Autonomic Test Case Generation of Failing Code Using AOP

by

Giovanni Murguia
B.Sc., Instituto Tecnológico y de Estudios Superiores de Monterrey, Mexico 2008

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

Autonomic Test Case Generation of Failing Code Using AOP

by

Giovanni Murguia
B.Sc., Instituto Tecnológico y de Estudios Superiores de Monterrey, Mexico 2008

Supervisory Committee

Dr. Hausi A. Müller, Supervisor
(Department of Computer Science)

Dr. Alex I. Thomo, Departamental Member
(Department of Computer Science)

Supervisory Committee

Dr. Hausi A. Müller, Supervisor
(Department of Computer Science)

Dr. Alex I. Thomo, Departamental Member
(Department of Computer Science)

ABSTRACT

As software systems have grown in size and complexity, the costs of maintaining such systems increases steadily. In the early 2000's, IBM launched the autonomic computing initiative to mitigate this problem by injecting feedback control mechanisms into software systems to enable them to observe their health and self-heal without human intervention and thereby cope with certain changes in their requirements and environments. Self-healing is one of several fundamental challenges addressed and includes software systems that are able to recover from failure conditions. There has been considerable research on software architectures with feedback loops that allow a multi-component system to adjust certain parameters automatically in response to changes in its environment. However, modifying the components' source code in response to failures remains an open and formidable challenge.

Automatic program repair techniques aim to create and apply source code patches autonomously. These techniques have evolved over the years to take advantage of advancements in programming languages, such as reflection. However, these techniques require mechanisms to evaluate if a candidate patch solves the failure condition. Some rely on test cases that capture the context under which the program failed—the patch applied can then be considered as a successful patch if the test result changes from failing to passing. Although test cases are an effective mechanism to govern the applicability of potential patches, the automatic generation of test cases for a given scenario has not received much attention. ReCrash represents the only known implementation to generate test cases automatically with promising results through the use of low-level instrumentation libraries.

The work reported in this thesis aims to explore this area further and under a different light. It proposes the use of Aspect-Oriented Programming (AOP)—and in particular of AspectJ—as a higher-level paradigm to express the code elements on which monitoring actions can be interleaved with the source code, to create a representation of the context at the most relevant moments of the execution, so that if the code fails, the contextual representation is retained and used at a later time to automatically write a test case. By doing this, the author intends to contribute to fill the gap that prevents the use of automatic program repair techniques in a self-healing architecture.

The prototype implementation engineered as part of this research was evaluated along three dimensions: memory usage, execution time and binary size. The evaluation results suggest that (1) AspectJ introduces significant overhead with respect to execution time, (2) the implementation algorithm causes a tremendous strain on garbage collection, and (3) AspectJ incorporates tens of additional lines of code, which account for a mean size increase to every binary file of a factor of ten compared to the original size. The comparative analysis with ReCrash shows that the algorithm and data structures developed in this thesis produce more thorough test cases than ReCrash. Most notably, the solution presented here mitigates ReCrash's current inability to reproduce environment-specific failure conditions derived from on-demand instantiation. This work can potentially be extended to apply in less-intrusive frameworks that operate at the same level as AOP to address the shortcomings identified in this analysis.

Contents

Supervisory Committee ii

Abstract iii

Table of Contents v

List of Tables viii

List of Figures ix

List of Listings x

Acknowledgements xii

1 Introduction 1

1.1 Motivation . . . 1
1.2 Problem . . . 2
1.3 Goal and approach . . . 3
1.4 Research questions . . . 3
1.5 Contributions . . . 4
1.6 Thesis overview . . . 4

2 Background and related work 6

2.1 Autonomic computing and self-healing systems . . . 6
2.2 Overview of self-healing architectures . . . 7
2.3 Java instrumentation . . . 9
2.4 Aspect oriented programming (AOP) concepts . . . 11
2.4.1 Crosscutting concerns . . . 11
2.4.2 Advices . . . 12
2.4.3 Join points . . . 13

2.4.4 Pointcuts . . . 13
2.4.5 Aspects in AspectJ . . . 13
2.4.6 Aspect weaving . . . 15
2.5 Automated testing concepts . . . 17
2.5.1 Stubs and mocks . . . 17
2.6 Java memory management . . . 20
2.7 Automatic program repair . . . 21
2.8 Related work . . . 21
2.8.1 ReCrash . . . 22
2.8.2 FERRARI and MAJOR . . . 23
2.9 Chapter summary . . . 23

3 Approach to generate failure-contextualized test cases automatically 24
3.1 Prototype phases . . . 24
3.2 Chapter summary . . . 26

4 Design and implementation 27
4.1 Design methodology . . . 28
4.2 Context-relevant data structures . . . 28
4.2.1 Object call . . . 29
4.2.2 Boundary . . . 32
4.2.3 Object use . . . 34
4.3 Context-meaningful join points . . . 34
4.3.1 Marking target classes . . . 35
4.3.2 Constructor join point . . . 36
4.3.3 Initialization block join points . . . 37
4.3.4 Method join points . . . 37
4.3.5 Field join points . . . 38
4.3.6 Excluding join points . . . 38
4.4 Advice definition . . . 40
4.5 Test generation algorithm . . . 44
4.6 Additional implementation details . . . 44
4.6.1 Serialization considerations . . . 45
4.6.2 Generics . . . 46
4.7 Chapter summary . . . 46

5 Evaluation of AspectJ and ReCrash **47**
5.1 Augmented binary size analysis 47
5.2 Execution time and memory analysis 51
5.3 AspectJ limitations for effective program monitoring 55
5.4 Implementation differences 56
5.4.1 Test usability and EACB coverage 56
5.4.2 Limitations on environment-dependent interactions 59
5.4.3 EACB coverage differences 60
5.5 Chapter summary 60

6 Conclusions **62**
6.1 Thesis summary 62
6.2 Contributions 63
6.3 Future work 64
6.4 Final words 66

References **67**

A AspectJ vs ReCrash script source code **73**

B AspectJ-woven empty constructor **75**

C Time and memory comparison details **81**

List of Tables

Table 2.1 Weaving mechanisms available in AspectJ 16

Table 4.1 Mapping between pointcut type and join points 40
Table 4.2 Advice definition 43

Table 5.1 Details of the target libraries 48
Table 5.2 Details of the environment used for benchmarking 54
Table 5.3 Details of the performance metrics obtained using the JMH . . . 54
Table 5.4 EACB comparison between RI and AI 60

List of Figures

Figure 4.1 Object call definition 31
Figure 4.2 Graphical representation of the boundaries observed in Listing 4.4 33
Figure 4.3 Boundary definition 34
Figure 4.4 Object use definition 35

Figure 5.1 Code percentage increase observed in target libraries augmentation using AI 50
Figure 5.2 Code percentage increase observed in target libraries augmentation using RI 50
Figure 5.3 Comparison of AI vs RI on the ANTLR library 51
Figure 5.4 Comparison of AspectJ's woven binary with and without monitoring vs RI on the ANTLR library 52
Figure 5.5 Binary growth with respect to the original file size 53

List of Listings

2.1 Basic implementation of a `ClassFileTransformer` 9
2.2 Aspect declaration syntax in AspectJ [22] 13
2.3 Pointcut declaration syntax in AspectJ [22] 14
2.4 Advice example 14
2.5 Simple class with a basic sum operation 17
2.6 Stub for the Operations class 18
2.7 Mock for the Operations class 19
2.8 Mocks for on-demand instances and static methods 19
4.1 An interaction whose response depends on the current state of the environment 30
4.2 Naive JUnit test 31
4.3 JUnit test mocking the method B.methodWithVariableResponses . . 32
4.4 Boundary lifespan 33
4.5 Class hierarchy modification. Interface CallTrackerMarker is introduced to all classes inside the org.apache package and subpackages . . 36
4.6 Problematic implementation of the toString() method 39
4.7 Pointcut declaration 41
4.8 Implementation of the `constructor` pointcut for the `after` advice specification 43
4.9 Process to load classes dynamically 46
5.1 Error obtained on the log4j weaving process 49
5.2 Pattern followed by ReCrash to create tests 56
5.3 Pattern followed by AI to create tests 58
5.4 Environment-dependent EACB 59
B.1 Empty constructor to be woven 75
B.2 The result of weaving an empty constructor using AspectJ 75
C.1 JMH source code to compare the instrumentation modes 81

C.2 Source code for the org.apache.commons.lang3.ArrayUtils.addAll(...) method 82

ACKNOWLEDGEMENTS

I would like to thank:

Dr. Hausi Muller, my supervisor, for supporting me and my family throughout this journey. This would have been impossible without his guidance, kindness, and understanding. Thank you for giving me the opportunity to work with you and learn so much.

My family, for supporting me every step of the way. For not letting me fall during the toughest of times. **Karla**, for enduring and overcoming adversity together with me on so many occasions that it becomes difficult to recall them all; for believing in me and helping me stay sane. **Abu Ade**, for helping us in so many ways that it would not fit here. **Jackie**, for bringing so much joy into our lives and making every day special. My parents, **Socorro and Manuel**, for helping me stay strong regardless of the distance. **Kitzia and Aram**, for taking care of everything back home. **Abu Efrain**, for sharing our motivation; celebrating our success, and giving perspective to our failures.

Tania Ferman, for guiding me from the very beginning and beyond. I was lost so many times but you helped me find my way every time.

My lab mates, for allowing me to learn from them. **Miguel**, for being an unofficial mentor. Your ideas, opinions, and feedback helped me mold my program. **Priya**, for helping me understand so many aspects of being a student at UVic and researching (not to mention the amazing pictures!). **Sunil, Karan, Ivan, Alvi, and Felipe**, for helping me stay motivated and learn from many different areas of research.

Every person we called to help us entertain our unrelenting Jackie. It is a very long list, but you helped me get to this stage.

Chapter 1

Introduction

1.1 Motivation

Software failures are inherent to software development. For example, the process of identifying, reproducing, and fixing bugs is a time-consuming activity that software developers have to perform routinely. As programs grow in size and complexity, the number of such failures increases. Historically, developers have devised mechanisms to aid in the contextualization of such failures. For example, program-generated logs are commonly used to print the state of the program at a certain point in time to a resource—usually a file. When a failure occurs, developers read the logs looking for indicators to diagnose the root cause. However, the efficacy of logs relates directly to the detail and quality of the log data, and the ability of the developer to interpret and correlate the data obtained from the logs with the program's logic. Furthermore, the execution of a program that writes log entries is sub-optimal due to:

1. **Use of machine resources.** Text manipulation and I/O calls[1] are examples of activities associated with logging that consume resources. Depending on the logging setup this can be extremely expensive. Writing log entries to a network resource synchronously will halt the program execution until a response is obtained.[2]

2. **Indirect maintenance costs.** Logging logic is designed and implemented by

[1] Input/Output normally requires system calls which add an overhead to the program.

[2] Log4J2 (https://logging.apache.org/log4j/2.x/) provides an asynchronous mode in which I/O communication is delegated to a surrogate thread. In spite of this improvement, thread communication and synchronization also add an overhead.

a developer, therefore making it error-prone as well. Furthermore, for it to be useful, updates to the business logic need to be reflected in the logging messages through logging logic adjustments. Neglecting such adjustments may diminish the quality of the logged data, and in severe cases it might contradict the actual context of a particular execution,[3] defeating its purpose.

An alternative approach to the developer-centric failure resolution are self-healing systems—as detailed in Chapter 2—in which the system is able to identify the erroneous state, recover from it and adjust itself to avoid crashing again under the same conditions. The advantage of this approach is a lower requirement of developer involvement in maintenance tasks, allowing them to concentrate on business adjustments.

This thesis aims to provide the mechanisms to create a contextualized representation of the failing program through a set of automatically created test cases, which would be a stepping stone to achieve a self-healing system that can recover from software failures. Although the aforementioned case is the principal objective of this work, those same test cases might be extremely useful to a developer to understand the conditions under which the program failed.

