Exception Handlers for Healing Component-Based Systems

HERVÉ CHANG and LEONARDO MARIANI, University of Milano Bicocca
MAURO PEZZÉ, University of Milano Bicocca and University of Lugano

To design effective exception handlers, developers must predict at design time the exceptional events that may occur at runtime, and must implement the corresponding handlers on the basis of their predictions. Designing exception handlers for component-based software systems is particularly difficult because the information required to build handlers is distributed between component and application developers. Component developers know the internal details of the components but ignore the applications, while application developers own the applications but cannot access the details required to implement handlers in components.

This article addresses the problem of automatically healing the infield failures that are caused by faulty integration of OTS components. In the article, we propose a technique and a methodology to decouple the tasks of component and application developers, who will be able to share information asynchronously and independently, and communicate implicitly by developing and deploying what we call healing connectors. Component developers implement healing connectors on the basis of information about the integration problems frequently experienced by application developers. Application developers easily and safely install healing connectors in their applications without knowing the internal details of the connectors. Healing connectors heal failures activated by exceptions raised in the OTS components actually deployed in the system.

The article defines healing connectors, introduces a methodology to develop and deploy healing connectors, and presents several case studies that indicate that healing connectors are effective, reusable and efficient.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—Error handling and recovery, diagnostics

General Terms: Design

Additional Key Words and Phrases: Exception handling in component-based software systems, healing connectors, healing patterns, self-healing, component-based software engineering, COTS components

ACM Reference Format:
DOI: http://dx.doi.org/10.1145/2522920.2522923

1. INTRODUCTION

Component-based software engineering (CBSE) is the branch of software engineering that deals with software applications built from reusable components. Reusable Off-The-Shelf (OTS) components are units of independent deployment and versioning...
that are released with both well-defined interfaces and explicit sets of dependencies [Szyperski 2002]. Application developers often reuse OTS components developed and released by independent organizations. In this article, we use the terms OTS components and component developers to indicate the components developed for reuse and the developers of OTS components, respectively.

Component developers implement and release OTS components with limited or no knowledge about the applications that reuse the components. OTS components are often released with incomplete specifications and without access to the source code, thus application developers integrate OTS components with only partial knowledge about their behavior. The lack of information available to component and application developers complicates the integration process, and increases the likelihood of integration faults. Subtle integration faults may derive from infrequent or specific interactions: For instance, well-known integration faults derive from application developers who misunderstand component APIs [Garlan et al. 1995; Yakimovich et al. 1999].

Although classic testing and analysis approaches are effective in revealing many integration faults, some faults may escape verification and validation, and may generate unexpected interactions that can result in runtime exceptions and infield system failures [Pezzè and Young 2007]. This happens especially when failures depend on rare interactions or on particular execution conditions difficult to reproduce in the testing environment.

Exception handling is the most common approach to cope with exceptional and unexpected events [Cristian 1989], and provides effective support for both dealing with runtime problems and structuring error-handling activities in the code [Garcia et al. 2001]. Exceptions can be used to signal many types of unexpected execution conditions, for example, the Sun Java Runtime Environment [Sun 2012] defines a core hierarchy of more than 460 types of exceptions that are thrown to signal exceptional conditions like incorrect arguments, inappropriate component states and I/O problems. Thrown exceptions are caught by special handler blocks that either react to the exceptional conditions or propagate the exceptions to the calling methods that iteratively react or further propagate the exceptions.

CBSE disciplines the way exceptions and exception handlers should be designed in software components: The concept of Idealized Fault-Tolerant Component (IFTC) defines how exceptions and exception handling should be structured in software components [Anderson and Lee 1990]. An IFTC separates the code dedicated to normal activities from the code dedicated to exceptional activities, and has well defined interfaces that clearly distinguish the propagation of normal requests and responses from the exceptional flows. CBSE also provides ways to specify the exceptional flow at the architectural level. For instance, the Aereal framework supports the specification and analysis of exceptional flows between architectural components [Filho et al. 2006].

The reuse of OTS components complicates the design of the exception handling code, because the information required to design proper exception handlers is distributed and rarely shared among component and application developers [Johnson 2005]. Component developers know both the exceptional cases that can be experienced when interacting with their components and the actions to be implemented by the applications to handle these cases, but have no control over the applications and thus cannot implement the suitable exception handlers. Application developers have full control over their applications, but do not have complete knowledge about the exceptional cases that can be generated by components and about the possible reactions to such cases.

This article specifically addresses the problem of automatically healing the infield failures that are caused by faulty integration of OTS components, and are not properly managed by manually designed exception handlers. We propose the use of healing connectors as a solution complementary to the design of exceptional flows at component
and architectural levels. Healing connectors are software modules that can be injected in applications at any time, even into systems that are deployed in the field. Healing connectors aim to handle the runtime failures caused by subtle interactions and rare conditions not covered by native exception handlers, and restoring proper behavior.

In a nutshell, our approach coordinates the distributed and asynchronous work of component and application developers to facilitate developing and deploying effective exception handlers. Component developers analyze common misuses that are experienced when reusing their released components, and are reported by application developers, for example through public communication channels like Internet forums and fault repositories. Then, they design suitable exception handlers to catch and handle the reported misuses, based on their knowledge about how the components should be used, independently from the applications. We call these exception handlers healing connectors to highlight the different nature and deployment of these handlers with respect to classic exception handlers: Classic exception handlers are designed by application developers while developing applications, while healing connectors are designed by component developers once components are already available in the market, and are deployed by application developers as connectors between components. Healing connectors are activated by raising exceptions, thus they can be safely installed and do not introduce runtime overhead when the applications run correctly. Information about the new connectors can be easily distributed, for example by means of simple publish-subscribe mechanisms. In the future, we envision specialized infrastructures that can automatically provide information about the availability of connectors to be automatically deployed in component based applications. There are only trust issues but no technical barriers to a fully automated approach.

Healing connectors represent an alternative to protective wrappers that wrap OTS components within a software layer that handles the mismatch between the failure assumptions of the embedded component and the rest of the system [de Castro Guerra et al. 2004]. Healing connectors address simpler runtime problems, and have two major benefits: (1) They implicitly elicit the communication and collaboration between component and application developers, facilitating the design of handlers that properly react to exceptions raised by OTS components, rather than assigning the whole responsibility to application developers who lack knowledge about OTS components, as in the case of protective wrappers; and (2) protective wrappers cope with application-specific architectural issues and must be redesigned every time a same-component is reused in a new system, while healing connectors can be publicly distributed by developers and can be automatically integrated into systems by application developers, resulting in a dramatic save of effort.

In this article, we advance the state of art in exception handling for OTS components by:

(1) introducing healing connectors, which are software modules that are proactively injected in component-based applications to heal infield failures;
(2) defining healing patterns, which are reusable and general solutions that support component developers in designing and implementing healing connectors;
(3) defining a methodology, which decouples the tasks of component and application developers who share information asynchronously and independently, and communicate implicitly by developing and deploying healing connectors;
(4) releasing a prototype Java implementation, which uses aspect-oriented programming to inject healing connectors in component-based application binaries;
(5) releasing a toolset, which supports the automatic generation and deployment of healing connectors;
(6) collecting empirical evidence, which indicates that our solution is effective, reusable and lightweight.

The work described in this article extends the early results presented in Chang et al. [2009] in many ways: We improved the design and increased the accuracy of the description of healing connectors, we advanced the work on healing strategies by defining healing patterns that include precise structural and behavioral descriptions, we developed a toolset that supports automatic generation of scaffolding classes and configuration files for all our healing patterns, we extensively experimented our solution with more than ten novel case studies that generalize the early results reported in Chang et al. [2009].

The article is structured as follows. Section 2 introduces healing connectors. Section 3 presents a methodology that coordinates the tasks of component and application developers. Section 4 defines a set of healing patterns that guide the definition and implementation of healing connectors. Section 5 describes our prototype toolset that automates the development and deployment activities of healing connectors. Section 6 discusses our empirical results obtained on representative applications. Section 7 discusses related work. Section 8 summarizes the main contributions of this work and outlines future research directions.

2. HEALING CONNECTORS

Healing connectors are software modules that implement exception handling mechanisms. They are designed by component developers to catch exceptions raised when invoking and executing OTS components and restore the normal execution of the application. Component developers implement healing connectors according to common faults and misuses experienced by application developers when integrating and running the OTS components within their systems. Application developers can communicate faults in many ways. Often they report problems through public communication channels, like Internet forums and fault repositories. We discuss in more details the methodology underlying our approach in Section 3.

The healing connectors that we defined so far implement simple operations that can heal failing executions. For example, they modify parameter values or sequences of method calls. The common healing solutions that we identified so far are captured by the healing patterns discussed in Section 4.

Connectors can be implemented with the many different technologies available for the automated injection of additional behaviors into binary applications, such as aspect-oriented frameworks [Constantinides et al. 2000], the TPTP probekit [IBM 2012], bytecode instrumentation tools [Apache Software Foundation 2012] and extended middleware [Simons and Stafford 2004]. In this section, we describe the structure and the behavior of healing connectors without referring to any specific code injection strategy. Section 5 describes our prototype implementation based on aspect-oriented programming technologies.

2.1. Structure of Healing Connectors

A healing connector is composed of three elements, as shown in Figure 1, an interceptor that includes the try and catch blocks to intercept exceptions raised by the OTS components, a set of healing strategies that are invoked from the catch block in the interceptor to repair the faulty execution, and an injection point that specifies the points where the connector must be injected, and typically consists of all the application program points that include calls to the OTS component involved in the integration fault.
The healing strategies executed by the healing connectors can be classified according to the Extender-Translator-Controller (ETC) classification scheme proposed by Keshav and Gamble [1998]. Extenders are classes that add features to other classes such as opening/closing files, buffering, polling or security checks. Translators are classes that transform data from one format to another without affecting the semantics of the information. Controllers are classes that coordinate communication between the components, such as determining the information to pass or discard, or determining the identity of the caller and callee components. In Section 4, we present a first set of healing strategies that are implemented as ETC classes. Note that an ETC class can use other ETC classes to reach its goal.

Healing strategies may require some information about the environment in which the applications are executed. For instance, a healing strategy that heals a failure in loading the appropriate configuration files by searching the configuration files in a set of default folders requires a specification of the folders. This specification can be provided only by the application developers, and cannot be hard coded into the healing strategies. To handle these cases, healing strategies can dynamically load configuration files edited by application developers.

2.2. Behavior of Healing Connectors

Healing connectors work in four main steps: detect exception, identify healing strategy, execute strategy and return to normal execution, as illustrated in Figure 2.

To detect the exceptions that they can handle, healing connectors listen to the types of exceptions related to the classes of problems that they can address. For instance, a simple but frequent integration problem that can be experienced when reusing the OTS Spring framework is the invocation of method `invoke` before method `prepare` when using an object of type `MethodInvoker`, raising an exception of type `IllegalStateException`. The interceptor for healing this problem listens to exceptions of type `IllegalStateException`.

1bug ID SPR-3386 in http://jira.springframework.org/browse/SPR-3386.
In this article, we focus on problems that can be detected by monitoring exceptions, and in this way we avoid runtime overhead when applications execute successfully. Even focusing only on the class of problems that raise exceptions, we can address a large number of important faults that frequently cause failures as studied and reported in Li et al. [2007]. However, the technique is not bound by this choice and can be extended to other classes of failures by incorporating different failure detection mechanisms, such as assertions to detect failures resulting from system specification violation [Rosenblum 1992].

The interceptors identify the healing strategies that may solve the problems behind the caught exceptions, by observing the type of the exceptions. If they do not find an appropriate strategy, they re-throw the exception to the application; otherwise, they delegate the execution of the healing process to the identified strategies. In general, interceptors can identify multiple strategies to repair a given problem, thus they load a set of healing strategies, and execute them sequentially until the problem is fixed. In applying healing strategies, healing connectors work similarly to recovery blocks, which define procedures that first check the status of a computation and then, when they detect a problem, execute multiple alternative recovery strategies [Horning et al. 1974; Anderson and Kerr 1976].

Healing connectors execute healing strategies on-the-fly and in-the-field. If they successfully heal the problem, they return the expected result and the application returns to normal execution. Otherwise, the healing connectors propagate the exception to the application.

Healing strategies are implemented as a chain of responsibility [Gamma et al. 1994], and are sequentially executed until they either fix the problem or all the strategies have been tried. Each healing strategy tries to identify the fault that caused the exception,
and either executes the corresponding healing actions if it identifies a fault, or delegates the healing action to the next healing strategy if either it does not identify a fault or the executed strategy is not successful, as shown in Figure 3.

Identifying faults can be as easy as checking the 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 interceptor detects an exception of type `IllegalStateException` that may be caused by the incorrect order of invocation of the methods `prepare` and `invoke` of class `MethodInvoker`, it delegates the execution to the healing strategy that invokes the method `isPrepared` to check if the called object of type `MethodInvoker` is ready or not.

If the healing strategy identifies a fault, it executes the corresponding healing actions to try to heal the fault. Healing strategies may include different types of actions, depending on the problem to be healed. For instance, in the previous example, if method `isPrepared` returns `false`, the strategy has successfully identified a fault, and can attempt to execute the corresponding healing actions that consist of invoking the method `prepare` on the callee object and then re-invoking the method `invoke`. In Sections 4

Fig. 3. The main steps that characterize the behavior of a healing strategy.
and 6, we present a first set of healing strategies that we derived from our experience and a set of empirical studies, respectively.

If the healing strategy succeeds, it returns the control to the client, and propagates return values, if any. Otherwise, the healing strategy delegates the healing to the next strategy if any, or re-throws an exception if no other strategies are available.

If the healing strategy does not identify a fault, it delegates the healing to the next strategy if any, or re-throws an exception if no other strategies are available. In the previous example, if the method isPrepared returns true, the strategy does not identify a fault, and delegates the healing to the next healing strategy in the chain.

The activities of the healing connectors are logged into log files, independently from the result of the healing process. Log files can be inspected offline by application developers to debug and permanently fix the problems healed by the healing connectors. In this way, healing connectors fix problems on-the-fly when they first occur, and application 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 OTS components and OTS-based applications, and introduce a methodology to release healing connectors, and integrate them into OTS-based applications. We present the methodology from the viewpoint of both component and application developers.

3.1. Component Developers
Healing connectors do not impact on the development of OTS components before release. Novel activities are required only during maintenance, to deal with integration problems of OTS components reported by application developers.

Currently, when application developers report problems with OTS components, component developers either release patches, when problems can be solved at the level of single OTS products, or document component misuses and possible solutions in Web forums, when the integration problems can be solved effectively by modifying the applications that use OTS components. Unfortunately, application developers are not always up-to-date about the latest documented problems, and OTS 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 solutions 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 OTS components, component developers can produce healing connectors that are automatically injected between the applications and the OTS components to solve the integration problems. Component developers implement healing connectors in four steps: check for fault types, identify healing conditions, identify a suitable healing pattern, and develop healing connectors according to the selected healing pattern.

Component developers can check the fault types, by verifying the type of exception raised by the integration problem and investigating the cause of the exception. For example, an instance of the Spring class MethodInvoker can raise the exception IllegalStateException when the preparation step is erroneously skipped.

Component developers can identify the healing conditions, by investigating the status of the OTS component when the exception is raised, and identifying 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.
Component developers can identify a suitable healing pattern, by checking if any of the strategies implemented by the patterns in the catalog can fix on-the-fly the execution that raised the exception.

Component developers can develop healing connectors, by implementing a solution based on the selected healing pattern (we present our catalog of healing patterns in Section 4). Development activities are supported both at the conceptual and implementation levels. At the conceptual level, component developers can use the structural and behavioral diagrams that describe how healing patterns work. At the implementation level, component developers can use our toolset described in Section 5 to automatically generate scaffolding classes. For example, the scaffolding of a healing connector for the missing initialization of the Spring class `MethodInvoker` can be automatically generated by our toolset; component developers shall add only the specific implementation of the healing strategy that consists of invoking the method `isPrepared` to check whether `MethodInvoker` is ready, and then invoking the methods `prepare` and `invoke`, to initialize the object and invoke again the method that raised the exception. Healing strategies can invoke methods with or without arguments. When invoking methods with arguments, healing strategies can define the argument values according to different strategies. For instance, they can use constant values, or values captured from previously executed calls, or new values defined as a function of the values observed in the previous steps of the computation.

Even if component developers know the OTS products very well, and fully understand the problems to be healed, healing connectors can introduce undesired side-effects in the applications. To verify the correctness of the connectors in the field, component developers design test cases that are executed when connectors are integrated into applications. Test cases shall check that (1) the misuse to be healed generates the expected exception indeed, (2) the fault identification condition characterizes the misuse, and (3) the connector fixes the problems on the fly. Figure 4 shows an example test case for the healing connector associated with the initialization problem of class `MethodInvoker`.

**3.2. Application Developers**

The application developers can integrate healing connectors transparently without knowing either the integration problems that can potentially exist in their applications.

```java
@Test
public void testMethodInvokerConnector() throws IllegalAccessException, InvocationTargetException{
    // deploy the healing connector
    try{
        // call invoke without prepared
        minvoker.invoke();
    } catch (IllegalAccessException ise){
        // connector should have healed the fault
        Assert.fail("Should not have raised IllegalAccessException");
    }

    finally{
        // undeploy the healing connector
        MethodInvokerConnectorDeployment.disableMonitoring();
    }
}
```

Fig. 4. JUnit test cases for the healing connector that fixes the missing initialization of the Spring class `MethodInvoker`.

Component developers can identify a suitable healing pattern, by checking if any of the strategies implemented by the patterns in the catalog can fix on-the-fly the execution that raised the exception.

Component developers can develop healing connectors, by implementing a solution based on the selected healing pattern (we present our catalog of healing patterns in Section 4). Development activities are supported both at the conceptual and implementation levels. At the conceptual level, component developers can use the structural and behavioral diagrams that describe how healing patterns work. At the implementation level, component developers can use our toolset described in Section 5 to automatically generate scaffolding classes. For example, the scaffolding of a healing connector for the missing initialization of the Spring class `MethodInvoker` can be automatically generated by our toolset; component developers shall add only the specific implementation of the healing strategy that consists of invoking the method `isPrepared` to check whether `MethodInvoker` is ready, and then invoking the methods `prepare` and `invoke`, to initialize the object and invoke again the method that raised the exception. Healing strategies can invoke methods with or without arguments. When invoking methods with arguments, healing strategies can define the argument values according to different strategies. For instance, they can use constant values, or values captured from previously executed calls, or new values defined as a function of the values observed in the previous steps of the computation.

Even if component developers know the OTS products very well, and fully understand the problems to be healed, healing connectors can introduce undesired side-effects in the applications. To verify the correctness of the connectors in the field, component developers design test cases that are executed when connectors are integrated into applications. Test cases shall check that (1) the misuse to be healed generates the expected exception indeed, (2) the fault identification condition characterizes the misuse, and (3) the connector fixes the problems on the fly. Figure 4 shows an example test case for the healing connector associated with the initialization problem of class `MethodInvoker`.

**3.2. Application Developers**

The application developers can integrate healing connectors transparently without knowing either the integration problems that can potentially exist in their applications.
or the healing strategies implemented by the healing connectors. The application developers download the connectors provided by the component developers for the OTS components integrated into their applications.

The connectors are automatically injected into the applications with procedures that depend on the technology. Our prototype uses aspect-oriented techniques. Regardless of the injection technology, connectors are activated only when exceptions are raised, thus developers can inject all available connectors without impacting on the application behavior.

The application developers validate the application augmented with healing connectors 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 with the healing connectors, to verify both that healing connectors do not introduce unexpected side effects when activated and do not incidentally interact or conflict with each other.

When healing connectors working in-the-field are activated, application developers can collect the log files that are automatically produced by the healing connectors and quickly produce a permanent fix for the experienced problems. When the application developers deploy a permanent fix to the problems, the corresponding exceptions should not be raised any more and the connectors will not be invoked again, but can stay silently in the application.

The execution of the system test cases, which cover normal scenarios, and the test cases associated with the installed healing connectors, which cover the scenarios that activate the connectors, provide confidence about the correctness of the system that includes the connectors, but cannot guarantees that the healing connectors introduce no side effects when activated. In the future, we intend to investigate the use of verifications techniques to verify systems augmented with healing connectors.

4. HEALING PATTERNS

In this section, we describe four healing patterns that correspond to general strategies to handle common OTS integration problems. Each pattern focuses on a specific class of integration problems that we experienced in our empirical investigation with various OTS products, such as the Sun JDK runtime environment and its libraries [Sun 2012], the Spring framework [SpringSource 2012], the JBoss systems [JBoss 2012] and various Apache systems [Apache 2012b]. We determined healing patterns by browsing fault repositories of OTS products, collecting information about commonly experienced misuses and integration problems, investigating the problems to identify classes of faults, and finally defining healing strategies to handle the classes of integration faults.

The four patterns presented in this article represent a first step towards the definition of a catalog of healing patterns that can be used by component developers to implement healing connectors for handling OTS integration problems. Even if this initial catalog is limited to four patterns, it has already demonstrated its effectiveness to heal many OTS integration problems, as further detailed in Section 6.

The healing solution implemented by each pattern is based on three steps: (i) the detection step determines the occurrence of an erroneous behavior, typically by catching a raised exception, (ii) the fault identification step analyzes the state of the system to identify the cause of the detected erroneous behavior, (iii) the failure recovery step repairs the identified problem. Each pattern describes both the actions that must be executed to solve the corresponding integration problem, namely the healing strategy, and the structure and behavior of the working solution, represented as UML class and sequence diagrams, to guide the implementation.

In the rest of this section, we describe our four healing patterns: Parameter Translator, Component Preparation, Alternative Operation, and Environmental Changes. The
Parameter Translator, Component Preparation and Environmental Changes patterns share the core idea of retrying a failed operation after eliminating the causes of the failure, while the Alternative Operation pattern exploits the idea of substituting a failed operation with another one (or a sequence of operations) that can lead to a result equivalent to the one expected from the failed operation.

We present the healing pattern in the Alexandrian form [Alexander et al. 1977] that consists of a Name that identifies the pattern, a Context that describes how the integration problem addressed by the pattern occurs, a Solution that describes the structure and behavior of the solution, a Resulting context that describes the effects and consequences of using the pattern, the Forces that describe criteria that can justify using or rejecting the pattern as solution to a specific problem, and finally Known uses that describe some real-world use cases of the pattern.

4.1. Pattern Name: Parameter Translation

Context. Client components can correctly bind to the target OTS component, call its methods and exchange syntactically valid parameters, but parameter values are sometimes semantically incorrect and thus cannot be adequately processed by the target OTS component. This scenario typically happens when application developers reuse the same OTS components in multiple projects without noticing that the new usage context implies the generation of unexpected parameter values.

Problem. The problem addressed by this design pattern is an invalid parameter fault that occurs when an application invokes a method of a component with a parameter value that does not meet the method requirements. The callee component cannot interpret the data to perform its operation and in most cases raises an exception to signal the problem to the caller. Examples include a caller component that invokes a method with parameter values outside the expected capacity or size limits, or in inconsistent formats like malformed string parameters. The responsibility of the failure is mainly on the caller component that invokes the method with the wrong parameter values.

Solution. The detection step consists of catching an exception that indicates a possible invalid parameter value. The specific type of the exception depends on the problem addressed by the component developers. For instance, healing can be activated by a URISyntaxException if the addressed problem is specifically due to a malformed URI or by a IllegalArgumentException if it is due to an incorrect argument such as a malformed string.

The fault identification step consists of inspecting the parameters that likely caused the exception and checking whether the parameters include illegal values that can be translated into a legal format. If the strategy identifies a translation to a legal format, the connector heals the parameter; otherwise, it either applies the next healing strategy, if any, or propagates the exception to the caller component.

The failure recovery step consists of a change parameter and retry strategy obtained by executing two actions, either replacing or translating the illegal parameter values with legal ones, and invoking again the method that raised the initial exception. As shown in Figure 5, the entities that collaborate in a change parameter and retry strategy are: (i) the caller and callee components, (ii) the interceptor that intercepts the invocations between the caller and callee components and catches the exception, (iii) the translator and controller that implement the solution to the detected problem by respectively transforming the parameter values (method transformData of class Translator) and reinvoking the original operation (method retryCall of class Controller). Figure 6 shows the collaborations between these elements.
**Resulting Context.** When invalid parameter values result in the generation of an exception, an implementation of the parameter translation healing pattern can fix on-the-fly the invalid parameter values generated by the caller, thus transparently enabling the callee to complete the requested operation.

**Forces**

— To implement a Parameter Translation healing pattern, component developers need to translate a semantically erroneous but syntactically compatible request into a request that is both syntactically and semantically legal. This translation can be done only if the erroneous request already includes all the information necessary to build the corresponding legal request.

— The data produced by a client component and the data accepted by a target OTS component satisfy a (sometimes implicit) data model. Communication between the client and the OTS components is successful only if such data models are compatible. In many cases, incompatibilities are limited to few specific cases that can be easily and quickly fixed with a simple parameter translation. This is the best use case for this healing pattern. In other cases, the effort necessary to design parameter translation can be high and the use of this healing pattern, even if feasible, is discouraged.

**Known Uses**

— Developers reusing the ActiveMQ messaging broker reported that constructing a JavaNet URI object with underscore characters raises a `URISyntaxException` (see bug ID AMQ-1188\(^2\)). We successfully implemented this pattern to heal this fault by automatically replacing the string parameter hostname with its corresponding IP address and re-invoking the original operation.

— Developers reusing the Apache HTTPClient library reported that constructing an `GetMethod` object with invalid characters, such as `[and]`, results in an `IllegalArgumentException` (see bug ID HTTPCLIENT-678\(^3\)). We successfully implemented the parameter translation healing pattern to heal this fault by automatically translating the invalid characters of the parameter and re-invoking the original operation.


4.2. Pattern Name: Component Preparation

**Context.** Client components can correctly bind to the target OTS component, call its methods and exchange data, but in some cases, the target OTS component is not in a state ready to serve the request issued by the caller. This scenario typically happens when an OTS component has an internal state that constrains the requests that can be accepted and application developers misunderstood the OTS component API by erroneously implementing client components that generate unacceptable sequences of requests.

**Problem.** The problem addressed by this design pattern is an *incorrect usage of interfaces fault* that occurs when an application invokes a method of a component and the component cannot serve it. The callee component cannot execute the requested operation and usually raises an exception to indicate that an unexpected request has been received. Examples include using OTS components that require initialization without having initialized them before the invocation, and incidentally using dead connections. The responsibility of the failure is mainly on the caller component that invokes operations without following the interaction protocol required by the OTS components.

**Solution.** The *detection* step consists of catching exceptions that indicate that the target OTS component is not ready to serve a given request. The specific type of the
exception depends on the problem to heal, even if healing is typically activated with an exception of type `IllegalStateException`.

The *fault identification* step consists of inspecting the state of the OTS component through inspector methods, and checking whether the OTS component state can be safely turned into a legal state that would accept the operation that previously failed. If the strategy can turn the OTS component into a legal state, the connector heals the state; otherwise it either applies the next healing strategy, if any, or propagates the exception to the caller component. Note that component developers know which are the pure inspector methods that can be executed from healing connectors because they develop both the healing connectors and the OTS components.

The *failure recovery* step consists of a *call operations and retry strategy* that is obtained by invoking a suitable sequence of methods to change the component state and invoking again the method that raised the exception. As shown in Figure 7, the entities that collaborate to achieve this solution are: (i) the caller and callee components, (ii) the *interceptor* that catches the exception generated by the interaction between the caller and callee components, (iii) the two *controllers* that implement the healing strategy by executing a sequence of operations that change the state of the OTS component (execution of method `callOperations` implemented by class `Controller`) and reinvoking the original operation (execution of method `retryCall` implemented by class `Controller`). Figure 8 shows the collaboration between these elements.

Our approach assumes that patterns are manually implemented. Techniques for mining common usage patterns of APIs, such as the ones proposed by Wasylikowski and Zeller [2011], can be used to automatically generate healing connectors that implement this pattern.

**Resulting Context.** When an exception due to an unprepared OTS component is detected, this healing pattern fixes the state of the callee component on-the-fly, thus enabling the callee to transparently complete the requested operation.

**Forces**

— Component developers can design a healing connector that automatically prepares a OTS component to serve a request that is not accepted at a given time only if the state that should be assigned to the OTS component can be derived from the current state of the component and the information associated to the request that generated the exception. This condition can be easily satisfied when the preparation can be executed without using information available to client components only.
Known Uses

—Developers reusing the Spring Framework reported that invoking method `invoke` without invoking method `prepare` before raises an `IllegalStateException` (see bug ID SPR-3386\(^4\)). We successfully applied the Component Preparation pattern to heal this fault by automatically invoking the method `prepare`, thus initializing the object, before re-invoking the method `invoke`.

—Developers reusing the Sun JavaMail library reported that sending a message using a disconnected JavaMail `Transport` raises an `IllegalStateException` (see bug ID GERONIMO-1669\(^5\)). We successfully instantiated this pattern to heal this fault by automatically invoking method `connect` before re-invoking the method `send`.

4.3. Pattern Name: Alternative Operation

Context. Client components can correctly bind and interact with the target OTS component, but an operation implemented by the target OTS component includes a latent fault that generates an exception. This pattern applies when component developers do not fix or postpone the fixing of faults typically because of the costs of fixing.

\(^4\)http://jira.springframework.org/browse/SPR-3386.
**Problem.** The problem addressed by this design pattern is a faulty method fault, which occurs when an application invokes a faulty method of an OTS component. The invoked operation returns an exception instead of returning the expected result. An example of unit faults that are often lately not fixed are faulty deprecated methods. The responsibility of the failure is shared between the caller and callee components depending on the specific case. For example, a failure generated by a faulty algorithm is mainly a responsibility of the callee component, while a failure generated by a faulty deprecated method is mainly a responsibility of the caller component.

**Solution.** The detection step consists of catching exceptions, depending on the specific type of fault that is expected.

The fault identification step consists of checking both the parameters and the OTS component state to determine if the exception has been raised by the known faulty behavior of the OTS components. The implementation of this step varies case by case.

The failure recovery step consists of a replace calls strategy obtained by automatically substituting the failed invocation with invocations of methods that produce results equivalent to the method that failed. As shown in Figure 9, the elements that are part of the solution are: (i) the caller and callee components, (ii) the interceptor that intercepts the invocations between caller and callee components and catches the exception, (iii) the controller that redirects the invocation to an equivalent sequence of operations (implemented by the method callOperations in class Controller). Figure 10 shows the collaborations between these elements.

Our approach assumes that patterns are manually implemented. Techniques for automatically identifying workarounds, such as the ones proposed by Carzaniga et al. [2010], can be used to automatically generate healing connectors that implement this pattern.

**Resulting Context.** When an exception is detected, this healing pattern can modify the execution at runtime and on-the-fly, by turning the failing invocation into a sequence of equivalent invocations. The execution transparently returns to the client component.

**Forces**

—Component developers can effectively use this healing pattern whenever there exist simple workarounds for faults that are expensive to be fixed.
The implementation of this pattern requires the existence of operations alternative to the faulty one. This requirement is usually satisfied with deprecated methods. Recent studies show the redundancy is more frequent than initially expected [Carzaniga et al. 2009].

**Known Uses**

—Developers reusing the Sun JRE 1.6 JavaNet reported that invoking the deprecated method `url.toURI` when the `url` object includes field values that do not strictly satisfy the RFC2396 format raises a `URISyntaxException` (see, for instance, bug ID TAPESTRY-2074\(^6\)). We successfully implemented this pattern to heal this fault by automatically replacing the invocation of method `url.toURI` with the construction of a new URI object using the fields of the URL object.

—Developers reusing the Sun JRE JavaLang 1.6 reported that invoking method `classLoader.loadClass` with an array syntax parameter under Java 6 results in

---

\(^6\)https://issues.apache.org/jira/browse/TAPESTRY-2074.
Fig. 11. Structure of the environmental changes healing pattern.

4.4. Pattern Name: Environmental Changes

**Context.** OTS components may require some set-up operations to behave correctly. When application developers do not fully understand how to complete the setup, applications that include badly configured OTS components can experience failures.

**Problem.** The problem addressed by this design pattern is an environmental fault that occurs when either the environment that executes the OTS component is not properly setup or the OTS component has not been properly setup according to the environment. Executions affected by an improper configuration often result in exceptions caused by folders that are not found, files that are not readable or connections that cannot be established. In many cases, the responsible of this kind of problems are the system administrators who do not properly configure the OTS components and the environment at deploy time.

**Solution.** The detection step consists of catching exceptions, depending on the specific type of fault that is expected.

The fault identification step consists of checking the state of the environment and the configuration of the OTS component to identify incorrect configurations that can be healed.

The failure recovery step consists of a change environment and retry strategy that modifies the environment to enable the operations that failed and re-invokes the method that raised the exception. As shown in Figure 11, the entities that collaborate in the solution are: (i) the caller and callee components, (ii) the interceptor that catches the exception, (iii) the extender that suitably modifies the environment (execution of a ClassNotFoundException (see for instance the bugs GERONIMO-3142 or JBAS-4491). We successfully implemented this pattern to heal this fault by automatically replacing the invocation of method classLoader.loadClass with the invocation of operation Class.forName. Notice that by replacing the call to loadClass with a call to Class.forName we may not always fix the fault, because the call to Class.forName may cause the execution of a different ClassLoader. The fault identification step of the healing connector must also check whether the ClassLoader used in the invocation of loadClass is the context ClassLoader, which is the one used by Class.forName. If this condition does not hold, the healing cannot be executed.

8https://jira.jboss.org/jira/browse/JBAS-4491.
method changEnvt implemented by class Extender) and the controller that re-invokes the operation that failed (method retryCall of class Controller). Figure 12 shows the collaborations between these entities.

Resulting Context. When a configuration problem is detected, this healing pattern activates a healing solution that changes either the environment or the component configuration to set a proper configuration and successfully complete the execution. This solution is usually slower than having the OTS component working with an optimized configuration defined by system administrators, and acts as a temporary replacement of the correct configuration until the system administrator fixes the issue.

Forces

—Component developers can design general solutions to common configuration problems that can be addressed with a general approach. However, component developers need information about the local environment to implement the healing strategies. To satisfy this requirement, healing connectors support the use of configuration files
that can be configured by system administrators and that include the information necessary to execute the healing strategy.

—This healing pattern is cost-effective when the configuration file provided with the healing connector is simpler to be configured than the OTS component configuration file, which happens when the probability to produce an improper configuration file for the healing connector is low.

Known Uses

—Developers reusing the Xalan XSLT processor reported that failing to deploy the required jar files leads to a TransformerConfigurationException (see bug ID MAGNOLIA-1958\(^9\)). This pattern can be successfully applied to heal this fault by automatically loading the missing jar files and deleting the corrupted directories before reinvoking the original operation.

—Developers reusing the MySQL Java Database Connectivity (JDBC) driver implementation reported that failing to correctly install the database driver implementation leads to a SQLException when invoking DriverManager.getConnection (see, for instance, several messages reported in MySQL official forum). This pattern can be successfully applied to heal this fault by dynamically loading the missing MySQL driver jar file and retrying the call to the original getConnection method.

5. A HEALING CONNECTOR PROTOTYPE TOOLSET

In this section, we describe a prototype implementation of the interceptor and the injection mechanisms that comprise the toolset to support component developers in the development of healing connectors, and a prototype implementation of a toolset that supports application developers in the deployment of healing connectors. The toolset automates many tedious and error prone activities related to the implementation of scaffolding classes and the definition of deployment and configuration files. The toolset is implemented as a collection of Eclipse plug-ins.

5.1. Healing Connectors Implemented with AOP

Our prototype implementation of healing connectors and injection mechanism is based on AspectWerkz [2012]: the interceptor is implemented as an aspect, the healing strategies as standard Java classes, and the connector injection points as pointcuts that point at the locations of the application program that include calls to the OTS component. These locations are specified by using expressions that only refer to the OTS component interface, and thus without requiring any knowledge about the application that integrates the OTS component. The AspectWerkz weaver provides the injection mechanisms.

Figure 13 shows an excerpt of the MethodInvokerInterceptor interceptor designed to catch the missing invocation to the IllegalStateException that the invoke method of the MethodInvoker class in the OTS Spring framework raises if the prepare method has not been previously executed.

When the healing connector is deployed in a target system that includes the Spring framework, every call to method invoke triggers the execution of the advice method detectFailure of interceptor MethodInvokerInterceptor which propagates the original invocation by executing jp.proceed(). If jp.proceed() raises an exception, the interceptor catches the exception, builds a failure context object encapsulating the exception and runtime information contained in the joinpoint jp, loads the healing strategies and delegates the execution of the healing to the strategies. Figure 14 shows the specification of the injection points defined as pointcuts that bind the execution points of the Spring framework.

of the advice method `detectFailure` of aspect `MethodInvokerInterceptor` *around* any call to method `invoke` of class `MethodInvoker`.

There are alternative ways of implementing the healing connectors with AspectWerkz. For instance, aspects could be bound to the exceptions raised by the methods rather than to the method calls. In this case, the code in an aspect could be restricted to the code in the `catch` clause of Figure 13. We preferred to implement interceptors using the `around` advice, which allows to define code that must be executed before and after a method is invoked, because the implementation is ready to support the implementation of new healing patterns that may require the execution of operations before the target method is invoked even if exceptions are not returned.

### 5.2. Generating Healing Connectors

Our toolset produces the Java classes and XML files that comprise the healing connectors. An Eclipse wizard collects data from component developer. The healing connectors are generated in three semi-automated steps: collect information, generate connector, and complete and package. Figure 15 shows the three main steps that are supported by our wizard, and the main components and actors involved in the execution of these steps.

In the *collect information* step, component developers specify the main information needed to generate a healing connector, supported by our wizard that (partially) checks the completeness and consistency of the provided data. In particular, the wizard checks whether all the necessary information is specified, that is all the necessary fields are filled, and no clearly incorrect data are provided, that is the names of methods and
classes correspond to existing entities. Such checks are not exhaustive in the sense that the tool cannot forbid component developers from binding healing connectors to wrong methods. However, these checks can effectively exclude that many wrong inputs are accepted. The main data that are specified in this step are: the name of the healing pattern, the method signature where the healing connector must be attached, and the names of the exceptions to catch.

In the **connector generation** step, our tool automatically generates the healing connector and its XML configuration files according to the data collected in the previous step. A healing connector consists of a bundle with Java interfaces and classes (essentially including the **Interceptor** and ETC classes as shown in Figure 1) and XML configuration files. The **Interceptor** component detects failures, and is implemented as a plain Java class with catch clauses that catch the types of exceptions indicated by component developers in the collection step. In addition to the automated generation of the catch clauses, the **Interceptor** already includes the code necessary to extract and embed runtime data related to the failure in a **failureCtxt** object. Depending on the selected healing pattern, one or more **Extender-Translator-Controller** classes can be automatically generated. Since ETC classes can be part of a **chain of responsibility**, they implement the **HealingStrategy** interface. Our toolset only generates scaffolding ETC classes, thus their behaviors must be manually coded by component developers. Our toolset also automatically produces a suitable set of interfaces and deployment descriptors that specify where the healing connector must be injected, and XML configuration files that include a set of configurable properties that are relevant for the specific healing patterns, and facilitate reuse and adaptability of the healing connector to different OTS components and applications. Finally, the bundle with the healing connector also includes the test cases that validate the correctness of the healing connector once injected in the target environment. The generation of XML and Java source code files is implemented using the Eclipse Java Emitter Templates (JET) engine [Eclipse 2012].

In the **complete and package** step, component developers manually edit the Java classes that implement the specific healing strategy that heals the problem addressed.
Healing connectors can be distributed to application developers by attaching healing connectors to the corresponding faults that are present in fault repositories, can be uploaded on public web repositories, or distributed through the Maven system.

5.3. Deploying Healing Connectors

Our toolset supports application developers in injecting healing connectors within OTS-based systems. The injection process is structured in three steps: collect information, inject connectors and test case execution. An Eclipse wizard supports the execution of these three steps. Figure 16 shows these steps, and the main components and actors involved in their execution.

In the **collect information** step, application developers provide some information about the target OTS component: the path to the binaries of the application and the path to the healing connectors to be injected.

In the **injection** step, our tool uses the information collected in the previous step to automatically inject the connectors classes into the application. The tool generates both a backup copy of the original application binaries and the new application binaries that include healing connectors. Connector injection is implemented with AspectWerkz.

In the **test case execution** step, our tool automatically executes the test cases embedded in the healing connector to simulate the addressed misuse and verify that the resulting integration problem is effectively healed. Our tool also detects and executes the test cases that verify the behavior of the previously deployed connectors, to check for possible interference with the newly deployed healing connector. It is a responsibility of application developers to execute the system test cases to validate that healing connectors do not introduce side effects on the normal behavior of the target application when inactive.
Our tool can also be used to remove the healing connectors that have been already deployed. In particular, the tool keeps a copy of the application and the list of deployed connectors. To remove obsolete connectors, the tool builds a new copy of the application and deploys every connector excluding the ones that must be removed.

6. EMPIRICAL VALIDATION
In this section, we present the results of our empirical validation of the solution presented in this article. We discuss the applicability of healing patterns, describe the experimental setup, and present the results of the experiments about effectiveness and reusability of healing patterns. We conclude by discussing the threats to validity.

6.1. Applicability
To estimate the applicability of our healing patterns, we refer to third-party empirical studies about the type of faults that typically affect software systems. In particular, we refer to the study by Pan et al. [2008] who investigated several open source projects, and classified the typical fixes that are implemented by application developers.

According to the results reported in this study, the healing patterns proposed in this paper can be used to address a relevant range of faults. In fact, three of our healing patterns can be straightforwardly used to implement two of the nine categories of fixes identified by Pan, Kim and Whitehead: the *method call* and *sequence* categories. According to the empirical investigation by Pan, Kim and Whitehead, these two categories correspond to a relevant amount of the fixes implemented for the studied systems: between 22.1% and 31.1% of the total amount of fixes in the considered case studies. The high frequency of the fixes and the heterogeneity of the applications investigated in the empirical study suggest that our healing patterns implement healing procedures that can address many common problems.

Our *environmental changes* pattern does not correspond to fixes reported by Pan, Kim and Whitehead, since they studied only code level problems. Thus, our solution can address a set of fixes larger than the already relevant amount of fixes indicated in the empirical study by Pan, Kim and Whitehead. The empirical experience reported in the next section confirms that several real integration faults with OTS components can be addressed with our healing patterns.

The healing patterns defined in this article represent a starting point for the research about in-field healing through healing connectors. In the future, we envision the definition of additional healing patterns, for instance, inspired by the error recovery strategies discussed in Rinard [2007], thus further extending the set of faults that can be addressed with the solution presented in this article.

6.2. Experimental Setup
We identify the case studies for the empirical validation by browsing fault repositories and official and unofficial technical forums available on the Web to find component based applications with integration problems that derive from OTS components. We found many integration faults involving OTS components that are due to misuse, misunderstanding or incorrect configuration of the OTS components. For our empirical investigation, we selected 18 case studies that well represent open source and commercial software systems that integrate OTS components. We selected the case studies according to the following criteria:

—the coverage of the healing patterns defined in this article;
—the sample of different domains, including enterprise systems, standard Java systems, Web applications, and middleware;
— the nature of the OTS components, spanning from complex frameworks, like Spring, to simple libraries, like Xalan;
— the size of the application, ranging from small applications (about 50 Loc) to large applications (more than 500 KLoc);
— the type of the faults that must depend on an application that inappropriately uses an OTS component.

According to the data reported in the former subsection, the case studies selected in this way could match at least 22% of the faults in an application.

We reproduced in-house the problems that we found, and we distinguished three cases depending on the availability of the OTS components and the application:

— both the OTS components and the application are available. In this case, we executed the experiments directly on the application.
— the OTS components are available, but the application is not. In this case, we developed in-house applications that use the OTS components and reproduce the reported faults, to execute the experiments.
— the OTS components are not available. We did not include this case in our experiments because the presence of third-party components is necessary for the validity of the experiments.

For each problem, we generated healing connectors from our catalog of healing patterns, either by designing a new healing connector or by reusing an existing healing connector developed for another case study. We generated and deployed all healing connectors with the toolset described in Section 5.

6.3. Effectiveness

This set of experiments aims to study the feasibility of the solution presented in this article. We evaluate feasibility as the range (types of applications, types of OTS components, size of applications, types of integration problems) of the cases that can be successfully addressed by healing connectors derived from our healing patterns. For each case study, we proceeded as follows. We started by reproducing the failure reported on the Web: We installed the system to be investigated in-house, reproduced the execution or the configuration that is reported to cause the OTS integration problem, and verified that it leads to a failure. In the considered cases, the failures consist of system crashes or incorrect system responses combined with exception messages traced in log files or displayed on the system console. We then tried to solve the problem with healing connectors: We generated healing connectors from our healing patterns or reused previously generated connectors when available; we injected the connectors in the application, executed both the application test cases, when available, and the connector test cases, and verified both the successful healing of the application and the absence of side-effects derived from the added connectors.

We have been able to reuse many connectors to heal integration problems both for the same OTS components in different applications, and for different OTS components. We implemented only 10 healing connectors to heal the 18 selected OTS integration faults.

Table I lists the Applications and the considered Failures. For each application, we give 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 OTS component that causes the problems when integrated in the application, provide a short informal Description of the integration problem and some Bug information that points to the bug report, and give the bug priority (Blocker, Critical, Major, Minor, Trivial) when applicable. Since the investigated bugs are caused
<table>
<thead>
<tr>
<th>Application</th>
<th>Category</th>
<th>Size (loc)</th>
<th>OTS Comp.</th>
<th>Description</th>
<th>Bug ID, Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache ActiveMQ (v.4.1.0)</td>
<td>Messaging broker</td>
<td>120K</td>
<td>Sun JRE (JavaNet component)</td>
<td>Starting ActiveMQ raises an exception URISyntaxException when the hostname</td>
<td><a href="http://issues.apache.org/activemq/browse/AMQ-1188">http://issues.apache.org/activemq/browse/AMQ-1188</a>, Minor</td>
</tr>
<tr>
<td>Apache ServiceMix (v.3.2.1)</td>
<td>Enterprise service bus</td>
<td>111K</td>
<td>Sun JRE (JavaNet component)</td>
<td>Starting ServiceMix raises an exception URISyntaxException when the hostname</td>
<td><a href="https://issues.apache.org/activemq/browse/SM-492">https://issues.apache.org/activemq/browse/SM-492</a>, Major</td>
</tr>
<tr>
<td>Application reproducing bug report HTTPCLIENT-678</td>
<td>Java application</td>
<td>39</td>
<td>Apache HttpComponents (HttpClient component)</td>
<td>HttpMethod raises an exception IllegalArgumentException when sending a request containing illegal characters</td>
<td><a href="https://issues.apache.org/jira/browse/HTTPCLIENT-678">https://issues.apache.org/jira/browse/HTTPCLIENT-678</a>, Major</td>
</tr>
<tr>
<td>Application reproducing failure reported in official forum</td>
<td>Java application</td>
<td>98</td>
<td>Apache HttpComponents (HttpClient component)</td>
<td>HttpMethod raises an exception IllegalArgumentException when sending a request containing illegal characters</td>
<td>Failure reported in Jakarta Commons mailing-list</td>
</tr>
<tr>
<td>JBoss Web services</td>
<td>Web services framework</td>
<td>89</td>
<td>Sun JRE (JavaLang component)</td>
<td>Classloader raises an exception ClassNotFoundException when the web service endpoint class name contains extra spaces</td>
<td><a href="https://jira.jboss.org/jira/browse/JBWS-1107">https://jira.jboss.org/jira/browse/JBWS-1107</a>, Major</td>
</tr>
<tr>
<td>Application reproducing bug report SPR-2311</td>
<td>Spring application</td>
<td>59</td>
<td>Spring Framework (JDBC Component)</td>
<td>DriverManagerDataSource raises an exception SQLException when loading a database driver with an url containing spaces and backspaces</td>
<td><a href="http://issues.springframework.org/browse/SPR-2311">http://issues.springframework.org/browse/SPR-2311</a>, Minor</td>
</tr>
<tr>
<td>Application reproducing bug report SPR-713</td>
<td>Spring application</td>
<td>73</td>
<td>Spring Framework (Beans Component)</td>
<td>ClassUtils raises an exception ClassNotFoundException when trying to load a Class property with a name surrounded with whitespaces</td>
<td><a href="http://jira.springframework.org/browse/SPR-713">http://jira.springframework.org/browse/SPR-713</a>, Major</td>
</tr>
<tr>
<td>Application reproducing bug report GERONIMO-1669</td>
<td>J2EE Web application</td>
<td>168</td>
<td>Apache Geronimo (JavaMail component)</td>
<td>Disconnected smtp transport raises an exception IllegalStateException when sending a mail Message</td>
<td><a href="http://issues.apache.org/jira/browse/GERONIMO-1669">http://issues.apache.org/jira/browse/GERONIMO-1669</a>, Major</td>
</tr>
<tr>
<td>Test case reproducing bug report SPR-3386</td>
<td>Java application</td>
<td>16</td>
<td>Spring Framework (SpringCORE component)</td>
<td>Unprepared method invoke raises an exception IllegalStateException when invoking the specified method</td>
<td><a href="http://jira.springframework.org/browse/SPR-3386">http://jira.springframework.org/browse/SPR-3386</a>, Major</td>
</tr>
</tbody>
</table>

(Continued)
<table>
<thead>
<tr>
<th>Application</th>
<th>Category</th>
<th>Size (loc)</th>
<th>OTS Comp.</th>
<th>Description</th>
<th>Bug ID, Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Tiles (v.2.0.6)</td>
<td>Web application template framework</td>
<td>11K</td>
<td>Sun JSE (JavaLang component)</td>
<td>Protected reflected method raises an exception IllegalAccessException when accessed with limited permission</td>
<td><a href="https://issues.apache.org/struts/browse/TILES-267">https://issues.apache.org/struts/browse/TILES-267</a>, Major</td>
</tr>
<tr>
<td>Apache Tiles (v.2.0.8)</td>
<td>Web application template framework</td>
<td>11K</td>
<td>Sun JSE (JavaLang component)</td>
<td>Protected reflected method raises an exception IllegalAccessException when accessed with limited permission</td>
<td><a href="https://issues.apache.org/struts/browse/TILES-286">https://issues.apache.org/struts/browse/TILES-286</a>, Major</td>
</tr>
<tr>
<td>Apache Geronimo (v.2.0-M3)</td>
<td>J2EE Application server</td>
<td>180K</td>
<td>Sun JRE 1.6 (ClassLoader component)</td>
<td>At startup, a faulty implementation of classloader.loadClass raises an exception ClassNotFoundException when used to load an array with name specified with array syntax</td>
<td><a href="http://issues.apache.org/jira/browse/GERONIMO-3142">http://issues.apache.org/jira/browse/GERONIMO-3142</a>, Major</td>
</tr>
<tr>
<td>JBoss Application platform (v.5.0.0.B2)</td>
<td>J2EE Application server</td>
<td>686K</td>
<td>Sun JRE 1.6 (ClassLoader component)</td>
<td>At startup, a faulty implementation of classloader.loadClass raises an exception ClassNotFoundException when used to load an array with name specified with array syntax</td>
<td><a href="http://jira.jboss.org/jira/browse/JBAS-4491">http://jira.jboss.org/jira/browse/JBAS-4491</a>, Major</td>
</tr>
<tr>
<td>Apache Tapestry (v.5.0.7)</td>
<td>Web application framework</td>
<td>157K</td>
<td>Sun JRE 1.6 (JavaNet component)</td>
<td>Retrieving the URI equivalent to a URL raises an URISyntaxException when the URL is not strictly correctly formatted.</td>
<td><a href="https://issues.apache.org/jira/browse/TAPESTRY-2074">https://issues.apache.org/jira/browse/TAPESTRY-2074</a>, Blocker</td>
</tr>
<tr>
<td>Test case reproducing bug report</td>
<td>Java application</td>
<td>53</td>
<td>Sun JRE (JavaNet component)</td>
<td>Retrieving the URI equivalent to a URL raises an URISyntaxException when the URL is not strictly correctly formatted.</td>
<td><a href="https://issues.apache.org/jira/browse/CXF-1583">https://issues.apache.org/jira/browse/CXF-1583</a>, Major</td>
</tr>
<tr>
<td>Magnolia CMS (v.3.5.1)</td>
<td>Enterprise content management system</td>
<td>58K</td>
<td>Xalan XSLT</td>
<td>Magnolia cannot run and raises an error TransformerFactoryConfigurationException when initializing its content repositories</td>
<td><a href="http://jira.magnolia.info/browse/MAGNOLIA-1958">http://jira.magnolia.info/browse/MAGNOLIA-1958</a>, Blocker</td>
</tr>
<tr>
<td>Application reproducing failure reported in official forum</td>
<td>Java application</td>
<td>30</td>
<td>MySQL JDBC driver</td>
<td>Developer application cannot find a suitable driver implementation to create a connection to MySQL database and raises an exception SQLException</td>
<td>Failure reported in MySQL forum</td>
</tr>
<tr>
<td>Application reproducing failure reported in official forum</td>
<td>Java application</td>
<td>22</td>
<td>PostgreSQL JDBC driver</td>
<td>Developer application cannot find a suitable driver implementation to create a connection to PostgreSQL database and raises an exception SQLException</td>
<td>Failure reported in PostgreSQL forum</td>
</tr>
</tbody>
</table>
by applications that inappropriately use the OTS components, most of the bug reports refer to the application bug repository. In few cases, the bug report refers to the OTS component, this happens when developers decided to make the implementation of the OTS component more robust to misuses, even if the problem was caused by the application and not by the OTS component.

Table II summarizes the experimental results. The table lists the analyzed Cases, and the designed Healing Connectors. For each case, we report the application and the OTS Component that causes the problem, and a description of the problem. For each healing connector, we provide a short Description of the healing pattern implemented by the connector, indicate the amount of times the healing connector is activated in the failing execution (# Activations), and refer to the healing Pattern implemented by the connector: P1 stands for pattern Parameter translation, P2 stands for pattern Component preparation, P3 stands for pattern Alternative operation and P4 stands for pattern Environmental changes. The connectors developed from our connector patterns heal all faults. In some cases, the same connector heals the effects of the same fault multiple times in a single execution, this is the reason for values greater than 1 in column # Activations.

In all the cases reported in Table II, healing connectors successfully healed OTS integration faults without introducing side effects. The range of applicative domains and size covered by the considered applications and OTS components suggest that healing connectors are a feasible and general solution for many component-based software systems. The experiments shows also that our healing patterns are widely applicable, confirming the empirical data reported in Pan et al. [2008].

While empirically experiencing the effectiveness of healing connectors, we also measured the overhead introduced in the system by the presence of the healing connectors. Results confirmed that the solution is lightweight. The overhead produced by the healing connectors while inactive is negligible; while the overhead produced by the activation of healing connectors has been always extremely low (30 ms in the worst case).

6.4. Reusability

The solution presented in this article supports three levels of reuse: conceptual reuse of the healing patterns, adaptation of the healing connectors, and reuse of the healing connectors.

Conceptually reusing a healing pattern consists of applying a healing pattern to solve multiple problems. Since healing patterns capture general problems and corresponding solutions, we expect high conceptual reuse.

Adapting a healing connector consists of reusing a concrete connector across applications by modifying only configuration and deployment files. We expect to adapt connectors to address similar problems for different OTS systems and OTS components that admit similar solutions.

Reusing a healing connector consists of reusing a concrete connector without changes. We expect to reuse healing connectors when the same OTS integration fault is experienced in different systems that include the same OTS component.

The reuse experimented with 18 healing connectors does not provide statistically significant data, but can give some confidence about the degree of reuse expected from our technology. Table III summarizes the empirical results on the reuse of healing connectors.

Column Healing pattern indicates the healing patterns that have been used to generate concrete healing connectors. Column \# of applications indicates the number of times a healing pattern has been used to produce a concrete healing connectors. This value represents the level of conceptual reuse of our solution. As expected, healing
<table>
<thead>
<tr>
<th>Application</th>
<th>OTS Comp.</th>
<th>Description</th>
<th>Healing Connector</th>
<th># Act.</th>
<th>Pat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache ActiveMQ (v.4.1.0)</td>
<td>Sun JRE (JavaNet component)</td>
<td>Starting ActiveMQ raises an exception URISyntaxException when the hostname contains underscore characters</td>
<td>Replace the hostname string parameter by its IP address, and re-invoke the original operation</td>
<td>3</td>
<td>P1</td>
</tr>
<tr>
<td>Apache ServiceMix (v.3.2.1)</td>
<td>Sun JRE (JavaNet component)</td>
<td>Starting ServiceMix raises an exception URISyntaxException when the hostname contains underscore characters</td>
<td>Replace the hostname string parameter by its IP address, and re-invoke the original operation</td>
<td>1</td>
<td>P1</td>
</tr>
<tr>
<td>Application reproducing bug report HTTPCLIENT-678</td>
<td>Apache HttpComponents (HttpClient component)</td>
<td>HttpMethod raises an exception IllegalArgumentException when sending a request containing illegal characters</td>
<td>Correct the illegal characters, and re-invoke the original operation</td>
<td>1</td>
<td>P1</td>
</tr>
<tr>
<td>Application reproducing failure reported in official forum</td>
<td>Apache HttpComponents (HttpClient component)</td>
<td>HttpMethod raises an exception IllegalArgumentException when sending a request containing illegal characters</td>
<td>Correct the illegal characters, and re-invoke the original operation</td>
<td>1</td>
<td>P1</td>
</tr>
<tr>
<td>JBoss Web services</td>
<td>Sun JRE (JavaLang component)</td>
<td>Classloader raises an exception ClassNotFoundException when the web service endpoint classname contains extra spaces</td>
<td>Trim the extra spaces, and re-invoke the original operation</td>
<td>1</td>
<td>P1</td>
</tr>
<tr>
<td>Application reproducing bug report SPR-2311</td>
<td>Spring Framework (JDBC Component)</td>
<td>DriverManager-DataSource raises an exception SQLException when loading a database driver with an url containing spaces and backspaces</td>
<td>Trim the extra spaces, and re-invoke the original operation</td>
<td>2</td>
<td>P1</td>
</tr>
<tr>
<td>Application reproducing bug report SPR-713</td>
<td>Spring Framework (Beans Component)</td>
<td>ClassUtils raises an exception ClassNotFoundException when trying to load a Class property with a name surrounded with whitespaces</td>
<td>Trim the extra spaces, and re-invoke the original operation</td>
<td>1</td>
<td>P1</td>
</tr>
<tr>
<td>Application reproducing bug report GERONIMO-1669</td>
<td>Apache Geronimo (JavaMail component)</td>
<td>Disconnected smtp transport raises an exception IllegalStateException when sending a mail Message</td>
<td>Call the transport connect() operation, and re-invoke the original operation</td>
<td>1</td>
<td>P2</td>
</tr>
</tbody>
</table>

(Continued)
<table>
<thead>
<tr>
<th>Case</th>
<th>Healing Connector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test case reproducing bug report SPR-3386</td>
<td>Spring Framework (SpringCORE component) <strong>Unprepared method invoke raises an exception</strong> <em>IllegalStateException</em> when invoking the specified method</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Call the method invoker prepare() operation, and re-invoke the original operation</td>
</tr>
<tr>
<td><strong>Act.</strong></td>
<td><strong>Pat.</strong></td>
</tr>
<tr>
<td>1</td>
<td>P2</td>
</tr>
<tr>
<td>Apache Tiles (v.2.0.6)</td>
<td>Sun JSE (JavaLang component) <strong>Protected reflected method raises an exception</strong> <em>IllegalAccessException</em> when accessed with limited permission</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Call the setAccessible operation, and re-invoke the original operation</td>
</tr>
<tr>
<td><strong>Act.</strong></td>
<td><strong>Pat.</strong></td>
</tr>
<tr>
<td>5</td>
<td>P2</td>
</tr>
<tr>
<td>Apache Tiles (v.2.0.8)</td>
<td>Sun JSE (JavaLang component) <strong>Protected reflected method raises an exception</strong> <em>IllegalAccessException</em> when accessed with limited permission</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Call the setAccessible operation, and re-invoke the original operation</td>
</tr>
<tr>
<td><strong>Act.</strong></td>
<td><strong>Pat.</strong></td>
</tr>
<tr>
<td>1</td>
<td>P2</td>
</tr>
<tr>
<td>Apache Geronimo (v.2.0-M3)</td>
<td>Sun JRE 1.6 (ClassLoader component) <strong>At startup, a faulty implementation of classloader.loadClass raises an exception</strong> <em>ClassNotFoundException</em> when used to load an array with name specified with array syntax</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Substitute the invocation of classLoader.loadClass() with Class.forName()</td>
</tr>
<tr>
<td><strong>Act.</strong></td>
<td><strong>Pat.</strong></td>
</tr>
<tr>
<td>2</td>
<td>P3</td>
</tr>
<tr>
<td>JBoss Application platform (v.5.0.0.B2)</td>
<td>Sun JRE 1.6 (ClassLoader component) <strong>At startup, a faulty implementation of classloader.loadClass raises an exception</strong> <em>ClassNotFoundException</em> when used to load an array with name specified with array syntax</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Substitute the invocation of classLoader.loadClass() with Class.forName()</td>
</tr>
<tr>
<td><strong>Act.</strong></td>
<td><strong>Pat.</strong></td>
</tr>
<tr>
<td>7</td>
<td>P3</td>
</tr>
<tr>
<td>Apache Tapestry (v.5.0.7)</td>
<td>Sun JRE 1.6 (JavaNet component) <strong>Retrieving the URI equivalent to a URL raises an URISyntaxException when the URL is not strictly correctly formatted</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Substitute the invocation of url.toURI() with the construction of a new URI object using the fields of the URL</td>
</tr>
<tr>
<td><strong>Act.</strong></td>
<td><strong>Pat.</strong></td>
</tr>
<tr>
<td>1</td>
<td>P3</td>
</tr>
<tr>
<td>Test case reproducing bug report</td>
<td>Sun JRE (JavaNet component) <strong>Retrieving the URI equivalent to a URL raises an URISyntaxException when the URL is not strictly correctly formatted</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Substitute the invocation of url.toURI() with the instantiation of a new URI object</td>
</tr>
<tr>
<td><strong>Act.</strong></td>
<td><strong>Pat.</strong></td>
</tr>
<tr>
<td>1</td>
<td>P3</td>
</tr>
<tr>
<td>Magnolia CMS (v.3.5.1)</td>
<td>Xalan XSLT <strong>Magnolia cannot run and raises an error TransformerFactoryConfigurationException when initializing its content repositories</strong></td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Dynamically load the jar files, delete the corrupted repository directories, and re-invoke the original operation</td>
</tr>
<tr>
<td><strong>Act.</strong></td>
<td><strong>Pat.</strong></td>
</tr>
<tr>
<td>1</td>
<td>P4</td>
</tr>
</tbody>
</table>
Table II. (Continued)

<table>
<thead>
<tr>
<th>Case</th>
<th>Healing Connector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application reproducing failure reported in official forum</td>
<td>MySQL JDBC driver</td>
</tr>
<tr>
<td>Developer application cannot find a suitable driver implementation</td>
<td>Dynamically load the mysql jdbc driver jar file, and re-invoke the original operation</td>
</tr>
<tr>
<td>create a connection to MySQL database and raises an exception</td>
<td></td>
</tr>
<tr>
<td>SQLException</td>
<td>1 P4</td>
</tr>
<tr>
<td>Application reproducing failure reported in official forum</td>
<td>PostgreSQL JDBC driver</td>
</tr>
<tr>
<td>Developer application cannot find a suitable driver implementation</td>
<td>Dynamically load the postgresql jdbc driver jar file, and re-invoke the original operation</td>
</tr>
<tr>
<td>create a connection to PostgreSQL database and raises an exception</td>
<td></td>
</tr>
<tr>
<td>SQLException</td>
<td>1 P4</td>
</tr>
</tbody>
</table>

Table III. Reuse of Healing Connectors

<table>
<thead>
<tr>
<th>Healing pattern</th>
<th># of applications</th>
<th># of generated heal. conn.</th>
<th>Concrete Connectors</th>
<th>Reuses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter translation (P1)</td>
<td>7</td>
<td>3</td>
<td>HC1&lt;sub&gt;P1&lt;/sub&gt;</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HC2&lt;sub&gt;P1&lt;/sub&gt;</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HC3&lt;sub&gt;P1&lt;/sub&gt;</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>3</td>
<td>HC1&lt;sub&gt;P2&lt;/sub&gt;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HC2&lt;sub&gt;P2&lt;/sub&gt;</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HC3&lt;sub&gt;P2&lt;/sub&gt;</td>
<td>2</td>
</tr>
<tr>
<td>Component preparation (P2)</td>
<td>4</td>
<td>2</td>
<td>HC1&lt;sub&gt;P3&lt;/sub&gt;</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HC2&lt;sub&gt;P3&lt;/sub&gt;</td>
<td>2</td>
</tr>
<tr>
<td>Alternative operation (P3)</td>
<td>4</td>
<td>2</td>
<td>HC1&lt;sub&gt;P4&lt;/sub&gt;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HC2&lt;sub&gt;P4&lt;/sub&gt;</td>
<td>2</td>
</tr>
<tr>
<td>Environmental changes (P4)</td>
<td>3</td>
<td>2</td>
<td>HC1&lt;sub&gt;P4&lt;/sub&gt;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HC2&lt;sub&gt;P4&lt;/sub&gt;</td>
<td>2</td>
</tr>
</tbody>
</table>

patterns have been conceptually reused many times to generate concrete healing connectors. In the experience reported in this article, 4 healing patterns have been successfully applied to heal 18 OTS integration faults.

Column # of generated heal. conn. indicates the number of concrete healing connectors that have been generated for each healing pattern to address the 18 case studies. The number of concrete healing connectors that have been generated from each pattern is always lower than the number of experienced cases, indicating that we have often been able to reuse the same concrete connectors.

Column Concrete connectors provides further details about the number of uses and reuses of each single healing connector. Column Connector ID identifies the concrete connectors that have been generated. Column # uses indicates the number of times that each concrete healing connector has been applied. Most concrete healing connectors
have been applied multiple times. Column Reuses indicates the number of time each
healing connector has been reused. In particular, Columns ♯ adaptations and ♯ reuses
as is indicate the number of times that the concrete healing connectors have been
reused by adaptation and with no changes, respectively.

Healing connectors have been reused with no changes when the same OTS component
is integrated in different applications that suffer from the same OTS integration fault.
This result confirms that application developers can select the healing connectors to
be downloaded and applied by considering only OTS components, and by ignoring the
identity of the system that integrates these components. We successfully adapted two
healing connectors (cases $HC3_{p1}$ and $HC2_{p4}$) to different case studies, that is different
applications that include different OTS components, by simply changing configuration
files.

The results reported in Table III suggest that the strategies defined in healing patterns
can be reused. In particular, concrete healing connectors can be often reused
when the addressed OTS component is reused. Concrete healing connectors can be
adapted to different scenarios, when used to heal semantically and structurally equivalent interactions for different pairs of applications and OTS components.

6.5. Threats to Validity

In this section, we discuss the main threats to validity of the empirical work, and how
we addressed them.

The representativeness of the case studies is a key element for the generality of the
results. A set of case studies that do not represent the general class of component-based
systems and OTS components represent a threat to external validity. In particular, the
empirical results may be valid only for the considered case studies.

To reduce this threat, we selected a large set of systems (18), and we analyzed many
OTS integration fault reports. The resulting empirical work covers different types of
systems, many types of OTS components and range from small-scale components, such
as the Xalan library, to large-scale components, such as the Geronimo server. The
empirical results reported in Pan et al. [2008] confirm that the kind of integration
problems we address are relevant and frequent. In summary, we believe that our
sampling gives good confidence about the generality of the results.

The in-house implementation of healing connectors is another potential threat to va-
lidity. The methodology presented in this article assumes that component developers
release healing connectors for their OTS component, while in the empirical validation
we implemented healing connectors in-house. We assume that our connectors cannot
work better than the healing connectors developed directly by the component develop-
ers, thus the results obtained with our connectors represent a suboptimal solution.

Another issue related to the implementation of the healing connectors is the fact that
their implementation may require specific knowledge about the context in which the
target OTS component is used. Although additional knowledge might be necessary, we
never experienced this case in our empirical evaluation.

Faults in our healing connectors or side-effects when injecting or activating the
connectors might be a threat to internal validity. To reduce these threats, we manually
inspected all developed connectors, we injected each connector individually, we verified
that each healing connector is successfully activated (without side-effects) only when
the addressed integration fault is experienced. The absence of side-effects has been
verified by executing both the system and connector test cases, carefully inspecting
logs and outputs, and double checking the complex cases by tracing step-by-step the
executions that activate the connectors.

Some fault reports that we used to generate some case studies provide incomplete
information, and limited or no test cases that can be used to reproduce the failing
interaction in-house. We developed applications that can lead to the failures reported in the descriptions. Applications developed in house can threat the construction validity because they may be different from those used by the fault reporters. To reduce this threat, we carefully examined the information contained in each fault report and all the available documentation of the OTS components such as API documentation and user guides. Moreover, we discarded the reports with ambiguous information and, for the selected ones, we reproduced the failure in our controlled environment before reproducing it in the subject systems.

The empirical work about the reusability of the healing connectors is limited and cannot be generalized. However, the empirical investigation based on problems that have been really experienced by application developers provides a good confidence about the possibility of reproducing the degree of reusability experienced in the investigated scenarios also to other cases.

7. RELATED WORK

Exception handlers and other mechanisms to handle at run-time and in-the-field unexpected component interactions have been investigated from different and sometimes overlapping viewpoints, and with different techniques: exception handling mechanisms, exception handling in CBSE, runtime failure detection techniques, error recovery strategies, self-healing solutions, fault tolerance techniques, and patching infrastructures. In this section, we present the recent work for each category, and discuss relations, synergies and complementarities between these techniques and the one presented in this article.

Exception Handling Mechanisms

Exception handling is a well-known construct supported by most modern programming languages to handle the exceptional cases that may raise at run-time [Cristian 1989]. Research on exception handling primarily focused on improving the design of the exceptional flows and increasing the robustness of exception handling code.

The research on the design of exceptional flows concentrated on the definition of good design practices, on new exception handling mechanisms and on novel models for exception handling. For instance, Cristian [1982] illustrates how to effectively use exception handling and automatic backward recovery strategies for specific classes of design faults. Miller and Tripathi [1997] discuss major issues about designing exceptions in object-oriented systems due to general concepts of object-orientation, such as abstraction, encapsulation and modularity. Garcia et al. [2001] compare exception mechanisms implemented in different object-oriented languages. Xu et al. [1998] define an object-oriented exception model for distributed and concurrent systems. Pitman [2001] surveys exception handling in the LISP language pointing out the advanced features provided by LISP. Lippert and Lopes [2000] investigate the use of aspect-oriented programming (AOP) techniques to simplify the design of exception handling code. Cabral and Marques [2008] defined a model for automatic exception handling, where programmers write handlers only when the default behavior is not satisfactory.

Healing connectors do not represent an alternative way to catch exceptions, but generalize classic exception handling augmenting existing exception handling mechanisms, such as the one in the Java programming language, with handlers that are designed to heal specific integration problems with OTS components. Differently from exception handlers that are included into applications at design-time, healing connectors are injected a-posteriori into applications using AOP techniques. Finally, connectors are used to satisfy the key needs of application and component developers, rather than the general needs of programmers.
Techniques to improve the robustness of exception handlers are mostly based on static analysis and coverage techniques, such as N-version programming, group collaboration and dependability cases [Maxion and Olszewski 2000]. In particular, Robillard and Murphy 2003 define an exception-flow model and associated static analysis to reveal potentially inadequate handlers, Sinha and Harrold [1999] define a set of criteria to test correctness of exception-handlers. Fu et al. [2005] define a compile-time analysis technique that identifies vulnerable program points and evaluates the suitability of the corresponding exception handlers.

Differently from this work, healing patterns do not aim to improve the robustness of exception handlers, rather they implement strategies that can manage common OTS integration faults that generate exceptions when revealed. Techniques to increase robustness of exception handlers are complemental to the design of healing connectors, and can be used to increase the quality of healing connectors.

Fetzer et al. [2004] defined both a technique to identify nonatomic methods, that is methods that can leave an object in an inconsistent state when exceptions are generated, and an approach to turn nonatomic methods into atomic methods. The results of Fetzer et al. [2004] can be potentially extended to check the correctness, in term of atomicity, of a system augmented with healing connectors. Investigate this possibility is part of our on-going research.

**Exception Handling in CBSE**

In component-based software engineering (CBSE), the software units that comprise a system are frequently designed by independent organizations. Thus, the exceptional flow that crosses component boundaries must be carefully designed. Software engineers defined a number of design principles that help developers in the implementation of well-defined interfaces that properly handle both the regular and exceptional flows. In particular, the notion of Idealized Fault-Tolerant Component precisely describes how components, including OTS components, should be designed to be properly integrated with other components without producing anomalous flows in case of exceptions [Anderson and Lee 1990]. Good design practices do not cover only the one to one integration of components, but a number of research results provide ways to model, structure and validate the exceptional flow at the architectural level [Brito et al. 2009; Filho et al. 2006; de Castro Guerra et al. 2004; Romanovsky 2001; Rubira et al. 2005; Garcia et al. 1999].

Developers do not always follow these design-practices and implement applications that are fragile when exceptions must be caught and handled. A number of empirical studies confirm this phenomenon [Cabral and Marques 2007; Maxion and Olszewski 1998; Li et al. 2007]. This trend harms the correctness of applications and highlights both the need of increasing the knowledge about how designing exceptional flows among developers and the need of additional design and testing techniques that can be cost-effectively applied.

The a-posteriori strategy of injecting healing connectors to fix common errors due to misuse of OTS components presented in this article represents a practical complement that increase the robustness of the applications that may suffer from improperly designed exceptional flows. Some of the well-known strategies to improve exception handling in component-based software include the development of component wrappers that add to components a layer dedicated to exception handling [de Castro Guerra et al. 2004; Romanovsky 2001]. Wrappers play a role similar to our healing connectors in the sense that they capture and react to exceptions. However, wrappers are developed at design-time and can be used to implement complex (intercomponent) exception handling strategies, while healing connectors are injected into systems a-posteriori and are used to heal the common but simple integration problems that are
frequently unrevealed by testing. Our solution does not represent an alternative to a disciplined design of the exceptional flow at the component and architectural levels, but complement this design effort with the capability to automatically heal common integration problems the first time they occur and keep the execution correct while waiting for permanent fixes.

In component-based development software connectors play a role as relevant as the role played by components. Software connectors are first-class architectural elements that mediate the interactions between components. State-of-the-art connectors address a large variety of interaction models with different capabilities. Mehta et al. [2000] define a general taxonomy for software connectors and Becker et al. [2006] define a selection of adaptation patterns that can be implemented by connectors to handle functional and nonfunctional mismatches. The healing connectors presented in this article and some specific connector types share the overall goal of handling component interactions and possible incompatibilities. For example, our healing connectors include components that can be compared to adaptor and interceptor connector patterns that are described, respectively, in Mehta et al. [2000] and Becker et al. [2006]. However, healing connectors differ from regular connectors because they are injected when the target system is already deployed in the field, they are specifically designed to address the key needs of component and application developers, implement on-the-fly recovery of component integration problems and are activated only when known integration faults are recognized.

### Runtime Failure Detection

Runtime failure detection are techniques that can detect failures dynamically and automatically when they occur. Failure detection techniques are commonly used to trigger special procedures that can cope with the observed failure, for instance they can run healers or exception handlers. The most common approaches to failure detection are exception handlers and assertions.

Exception handlers can manage unexpected events that indicate exceptional cases and failures, directly in the field 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 the study by Cabral and Marques [2007]. Poor exception handling practice may even produce additional program failures thus decreasing software reliability. Li et al. [2007] show that between 12% and 16% of the failures reported in J2EE application servers are caused by poor exception handling. The limitations of exception handling are not intrinsic in the mechanism, but depend mainly on design and programming practice. In general, exception handling is a powerful mechanism for supporting failure detection at runtime. In this article, we exploit exception handling mechanisms to conveniently capture unexpected events, and trigger the problem identification and healing steps.

Assertions embedded in the code indicate conditions that are expected to hold during program execution [Rosenblum 1992]. Runtime violations of assertions indicate failures. Assertions can well complement exceptions, and can be used to identify a rich set of verification points into programs. Several assertion frameworks raise exceptions when conditions are violated (e.g., JML [Leavens et al. 2006]), thus they can be easily integrated with the solution presented in this article to extend the class of failures that can be caught and healed by healing connectors. However, healing connectors do not currently exploit this possibility because we preferred to use an extremely cheap failure detection mechanism, as exception handling is, rather than a more expensive but wider applicable mechanism, like assertions.
Error-Recovery Strategies

Error recovery strategies are techniques that can restore correctness of the execution after failures have been observed. Solutions can implement state and operation recovery.

State recovery strategies can restore a (partial) correct state of the application after failures have been observed. Common techniques include checkpointing and rollback, roll-forward and reboot. Checkpointing and rollback mechanisms [Elnozahy et al. 2002; Pradhan and Vaidya 1994] partially rollback the system to a consistent state, and retry the failed operation either as is or after some changes to increase the probability of success [Qin et al. 2005; Lorenzoli et al. 2007]. Roll-forward mechanisms react to failures by continuing the execution from the “closest” known consistent state in the future [Pradhan and Vaidya 1994]. Reboot-based strategies clean-up the state and restart the likely faulty components to reduce the probability of future failure occurrences [Candea et al. 2004]. The support to define operations that must be executed with a kind of transactional semantic has been even investigated at the language level. For instance, Bristlecone is a language that allows to implement a program as a collection of tasks decorated with rollback and error recovery facilities [Demsky and Sundaramurthy 2011].

These solutions share with healing connectors the high level goal of avoiding systems from interrupting their execution or generating incorrect replies. However, they differ in the way they achieve this high-level goal. Error-recovery strategies require relevant coding effort to be implemented and consume a big amount of resources when executed; while healing connectors are less demanding, by requiring little coding effort and few resources to be executed. Moreover, error-recovery strategies do not heal executions, but react to failures. On the contrary, healing connectors aim to repair executions in-the-field and on-the-fly before failures are experienced by the users of the application.

Operation recovery strategies focus on delivering the requested failing service even in the presence of exceptional conditions. Well-known techniques are based on the retry model and on design diversity [Avizienis 1985; Randell 1975]. Retry-based techniques address transient failures by retrying the failed operations one or more times. Techniques based on design diversity mask the failing operation using an alternative implementation of the same failing operation, obtained by taking different design decisions. Retry-based techniques are complemental to healing connectors because they address transient faults, while the healing patterns presented in this article address functional integration problems. Operations designed according to the design diversity principle can represent another way to heal failing operations, but the practical applicability of this approach is hindered by the high cost of developing multiple independent implementations of the same operations.

Self-Healing

Self-healing approaches aim to augment software systems with capabilities to automatically heal software at runtime. Some self-healing techniques focus on nonfunctional problems, for instance, Vaidyanathan and Trivedi [2005] propose a technique to address aging problems, Candea et al. [2004] address transient problems, Krena et al. [2007] focus on concurrency problems. These techniques are complemental to the healing patterns presented in this article that address functional integration faults.

Self-healing techniques that address functional faults can be grouped in two main classes: healing based on functional redundancy and ad-hoc healing. Healing based on functional redundancy consists of identifying and replacing failing sequences of actions with equivalent, but correct, sequences of actions. For example, Carzaniga et al. [2010] propose a technique to automatically derive workarounds from
Specifications. Lack of complete specifications, as often happen with OTS components, limits the applicability of this technique.

Other researchers propose to embed ad-hoc healing mechanisms into software systems. For instance, Demsky and Rinard [2003] propose techniques to handle consistency problems with ad-hoc mechanisms can be used to automatically heal specific unit faults within OTS products, but do not replace our solution based on healing connectors to handle misuses and integration problems.

A self-healing solution based on exception handling has been recently defined for service-based applications [Friedrich et al. 2010]. This solution uses exception handlers that implement healing actions similar to the ones implemented by our healing connectors. However, the purpose and technical realization of the two solutions are radically different. Healing in service-based applications mainly address dynamic service composition. On the contrary healing connectors (and the associated methodology) aim to decouple and coordinate the activities of component and application developers: healing connectors are concrete artifacts that can be easily distributed to masses of application developers who use the same OTS products to improve the reliability of the applications. The realization of the two technologies is also different: healing service-based application is achieved by changing BPEL processes, while healing connectors turn failing executions into correct executions by directly interacting with a OTS component.

In addition to self-healing, software engineers recently produced solution for automatically repairing software. For instance, Wei et al. [2010] proposed a technique for the automatic generation of fixes. Despite the usefulness of automatically synthesizing fixes, our work addresses healing of integration problems after OTS components and systems have been released, before a definitive fix is produced. In principle, our healing patterns could be complemented with techniques for the automatic generation of fixes that exploit the information produced by healing connectors to rapidly suggest a definitive fix to application developers.

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 [Abbott 1990; Avizienis et al. 2004]. Approaches to fault-tolerant systems share the overall goal with self-healing and error-recovery approaches, since all these classes of approaches aim to handle failures in the field by minimizing 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 sometimes satisfy different requirements.

Patching Infrastructures
Releasing patches to be installed to fix faults in the field is a common engineering practice. Updating and upgrading software systems is often facilitated by infrastructures for managing upgrades, like the Windows update system [Microsoft 2012]. Releasing new versions of components that are frequently misused could be seen as an alternative to releasing healing connectors. However, continuously upgrading a component with the capability of self-healing against every possible misuse of its API is not a good engineering practice. In fact, the component would end up including a lot of code that is difficult to maintain and that will be executed very rarely, to tolerate faults located in the caller code and not in the component. On the contrary, healing connectors can be released and maintained following an appropriate methodology. Component developers
maintain the connector code and the component code separately. Application developers can add healing connectors to the application and remove them once the caller code is robust enough.

State-of-the-art patching infrastructures handle effectively software updates at user sites, as it happens when software providers release patches to all users through their infrastructure. These infrastructures can cope with end-user applications but are neither designed for fixing integration faults and misuses between applications and OTS components nor to deploy and validate healing connectors. Investigating how to extend patching infrastructures to support our solution is part of our ongoing research.

8. CONCLUSIONS

In this article, we presented a technique to develop effective and efficient mechanisms to heal problems that arise when integrating OTS components. The technique supports component developers in designing reusable healing connectors. Component developers design healing connectors on the basis of failure reports produced by application developers, without knowing the applications that use or will use the connectors. Application developers integrate healing connectors similarly to exception handlers, without requiring information about the details of the connectors.

The experience reported in the paper indicates that healing connectors are a lightweight solution that can effectively reduce the occurrences of failures caused by typical problems that arise when integrating OTS components, and suggests the existence of common patterns that can be reused to design catalogs of healing patterns and libraries of healing connectors.

Healing connectors decouple the job of component and application developers, who can work with the information currently shared on the Web. For the future, we envision frameworks that automatically distribute and deploy healing connectors on the basis of registration policies that indicate which connectors should be deployed for which applications.

REFERENCES


GAMMA, E., HELM, R., JOHNSON, R., AND VLISHIDES, J. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.


Received October 2011; revised June 2012; accepted September 2012