On the Interplay of Exception Handling and Design by Contract: An Aspect-Oriented Recovery Approach

Henrique Rebêlo[1] Roberta Coelho[2] Ricardo Lima[1] Gary T. Leavens[3]
Marieke Huisman[4] Alexandre Mota[1] Fernando Castor[1]

[1] Federal University of Pernambuco, PE, Brazil
[2] Federal University of Rio Grande do Norte, RN, Brazil
[3] University of Central Florida, Fl, USA
[4] University of Twente, Netherlands

ABSTRACT

Design by Contract (DbC) is a technique for developing and improving functional software correctness through definition of "contracts" between client classes and their suppliers. Such contracts are enforced during runtime and if any of them is violated a runtime error should occur. Runtime assertions checkers (RACs) are a well-known technique that enforces such contracts. Although they are largely used to implement the DbC technique in contemporary languages, like Java, studies have shown that characteristics of contemporary exception handling mechanisms can discard contract violations detected by RACs. As a result, a contract violation may not be reflected in a runtime error, breaking the supporting hypothesis of DbC. This paper presents an error recovery technique for RACs that tackles such limitations. This technique relies on aspect-oriented programming in order to extend the functionalities of existing RACs stopping contract violations from being discarded. We applied the recovery technique on top of five Java-based contemporary RACs (i.e., JML/jml, JML/ajml, JContractor, CEAP, and Jose). Preliminary results have shown that the proposed technique could actually prevent the contract violations from being discarded regardless of the characteristics of the exception handling code of the target application.

1. INTRODUCTION

Design by Contract (DbC) is a technique for developing and improving functional software correctness [14]. The key mechanism in DbC is the use of the so-called "contracts". A contract formally specifies an agreement between a client and its suppliers. Client classes must satisfy the supplier class conditions before calling one of its methods. When these conditions are satisfied, the supplier class guarantees certain properties, which constitute the supplier class's obligations. However, when a client or supplier breaks a condition (contract violation), a runtime error occurs. The use of such pre- and postconditions to specify software contracts dates back to Hoare's 1969 paper on formal verification [7]. The novelty with DbC is to make these contracts executable.

In this context, runtime assertion checkers (RACs) are a well-known technique used for enforcing these contracts as the program executes [1, 16, 17, 4]. We can cite, for instance, the Java-based contemporary RACs: (i) JML/jmlc [1], (ii) JML/ajmlc [16], (iii) JContractor, (iv) Contract Enforcement Aspect Pattern (CEAP) [17], and (v) Jose [4]. Although they are largely used to implement the DbC concepts in Java, a previous study [8] has shown that the interaction between some exception handling (EH) contexts (i.e., *finally blocks*) and the contemporary RAC techniques can cause the latter to fail to properly communicate exceptions that report contract violations. Current RACs represent current contract violations as exceptions. As a consequence, specific exception handling contexts on the target application have the ability to accidently capture such exceptions, silencing the contract violation and allowing the system to continue its execution on an illegal state.

However, a deep analysis of this limitation led us to conclude that the conflicts that arise from the interplay between RACs and the EH code are not caused by specific EH contexts. They are related to the way exception handling mechanisms are implemented in modern languages like Java. The conventional exception handling facilities promotes the problem of accidental exception capture due to the dynamic nature of exception-handling semantics (see Section 2). Hence, none of the existing contemporary RACs [1, 16, 17, 4] can prevent contract violation errors from being discarded during program's execution.

In this paper we present an approach to prevent contract violations from being discarded regardless of the exception handling contexts present on the target applications. Our approach relies on the use of aspect-orientation [10] to monitor the exceptions that represent contract violations and force such exceptions to be signaled — preventing EH contexts from negatively affecting RAC generated exceptions. We claim that our approach focuses on "recovery" because we address situations where the RAC infrastructure cannot behave as expected. In these situations, we could say that the RAC infrastructure "fails". In the event of a contract violation, the RAC infrastructure should signal an exception indicating this event. This exception prevents parts of the application from running in a situation where they would not be consistent. By ensuring exception propagation in situations where it would be accidentally caught, we steer the RAC infrastructure back to its expected behavior, thus recovering it from its erroneous state.