1.2 Problem

There has been significant work in the field of automatic program repair (APR) [10, 17, 26, 36]. APR can potentially be part of the executor in a self-healing system that fixes its own software failures that lead to an aborted program or *crash*. These techniques require some kind of metric to measure if a candidate patch effectively allows the system to avoid crashing under the conditions previously recorded. In particular, some techniques use a test case for this purpose given that test cases can isolate the failing code while controlling their interactions.

However, generating test cases automatically from a running program has been only lightly explored. ReCrash [2] accomplishes the automatic generation of test cases in response to program failures, but it handles failure conditions derived from on-demand instantiation ineffectively.

[3] Consider, for example, a variable that was renamed and repurposed, while the logging message is misleadingly referring to the previous use.

1.3 Goal and approach

The goal of this thesis is to contribute to the autonomic generation of test cases for failing code. A *test case* is a short program that configures the program under test and executes specific functions of it, with the purpose of automating the testing process. The test cases generated are intended to serve as a measuring mechanism of the efficacy of a candidate patch that aims to solve the original error condition. We define a *patch* as a source code change that attempts to solve a specific problem. Although the initial program failure is not avoided through this technique, subsequent crashes can be avoided once a successful patch is applied to the source code.

To accomplish this objective, this thesis follows the steps described below.

1. Design data structures to logically organize the state of a monitored program with sufficient data to reproduce the context of the interactions among components.

2. Define monitoring elements that can be injected to a given program, and mechanisms to extract the program's execution context to populate the data structures defined previously.

3. Define an algorithm to generate test cases from the state retained in the data structures. This algorithm can then be executed in response to a software failure.

4. Construct a prototype to validate the components defined in this thesis and to evaluate the implementation in controlled scenarios.

5. Evaluate the implementation in terms of binary size, execution time, and memory consumption. We select three publicly available libraries and augment them with the prototype constructed and ReCrash—the most relevant test-generation library for software failures.

1.4 Research questions

RQ1 Which Aspect Oriented Programming (AOP) elements can be used to monitor a program and reproduce its state in a test case for programmatic errors?

RQ2 What are the limitations of AspectJ to effectively and comprehensively monitor Java programs?

RQ3 Which data structures can be used to support the construction of an application-context monitor?

RQ4 What is the performance cost of monitoring Java applications using AOP and AspectJ compared to a purely instrumentation-based one, such as ReCrash?

1.5 Contributions

C1 Identification of AspectJ's relevant join points and design of pointcuts and advices to effectively monitor an application and replicate the runtime context of a failure condition.

C2 A compendium of AspectJ's limitations observed during the construction and evaluation of AspectJ's monitor implementation.

C3 A thorough definition of data structures that can represent a program's execution context at one given point of time and that can be used to reproduce that context on demand.

C4 An AspectJ-based reference implementation of a monitor and test-generation component and side-to-side performance comparison (i.e., binary size, execution time, and memory usage) of the AspectJ implementation listed in C2 and ReCrash's implementation.

1.6 Thesis overview

This chapter explains the motivation behind this work and underlines the research questions that guide the following chapters; it also outlines the main contributions obtained as a result of this thesis. The remaining chapters are organized as follows:

Chapter 2 introduces concepts that are required to understand the underlying context of the problem as well as the particular elements utilized in later chapters.

Chapter 3 connects these concepts and explains the technological decisions made.

Chapter 4 presents the design and implementation details.

Chapter 5 discusses the findings obtained in the evaluation of the prototype implementation and compares it to ReCrash.

Chapter 6 presents a summary of the work done in this thesis and introduces points to consider for future work.

Chapter 2

Background and related work

2.1 Autonomic computing and self-healing systems

Autonomic computing is a popular concept proposed by IBM in 2001 [13, 14] when it was clear that software systems were becoming more sophisticated and the resources required to maintain such systems were increasing at an alarming pace. Hence, the goal of autonomic computing was to tackle the intricateness of large-scale systems, via the implementation of feedback control mechanisms meant to augment such systems, allowing them to improve themselves based on user-defined criteria.

IBM introduced the concept of the *autonomic element* [19] as a component to be used in the orchestration of autonomic systems. An autonomic element consists of one or more managed elements—the components being controlled—and an autonomic manager which controls the managed elements. The autonomic manager comprises a feedback loop to coordinate the actions needed to control the autonomic element and act depending on the state of it. This is known as the monitor-analyze-plan-execute (MAPE) loop or monitor-analyze-plan-execute-knowledge (MAPE-K) loop [32].

One of the fundamental challenges of autonomic computing is self-healing, which comprises systems that are capable of detecting, analyzing and adapting—without human intervention—to faults originating within the system [1, 33]. The common architecture of a self-healing system is composed of four main components:

- **Monitors**. Observe the system with minimal obstruction and gather relevant data that will be used to infer the state of the system.
- **Interpreters of monitored data**. Analyze gathered data and identify if the system has suffered from an internal fault.

- **Repair-plan creators**. Define the steps to follow which will prevent the system from failing again under the presence of the monitored conditions.
- **Executors of the repair plan**. Apply the steps defined to the target system.

The work of this thesis focuses on the implementation of the first two components, laying out the ground for the latter components with the ultimate goal of addressing programmatic errors autonomously.

2.2 Overview of self-healing architectures

Over the past decade, multiple prominent reference architectures for self-healing systems emerged. This section summarizes the most relevant architectures.

One type of self-healing architecture reuses the available infrastructure to augment multiple systems with a centralized control model. Rainbow [8] uses external agents that constantly monitor a defined set of parameters on the target system and act when such parameters fall beyond an established threshold. The main advantage of this approach is that monitored systems are loosely coupled to the control model, allowing to apply the same architecture to any system capable of exposing hooks to monitor and adapt, regardless of the system's context. However, the authors reported a round-trip overhead in agent-system communication. Consequently, highly-dynamic systems must first evaluate if the aforementioned overhead is acceptable for the operation of the target system.

Service-Oriented Architectures (SOA) have received considerable research work attention aiming to augment the base architecture with self-healing components. Moses [4] and SASSY [29] are two such approaches. Moses relies on a Business Process Execution Language (BPEL) to manage adaptations in concrete services, prioritizing non-functional requirements. The control model is an external MAPE loop that serves as a message-broker for clients of the target system. Unlike Moses, SASSY customizes the architectural decisions using the knowledge from domain experts who establish a set of system services architectures and parameters that outline the boundaries that keep the system under acceptable levels of quality of service (QoS). A key difference with respect to Moses is that SASSY creates a model of system components, allowing it to use heuristics to modify the architecture dynamically.

Dynamico is a reference architecture for self-adaptive systems proposed by Villegas et al. [42] which supports dynamic adaptation goals for systems whose specific envi-

ronments change frequently and require a more flexible definition of the adaptation goals. It is composed of three levels of feedback loops: one that manages the control objectives, one to manage the target system adaptation, and one to manage context monitoring. This innovative architecture was further evaluated by Tamura et al. in [39] comparing it against the prominent Rainbow architecture discussed previously. For the comparison, a controlled news website—Znn.com—was the target system to be adapted according to certain adaption goals. As a result of the adaptations, the Apache server's configuration parameters were modified to respond to changes in the environment.

Self-healing architectures based on software components have not received substantial research work, compared to service-based architectures. In a service-based architecture, each service can be considered as an independent program that interacts with others through messages;[1] in comparison, a component-based architecture is a single program and components interact among them through method calls. To the best of the author's knowledge, the only work of this type of architecture has been Vuori's [43] architecture, where the control model is a software component that is considered within the system's design; other control-oblivious components contain module-specific functionality to heal themselves. If a monitored component falls beyond an acceptable threshold, it is considered to be *sick* and is therefore isolated from the rest of the components until its internal healing process finishes. If there is an alternate component that may replace the sick one temporarily, it is effectively used to replace the other; if no such component exists, the performance of the system may either degrade or even stop the system execution altogether until the offending component returns to an acceptable state. One unique feature of this approach is that once the component finishes the healing process, it is validated against test data to ensure the problem is fixed and it is ready to be reincorporated into the system.

The work on this thesis is intended to support a hybrid architecture that incorporates flexible adaptation goals to support a mechanism to dynamically modify which parts of the system should be monitored as an alternative to global monitoring; and a component-based architecture, given that setting up a service-based architecture to adapt an inherently self-contained software component is impractical.

[1] Sometimes those messages are sent through the network to remote locations.

2.3 Java instrumentation

The instrumentation API was introduced in Java 5 and allows to modify binaries as they are loaded into the JVM through the use of *agents* [6]. Agents are Java programs that define how the binaries should be modified using `ClassFileTransformer`s. The instrumentation API is cumbersome and error-prone. Let us consider Listing 2.1.[2] Line 11 receives the actual bytes of the class, and it gives an opportunity to modify the bytecodes before returning it.

Listing 2.1: Basic implementation of a `ClassFileTransformer`

```
1  import java.lang.instrument.ClassFileTransformer;
2  import java.security.ProtectionDomain;
3
4  public class Transformer implements ClassFileTransformer {
5    @Override
6    public byte[] transform(
7      ClassLoader loader,
8      String className,
9      Class<?> classBeingRedefined,
10     ProtectionDomain protectionDomain,
11     byte[] classfileBuffer) {
12       byte[] modifiedByteCode = classfileBuffer;
13       // Here one would apply transformations to the
            bytecodes
14       // The following is an example of introducing two
            variables to the source code of a single class to
             keep track of the execution time
15       byte[] byteCode = classfileBuffer;
16       String finalTargetClassName = this.targetClassName
17       .replaceAll("\\.", "/");
18       if (!className.equals(finalTargetClassName)) {
19         return byteCode;
20       }
21
```

[2]The transformation code is part of an instrumentation tutorial in https://www.baeldung.com/java-instrumentation .

```
22        if (className.equals(finalTargetClassName)
23        && loader.equals(targetClassLoader)) {
24
25          LOGGER.info("[Agent] Transforming class MyAtm");
26          try {
27            ClassPool cp = ClassPool.getDefault();
28            CtClass cc = cp.get(targetClassName);
29            CtMethod m = cc.getDeclaredMethod(
                WITHDRAW_MONEY_METHOD);
30            m.addLocalVariable("startTime", CtClass.longType
                );
31            m.insertBefore("startTime = System.
                currentTimeMillis();");
32
33            StringBuilder endBlock = new StringBuilder();
34
35            m.addLocalVariable("endTime", CtClass.longType);
36            m.addLocalVariable("opTime", CtClass.longType);
37            endBlock.append("endTime = System.
                currentTimeMillis();");
38            endBlock.append("opTime = (endTime-startTime)
                /1000;");
39
40            endBlock.append("LOGGER.info(\"[Application]
                Withdrawal operation completed in:" + "\" +
                opTime + \" seconds!\");");
41
42            m.insertAfter(endBlock.toString());
43
44            byteCode = cc.toBytecode();
45            cc.detach();
46          } catch (NotFoundException |
               CannotCompileException | IOException e) {
47            LOGGER.error("Exception", e);
48          }
```

```
49          }
50          return modifiedByteCode;
51      }
52  }
```

Nevertheless, applying changes to the class bytes directly is a difficult task—analogous to working with assembly code directly—but libraries such as javassist[3] provide an API to use text that will be then converted to the appropriate bytecodes. While this is better than working with bytecodes directly, the API is still challenging to understand and use effectively. The task to accomplish in Listing 2.1 is a simple one. Still, it requires of over 30 lines of convoluted code.

2.4 Aspect oriented programming (AOP) concepts

AOP is a programming paradigm whose purpose is to encapsulate non-functional requirements[4], known as cross-cutting concerns, into reusable, declaratively-applied components known as *aspects* [22]. The concepts introduced in this section set the stage to understand how the prototype developed in this thesis is designed.

2.4.1 Crosscutting concerns

A *concern* is a requirement—functional or non-functional—that a system must comply with to operate effectively [22]. In a system we can identify *primary concerns*, such as withdrawing money from an account, and *secondary concerns*, such as traceability of operations. El-Hokayem, Falcone and Jaber [5] define *crosscutting concerns* as "Concerns [that] are often found in different parts of a system, or in some cases multiple concerns [which] overlap one region". A crosscutting concern is a secondary concern that is inextricably linked to multiple primary concerns, and therefore is difficult to encapsulate without introducing a dependency on the primary concerns. A primary concern is addressed by the program itself. For example, withdrawing money from an account is a primary concern, whereas recording the time of the transaction is a secondary one that could be shared with a deposit operation. Therefore it is a crosscutting concern. A canonical example of a crosscutting concern is caching invocation results, which is not the main purpose of a program because it is likely linked

[3] http://www.javassist.org/

[4] For example, enforcing an authorization step to all interactions with the system.

to a performance, non-functional requirement. In this example, the response time of cached invocations is improved, but building and maintaining a cache is unrelated to the logic behind heterogeneous invocations.

Hokayem, Falcone and Jaber [5] identify *code-scattering* and *tangling* as the two major consequences of not externalizing crosscutting concerns. Code-scattering refers to logic that is fragmented in multiple parts of the program (e.g., logging the parameters of a defined set of methods). Tangling refers to code that is not related to the primary objective of a component (e.g., authorizing access to parts of the program depending on a given role). This thesis uses AOP to abstract the logic required to monitor the execution context of a program, which is a crosscutting concern. Because several parts of the system need to be observed by this component—as discussed in Section 4.3—the risk of code-tangling and scattering is huge if it was not introduced to the system through an instrumentation mechanism.

2.4.2 Advices

The concept of *advice* was introduced by Teitelman in 1966 [40], as an innovation to modify programs of his PILOT system. He defines advices as "new procedures [inserted] at any or all of the entry or exit points to a particular procedure (or class of procedures)". Teitelman also states that since advices are procedures with independent entry and exit points, they can alter the conventional flow of a program, allowing to override adviced procedures completely. This definition is basically preserved in AOP, where advices are the logic that is inserted into well-defined parts of an existing program. An *advice specification* states the conditions under which the advice is applied:

- **before**: the advice is executed before forwarding the program execution to the selected join point.
- **after**: the advice is executed after the join point completes.
- **around**: the join point execution is intercepted, and the advice determines the conditions to forward control to the join point.

In the cache example, an advice would be responsible for creating the cache, forwarding invocations to the original code for cache-misses, returning cached values, and expiring cache entries. An *around* advice would provide the conditions to fulfill this requirement.

2.4.3 Join points

A *join point* represents a program's execution point in which advices can be applied. Join points are part of what is known as aspect-aware interfaces [21], which are the properties that constitute interfaces in AOP. Examples of join points are method or function calls, variable assignments, control flow structures, etc. An AOP library defines which join points are supported and provides mechanisms to declare them. For example, Spring AOP only supports method join points[5] whereas AspectJ supports instance-variable operations, method and constructor call and execution, exception handlers, among others.[6] We will refer to *advicing* as the process of executing an advice on a join point. In the cache example, advices would be applied to method call or method execution join points, to intercept method invocations.

2.4.4 Pointcuts

A *pointcut* combines a join point or group of join points of interest together with the advice specification and defines the conditions under which it can be adviced, expressed by a *pointcut type*. For example, an *execution* pointcut occurs when the join point is executed, whereas a *call* pointcut takes place before the actual execution. The AOP library will define which pointcuts it supports and the capabilities of each pointcut type.[7] In the cache example, an *execution* pointcut would be defined *around* the methods to be cached.

2.4.5 Aspects in AspectJ

An *aspect* combines all the previous concepts into a single entity. In the case of AspectJ, an aspect definition resembles a class definition. Notably, aspects may maintain state via data members and methods, and they are subject to inheritance mechanisms being objects themselves, i.e. an aspect may extend a class or an aspect, implement interfaces, include abstract members, and so forth.

---

[5] According to Spring's documentation, section "Spring AOP Capabilities and Goals" in https://docs.spring.io/spring/docs/5.2.x/spring-framework-reference/core.html#aop-introduction-spring-defn.

[6] The entire list of join points supported by AspectJ is available at https://www.eclipse.org/aspectj/doc/next/progguide/quick.html.

[7] A complete list of the pointcuts supported by AspectJ can be found in https://www.eclipse.org/aspectj/doc/next/progguide/semantics-pointcuts.html .

Listing 2.2: Aspect declaration syntax in AspectJ [22]

```
[access specification] aspect <AspectName>
[extends class-or-aspect-name]
[implements interface-list]
[<association-specifier>(Pointcut)] {
        ... aspect body
}
```

Furthermore, an aspect may declare named pointcuts using a syntax similar to method declarations.

Listing 2.3: Pointcut declaration syntax in AspectJ [22]

```
[access specifier] pointcut pointcut-name([args]) :
pointcut-definition
```

Finally, an aspect may contain zero or more advices. Listing 2.4 shows an example of a basic advice which prints a message for every field assignment of subtypes of the user-defined `MyMarker` type.

Listing 2.4: Advice example

```
public aspect MyAspect {

  public pointcut myFields() : set(* MyMarker+.*);

  before() : myFields()  {
    System.out.println("Before a field assignment.");
  }
}
```

AspectJ is not the only implementation of AOP. However, it is a well-established, robust project that started in 2001 and has been regularly maintained. Furthermore, it is well known in the industry and because of this other AOP frameworks borrow its syntax.[8]

[8]For example Spring AOP https://docs.spring.io/spring/docs/2.5.x/reference/aop.html .

2.4.6 Aspect weaving

Aspect weaving refers to the process of modifying a program's instructions to apply advices, according to the pointcut definition. An *aspect weaver*—or just *weaver*—is the component that consumes aspects and combines the aspect instructions with the target program. AspectJ supports three mechanisms to weave aspects, as shown in Table 2.1

Table 2.1: Weaving mechanisms available in AspectJ

Weaving mechanism	**Description**	**Software phases affected**	**Considerations**
Source code weaving	A modified compiler acts as the weaver. It pre-processes source code of primary concerns in conjunction with the source code of aspects.	Compilation process	• Availability to the source code for the primary concern is mandatory. • Updates to aspect definitions require the recompilation of both primary and secondary concerns' source code.
Binary weaving	Shares the modified compiler used for source code weaving. It injects aspects to existing Java binary files as a post-processing task and creates a new binary file.	Compilation process	• Source code of primary concerns is irrelevant to the compilation. • Updates to either primary or secondary concerns still require re-compilation for weaving purposes.
Load-time weaving	A Java agent modifies the class-loading process to inject aspects to Java classes as they are loaded into the Java Virtual Machine (JVM)	Execution process	• Compilation of primary and secondary concerns occurs independently from each other. • Execution of the main program requires additional JVM options to use a weaving agent.

As it can be seen, AOP provides a simple yet powerful model to encapsulate code and insert it at arbitrary locations using a well-defined syntax. This simplicity contributes to alleviate the risk of introducing bugs using low-level instrumentation.

2.5 Automated testing concepts

Testing is a fundamental part of the software development process. Initially, it is used to validate that functional and non-functional requirements are met. Then, as the program evolves, to ensure that the deployment of new features does not negatively impact other parts of the system [20].

Automated tests allow a fast, consistent, and reproducible environment in which tests are executed by a dedicated program, and test results are aggregated and reported for further analysis. These tests may be written at different levels, from unit tests that cover individual pieces of software, to integration and resiliency testing, which aim to improve the overall robustness of the system. This thesis aims to create unit tests that reproduce failures on individual components.

2.5.1 Stubs and mocks

The purpose of unit tests is to evaluate individual components, minimizing the interaction with external components. However, this is a complex task for non-trivial programs, as it is cognitively easier to split functionality into *chunks* of code that are simpler to manage and understand [31]. Therefore, it is not uncommon for a component to have multiple dependencies that complement its functionality.

One technique to isolate a given component's execution from other components is to write what is canonically known as a *stub* [23]. A stub is a component that extends or implements a given interface, to return fixed responses to method calls. This technique allows us to decouple the class-under-test so that errors in other classes do not affect the test result of the target component. A developer usually writes these stubs on demand, but as more components require the same type of stub, increases the difficulty of managing a family of stubs. For example, let us consider the class in Listing 2.5 and the corresponding stub in Listing 2.6.

Listing 2.5: Simple class with a basic sum operation

```
public class Operations {
  public int sum(int a, int b) {
```

```
    return a + b;
  }
}
```

The stub is removing all the business logic from the method and instead, it returns a fixed value. This ensures that any object that is under test and which depends on `Operations` or, in this case, the `OperationsStub` implementation, will not be affected if the `sum` method had programmatic errors.

This technique, though simple, is not the preferred approach to simulate interactions among components. The reason is that if a class had N number of methods, and the developer only wanted to simulate one interaction, she would have to provide empty implementations of N-1 methods. Furthermore, if the structure of the component being stubbed was modified, e.g. adding or removing methods, all stubs would need to reflect the modification.

Listing 2.6: Stub for the Operations class

```
public class OperationsStub extends Operations {
  public int sum(int a, int b) {
    return 5;
  }
}
```

Mocks, on the other hand, are components that are usually provided by a mocking framework [38]. They allow isolating dependencies following a clean, well-established syntax. Mocks allow to set up expectations on *fake* implementations within the test file itself. This mechanism makes it more natural to set up fixed behaviors for the test writer, and are clearer to understand for the test maintainer. Mockito[9] is one such framework for the Java programming language, which is popular in the open-source community [30]. Listing 2.7 shows how a mock component is configured for the `Operations` class discussed previously using Mockito. Line 1 creates the mock object for the class or interface provided—in this case for the `Operations` class. The instance returned is meant to be used to intercept method calls and record the expected behavior, which is done in Line 2. Mockito allows specifying behavior for a specific set of parameters, or for all possible values, through the use of static methods such as `anyInt()`. Then, it can be configured to throw exceptions and return values, among other possible behaviors, for the specified method call.

[9] https://site.mockito.org/

Listing 2.7: Mock for the Operations class

```
1 Operations opsMock = Mockito.mock(Operations.class);
2 when(opsMock.sum(anyInt(), anyInt())).thenReturn(5);
```

In version 1.x of Mockito, mocks are created using dynamic proxies[10]. This design decision imposes limitations on the use cases that can be covered using Mockito. For example, static methods, final classes, and on-demand instances are not candidates for mocking because dynamic proxies are incapable of overriding the default object-creation process. PowerMock[11] is an alternate framework that extends Mockito's functionality through bytecode modifications. This allows us to cover scenarios such as the ones mentioned previously.

To illustrate PowerMock's capabilities, let us consider Listing 2.8. In the first case, `OnDemand` represents a class that is instantiated directly inside a method, and which cannot be injected as a dependency of the target object being tested. The configuration shown in Lines 2 to 3 configure PowerMock to intercept the constructor call on `OnDemand` via bytecode modification and return the supplied mock object. Lines 7 to 8 show the configuration used to mock calls to a static method in the class `StaticMethods`. Similarly to on-demand instances, the bytecode is modified to intercept the method call to the static method and fixate the behavior that it should display.

Listing 2.8: Mocks for on-demand instances and static methods

```
1 // Handling on-demand instances
2 OnDemand onDemand = Mockito.mock(OnDemand.class);
3 PowerMockito.whenNew(OnDemand.class).withNoArguments().
     thenReturn(onDemand);
4
5 // Handling static methods
6 PowerMockito.mockStatic(StaticMethods.class);
7 PowerMockito.when(StaticMethods.someStaticMethod()).
     thenReturn("static method with fixed value");
```

[10] https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html
[11] https://github.com/powermock/powermock

2.6 Java memory management

One of the most important features of Java, when it was designed, was its memory management paradigm, which distinguishes it from other programming languages where the developer is responsible for the allocation and deallocation of memory (e.g., C or C++) by introducing the concept of a garbage collector. The purpose of the garbage collector is to identify object instances that are not reachable and release their resources without direct intervention from the developer.

Java's memory space is divided into multiple sections. The first two sections in which it is divided are the stack and the heap. The stack keeps track of the bytecode instructions that the running thread is executing. The heap stores objects and class definitions; every object that is created is dynamically allocated on the stack and a *reference* is created to gain access to the object. References are memory address where the object memory starts.

To properly manage the deallocation of objects from the heap, Java uses a garbage collector, which is a component that keeps track of object references and releases memory only when all references to an object have been freed.[12] If a single reference exists, independently of its location, the object is kept in memory. This mechanism avoids a problem known as *dangling pointers* where an object's memory is reclaimed but there are still references pointing to that memory address, which is common in languages with explicit memory management [18]. For garbage collection to be effective, it needs to pause the executing program—a process known as *stop-the-world pauses*—so that the algorithm can find which objects have still references and safely deallocate those that are not in use. This is important because programs that rely on dynamic memory management through the use of a garbage collector are affected by a) the number of times that the garbage collector runs in a unit of time, and b) the time it takes to complete each garbage collection execution.

Generational garbage collectors divide the heap into two main areas: *old generation*—or just old gen—and *young generation*, which is further divided into the *eden* and *survivor* spaces [34]. New object instances are allocated in the eden part of the young generation area. When a garbage collection cycle runs, it frees the memory of objects that do not have active references pointing at them. Objects that are still being used are moved to either the survivor area or the old gen, depending on the

[12] Weak and soft references are an exception to this mechanism, but this study focuses in strong references which are more common.

garbage collection algorithm; this process has the positive side effect of compacting regions as they are being cleared. The reason is that those objects have a higher probability of staying loaded in memory. This has a performance gain since garbage collection runs are executed on a smaller area of the heap, where ephemeral objects are to be expected. When one memory area or subarea is filled, a garbage collection execution runs on that area.

2.7 Automatic program repair

Automatic Program Repair (APR) focuses on techniques meant to patch program failures with minimal human interaction. Some approaches use machine learning techniques to train a model using some sort of data bank that is mined to modify the offending code until a patch successfully resolves the root cause of the failure, for example [25, 36, 11, 12, 10]. Other approaches, such as the ones presented in [24, 26, 44, 17, 9, 37], teach the model the elements that correct code frequently uses, and the algorithm modifies the source in an attempt to make it more *correct* according to the patterns observed.

Both approaches share a common requirement: to determine when a program failure—or a type of error—is fixed, they require a signal or form of evaluation that yields a positive result when a patch is successful. In particular, some of them rely on a test case that captures the failure conditions and initially fails, similar to a Test Driven Development strategy [15]. This restriction forces the involvement of developers to analyze the failure, determine the conditions, and write a test case, which would make it unusable in a self-healing system.

2.8 Related work

Existing research work has focused mostly on the generation of test cases for existing source code. Pacheco and Ernst, for example, designed Randoop using a specific type of random testing called *feedback-directed random testing* where test cases are created incrementally, taking into account the input from previous generated tests [35]. Fraser and Arcuri, on the other hand, use a genetic algorithm in EvoSuite to automatically generate test cases that try to cover as much code as possible [7]. However, both of these techniques start with random input that is iteratively improved to try to

generate high-quality tests. A complementary approach is to monitor an executing application and generate a test case if an error is thrown; this is the case of ReCrash.

2.8.1 ReCrash

To the best of the author's knowledge, the most important work that is related to automatic test generation for faulty code is ReCrash [2]. Artzi, Kim, and Ernst propose a technique to reproduce program errors, facilitating the maintenance phase of software systems. It consists of two phases: monitoring and test generation. During monitoring, a shadow stack is generated for a running program. The motivation of the stack is to store the execution context (in particular the value of variables and method-call parameters) such that, if the program throws an unhandled error, it can be used to reproduce the error under the same conditions as the executing program. In the test generation, the shadow stack data is used to generate a self-contained test case that enables developers to understand the conditions under which the error occurs and fix the code with minimal effort. As the authors identified, the cost of maintaining the aforementioned shadow stack is not negligible, and they proposed techniques to optimize the monitoring phase, namely limiting the depth of the stack copies and limiting the number of methods under monitoring.

The implementation of ReCrash relies on instrumentation of the Java Virtual Machine (JVM), specifically ASM.[13] While a powerful tool to alter the byte-code of existing programs, the instrumentation API is brittle in regards to robustness; the creation of Java agents is error-prone due to the low-level in which instrumentation operates.

The principal constraint of ReCrash is the definition of the shadow stack. Serialized representations of the program objects impose a limitation in the usefulness of such objects to reproduce the program context. For example, an instance of the class `FileInputStream`[14] depends on the host file system, making its serialized object unpredictable and potentially introducing a new bug in the test case. Besides, as the authors identified and circumvented by imposing restrictions, the shadow stack is susceptible to consume considerable amounts of memory for complex object graphs.

[13] https://github.com/xingziye/ASM-Instrumentation
[14] https://docs.oracle.com/javase/7/docs/api/java/io/FileInputStream.html

2.8.2 FERRARI and MAJOR

One important aspect to consider while using instrumentation techniques—including AspectJ—to monitor Java applications is that Java core classes are unusually candidates for bytecode modifications. Binder, Hulaas, and Moret [3] discuss how disrupting the bootstrapping process of the JVM can have adverse effects on the normal execution, including JVM crashes. As a result, they proposed FERRARI, a hybrid static and dynamic instrumentation mechanism to control how Java core classes are instrumented, effectively instrumenting core classes. Then, Villazon et al. [41] leveraged FERRARI to modify the standard AspectJ weaver, allowing it to weave all JVM classes, including core Java classes. This is relevant to the work of this thesis because it enables us to monitor and potentially capture the context of not only custom libraries, but even those provided by the JVM implementation. Even more remarkably, Villazon reimplemented ReCrash using its modified AspectJ weaver. While this is an improvement to the low-level instrumentation ReCrash was originally designed with, it uses the same concept of a shadow stack. As a result, it suffers from equivalent limitations, such as the presence of big object graphs being kept in memory due to the use of object references instead of lightweight representations.

2.9 Chapter summary

This chapter introduced concepts to better understand where this thesis's work is intended to be applied; it described the autonomic computing initiative and, in particular, self-healing architectures. Then, it introduced the ideas and tools that are used in this thesis to achieve context-monitoring and test case generation, most notably join points, pointcuts, advices, and aspects. It discussed memory management in Java to establish the connection between the different memory areas and the garbage collector, which are relevant for the evaluation of this thesis's prototype. Finally, it concludes with a discussion on the literature that addresses automatic program repair and test generation as a response to program failures.

Chapter 3

Approach to generate failure-contextualized test cases automatically

3.1 Prototype phases

Chapter 2 explained how Dynamico defined adaptation goals to reconfigure Znn.com's parameters and meet SLAs such as throughput. A similar approach can be used to adapt an executing system when software failures occur through the introduction of an APR subsystem in the adaptation feedback loop. However, such subsystems require metrics to evaluate candidate patches; the metric is defined typically by a failing test that reproduces the failure, such as in the work of Long and Rinard [26, 25]. In this way, a candidate patch that turns a failing test case into a passing test case can be considered as final. Thus, it is imperative to find a mechanism that automatically generates test cases that contain the context under which a program failed, which would be incorporated into the adaptation feedback loop as well.

The resulting test cases need to contain enough context to reproduce the failing case. Thus, the subsystem must hold a monitoring component that manages this context. AOP allows us to define join points expressively, and therefore it is of interest if it contains enough elements to gather failure-relevant context. Although it is uncertain at which point a program may fail, it is possible to use AOP to insert code surrounding each method call and alert the monitor when an unhandled error occurred. The context gathered at that point then represents the conditions under

which the program failed. The work in this thesis demonstrates that AspectJ—the canonical implementation of AOP for Java—can be used to monitor applications and retain data that allow constructing the context, albeit with some limitations and a considerable performance hit. This phase is equivalent to ReCrash's monitoring phase, thus it will be identified as well as the "monitoring phase".

After the monitoring phase has exposed the context upon a software failure, the test creation phase translates that context into JUnit test cases. Then, those test cases can be plugged into the program's test suite to begin the process of an APR system.

The general idea of the monitoring phase is to maintain a list of all objects created and to map each method call with instance creation, method calls onto those instances, and instance variable accesses. This allows the creation of a graph that represents the entire context of the program in execution. Additionally, the aspect also creates a boundary to define the beginning and end of the method construction. Similarly, a pointcut is appended to every public method, defining the beginning and end boundaries of the method call; and the method parameters and return instances are stored. Should an error occur within that boundary, every instance created and used within that boundary is used to construct the runtime context of the method where the exception was thrown. That context is composed of new instances, detail of the method calls on such instances as well as to any instance objects. Finally, the state of used instance variables is recorded as well.

The test case generation phase uses the runtime context to write a test case that reflects the precise conditions in which the error occurs. It is important to note that anything that falls beyond the method boundary is irrelevant. The execution context is delimited through the use of mock objects. These objects represent method calls to dependencies of the instance without it being necessary to represent the inner state of the dependencies themselves. The implementation of this thesis uses Mockito and PowerMock to define mocks in the JUnit test cases, due to the fact that PowerMock allows to mock method calls of instances created within a method and static method calls, which are essential to effectively handle these common cases.

Similarly to ReCrash's implementation, the AspectJ implementation constructed in this thesis is composed of two phases: a) a monitoring phase where context is constantly updated to reflect the exact application state at that moment, and which is executed along with the program being monitored; and b) a test generation phase which runs offline to consume the context obtained in the monitoring phase. The

context is kept in memory during normal execution, and it is serialized to a JSON file when the program fails.

The implementation on AspectJ distinguishes itself from ReCrash in three main aspects:

- AOP is a higher-level mechanism compared to direct bytecode instrumentation. It is analogous to the difference between using assembly language instead of a high-level language. And similarly to this analogy, there is a cost involved in this choice of abstraction. Chapter 5 shows that the performance overhead of using AspectJ is excessive and it reduces its applicability.

- AspectJ's implementation does not store object references. Instead, it stores interaction representations that allow it to reliably reproduce scenarios where object references are not enough (e.g., interactions with files or network resources that cannot be serialized by ReCrash).

- References are not being kept in the monitoring stack. Therefore, the memory will be released earlier than in ReCrash's implementation. However, this has the side effect of depleting the eden memory area often with slow garbage collection cycles. This behavior is described in detail in Chapter 5.

3.2 Chapter summary

This chapter explained the approach that this thesis follows to achieve an AOP-based tool to respond to program failures. Following an approach similar to ReCrash, it is composed of two phases: a monitoring phase that is executed along with the program, and a test generation phase that can occur at the moment of failures or, on-demand, at a later time. The context is retained in the form of a JSON file with textual object representations, as opposed to ReCrash's object references.

Chapter 4

Design and implementation

As outlined in Section 2.4, unobstrusive and well-defined advices provide a relevant alternative to ReCrash's instrumentation-based introspection, through the use of simple yet powerful pointcut model. A simpler model can help overcome the inherent complexity of the low-level instrumentation API and, therefore, easing the process of monitoring a subset or new code blocks. Analogous to ReCrash's approach to reproduce failures in test cases, this thesis considers two main components: a *monitor* which executes in parallel with the main program, and an offline *test-case generator* which uses the monitored context to recreate the error conditions as a JUnit test case. A test case generator is a program that reads the context persisted by the monitor and writes a test case in the form of a Java class file. The result of executing the test case is either a success state or a failure state. Initially, the test case written by the test case generator must fail, as the program has not been modified and it was executed under the observed failure context. If the program source code is modified in such a way that the test case ends in a success state, then the failure conditions was properly handled and it must not fail again if the same conditions occur outside of the test case. One of the contributions of this thesis is the definition of a high-level algorithm to create AOP-centric monitors in languages where AOP libraries are available. However, the heterogeneity of programming languages and AOP libraries impede the complete generalization of such algorithm, requiring it to be customized for the target environment. Despite this limitation, the proposed data structures contain the minimum elements that a monitor must retrieve to provide sufficient context for the test-case generator.

This chapters presents several code snippets to clarify the purpose of the data structures defined, comparing testing approaches to address specific failure scenarios.

4.1 Design methodology

This thesis proposes the following design methodology for the AOP-based monitor and test-case generator components.

Define minimal context-relevant data structures. AOP provides substantial data associated with the join point. However, the monitor should be defined to discriminate among context-rich data to avoid an excessive use of memory with irrelevant details. Additionally, the data structures should logically organize the data in such a way to allow the test-case generator to produce meaningful test-cases. A test case is meaningful if it reproduces a particular use case that the original program might run into. For this thesis, that use case represents the failure conditions.

Identify context-meaningful join points. Not all join points are relevant to the assembly of a practical, non-exhaustive representation of a program's execution graph. For example, assignments to instance fields are join points that do not enrich the failure context, therefore being irrelevant to the monitoring phase. Consequently, a key step in this methodology is to constrain the join points to be observed by the monitor. For example, control flow join points would not reveal data to be used by the previously defined data structures.

Define advices. Each defined join point requires at least one advice that executes an action to manage the execution context. The advice definition should allow the recollection of sufficient data to fill the context-relevant data structures. Advices should take into account relevant join points and identify which pointcut type is appropriate and exposes the current stage of the execution.

4.2 Context-relevant data structures

This section introduces the data structures that will be used throughout the implementation to recreate failure conditions, namely object call, boundary, and object use.

4.2.1 Object call

An *object call* models the execution of an individual method call. This includes parameters and returned values, if any. Figure 4.1 shows how the data structure is defined. The way an object call contributes to the context relies on the representation of an individual, unique interaction between two components, namely a method caller and a method callee. When a failure occurs, boundary-related object calls are used to recreate those interactions that lead to the error, thus effectively isolating the target component. Note that cycles in the interactions would be a condition that would make both the original program and the monitor to fail, and is therefore a scenario that would not be handled properly by this approach. This is relevant for interactions whose response depends on the current state of the environment. An example of this is depicted in the code in Listing 4.1. Method `A.targetMethod`'s execution might either succeed or fail depending on the value returned by method `B.methodWithVariableResponses` at line 12. Listing 4.2 shows a candidate JUnit test to cover this interaction.

Listing 4.1: An interaction whose response depends on the current state of the environment

```
public class A {
  B b;

  public static void main(String... args) {
    A a = new A();
    a.b = new B();

    a.targetMethod();
  }

  public void targetMethod() {
        int random = b.methodWithVariableResponses();
        if (random > 100_000) {
          throw new IllegalStateException();
        }
  }
}

class B {
  public int methodWithVariableResponses() {
    return new java.util.Random().nextInt();
  }
}
```

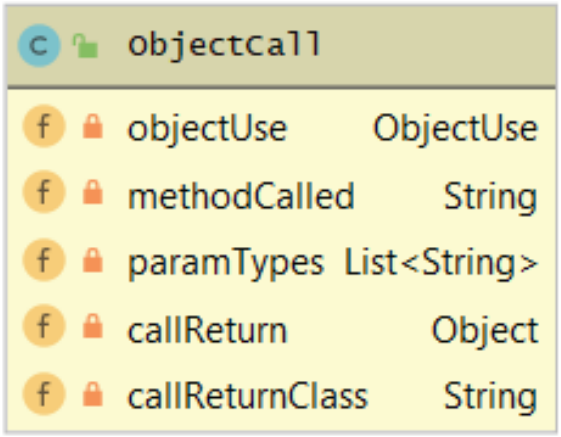

Figure 4.1: Object call definition

Listing 4.2: Naive JUnit test

```
public class ATest {
  @Test(expected = IllegalStateException.class)
  public void testTargetMethod() {
    A a = new A();
    a.b = new B();

    a.targetMethod();
  }
}
```

However, since the callee creates a pseudo-random number, the test in Listing 4.2 will succeed only occasionally, hindering its usefulness in a test suite. A hand-written stub for class B would be ideal for this use case, but a mock can work with minimal effort—compared to writing a stub just for this test—and it eases the understanding of the conditions under which the program failed. Listing 4.3 shows a test case following this approach, where lines 11 and 12 set a fixed interaction, thus allowing the test to succeed for all executions.

Listing 4.3: JUnit test mocking the method B.methodWithVariableResponses

```
1  @RunWith(org.mockito.junit.MockitoJUnitRunner.class)
2  public class ATest {
3    @Mock
4    B b;
5
6    @InjectMocks
7    A a;
8
9    @Test(expected = IllegalStateException.class)
10   public void testTargetMethod() {
11     org.mockito.Mock.when(b.methodWithVariableResponses())
12       .thenReturn(100_001);
13
14     a.targetMethod();
15   }
16 }
```

An object call captures the interaction shown in Listing 4.1 and can be used to create a test case similar to that shown in Listing 4.3 programmatically.

4.2.2 Boundary

A *boundary* delimits the borderlines within which local object calls need to be recorded to reproduce a program failure. Boundaries can be nested since boundary events themselves can lead to program failures. The selection of language structures whose boundary representation provides relevant context data is driven by the language grammar. In Java, public, protected and package-level methods—both static and instance—and constructors, and initializer blocks—both static and instance, as well—are structures that can be isolated to reproduce the runtime context effectively.

A boundary lifespan is contained in the executing block. Listing 4.4 shows a simple example where boundaries are marked with code comments; note that `Boundary 3` is a nested boundary. Figure 4.2 depicts the boundaries for this example. Although `Global boundary` is not explicitly defined in the code snippet, every program has this top-level boundary implicitly.

Listing 4.4: Boundary lifespan

```
1  public class TargetClass {
2    static {
3      // Boundary 1 begins here
4      // Boundary 1 ends here
5    }
6
7    public TargetClass() {
8      // Boundary 2 begins here
9      targetMethod();
10     // Boundary 2 ends here
11   }
12
13   public void targetMethod() {
14     // Boundary 3 begins here
15     // Boundary 3 ends here
16   }
17 }
```

Figure 4.3 depicts the boundary data structure used for the prototype built for this thesis.

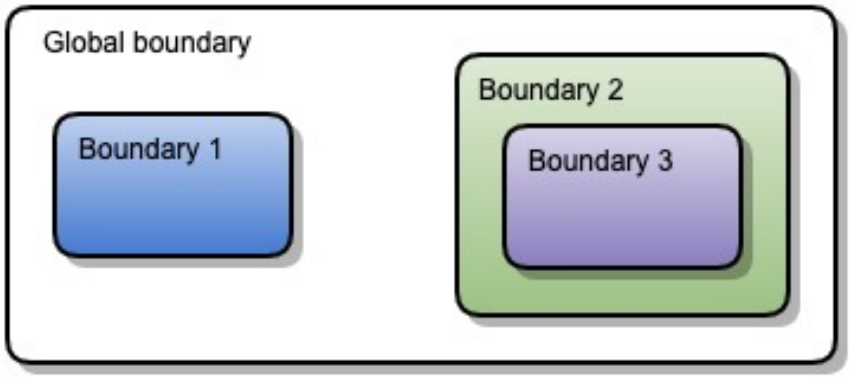

Figure 4.2: Graphical representation of the boundaries observed in Listing 4.4

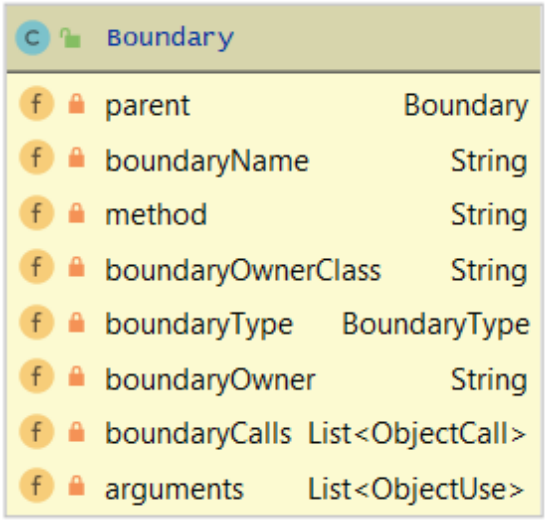

Figure 4.3: Boundary definition

4.2.3 Object use

An *object use* is intrinsically linked to an object call because it captures the context of the instance under which a method is invoked. Figure 4.4 illustrates the data structure corresponding to object use. The context is conformed by:

- **An object type** because of the contextual differences among instance variables, local variables, method parameters, or method-result instances.
- **An instance counter** to distinguish between instances and connect a single instance to multiple uses, namely fields and static variables, new instances, method-returned instances, and arguments.
- **Parameter-centric data**, such as parameter value for basic types, namely primitives, primitive wrappers and String, and method call position, which is required to recreate method invocations.

4.3 Context-meaningful join points

AOP defines a wide selection of join points, but only a subset of them are relevant to capture the minimal execution context required to reproduce a failure in a test

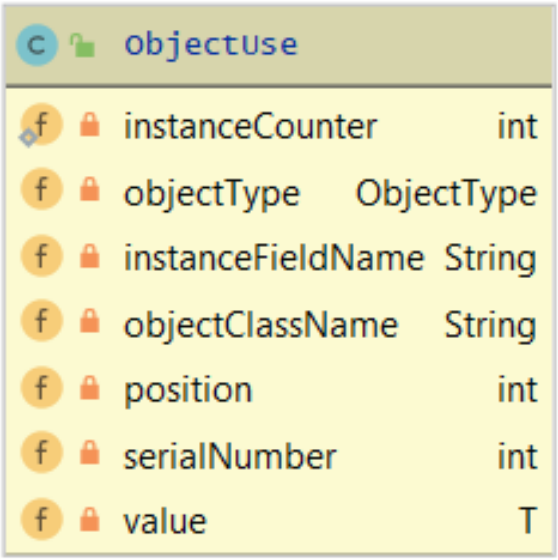

Figure 4.4: Object use definition

case. We call *externally accessible code blocks* (EACBs) to public, protected, default-scoped, and static code blocks. EACBs are relevant as they can be acted on by external triggers including method calls or an instance creations. Conversely, private code-blocks can only be used internally, and therefore they can be reached through an EACB. Furthermore, any interaction with external components must occur through an EACB as well.

This section groups context-meaningful join points for Java using AspectJ on per-boundary basis, as defined in Section 4.2.2, and defines a mechanism to *mark* subsets of target classes.

4.3.1 Marking target classes

AspectJ's aspects can not be weaved in all classes since some classes need to be loaded earlier in the program execution. For example, weaving core classes such as `java.lang.Object` can potentially have harmful effects in the JVM [41]. To overcome this restriction, our prototype implementation defines a subset of classes and modifies their class hierarchy introducing marker interfaces, such as the one shown in Listing 4.5.

Listing 4.5: Class hierarchy modification. Interface CallTrackerMarker is introduced to all classes inside the org.apache package and subpackages

```
public aspect CallTrackerAspect {
  declare parents: org.apache..* implements
     CallTrackerMarker;
  ...
  public interface CallTrackerMarker {}
}
```

The interface declared at line 4 is then added to the hierarchy of all types specified in line 2. This setup allows us to bind join points to the marker interface, decoupling class marking from join point definitions.

4.3.2 Constructor join point

Constructors are code blocks whose executions occur at most once for every class instantiation (instances created through serialization overrides construction invocation). Constructor-bound join point's contribution to the execution graph context is twofold:

1. Binding to a boundary new instances, which allows us to define an association of the object used in an object call to the appropriate object type and distinguish it from fields, arguments, and returned instances.

2. Binding constructor parameter values or references to the constructor boundary, defined as object uses. If the constructor fails, these parameters allow us to write a test case that invokes the correct constructor with equivalent parameters.

Join point (4.1) is used to select constructors and initialization blocks on all marked classes. The `+` on the right of the marker interface denotes all implementations, which will is controlled by the marker on Listing 4.5. The `new` portion of the join point selects constructors and the `..` notation is used to select constructors indistinctly of the type and number of the parameters. Only the private access modifier is not an EACB, thus the join point excludes it from the selection.

$$!private\ \ CallTrackerMarker+.new(..) \tag{4.1}$$

4.3.3 Initialization block join points

Initialization blocks are blocks of code that allow the introduction of initialization steps in programs. Their lifespan begins earlier than the constructor's but it ends until the constructor has completed. Similarly to constructors, each initialization block is executed at most once per class instantiation.

Join point (4.2) selects initialization blocks. It is almost identical to Join point (4.1), but it does not restrict on any modifier given that init blocks are executed regardless of it.

$$CallTrackerMarker + .new(..) \tag{4.2}$$

4.3.4 Method join points

Methods are code blocks that are associated with a class. They can be invoked by the class they are defined in or, depending on their access modifier, by external classes. Their contributions to the context are:

1. Similarly to constructor join points, binding method parameters values, or references to the method boundary.
2. Linking the current method execution to the previous boundary. This is vital to accurately create the execution graph.
3. Binding the method result—if any—to the object call.

Join point (4.3) is used to select instance methods on all marked classes. The first * denotes all methods regardless of their return type, including `void` methods; while the second * selects all methods irrespective of their name.

$$!private \quad * \quad CallTrackerMarker + . * (..) \tag{4.3}$$

Join point (4.4) complements Join point (4.3) by including static methods. The only addition to the join point is the `static` modifier.

$$!private \ \ static \quad * \quad CallTrackerMarker + . * (..) \tag{4.4}$$

4.3.5 Field join points

Fields are variables that pertain to a class and can hold both primitive types and object references. Their contribution to the context is:

1. The unequivocal identification of the field a method was invoked on. This is crucial to assign expectations to the appropriate instance.

2. The storage of the object state for the lifespan of the executing boundary.

Join point (4.5) is used to select all fields in implementations of the marker interface. It is similar to Join point (4.3) but it excludes parameters, making it a field join point.

$$* \quad CallTrackerMarker + .* \tag{4.5}$$

Join point (4.6) adds specificity to Join point (4.5), by replacing the first * with subtypes of the marker interface, which will restrict the join point to select only fields that hold instances that might be used in an object call.

$$CallTrackerMarker + \quad CallTrackerMarker + .* \tag{4.6}$$

Finally, since static variables do not belong to a specific instance, and therefore require a distinct treatment during test creation, Join point (4.7) includes the `static` modifier to Join point (4.6).

$$static\, CallTrackerMarker + \quad CallTrackerMarker + .* \tag{4.7}$$

4.3.6 Excluding join points

AspectJ compiles aspects into classes, which makes aspects subject to aspect-weaving themselves. Thus, it is important to exclude monitoring aspects in monitoring advices to avoid monitoring irrelevant classes and potentially creating circular calls that turn into `StackOverflowException`s.

Join point (4.8) selects support methods weaved to the marked classes. Such methods manage the current role of the instance (e.g. argument, new instance, etc.). Apart from being irrelevant to the original execution context, they lead to circular calls.

$$* \quad CallTrackerMarker.*(..) \tag{4.8}$$

Join point (4.9) is similar to Join point (4.8), but it selects marker support fields that share the same purpose as marker support methods. Comparably, such fields are irrelevant to the original execution context.

$$* \quad CallTrackerMarker.* \tag{4.9}$$

Join point (4.10) selects the `java.lang.Object.toString()` method. Weaving this method may lead to big invocation graphs with little value or, in worst cases, circular calls when the implementer uses the internal state to construct the textual representation of the object, as shown in Listing 4.6.

$$public \quad * \quad *.toString(..) \tag{4.10}$$

Listing 4.6: Problematic implementation of the toString() method

```
class A {
  private B b;

  public String toString() {
    return "A [" + b.toString() + "]";
  }
}

class B {
  private C c;

  public String toString() {
    return "B [" + c.toString() + "]";
  }
}
...
```

Table 4.1: Mapping between pointcut type and join points

Pointcut type	**Purpose**	**Join points to use**
execution	Intercept method and constructor execution.	(4.1), (4.3), (4.4), (4.8), (4.10)
this	Capture the EACB executing instance.	(4.1), (4.3), (4.5), (4.6)
initialization	Intercept initialization block execution.	(4.1)
get	Intercept field access within an EACB.	(4.5), (4.6), (4.7), (4.9)

4.4 Advice definition

This section defines the advices that use the join points identified in Section 4.3 to provide enough data to populate the data structures defined in Section 4.2. Advice definition is organized on a per-join point basis applying the pointcut declarations shown in Listing 4.7. Table 4.1 summarizes the relation between pointcut types and join points. Only `before` and `after` advice specifications are used; though it is possible to obtain equivalent results using `around`. Our implementation avoids `around` to have a clear separation of concerns.

The contribution of each advice is clearer by specifying pre-execution and post-execution actions, as shown in Table 4.2. There are some characteristics that must be considered when reading this table:

1. In the **Pointcut name** column names repeat in several rows because the contribution they provide to the data structures overlap, and it becomes simpler to understand this overlap by looking at the pointcut name and focusing on one row at a time.

2. In the **Contributions to data structures** column the object each advice contributes to is specified in bold at the beginning of the cell. Clarifications are done inside parenthesis to provide more information about how the data structure is used inside the advice. Then, the contributed fields are listed. A contributed field means that the field is populated on the data structure because the corresponding data is available in the advice, and during the corresponding advice specification the data will contain the appropriate context.

Listing 4.7: Pointcut declaration

```
// Auxiliary pointcuts
pointcut notIrrelevantMethods(): !execution(* CallTrackerMarker.*(..))
                                 && !execution(public * *.toString(..));

pointcut fieldsNotInCallTrackerMarker() : !get(* CallTrackerMarker.*);

// Core pointcuts
pointcut constructor(!private CallTrackerMarker accessTracked): execution(CallTrackerMarker+.new(..))
                                                                && this(accessTracked);

pointcut initBlock(CallTrackerMarker accessTracked): initialization(CallTrackerMarker+.new(..))
&& this(accessTracked);

pointcut method(!private CallTrackerMarker accessTracked): execution(* CallTrackerMarker+.*(..))
                                                           && notIrrelevantMethods()
                                                           && this(accessTracked);

pointcut staticMethod(): execution(!private static * CallTrackerMarker+.*(..));

pointcut markerFields() : get(CallTrackerMarker+ CallTrackerMarker+.*);

pointcut staticFields() : get(static CallTrackerMarker+ CallTrackerMarker+.*);
```

```
pointcut serializableFields() : get(Serializable CallTrackerMarker+.*)
                          || get(boolean CallTrackerMarker+.*)
                          || get(byte CallTrackerMarker+.*)
                          || get(char CallTrackerMarker+.*)
                          || get(short CallTrackerMarker+.*)
                          || get(int CallTrackerMarker+.*)
                          || get(long CallTrackerMarker+.*)
                          || get(float CallTrackerMarker+.*)
                          || get(double CallTrackerMarker+.*);
```

Table 4.2: Advice definition

Pointcut name	**Advice specification**	**Contribution to data structures**
constructor, method, initBlock, staticMethod	before	**Boundary (creates boundary)** boundaryName, parent, boundaryType, boundaryOwnerClass, method, arguments.
	after	**Boundary (removes boundary)**
constructor, method, staticMethod	before	**ObjectUse (for each argument, appended to the previous boundary)** objectType, objectClassName, position, serialNumber, value.
markerFields, staticFields	after	**ObjectUse** instanceFieldName
serializableFields	after	**ObjectUse (appended to the current boundary)** objectType, targetClass, value, instanceFieldName
staticMethod, method	after	**ObjectCall (appended to the previous boundary)** objectUse, paramTypes, callReturn, callReturnClass

Having defined the advices, the monitoring implementation is straightforward. Each advice is implemented to map the data available through AspectJ's API. For example, let us consider the first row of Table 4.2. The implementation of the `constructor` pointcut with an `after` advice specification is shown in Listing 4.8. Note that the instance marked with the marker interface `CallTrackerMarker` is exposed through the `constructor` pointcut, and then it is made available to the advice as a parameter to the `after` advice specification. `ReferencesStack` is a convenience wrapper on a regular stack of boundaries and the code in Line 2 just removes the boundary at the top of the stack, effectively closing the current constructor boundary.

Listing 4.8: Implementation of the `constructor` pointcut for the `after` advice specification

```
1 after(CallTrackerMarker accessTracked): constructor(
    accessTracked) {
2   ReferencesStack.removeBoundary();
```

```
3 }
```

4.5 Test generation algorithm

To generate a test from a JSON-based[1] stack, we propose an iterative algorithm over the boundaries available in the stack. The pseudocode is shown in Algorithm 1.

Generating Java code programmatically is a cumbersome, error-prone activity. Therefore, this thesis's prototype implementation leverages JavaPoet, a Java API to generate Java source files [2].

Algorithm 1 abstracts the logic needed to build the code generator. Some important caveats are:

1. Triggering EACBs will vary greatly among one another. For example, a method invocation EACB is triggered just by invoking the method, but a static initialization block will only run once when the class is loaded by the class loader. This may be problematic, for example, if the test instance variables include the class under test.

2. Setting up mock expectations will vary depending on the `ObjectUse.objectType` of the `ObjectCall`. For example, setting up a static method invocation requires the use of PowerMock because our version of Mockito uses dynamic proxies to create mock objects. Other `ObjectUse.objectType`s are simpler and can be implemented without external libraries.

3. This thesis's implementation creates unique `ObjectUse`s for every type during instantiation. This was an implementation decision to ease matching different `objectTypes` that pertain to the same instance. However, this means that during the test generation phase a single mock object may potentially have different calls chained to it.

4.6 Additional implementation details

This section discusses some details found during this thesis's prototype implementation.

[1] https://www.json.org/
[2] https://github.com/square/javapoet

Algorithm 1 Test generation algorithm

```
1: procedure CreateTestCase(stack)
2:     boundary ← stack.pop
3:     while boundary ≠ emtpy do          ▷ The same process for every boundary
4:         objectCalls ← boundary.boundaryCalls
5:         if objectCalls ≠ emtpy then
6:             create top class element
7:             load boundary owner class
8:             objectUses ← boundary.fieldUses
9:             for all objectUse ← objectUses do
10:                declare objectUse as instance variable    ▷ Mock objects that will simulate the EACB interactions with fields
11:            end for
12:            create setup method
13:            for all objectCall ← objectCalls do
14:                objectUse ← objectCall.objectUse
15:                if objectUse not in instance variables then
16:                    add objectUse to instance variables
17:                end if
18:                set up mock expectation objectCall in setup method    ▷ This is where the interaction is emulated
19:            end for
20:            create test method
21:            trigger EABC          ▷ this can be an object call, constructor call, etc.
22:            persist test case
23:        end if
24:    end while
25: end procedure
```

4.6.1 Serialization considerations

This thesis's prototype implementation initializes monitoring-specific fields in the constructor. Therefore, alterations to the serialization process[3] impact AOP's inserted fields because values can be used before they are initialized. If the object is created through `readObject(...)` as opposed to the normal constructor process, then the constructor is not invoked and the fields the algorithm depends on are not initialized. The evaluation showed that objects created through deserialization lack the context under which they were created and therefore the monitoring process was broken.

[3]See for example how the serialization protocol can be customized in https://docs.oracle.com/javase/7/docs/platform/serialization/spec/output.html#861

4.6.2 Generics

Java 5 introduced the concept of generic types to the language. This improved type safety and it was quickly incorporated in core classes—notably the collections API—and common libraries depending on the library nature. Thus, the AspectJ implementation leverages the Reflection API[4] to extract the actual types at runtime, and class names are stored in the context-aware data structures. Then, the test generation component loads dynamically the classes as shown in Listing 4.9.

Listing 4.9: Process to load classes dynamically

```
final Class<?> receiverClass = Class.forName(fieldUse.
   getObjectClassName());
```

4.7 Chapter summary

This chapter defined the data structures that represent the interaction context of a running program, namely, object call, boundary, and object use. It explains the selection of AspectJ join points used to monitor a Java program effectively, in particular those that represent context-relevant interactions, defined as EACBs. The chapter then describes the design of advices that are attached to those join points and map the available context-data to the data structures. Finally, it introduces the test-generation algorithm that takes a stack of boundaries and produces test cases that emulate interactions among EACBs.

[4]https://docs.oracle.com/javase/tutorial/reflect/index.html

Chapter 5

Evaluation of AspectJ and ReCrash

This chapter compares ReCrash's implementation (RI) with this thesis in terms of augmented code size, as well as time and memory overhead. The motivation behind this comparison is because both ReCrash and AspectJ's implementations (AI) modify the program by inserting bytecode instructions in the original binaries, and even though AspectJ provides a more expressive interface, it also requires more instructions to achieve such flexibility. This chapter quantifies the weaving process to understand how slower woven code is compared to low-level instrumentation. Finally, this chapter analyzes both implementations in a real world project, using the framework *defects4j*[1]. Based on our analysis, we conclude that on the one hand, RI is fast, but memory-intensive and it does not cover all EACBs; on the other hand, AI makes more efficient use of memory and addresses many more EACB's, but is around 20 times slower.

5.1 Augmented binary size analysis

Binary code augmentation via instrumentation consists of the insertion of arbitrary bytecodes at predefined places on target binaries. The analysis conducted in this section reveals how binaries are affected by ReCrash and AspectJ's binary augmentation. This is the first dimension of the evaluation of AspectJ as a context-monitoring tool. Being aware of this impact helps us to understand how much extra-processing will be required to monitor the running context of target binaries. We selected three libraries

[1]https://github.com/rjust/defects4j

Table 5.1: Details of the target libraries

Library	Library version	Applicable classes count
antlr	2.7.2	163
commons-codec	1.11	88
commons-io	1.3.2	57

for this analysis, and they will be referred to as *target libraries* in this chapter. These libraries were selected because (i) the author has experience using them and considers they are heterogeneous in terms of purpose, and (ii) only a very reduced subset of libraries succeeded in the augmentation process of both RI and AI.

- **antlr**[2] Used in parser generators. Allows to read a language that performs lexical and grammatical analysis. It also creates Java source code that can be used to build compilers.
- **commons-codec**[3] Allows to encode and decode bytes in different formats, including Base64 and URLs. It supports both memory and streamed representations.
- **commons-io**[4] Provides support for the development of I/O-centric applications by abstracting boilerplate-code into static methods, and by implementing commonly used readers and writers, among other utilities.

Table 5.1 shows the details of the aforementioned libraries. Interfaces were not included in the analysis, due to the fact that they are not woven by AspectJ nor instrumented by ReCrash. The selection of libraries was hindered by the limitations of both AspectJ's weaver and RI's instrumentation mechanics. Listing 5.1 shows one such error. This error highlights the fact that AI is restricted by the applicability of AspectJ itself in the target library.

For the comparison, we recorded the augmented code size for each `.class` file and compared it against the unaugmented code. Appendix A provides the source code of a bash script that augments any library using ReCrash and AspectJ. Since each class contains different elements (e.g., the number of public methods, constructors, initialization blocks) the size increase differs from file to file. The violin graphs in

[2]https://www.antlr.org/
[3]https://commons.apache.org/proper/commons-codec/
[4]https://commons.apache.org/proper/commons-io/

Figures 5.1, 5.2, 5.3, 5.4 and 5.5 represent the distribution of all the `.class` files in each library. The horizontal line represents the mean, and as more values fall in a certain range, the graph's width increases in such range.

Listing 5.1: Error obtained on the log4j weaving process

```
Ran out of memory creating debug info for an error
java.lang.OutOfMemoryError: GC overhead limit exceeded
at org.aspectj.apache.bcel.classfile.LineNumberTable.
   unpack(LineNumberTable.java:119)
at org.aspectj.apache.bcel.classfile.LineNumberTable.
   getLineNumberTable(LineNumberTable.java:167)
at org.aspectj.apache.bcel.generic.MethodGen.<init>(
   MethodGen.java:261)
java.lang.OutOfMemoryError: GC overhead limit exceeded
ABORT

Exception thrown from AspectJ 1.8.2
```

Figure 5.1 compares the target libraries augmented with AI. It shows that the mean remains stable at a size increase of 10x the original size, but the majority of files fall around 5x.

Figure 5.2 compares the target libraries augmented with RI. It shows that the mean remains relatively stable at around 1.2x. However, the distribution is not as stable as AI, i.e. the minimum and maximum sizes vary greatly among the augmented libraries. This is expected, since RI only targets public methods, therefore it introduces code only in some parts of the binaries.

The binary size increase means that more instructions are inserted to the original program. Those instructions are executed in the stack memory area and remain constant throughout the program's life; unlike object instances that are created in the heap. Java's garbage collector is executed only on the heap.

Figure 5.3 puts in perspective the binary size increase percentage in both implementations. It shows how AI introduces a massive amount of code per `.class` file, in comparison with ReCrash's. One of the reasons is that AI targets many more EACB's, which translates into code inserted in more places with respect to ReCrash's. In addition to this, AspectJ introduces several lines of code to support the join point model. Appendix B shows the code of a decompiled constructor after it was woven

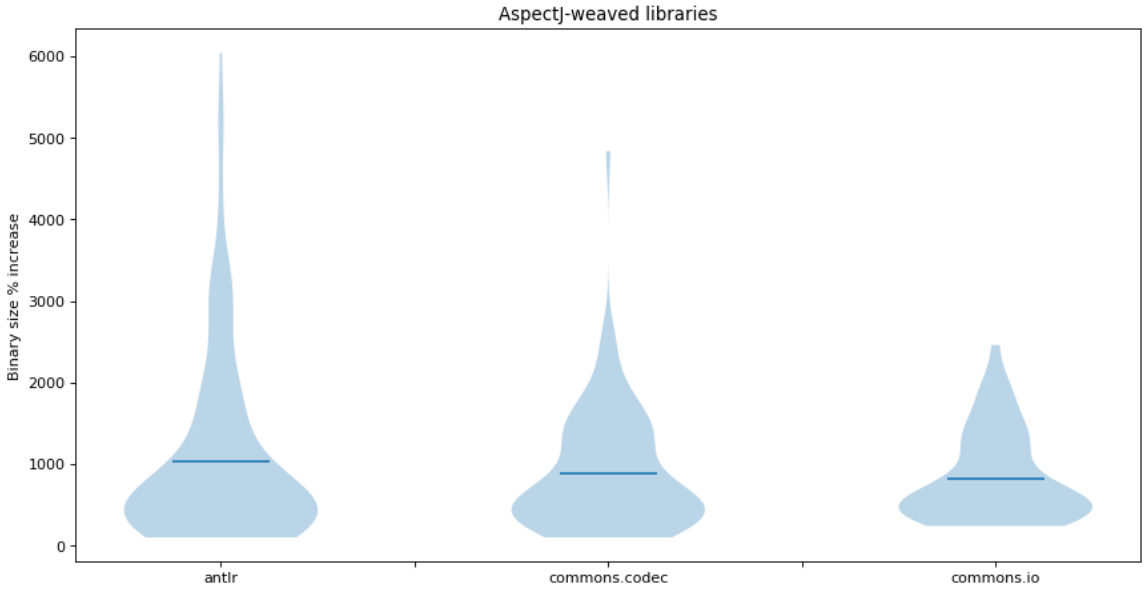

Figure 5.1: Code percentage increase observed in target libraries augmentation using AI

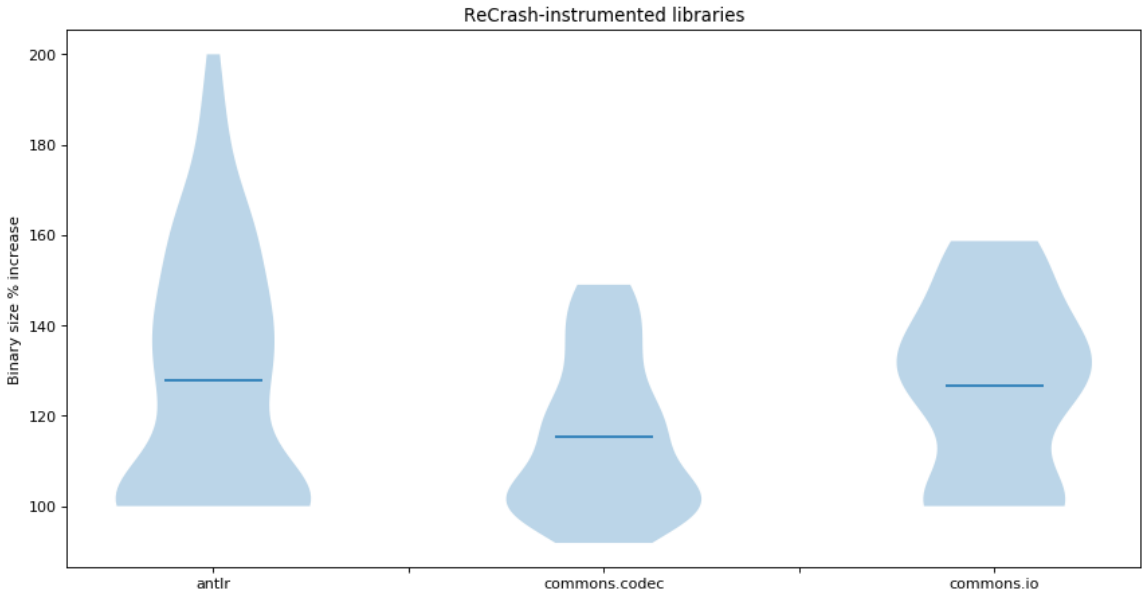

Figure 5.2: Code percentage increase observed in target libraries augmentation using RI

using AI. Most of the code is unrelated to the monitoring logic; it sets up code hooks to be invoked under the conditions defined through the aspect's join points. The additional instructions impact the overall execution time, but are not managed by the garbage collector and therefore are irrelevant to the memory management analysis.

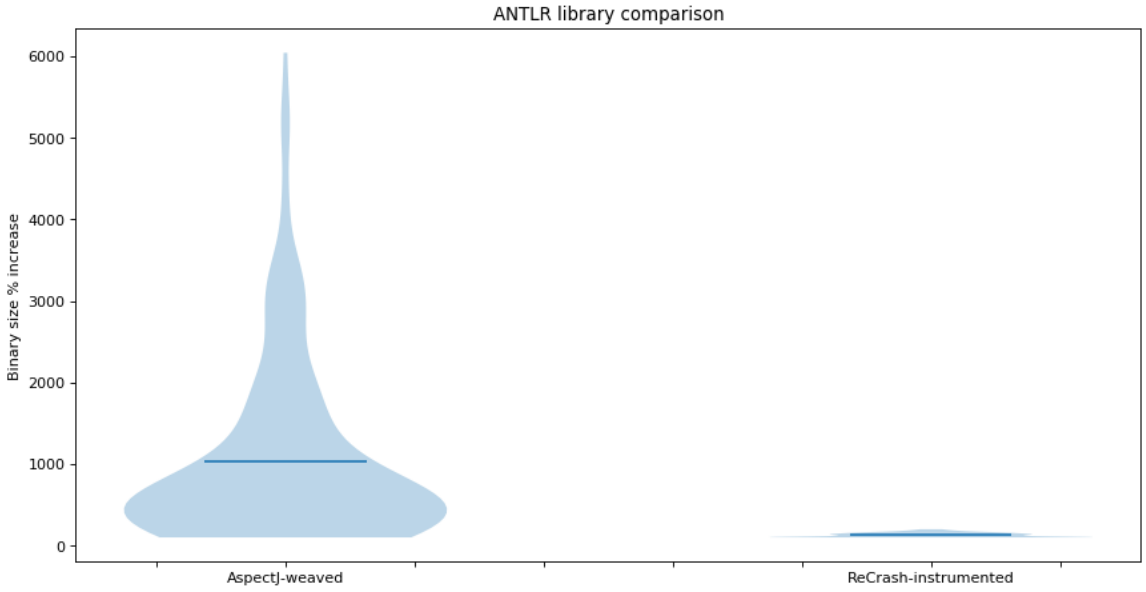

Figure 5.3: Comparison of AI vs RI on the ANTLR library

Figure 5.4 shows the code increase distribution for AI and RI again, but the leftmost figure represents the library weaved without any monitoring logic. By looking at the first and second graph we can conclude that the monitoring logic has little impact on the size increase observed in Figure 5.3, but the overall binary growth is still significant.

Figure 5.5 shows that the binary size increase follows a rational function of the form $y = a/x$ for the three libraries. This is reasonable, considering that denser classes will observe proportionally fewer instrumentation-inserted bytecode instructions.

5.2 Execution time and memory analysis

The second dimension of this analysis corresponds to the time and memory overhead caused by the instrumentation of binaries. This is of interest because it will dictate how feasible it is to use a monitored binary in a productive environment.

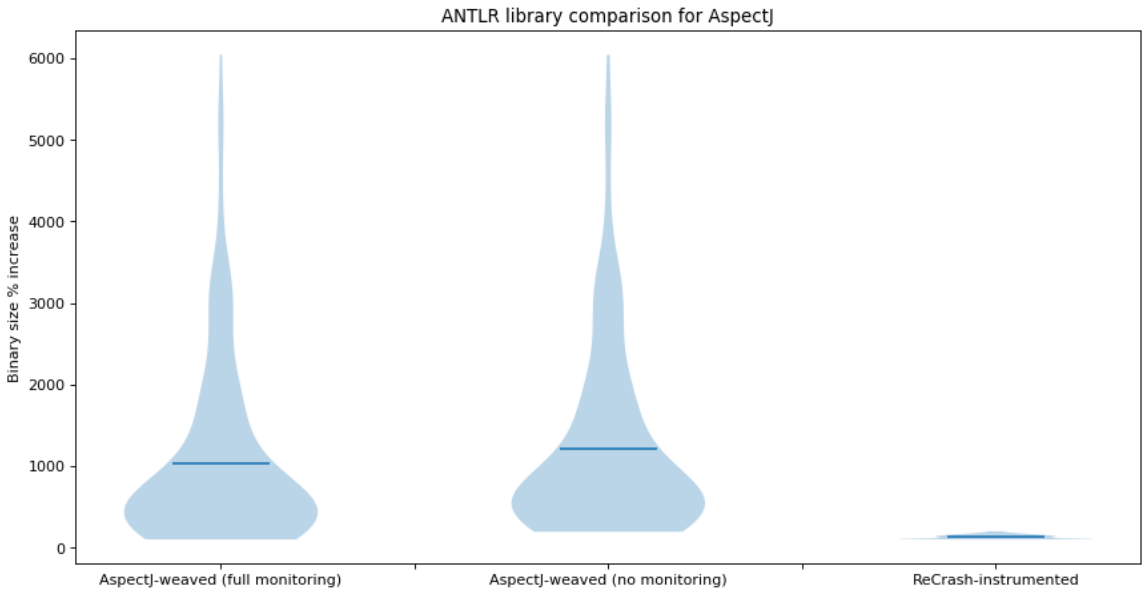

Figure 5.4: Comparison of AspectJ's woven binary with and without monitoring vs RI on the ANTLR library

The Java Microbenchmarking Harness (JMH) is an Oracle-provided tool that allows us to use Java annotations to configure programmatically benchmarking tests on individual methods. The analysis described in this section takes advantage of this tool to benchmark a single method, using both the unmodified library and the AspectJ-weaved and ReCrash-instrumented ones. The method selected is `org.apache.commons.lang3.ArrayUtils.addAll(...)` from the library `commons-lang` obtained from defects4J, because its implementation is non trivial, and it allows us to test a method that involves nested EACBs. Table 5.2 summarizes the environment under which we executed the benchmarking, and Table 5.3 shows the results of the execution. The remainder of this chapter discusses the results obtained and suggests the conditions under which the AspectJ monitoring can be used.

For this analysis, we configured JMH to start with five warm-up iterations. The purpose of these iterations is to prepare the JVM so that the Just In Time (JIT) compiler can perform optimizations on the code before recording metrics. After the warm-up phase completes, JMH runs 20 iterations of 10 seconds each and records the data shown in Table 5.3 for the current iteration. Finally, it averages the partial results and summarizes the execution data obtained.

Firstly, the analysis had to be performed on 20 iterations because larger values

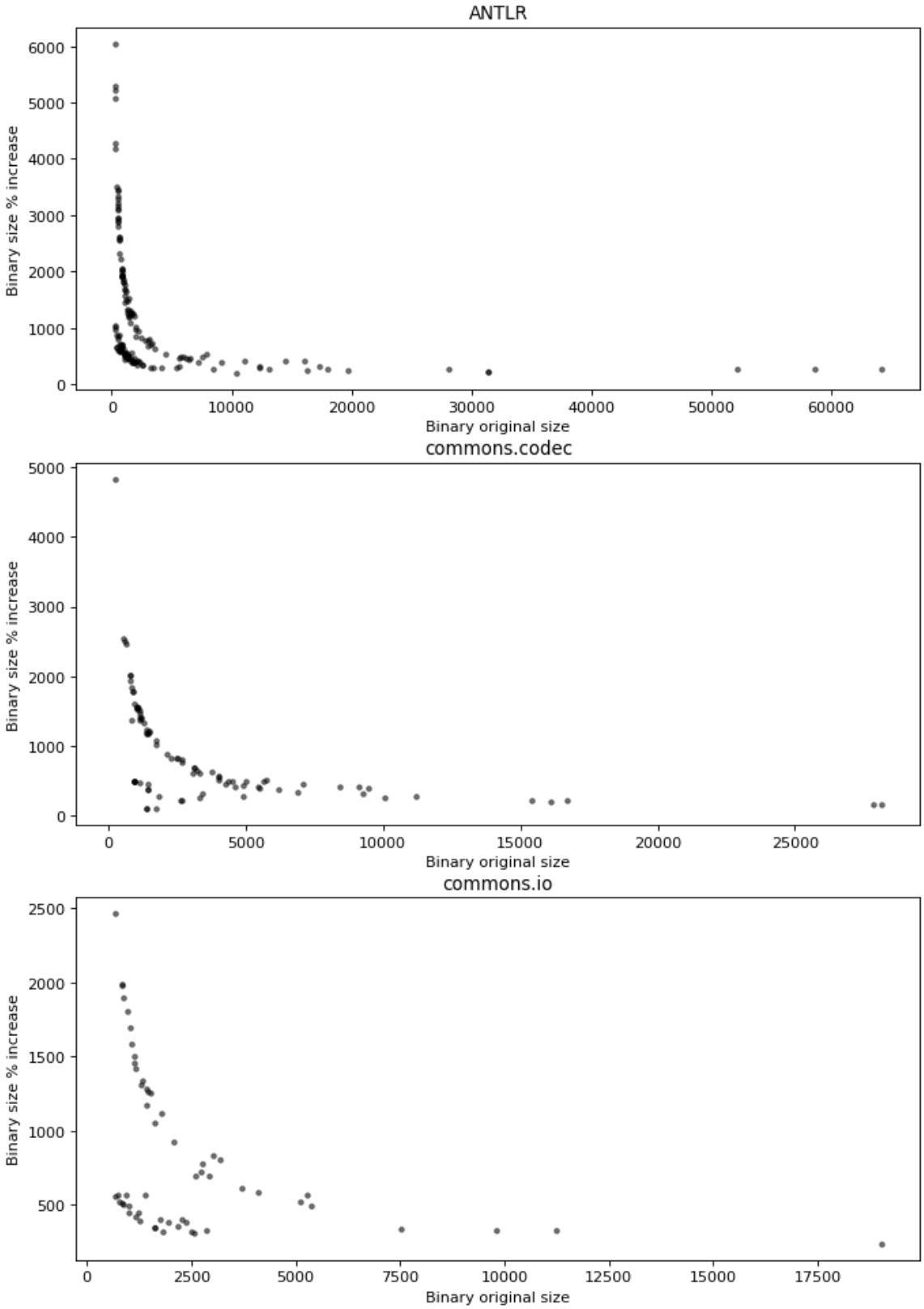

Figure 5.5: Binary growth with respect to the original file size

Table 5.2: Details of the environment used for benchmarking

Characteristic	**Value**
OS Name	Microsoft Windows 10 Home
Version	10.0.17763 Build 17763
Processor	Intel(R) Core(TM) i7-7700HQ CPU at 2.80GHz
Cores	4
Logical processors	8
RAM memory	16 GB

Table 5.3: Details of the performance metrics obtained using the JMH

Metric	**Values observed**			**Unit**
	Original	AspectJ	ReCrash	
Execution	29,671,112	387,985	7,387,031	operations per second
Allocated memory	453	432	1,277	MB per second
Allocated memory (Eden)	437	328	1221	MB per second
Allocated memory (Old Gen)	N/A	$\approx 10^{-5}$	N/A	MB per second
Allocated memory (Survivor)	0.05	2.98	0.09	MB per second
GC runs	247	28	240	execution count
GC execution time	171	92,124	175	milliseconds

resulted in high garbage collection times for the AI and iterations fail due to timeouts, set for 10 minutes per iteration. Table 5.3 demonstrates this tendency. The GC execution time for AI is two orders of magnitude greater than the other two. In addition, the number of GC executions in AspectJ is about one-tenth of the other two. We can also see that it is the only case where the Old Gen memory area had significant enough use to be reported and the Survivor area is a hundred times greater, which suggests a large amount of objects created and then available for garbage collection shortly after. We can confirm this because AI creates at a minimum of

seven new objects for every object instantiation. Then, it creates even more objects for arguments of method calls and constructors, as well as for instance variable usages. The latter creates a strain on the GC as memory gets used and released in greater chunks. It is also worth noting that ReCrash allocates three times more memory than the unaugmented library and AspectJ. This is because ReCrash maintains its stack updated with the actual object instances, instead of a representation. This allows ReCrash to reproduce the context of the program at an arbitrary point, but it also keeps references that would otherwise be released earlier.

5.3 AspectJ limitations for effective program monitoring

This section summarizes the limitations observed during the implementation and evaluation of this thesis's proof-of-concept.

First and most remarkably, Sections 5.1 and 5.2 show that the performance cost of using AspectJ for widespread monitoring is too high to be used in most applications. