# Self-healing Strategies For Component Integration Faults

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

## Abstract

*Software systems increasingly integrate Off-The-Shelf (OTS) components. However, due to the lack of knowledge about the reused OTS components, this integration is fragile and can cause in the field a lot of failures that result in dramatic consequences for users and service providers, e.g. loss of data, functionalities, money and reputation. As a consequence, dynamic and automatic fixing of integration problems in systems that include OTS components can be extremely beneficial to increase their reliability and mitigate these risks.*

*In this paper, we present a technique for enhancing component-based systems with capabilities to self-heal common integration faults by using a predetermined set of healing strategies. The set of faults that can be healed has been determined from the analysis of the most frequent integration bugs experienced by users according to data in bug repositories available on Internet. An implementation based on AOP techniques shows the viability of this technique to heal faults in real case studies.*

## 1 Introduction

Modern software systems increasingly use Off-The-Shelf (OTS) components to build large scale applications, reduce system development costs and reuse third-party expertise. A wide range of ready-to-use commercial and open-source components are available to developers, including runtime platforms and middleware (*e.g.*, Apache http, JBoss AS, messaging systems), frameworks (*e.g.*, Spring, Struts), DBMS (*e.g.*, MySQL, PostgreSQL) and application libraries (*e.g.*, logging frameworks, XML parsers). Unfortunately, since OTS components are implemented independently and reused in partially known contexts, their integration within actual systems is challenging and can introduce subtle *integration faults*.

While syntactical integration problems are well-addressed by compilers and type checking systems [15], semantic faults are more subtle to be revealed because they are related to the functional behavior of components. A recent study of bug characteristics on the Apache web server and Mozilla softwares indicates that semantic bugs are increasingly dominant [19]: 81-86% of failures are caused by semantic faults consisting of wrong implementations, missing features, missing cases, typos and improper exception handling. In the total number of bugs, 96-98% of them are likely to cause incorrect functionality and 42-44% of them are likely to cause system crashes, thus threatening systems functionality and availability.

Classic verification and validation techniques focus on minimizing the introduction of semantic faults at design-time, and removing the defects introduced at development-time. Well known examples are defensive programming, formal verifications, static analysis and testing [8, 22]. Even if useful, these techniques are insufficient to remove all the faults, especially because the many different configurations and situations that may exist in the field cannot be foreseen.

Self-healing techniques represent a complement to traditional testing and verification techniques by enhancing systems with capabilities to *automatically* detect and *repair* software errors in the field [17].

There exist several techniques to detect semantic errors in the field, e.g., exception handling frameworks integrated in modern languages, such as Java or .NET, can be used to detect general unexpected events related to both systems' behavior and interactions between systems and their environment [9]; assertions can be used to verify whether boolean expressions that must hold at given program points hold at run-time [21]; and ACID transactions can be used to detect and manage improper use of shared resources [16]. While extremely useful for error detection, these techniques often do not provide associated healing mechanisms, thus cannot be used to repair the errors once detected.

In this paper, we focus on healing faults that result in raised exceptions, which represent a relevant portion of possible failures. For example, a recent study on 3 widely-used J2EE application servers (Geronimo, JBoss AS and JOnAS) reports that 70% of the failures are manifested by exceptions [18].

Exception frameworks provide mechanisms and constructs to implement suitable exception handlers aimed at managing exceptions and recovering from errors. However, because it is difficult to write correct exception handlers, exception mechanisms are currently *not* used at their best to recover from errors. In practice, they usually do nothing or apply very general exception-handling strategies. A recent field study of 32 Java and .NET applications shows that the amount of code used in error handling is much less than expected (only 3-7% for Java, 3% for .NET) [4]. Moreover, when exception handlers exist, they usually execute general actions (*e.g.*, exception propagation, graceful degradation and termination) which are likely to cause systems to misbehave, disrupt their services or crash, e.g., 12-16% of the 70% failures reported in [18] appear to be caused by poor exception handling.