This approach is implemented in the form of an aspect library, which can be associated with any Java-based RAC. The main contributions of this paper are as follows:

- we present the *fault model* that describes the exception handling scenarios that can prevent current RACs from effectively signaling contract violations (i.e., exception overriding, exception swallowing and unintended exception subsumption);
- we show how aspect-oriented concepts can be used to overcome the obstacles posed by such interplay between RACs and the exception handling code on the target application;
- we developed an aspect library, called EHSafetyRAC, that can be added to any Java-based RAC in order to ensure that contract violations, regardless of the exception handling contexts present on the target applications, are signaled;
- we applied the proposed solution to different Java-based RACs and assessed the effectiveness of our approach using a fault injection technique; our preliminary results show that EHSafetyRAC provides effective support for RACs.

The remainder of this paper is organized as follows. Section 2 presents a short discussion on the limitations of the contemporary runtime assertion checkers for Java-like programs under some contexts of exception handling. Section 3 presents the supporting ideas of the proposed approach and how aspect-oriented concepts can be used to concretize these supporting ideas. Section 4 summarizes our experience of applying the proposed solution to five different JML RACs. Finally, Section 5 presents our conclusions and directions for future work.

2. CHARACTERIZING THE INTERPLAY OF DBC AND JAVA EH MECHANISM

In order to support the reasoning about the interplay between specific exception handling contexts and contemporary RACs, we present the main characteristics of the Java exception handling mechanism (Section 2.1) and some scenarios that may negatively affect RACs (Section 2.2).

2.1 Java Exception Handling

In Java, **try** blocks define exception handling contexts, **catch** blocks define the exception handlers, and **finally** blocks define clean-up actions — executed whether or not exceptions are raised [5]. Exceptions are represented in a hierarchical structure, as illustrated in Figure 1. According to this structure every exception is an instance of the Throwable class. The user defined exceptions can be represented as a checked (extends Exception) or an unchecked exception (extends RuntimeException). By convention an Error represents an unrecoverable condition; often a JVM-related problem.

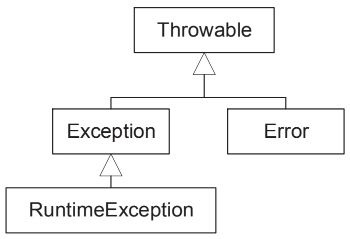

Figure 1: Java Exception Type Hierarchy.

```
> ajmlrac Example1
Exception in thread "main" org.jmlspecs.ajmlrac.
    runtime.JMLInternalPreconditionError: by method
     Example1.decrypt regarding specifications at
File "Example1.java", [spec_case]: line 2,
    character 22 (Example1.java:2), and
by method Example1.decrypt regarding code at
File "Example1.java", line 4 (Example1.java:4),
 when
 'key' is 10
 'access\_code' is 36
... more
```

An exception is raised by a method when an abnormal computation state is detected. In Java, whenever an exception is raised inside a method that cannot handle it, such an exception is signaled to the caller, and the search for the handler continues along the dynamic invocation chain until a handler is found. This way of binding exceptions with handlers based on the call chain is said to increase software reusability. Hence, the invoker of an operation can handle the exception in a larger context. Each handler is attached to a protected region (e.g. method, blocks of code) and associated with the exception type, which specifies the handling capabilities — i.e., which exceptions can handled. When an exception is signaled, it can be subsumed into the type associated with a handler, only if the exception type associated with the handler (i.e. the handler type) is a supertype of the type of caught exception.

2.2 Fault Model

This section presents a fault model that defines a set of types of faults that can happen on the interplay between Java exception handling mechanism and RACs (this fault model is also known as exception handling antipatterns [13]). Such faults can be manifested as a failure on the execution of a target application and consequently discard contract violation exceptions. They are: (i) unintended exception subsumption; (ii) exception swallowing, and (iii) exception overriding. This fault model defines the types of faults considered by our recovery approach. These faults can accidentally discard contract violation exceptions [15, 3]. Each fault type is described next based on a simple example.

Consider a method called decrypt, whose first argument is a key and the second one is a special access code, which determines whether one is allowed to use decryption. For simplicity, let us assume that the access_code argument should be an integer below 10. In JML [1] [12], such a method could be specified as follows [8]:

[1]We use JML for simplicity. However, any design by contract approach can be used to the following examples because the effect will be the same.

```
1 public void sneakyMethod() {
2  try {
3   int r = 36;
4   decrypt(key, r);
5  }
6  catch (Error e){
7  }
8 }
```

```
9  public void sneakyMethod() throws E {
10  try {
11   int r = 36;
12   decrypt(key, r);
13  }
14  finally{
15   throw new E();
16  }
17 }
```

```
18 public void sneakyMethod() {
19  try {
20   int r = 36;
21   decrypt(key, r);
22  }
23  finally{
24   return;
25  }
26 }
```

Figure 2: Examples categories of exception handling antipatterns [13] which cause the existing RAC approaches to fail report runtime violations [8].

```
//@ requires access_code < 10;
public void decrypt(int[] key, int access_code){...}
```

JML annotation comments start with an at-sign(@). Preconditions are introduced by the keyword **requires**. In this case, the precondition of method decrypt states that the access_code must be less than 10.

A success scenario. Consider a client method denoted by sneakyMethod that calls the method decrypt [8]. Also, assume we have compiled our example including its client code [8] by using the JML RAC compiler ajmlc [16]:

```
public void sneakyMethod(){
 int r = 36;
 decrypt(key, r);
}
```

Hence, any attempt to pass an access code that is out-of-range must always raise a JML RAC error which stops the program's execution:

Unintended Exception subsumption. This is a well-known problem [3] of poorly designed exception handling code. It happens when the declared type in a handler is a supertype on the exception hierarchy and mistakenly catches an exception of a subtype. Figure 2 illustrates an example of an unintended exception subsumption (lines 1–8). Since the method decrypt is used with illegal arguments, its execution should finish abruptly by throwing a JML precondition error. However, the Java virtual machine would not signal this runtime error, since such JML precondition error is caught (subsumed) by a handler (which targets java.lang.Throwable).

Exception overriding. This fault is related to the ability of a handler or finally block to throw another exception, substituting the original exception by the new one [3]. In this scenario we say that it overrides the exception that was initially thrown. Figure 2 presents an example where the method sneakyMethod terminates abruptly by throwing a checked exception of type E (lines 9–17). The problem is that the exception is thrown from within a **finally** block (lines 14–16). Thus, the JML precondition error thrown by the illegal call to the decrypt method is later overridden by another exception E (line 15).

Exception swallowing. Such problem happens when an exception is caught without being adequately handled (by logging or presenting a warning to the user). Figure 2 illustrates an example of exception swallowing (lines 18–26). After a JML precondition error is thrown from within the **try** block (lines 19–22), the **finally** block (lines 23–25) performs a **return** statement (line 24). As with the **return** statement, any **break** or **continue** statement would have a similar effect. Such problem can also happen when a contract violation error is subsumed by an unintended handler problem, and no adequate handling is given to it (as in the example on the left-hand side of Figure 2).

3. PROPOSED APPROACH

This section presents an overview of our approach that leverages aspect-orientation to implement the recovery actions needed for RACs to work in a consistent way. We also give an overview of the main concepts of aspect-oriented programming (AOP) [10]

3.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [10] proposes an approach for improving the separation of concerns in software design and implementation. It proposes a new abstraction, called Aspect, to capture concerns that cannot be easily expressed by the elements of traditional decomposition approaches (e.g., classes, procedures). Such concerns are usually spread over several system modules and tangled with other concerns. Some examples of such concerns are logging,monitoring, transaction management and security. AspectJ [11] is currently the most used aspect-oriented programming language. It incorporates aspect-oriented software development concepts into the Java programming language. The main concepts are the following: (i) join points – well-defined locations within the base code where an aspect can compose with the application (e.g. method calls, method execution); (ii) pointcuts – a collection of join points; and (iii) advice – a special method-like construct consisting of instructions that execute before, after, or around a join point. The around advice executes in place of the indicated join point, which allows the aspect to replace a method.

3.2 Contract Violations Vs Application-Specific Exceptions

Although they share the same type hierarchy, contract violations and application specific exceptions should be treated differently inside the applications. It should not be allowed for the application to handle contract violation errors or exceptions. Since a contract is violated, the result of the computation cannot be trusted. Regarding the exceptions signaled by RACs (contract violations), we argue that as the computation reaches an inconsistent state, such exceptions should be signaled to the system entry point and terminate the program execution. The program's execution might be stopped since there is no way of guaranteeing the result correctness. In our approach whenever a contract violation is signaled by a RAC, it reaches the program entry point. This solution is supported by aspect-orientation as detailed next.

```
1  public abstract aspect EHSafetyRAC {
2   // auxiliary field that stores a RAC error of the program execution
3   private Throwable contractViolation = null;
4
5   // abstract pointcuts
6   private pointcut monitoringTargets();
7   protected abstract boolean isExceptionMonitorable(Throwable e);
8
9   // events under monitoring
10  private pointcut exceptionOrErrorCreated(): call(RuntimeException+.new(..)) || call(Error+.new(..));
11  private pointcut monitoringContext(): monitoringTargets() && !within(EHSafetyRac+);
12
13  //monitoring contract violations
14  after() returning(Throwable e): exceptionOrErrorCreated() && cflow(monitoringContext()) {
15   if(this.isExceptionMonitorable(e) && contractViolation == null){
16    contractViolation = e;
17   }
18  }
19  after(): monitoringContext() {
20   this.rethrowContractViolation();
21  }
22  // auxiliary method
23  private void rethrowContractViolation() {
24   ...
25  }
26 }
```

Figure 3: The EHSafety code snippet.

3.3 Monitoring and Enforcing Contract Violations

The first supporting idea of our approach was to keep track (monitor) every contract violation that is thrown inside an application. In a conventional setting, the monitoring code is scattered across multiple modules and tangled with other application concerns. In order to avoid scattered and tangled code, our approach employs aspect-oriented programming techniques to modularize the monitoring concern. Thus, it entirely decouples the monitoring code from the business code. The monitoring code is able to crosscut specific points (join points) of a system, such as method and constructor executions.

The aspect-oriented monitor pattern [2] was originally conceived for tracking operation performance and thread status. In our approach we extended the notion of aspect-oriented monitoring to gather and filter information about contract violation exceptions (i.e. instances of Java Error or RuntimeException classes) from a set of join points.

Besides monitoring the contract violation errors, our approach ensures that the exceptions detected by RAC reach the program's entry point and stop the program's execution. As with the monitoring concern, the enforcement concern was also implemented by an aspect, as detailed next.

3.4 EHSafetyRac Implementation

The main design decision of our solution was the definition of an aspect library to implement the monitoring and enforcement behaviors to the problems discussed in previous section. Although prebuilt aspect libraries are a relatively new reuse artifact, several useful collections have already become available, including: the Spring AOP aspect library [2], the Glassbox Inspector [3], the JBoss Cache [4], and GOF patterns aspect library [6]. Such libraries typically implement crosscutting functionalities (e.g., performance monitoring, security, and transaction management) that would be spread in many application modules otherwise [10, 11].

Figure 3 illustrates the partial code of the contract monitoring and enforcement aspect (lines 1–26). As observed, the aspect contains one abstract pointcut and one abstract method to be implemented by concrete subaspects defined to specific applications and RACs. The abstract pointcut called monitoringTargets (line 6) denotes the set of points in the application execution (join points) under contract monitoring and enforcement. Hence, each application should provide a concrete implementation to it. The abstract method isExceptionMonitorable (line 7) is responsible for identifying whether or not a thrown exception is an instance of contract violation error (or exception). Thus, each specific RAC should provide a concrete implementation for such method.

This aspect keeps track of every contract violation that happens inside the application (lines 10–15). It is responsible for monitoring every new instance of a RuntimeException or Error that is created inside the application. Since this advice is an **after returning** advice, it is triggered only if the execution of the intercept constructor ends normally. Hence, if any contract violation happens inside the application, it is stored and re-signaled until it reaches the program entry point. This solution simulates a different exception handling mechanism based on program termination.

4. EVALUATION

This section summarizes our experience using EHSafetyRAC aspect library (Section 3) to avoid the negative effects of the fault model, previously discussed, on current Java-based runtime assertion checkers (RACs). The proposed solution was applied to five Java-based contemporary RACs: (i) JML/jmlc [1], (ii) JML/ajmlc [16], (iii) JContractor [5], (iv) Contract Enforcement Aspect Pattern (CEAP) [17], and (v) Jose [4]. The main goal of the study was to assess whether a programmer using the EHSafetyRAC can prevent contract violations from being discarded.

[2]http://www.springframework.org.

[3]https://glassbox-inspector.dev.java.net.

[4]http://www.jboss.org/jbosscache/.

[5]`http://jcontractor.sourceforge.net/index.html.`

The target system used in this study was the HealthWatcher [3] system, a web-based application that allows citizens to register complaints regarding issues in health care institutions. Basically, the HealthWatcher system is structured according to the layer architectural pattern, composed by three layers: GUI, Business, and Data.

We compared the following two use case scenarios: (i) the use of the original RACs to check a set of DbC contracts added in the target system and (ii) the use of the improved version of each RAC (combined with the proposed EHSafetyRAC solution) which checks the same contracts.

The code snippet below was extracted from the business layer. It illustrates the piece of code of the Facade class responsible for implementing the "Insert Employee" use case. This method is responsible for inserting a new Employee in the system. Actually, this method delegates this task to a method on the Data Layer (line 4).

```
1  public void insert(Employee employee) ... {
2   try {
3    getPm().beginTransaction();
4    employeeRecord.insert(employee);
5    getPm().commitTransaction();
6   } catch (ObjectAlreadyInsertedException e) {
7      getPm().rollbackTransaction();
8      throw e;
9   } catch (ObjectNotValidException e) {
10     getPm().rollbackTransaction();
11     throw e;
12  } catch (TransactionException e) {
13     getPm().rollbackTransaction();
14     throw e;
15  } catch (Exception e) {
16     getPm().rollbackTransaction();
17  }
18 }
```

Note that in order to successfully insert a new employee in such a system, a user name, login, and password should be provided. Any missing employee information violates the use case requirement. Hence, in order to prevent any malicious client from bypassing such a validation (leading to inconsistent data in the system), three basic preconditions are added to the insert method from the Data layer (line 4). The code snippet below illustrates the JML contracts defined to represent such preconditions:

```
1 //@ requires !employee.getLogin().equals("");
2 //@ requires !employee.getName().equals("");
3 //@ requires !employee.getPassword().equals("");
4 public void insert(Employee employee) {...}
```

Hence, any attempt to insert a new employee with missing required data, should result in a contract violation that should be presented to the user. However, if the developer is using CEAP [17] or Jose [4] RACs, such contract violation message is never presented to the user. The reason is the following: when a contract violation occurs in such RACs, a RuntimeException is thrown. Hence, if any runtime exception, different from the ones listed in the **try**−**catch** clause is subsumed by the general **catch** Exception clause defined in the Business layer (lines 15−17). As a result, any precondition violation raised in this context is mistakenly caught by such general handler. This fault in source code is related to the first EH antipattern depicted in Figure 2. In order to overcome such problem we combined every RAC with our EHSafetyRAC solution. It was responsible for avoiding the contract violations exceptions to be caught by unintended handlers. Such problem does not occur in JML/jmlc [1] and JML/ajmlc [16] RACs because they throw an Error that is not subtype of Exception.

