microsystems, 2003.

[34] Sun Developer Network. Bug database community. http://bugs.sun.com/, visited in 2008.

[35] K. Vaidyanathan and K. Trivedi. A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing*, 2(2), 2005.

[36] D. Yakimovich, J. M. Bieman, and V. R. Basili. Software architecture classification for estimating the cost of COTS integration. In *proceedings of the 21st IEEE International Conference on Software Engineering*, 1999.