The self-healing technique presented in this paper captures exceptions generated by semantic integration faults between core business applications and the various OTS components they may use (runtime platforms, libraries, etc.) and applies predetermined *healing strategies* to recover from failing operations. When strategies succeed in healing executions, execution can safely proceed. For example, consider an application that uses the Apache HTTP-Client library[1]. A call to the constructor of the class `GetMethod(String uri)` with the `uri` parameter `"http://website/img/logo[1].gif"` will return the exception `java.lang.IllegalArgumentException`[2], which may cause a failure at the caller side. Our technique is able to heal the execution of that constructor operation by automatically intercepting the raised exception, fixing the value of the uri parameter and invoking again the `GetMethod` constructor with the new parameter value.

The classes of faults that can be healed by the solution presented in this paper have been determined by analyzing bug reports in open repositories[3] available on the Web and selecting the most common semantic integration faults that cannot be healed by patching OTS. Examples of such integration faults are: misusing OTS API, using faulty or deprecated methods that have not been fixed across versions for technical or cost-effectiveness reasons, or failing in correctly setting up the execution environment where OTS are reused. For each selected class of faults, we determined (1) the exception that is raised when a fault in the considered class causes a failure, (2) the conditions that must be checked in the field to verify that a raised exception is caused by a fault that can be healed, and (3) the healing strategy that must be executed to repair a failing execution caused by a known fault.

Each known fault is associated with a *healing connector* that can be plugged into target systems to inject healing capabilities. A healing connector is activated when exceptions generated by known faults are raised. Every time a healing connector is activated, it checks whether it identifies a known fault by verifying that some conditions hold, and applies the associated healing strategies if it is the case. Otherwise, the raised exception is simply propagated back to the caller component. Healing connectors are used by system developers as black box elements, i.e., developers plug them into their systems and if any of the faults that can be healed exist, they are automatically fixed at run-time. Since healing connectors are activated only upon exceptions, they introduce no overhead in successful executions.

The rest of the paper is organized as follows. Section 2 presents the process that we followed for designing the healing strategies presented in this paper. Section 3 describes their implementation using AOP techniques. Section 4 presents real problems that can be automatically healed with our approach. Section 5 discusses related work. Finally, Section 6 concludes and presents our future research agenda.

## 2 Designing Healing Strategies

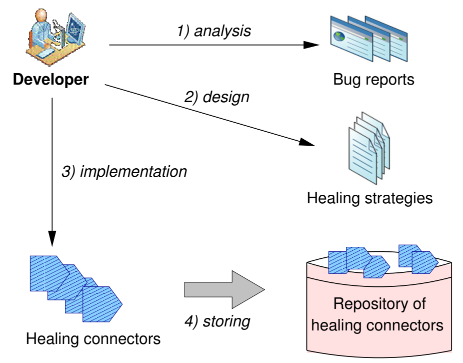

Figure 1: The four steps of the process to design the healing strategies.

We identified healing strategies and implemented healing connectors for common integration problems associated with widely used OTS components and middleware according to the process shown in Figure 1. We concentrated on OTS components and middleware because they are reused across many systems, and several similar integration faults tend to be present in different applications that integrate the

[1]http://hc.apache.org/httpcomponents-client/index.html

[2]This example is related to a real bug reported at http://issues.apache.org/jira/browse/HTTPCLIENT-678.

[3]We use "bug repository" to refer to issue tracking systems that contain both faults reports and features requests like Bugzilla and JIRA systems.

same components and middleware.

*At the analysis step*, we browsed bug repositories related to OTS components and analyzed the characteristics of experienced bugs contained in the bug reports, specifically focusing on semantic integration faults that are manifested by raised exceptions. So far, we analyzed open bug repositories of the Sun standard JDK and its libraries[4], the Spring framework[5], the JBoss platform[6] and various systems hosted in the Apache software foundation[7]. We also studied the normal behavior of target OTS components and identified the root cause of failures. In the best cases, this analysis allows us to: i) identify the experienced semantic bugs, ii) discover the root-causes of the failures and iii) extract (when available) the fixes to be applied in the field.

*At the design step*, healing strategies are designed based on the information extracted at the previous step. The healing strategies address integration faults with highly reused OTS components and can be used to fix errors in all systems that misuse the analyzed OTS components in a known way. The rest of this paper gives examples of healing strategies for different analyzed OTS components. The set of healing strategies can be extended by studying other OTS components following this same process. Our analysis conducted so far leads us to identify four classes of semantic faults related to individual operations and to propose the corresponding healing strategies.

Once identified, we implement these strategies as *healing connectors*. These healing connectors are stored in a repository for future use by developers. Strategies are *manually* designed to implement specific healing logics that are best suited to heal the corresponding semantic faults, and also *incrementally* whenever faults are experienced and bug fixes identified.

Developers can also follow our same process to extend the set of faults that can be healed. Note that in most cases, patching the target OTS components does not work because integration faults depend from the specific way an OTS is used and can be patched only when clients of OTS components are available.

Analysis of faults resulted in 29 different healing strategies. Our final ambition is to build a large number of healing strategies that can be seamlessly plugged into existing component-based systems to heal executions for widely (re-)used OTS components. Table 1 summarizes the OTS components that we analyzed and the number of healing strategies defined for each OTS component and middleware. A more detailed report is available on the Web[8].

The rest of this section presents the proposed healing

[4]http://bugs.sun.com/
[5]http://jira.springframework.org/
[6]http://jira.jboss.org/
[7]https://issues.apache.org/
[8]http://www.lta.disco.unimib.it/lta/personalPages/herveChang/recovery/

| OTS components | Category | Nb. of strategies |
|---|---|---|
| Sun Java SE (JDK)[a] | Java runtime platform | 18 |
| Optional Sun Java libraries[b] | Java libraries | 1 |
| Spring framework[c] | Java/JEE application framework | 2 |
| JBoss AS[d] | JEE application server | 2 |
| Apache ActiveMQ[e] | Message broker | 1 |
| Apache HTTP Components[f] | HTTP protocol libraries | 2 |
| Apache Cactus[g] | Server-side unit-test framework | 3 |

[a]http://java.sun.com/javase/
[b]http://developers.sun.com/
[c]http://www.springframework.org/
[d]http://www.jboss.org/jbossas/
[e]http://activemq.apache.org/
[f]http://hc.apache.org/
[g]http://jakarta.apache.org/cactus/

Table 1: Studied OTS components and number of designed strategies.

strategies. Since different faults impose different kinds of reactions, we classified faults that can be healed and we correspondingly defined categories of healing strategies.

## 2.1 Classes of Faults

In our early study, we identified four major classes of problems that can cause the execution of a method to result with an exception: invalid method parameters fault, wrong method usage fault, faulty method implementation and environment fault.

We have an *invalid method parameter* fault when a caller component invokes a method of a server component with parameter values that do not satisfy callee's expectations, such as it may happen for numeric values outside the expected range or malformed strings. This problem often results with an exception of type `IllegalArgumentException`. The caller component is responsible for this kind of problems because it is the generator of the incorrect inputs.

We have a *wrong method usage* fault when an operation is invoked even if the server component cannot serve it. Typical examples are missing initialization steps and incidental use of dead connections. A wrong method usage can result in the generation of various exceptions. The most frequent ones are `IllegalStateException` and `MessagingException`. The caller component is responsible for this kind of problems because it is the generator of the unexpected call.

We have a *faulty method implementation* when an operation includes an implementation fault. Typical examples of faulty methods that may be addressed with the technique presented in this paper are operations that do not correctly handle special characters and faulty algorithms due to missing cases. The execution of a faulty method can result in various exceptions, depending case by case. Component developers are responsible for this kind of problems because they released faulty components.

We have *environment* faults which are caused by the interaction between the applications and their execution environments. Typical examples are missing deployment descriptors and missing class files. Environment problems can result in the generation of several exceptions. Two frequent cases are `IOException` and `ClassNotFoundException`. System administrators who deploy systems in running environments are responsible for avoiding such problems.

If we consider the case of the HTTP-Client library example presented in the introduction, the described problem belongs to the first category since the characters '`[`' and '`]`' in the string parameter are invalid for URIs, according to the RFC-2396[9].

## 2.2 Healing Strategies

We designed four categories of healing strategies to address the four classes of faults presented in the previous section. Healing strategies aim at healing the execution of the failed operation and restore the correct functionality.

Since accessibility to component internals are usually limited or impossible for OTS components, our technique only relies on information provided by interfaces (methods signature, declared exceptions, java documentation...). Healing strategies are actuated at component interfaces without requiring any knowledge of component internals as well.

In this paper, our approach to heal executions is based on *retrying* the failed operation. However, as the retry model is effective for transient failures but insufficient to heal the deterministic semantic faults described above, the proposed strategies also perform various *operations* before retrying the failed original invocation. These changes do not affect component implementations but only concern actions that target component interfaces or the execution environment.

In particular, we elaborated the following four categories of healing strategies, which have a one-to-one mapping with the fault categories: change parameter and retry the failed operation, call operations before retrying the failed operation, replace the failed operation with other operations, and modify the environment and retry the failed operation.

The *change parameters + retry* strategy performs changes on parameters of the called operation followed by a retry invocation. It can be used to heal from invalid method parameters. Examples of concrete healing actions that we implemented for some of the common misuses of OTS operations are remove extra white-spaces, add a missing trailing slash, and change a parameter value with a default one.

The *call operations + retry* strategy performs calls to other operations before retrying the original one. It can be used to heal incorrect usage of OTS interfaces. Examples of concrete healing actions that we implemented for some of the common misuses of OTS components' interfaces that we identified in the analysis phase are adding a missing initialization step, and (re-)establishing connections before using them.

The *replace calls* strategy replaces the original invocations with a different invocation sequence. It can be used to heal from faulty methods. Examples of concrete healing actions that we implemented for some of the common faulty methods that we identified in the analysis phase is to replace the original invocation with another invocation to a non-faulty but equivalent interface method of a same OTS component.

The *change environment + retry* strategy performs changes in the environment followed by a retry. It can be used to heal from environment faults. Healing actions may include changes on almost everything that is external to the application but can affect its execution (file system, networks protocols or memory, or standard or third-party libraries, etc.). Examples of concrete healing actions that we implemented for some of the common environment faults that we identified in the analysis phase are programmatically creating the necessary directory for persisting data and dynamically loading missing class files.

In our previous example related to the HTTP-Client library, the healing strategy designed for this OTS component would detect the invalid characters '`[`' and '`]`' in the parameter and would replace them with their respective escape characters '`%5B`' and '`%5D`'. More in details, once the `IllegalArgumentException` exception has been intercepted, the associated healing strategy proceeds by retrieving the `uri` parameter of the `GetMethod` constructor, checking for the presence of characters not allowed for an URI, re-encoding the string argument to escape all the invalid characters (the new parameter is `"http://website/img/logo%5B1%5D.gif"`), and finally calling the constructor with this new uri.

# 3 Injection of Healing Strategies

Healing strategies are designed to heal specific faults and misuses of OTS components, i.e., a healing strategy heals a specific misuse/fault for a given OTS component, and are implemented as healing connectors.

[9]www.ietf.org/rfc/rfc2396.txt

Healing connectors can be injected into target systems at any time. The only requirement is the availability of the binaries for the components (OTS or not) responsible for a problem that can be healed.

Figure 2 shows the injection of the healing strategies in existing component-based systems. The core business application (top of the figure) uses both a runtime platform and a common library as OTS components (bottom of the figure). The self-healing layer lies between these two levels and is composed by different healing connectors activated by exceptions raised from methods that are known to include some of the faults that can be healed.

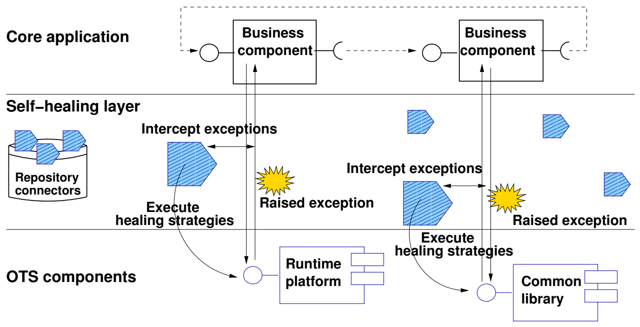

Figure 2: Integration in component-based systems.

We implemented our technology by using AOP techniques, which provide an easy way to inject healing capabilities in existing systems even if only binaries are available. In particular, we used the AspectWerkz framework[10], which is a fast and flexible Java AOP framework.

Each healing connector includes two main parts: an interceptor and code block and a healing code block.

Interceptors are implemented as advices, i.e., code blocks that are executed when target operations are invoked. Advices, and thus interceptors, can arbitrarily modify the execution context, e.g., it is possible to change parameters and to replace a method call with any operation sequence.

Healing code blocks consist of `handleFault` methods that execute the sequence of actions necessary to fix the current execution. Since several strategies can be tried for a given fault, the various fault handlers are chained using the *chain of responsibility* design pattern. Each fault handler implements a common interface, tries to heal from the original fault by performing its own healing strategy implemented in the `handleFault` method. A contextual object that contains the cause of the failure (e.g. exception type, error codes and messages) and the interception context (components, method name and parameters) is passed between the fault handlers.

*At runtime*, when an exception is returned by a method associated with healing connectors, the associated healing strategies are activated. The general behavior consists of identifying the possible causes of the observed failure and triggering the execution of the corresponding healing actions, if any. If healing is not possible, the original exception is propagated.

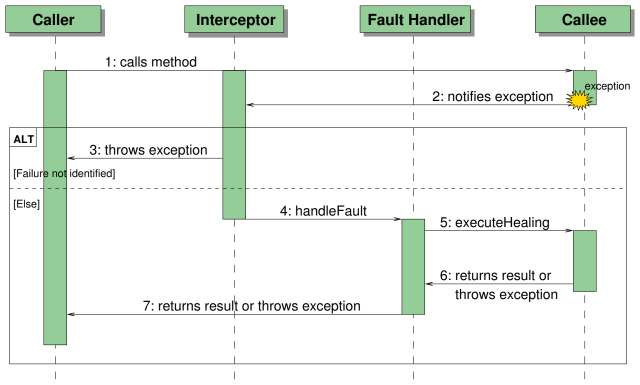

Figure 3: Messages exchanged between the caller and callee components and the healing objects (interceptor, fault handler).

Figure 3 shows the typical set of messages exchanged when a method associated with healing connectors generates an exception. The invocation to a target method (message 1) returns an exception that is intercepted by an interceptor (message 2). The interceptor then checks whether the current problem can be handed by a fault handler. If the problem cannot be handled, the original exception is propagated to the caller (message 3). If the problem can be handled, the interceptor retrieves the *chain of handlers* associated with the current problem and delegates the healing to the first handler in the chain (message 4). Each handler executes its healing action (message 5). If the healing is successful, the result of the operation is returned to the caller component. If the healing is not successful, the next handler is applied. If all the handlers fail, the healing process terminates with a failure and the original exception is propagated (message 6 and 7).

## 4 Preliminary Experiments

In this section, we present an early evaluation that shows effectiveness of the technique in healing faults in three case studies related to the Spring framework, the Sun standard J2SE distribution and the Sun database connectivity API, which together with the running example covers the four categories of strategies. In all the three cases, faults are healed and executions can safely proceed.

[10]http://aspectwerkz.codehaus.org/

**Call operations and retry:** We applied this strategy to heal from the fault described in Bug ID SPR-315[11]. This fault is related to the Spring framework and results in the generation of a `IllegalStateException` when calling the `invoke` method on a `MethodInvoker` object without having prepared it. As indicated in the java documentation, the correct usage of this method requires a previous call to the `prepare` method.

The healing connector developed to heal this fault implements a *call operations + retry* strategy which executes the following actions: i) it intercepts the `IllegalStateException` raised by the `invoke` method, ii) it checks whether the invoker has been prepared or not, iii) if yes, the exception is simply re-thrown, iv) if not, it calls the `prepare` method and retry a call to the `invoke` method. If the invocation succeeds the result of the `invoke` method is returned to the caller, otherwise the original exception is re-thrown.

**Replace calls:** We applied this strategy to heal the fault described in Bug ID 4976356[12]. This fault is related to the Sun J2SE *version 6* and results in a raised `ClassNotFoundException` when calling the `loadClass` method with an array syntax parameter. It has been frequently experienced when servers initialize or when applicationsserialize/deserialize arrays of objects (see for instance the fault description for Hibernate[13]). A `ClassNotFoundException` is usually raised when a classloader cannot find the class definition or fetch the byte codes from the source. However, in this case, the error is related to an undocumented feature change, as loading an array class with the array syntax has been disabled since JDK 1.6, but works correctly in previous versions. The faulty behavior can be reproduced by using the Sun JDK 1.6 and calling the `loadClass` method with any array syntax parameter, such as `"[Ljava.lang.String"`.

To suitably load an array class, a possible workaround reported by Sun is to use the extended form of `Class.forName()` instead of the `loadClass`.

The healing connector developed to solve this problem implements a *replace by an operation* healing strategy which performs the following actions: i) it intercepts the `ClassNotFoundException` raised by the `loadClass` method, ii) it checks whether the parameter of `loadClass` uses the array syntax, iii) if it is the case, it replaces the `loadClass` call by a call to the extended `Class.forName()` operation with the original array syntax parameter and the current classloader object. If the invocation succeeds, the loaded array is returned, otherwise the original exception is re-thrown.

[11] http://jira.springframework.org/browse/SPR-315

[12] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4976356

[13] http://opensource.atlassian.com/projects/hibernate/browse/HHH-2990

As a final note, Sun does not expect to modify the implementation of the class loading or provide a fix this faulty behavior. The use of our technique appears to be particularly relevant here, as a means to seamlessly integrate known patches that are not provided by OTS providers.

**Change environment and retry:** We applied this strategy to heal from a fault related to the Java Database Connectivity API included in Sun J2SE distribution. The failure caused by this fault consists of a `SQLException` raised by method `getConnection(url)` of class `DriverManager`. This error may have several source causes (network failures, database crash). Among the many possibilities, it can be the case that the `DriverManager` have failed in locating a suitable driver, as indicated in the `url` parameter. In this case, a `SQLException` object containing the `SQLState` value of 08001 with a "No suitable driver" error message is raised. The faulty behavior can be reproduced by establishing a connection to a database, for example the MySQL database with the url `jdbc:mysql://localhost:3306/test` without having deployed its driver implementation `mysql-connectorJ.jar` in the classpath.

The healing connector developed to address this problem implements a *change environment and retry* healing strategy which performs the following actions: i) it intercepts the `SQLException` raised by the `getConnection` method, ii) it checks whether the `SQLState` code is 08001, iii) if it is the case, it dynamically searches for the required missing `mysql-connectorJ.jar` jar file from a predefined location using the database *subprotocol* (mysql) and a table mapping. If the jar is found, the healing connector dynamically loads the jar file, loads the driver class and registers it to the `DriverManager`, and iv) retries the call to the `getConnection` method. If the invocation succeeds the resulting `Connection` object is returned to the caller, otherwise the original exception is re-thrown. Note that in this case, the technique healed from an incomplete deployment setup by dynamically changing the environment.

## 5 Related Work

Related work for healing or repairing functional faults can be classified in the following categories: *exception-handling mechanisms*, *fault-tolerant techniques*, *self-healing strategies* and few approaches that focus on *execution recovery* and *component adaptation*.

Exception handling provide general mechanisms to implement recovery techniques. However, since it is hard to write correct recovery procedures, the implemented *exception handlers* are usually limited to the following strategies: exception propagation, resources clean-up, retry or termination. Our approach rather aims at providing complete recovery capabilities through predetermined healing strategies, thus offering a higher-level of handling. It comes at the cost of implementing the strategies. However, since we are focusing on highly reusable OTS components, rather than specific core applications, we target to benefit from their frequent reuse in many systems. Other works have also been done about maintainable, testing and improvement of exception-handling [27, 26]. They are complementary to our solution because they focus on aspects as the design and maintenance of exception handlers, while we focus on the implementation of healing strategies.

A large set of techniques under the wide field of *fault-tolerance* are based on the ideas of exploiting redundant information and recovering to error-free states. For example, N-version programming [2] and recovery blocks [25] rely on multiple variants of a software module, with the hypothesis that they fail independently. These techniques can be effective to handle bugs that affect a minority of replicas but are not effective for failures common to several modules. Moreover, they rely on the ability to achieve truly different versions of a same module, which seldom happens in practice. Another work proposed to exploit redundancy in software systems in a different way. Failing operations are automatically replaced with equivalent sequences of operations implemented in the same application [6].

Our solution considers the use of redundancy as one of the many ways to implement healing strategies. Compared to the aforementioned approaches, our technique can only address known faults, but can integrate any developer-defined solution to heal faults. On the contrary, while redundancy based techniques are not limited to known faults, they rely on the existence of redundant modules and a way to identify the one to apply, which is hard in practice.

Checkpoints and recovery techniques are used to set applications to an error-free state to enable servicing subsequent requests. Recovery mechanisms are usually supported by exception handling and takes the forms of rolling back to a previous checkpoint [12, 20] or rolling forward a new legal state [23]. These techniques mostly focus on recovering from environment-dependent and transient failures by re-setting the application to an error-free state and "forgetting" failures, whereas our solution focuses on semantic faults and recovers a currently failed operation.

Other *self-healing strategies* address application-independent failures and mostly focus on non-functional properties. Examples include reconfigurable architectures [14, 13], service-discovery mechanisms [10], resources provisioning [28], and different kinds of component reboots [5]. These techniques attempt to address healing of faults without requiring a-priori knowledge of the target applications. However, the healing actions that are executed have little diversity and are extremely general. On the contrary, our technique defines specific healing strategies to address semantic faults. While it requires some knowledge about the OTS components that are integrated into a target system, our solution is in line with a previous work that found that the majority of application faults are independent of the operating environment and requires the use application-specific knowledge to be fixed [7].

Few techniques focus on *execution recovery* from different point of views. Examples consist of the implementation of outcome-tolerant conditional branches [29], for which program behavior may remain the same even if the execution took the wrong path; specific actions to repair data structures inconsistencies detected by violated constrains [11], and rolling-back/reexecuting the process in a modified environment [24]. The technique presented in this paper also aims to obtain correct executions, but recovery actions are actuated at the level of components interfaces or environment, to preserve component encapsulation. On the contrary, the aforementioned approaches repair actions at the code level (data structures and conditional branches).

Finally, several techniques use adaptation patterns to handle component incompatibilities. For example, a selection of adaptation patterns based on a taxonomy is proposed to handle functional and non-functional mismatches [3]. Compared to this approach, our solution is restricted to semantic aspects but at the same time identifies more detailed classes of faults related to method calls and uses the interceptor pattern to intercept raised exceptions and perform error recovery. Another work proposes a protective wrapper for an OTS PID controller that detects erroneous information going to and from the OTS and initiates simple recovery actions [1]. Compared to this technique, our solution propose a wider range of healing strategies that are only activated on raised exceptions.

## 6 Conclusion

Designing suitable self-healing strategies that address component integration faults is critical for modern systems built from integrating OTS components. Various techniques for detecting faults in the field have been proposed. However, they often do not define associated techniques to heal the detected faults [21, 9, 16].

In this paper, we have i) presented a technique for designing and integrating healing strategies for semantic integration faults detected by raised exceptions between core applications and OTS components, ii) described the main elements of the implementation based on aspect-oriented techniques, and iii) reported preliminary experiments that give evidence of the effectiveness of the technique to heal problems in Open Source OTS components.

More work is needed to completely validate our proposal and ongoing research is focusing on two main directions. First, we will increase the set of available healing strategies, and possibly enrich or refine the classes of faults that can be addressed by our technique. This work will be incrementally completed by continuing the analysis of OTS components API and the many bug reports in bug repositories. Second, we will perform further experiments with case studies of different types and size to show general applicability and effectiveness of the solution.

## Acknowledgement

This work is partially supported by the European Community under the IST program of the $6^{th}$ FP for RTD - project SHADOWS contract IST-035157.

## References

[1] T. Anderson, M. Feng, S. Riddle, and A. B. Romanovsky. Protective Wrapper Development: A Case Study. In *proceedings of the 2nd International Conference on COTS-Based Software Systems*, volume 2580 of *Lecture Notes in Computer Science*, 2003.

[2] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 11:1491–1501, 1985.

[3] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an Engineering Approach to Component Adaptation. In *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science*, 2004.

[4] B. Cabral and P. Marques. Exception Handling: A Field Study in Java and .NET. In *proceedings of the 21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, 2007.

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — A Technique for Cheap Recovery. In *proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[6] A. Carzaniga, A. Gorla, and M. Pezzè. Self-Healing by Means of Automatic Workarounds. In *proceedings of the 3rd Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2008.

[7] S. Chandra and P. M. Chen. Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software. In *Proceedings of the 1st International Conference on Dependable Systems and Networks*. IEEE Computer Society, 2000.

[8] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, 1996.

[9] F. Cristian. Exception Handling and Software Fault Tolerance. *IEEE Transactions on Computers*, 31(6):531–540, 1982.

[10] C. Dabrowski and K. Mills. Understanding Self-healing in Service-discovery Systems. In *proceedings of the 1st International Workshop on Self-Healing Systems*. ACM, 2002.

[11] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors in Data Structures. *SIGPLAN Notices*, 38, 2003.

[12] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Computing Surveys*, 34:375–408, 2002.

[13] D. Garlan and B. Schmerl. Model-based Adaptation for Self-healing Systems. In *proceedings of the 1st International Workshop on Self-Healing Systems*. ACM, 2002.

[14] I. Georgiadis, J. Magee, and J. Kramer. Self-organising Software Architectures for Distributed Systems. In *proceedings of the 1st International Workshop on Self-Healing Systems*. ACM, 2002.

[15] S. L. Graham and S. P. Rhodes. Practical Syntactic Error Recovery. *Communications of the ACM*, 18:639–650, 1975.

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[17] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, 2003.

[18] J. Li, G. Huang, J. Zou, and H. Mei. Failure Analysis of Open Source J2EE Application Servers. In *proceedings of the 7th International Conference on Quality Software*. IEEE Computer Society, 2007.

[19] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have Things Changed Now?: an Empirical Study of Bug Characteristics in Modern Open Source Software. In *proceedings of 1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[20] D. Lorenzoli, L. Mariani, and M. Pezzè. Towards Self-Protecting Enterprise Applications. In *proceedings of 18th IEEE International Symposium on Software Reliability Engineering*, 2007.

[21] B. Meyer. Applying "Design By Contract". *IEEE Computer*, 1992.

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

[23] D. K. Pradhan and N. H. Vaidya. Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture. *IEEE Transactions on Computers*, 43:1163–1174, 1994.

[24] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating Bugs as Allergies—A Safe Method to Survive Software Failures. In *proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[25] B. Randell. System Structure for Software Fault Tolerance. In *proceedings of the International Conference on Reliable software*. ACM, 1975.

[26] M. P. Robillard and G. C. Murphy. Static Analysis to Support the Evolution of Exception Structure in Object-oriented Systems. *ACM Transactions on Software Engineering and Methodology*, 12:191–221, 2003.

[27] S. Sinha and M. J. Harrold. Criteria for Testing Exception-Handling Constructs in Java Programs. In *proceedings of the 15th International Conference on Software Maintenance*, 1999.

[28] B. Urgaonkar and A. Chandra. Dynamic Provisioning of Multi-tier Internet Applications. In *Proceedings of the 2nd International Conference on Autonomic Computing*. IEEE Computer Society, 2005.

[29] N. Wang, M. Fertig, and S. Patel. Y-Branches: When You Come to a Fork in the Road, Take It. In *proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2003.