Table 1: Faults detected when performing RAC with Catch Exception (CE) and Catch Throwable (CT).

RAC Technique	Subsumption		Overriding	Swallowing
	CE	CT	CT	CT
JML/jmlc	✓	x	x	x
JML/jmlc with EHSRAC	✓	✓	✓	✓
JML/amlc	✓	x	x	x
JML/ajmlc with EHSRAC	✓	✓	✓	✓
JContractor	✓	x	x	x
JContractor with EHSRAC	✓	✓	✓	✓
CEAP	x	x	x	x
CEAP with EHSafetyRAC	✓	✓	✓	✓
Jose	x	x	x	x
Jose with EHSafetyRAC	✓	✓	✓	✓

Hence, to further investigate the effectiveness of EHSafetyRAC aspect library we used a fault injection [18] strategy. We manually injected instances of the each one of the exception handling faults described in the fault model, and created subaspects of EHSafetyRAC to each one of the Java-RACs involved in the study. The code snippet bellow illustrates the EHSafetyRAC for JML/jmlc and JML/ajmlc RACs:

```
1 public aspect EHSafetyRACForJML extends EHSRAC{
2  protected pointcut monitoringTargets():
3   execution(execution(public * hw..*.*(..));
4  protected boolean isExcepMonito(Throwable e) {
5    return (e instanceof JMLAssertionError);
6 }
7}
```

Line 2 denotes the definition of the (inherited abstract) pointcut which intercepts the execution of every public method of HealthWatcher system (from which we wanted to recover any contract violation when it occurs). Lines 4−10 depict the implementation of the (inherited abstract) method denoted by isExceptionMonitorable. It is responsible for identifying a particular contract exception thrown by a particular RAC. In this case, the JML contract violation is subtype of the JMLAssertionError type (line 7).

Table 1 illustrates the results of our fault injection study. As expected, all the RAC approaches fail to properly report the precondition violation. This happens because the injected **catch**(Throwable) catches both runtime exceptions or errors. As a result, all the runtime exception and Error-based RACs fail to report the precondition violation.

The combination between the RAC techniques with their corresponding EHSafetyRAC solutions recover the precondition violation raised during a mal-formed query to the method insert (of the business layer). We could observe that through this combination, the subsumed precondition was indeed recovered and properly signaled to the user. These results indicate that our approach based on EHSafetyRAC improved any kind of runtime assertion checker regarding exception safety propagation.

5. DISCUSSIONS

Concurrent Programs. In relation to concurrent programs, if an unhandled exception occurs in a separate thread, only that thread is terminated. In addition, if any thread is blocked by a monitor that the failing thread holds, the blocked threads will be free to acquire the monitor and access a potentially inconsistent part of the program. Finally, if a thread is waiting for a notification from the failed thread, the former will be blocked forever. Our approach currently does not address these problems. The main complicating factor, in this case, is that the exception handling mechanism of Java does not provide support for handling exceptions in concurrent programs. In the future, we intend to devise solutions to circumvent this limitation of Java's exception handling mechanism, to guarantee that contract violations do not leave the system in an unsafe state.

Advances in EH Mechanisms. A number of authors have identified problems with modern exception handling mechanisms and proposed solutions. Some of them have a strong connection with the approach we propose. For example, Jacobs and Piessens [9] have proposed failboxes to guarantee that code that depends on the successful completion of an operation is not executed if the operation fails. We use specific exceptions and guarantee that they are not accidentally caught to achieve the same goal. On the one hand, our approach does not require any modifications to the underlying programming language. On the other hand, failboxes provide stronger guarantees, allowing applications to resume execution with only partial functionality, while ensuring that failed parts are not used.

6. FINAL REMARKS

Aspect-oriented programming definitely is a promising technique for building dependable software. We have found that AOP employed as a error recovery technique has shown to be quite useful for developers. The EHSafetyRAC approach can recover errors (contract violations) detectable by runtime assertion checking even if such violations are hidden or overridden by programs due to the use of practices such as exception handling antipatterns.An appealing property of our approach is that it is neutral with respect to a runtime assertion checking technology. However, we have not yet investigated the promise of EHSafetyRAC with other aspect-oriented (AO) languages with different mechanisms than ones of AspectJ. In the near future, we intend to investigate other AspectJ-like languages to ensure that the suitability of our approach can be used by other AO languages. Our experience to date with EHSafetyRAC is limited to one system, HealthWatcher [3]. However, we argue that any other real system, with the aforementioned kinds of exception handling antipatterns, can benefit of the proposed error recovery approach. Hence, we expect that developers can apply our aspect-oriented recovery approach to help developing dependable software. The EHSafetyRAC technique discussed in this paper is the first initiative to automatically improve dependability of any runtime assertion checker of programs which contain exception handling antipatterns.

Acknowledgements

This work has been partially supported by the National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08. The work of Ricardo Lima, Roberta Coelho, and Fernando Castor were partially supported by CNPq under grants No. 314539/2009-3, No.620132/2008-6, and (308383/2008-7 and 475157/2010-9), respectively. Henrique Rebêlo is also supported by FACEPE under grant No. IBPG-1664-1.03/08. The work of Leavens was partially supported by a US National Science Foundation grant, CNS 08-08913.

7. REFERENCES

[1] L. Burdy et al. An overview of JML tools and applications. *Int. Journal on Soft. Tools for Tech. Transfer (STTT)*, 7(3):212–232, June 2005.

[2] R. Coelho et al. The application monitor aspect pattern. In *Proceedings of PLoP*, PLoP '06, pages 13:1–13:10, New York, NY, USA, 2006. ACM.

[3] R. Coelho et al. Assessing the impact of aspects on exception flows: An exploratory study. In *Proceedings of ECOOP*, ECOOP '08, pages 207–234, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] Y. A. Feldman et al. Jose: Aspects for design by contract80-89. *IEEE SEFM*, 0:80–89, 2006.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley, 2005.

[6] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37:161–173, November 2002.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[8] M. Huisman. On the interplay between the semantics of java's finally clauses and the jml run-time checker. In *Proceedings of FTfJP*, FTfJP '09, pages 8:1–8:6, New York, NY, USA, 2009. ACM.

[9] B. Jacobs and F. Piessens. Failboxes: Provably safe exception handling. In *Proceedings of ECOOP*, ECOOP'09, pages 470–494, Berlin, Heidelberg, 2009.

[10] G. Kiczales et al. Aspect-oriented programming. In *Proceedings of ECOOP*, number 1241 in ECOOP '97, pages 220–242. Springer-Verlag, June 1997.

[11] G. Kiczales et al. Getting started with aspectj. *Commun. ACM*, 44:59–65, October 2001.

[12] G. T. Leavens et al. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, Mar. 2006.

[13] T. McCune. Exception handling antipatterns. Article, 2006.

[14] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[15] R. Miller and A. R. Tripathi. Issues with exception handling in object-oriented systems. In *ECOOP*, pages 85–103, 1997.

[16] H. Rebêlo et al. Implementing java modeling language contracts with aspectj. In *Proc. of the 2008 ACM SAC*, pages 228–233, New York, NY, USA, 2008.

[17] H. Rebêlo et al. The contract enforcement aspect pattern. In *Proc. of the 2010 SugarLoafPLoP*, pages 99–114, 2010.

[18] J. M. Voas and G. McGraw. *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

A. Online Appendix

We invite researchers to replicate our preliminary study. Source code of the target system and their specification using five DbC approaches with and without EHSafetyRAC are available at: `http://cin.ufpe.br/~hemr/ftfjp11`.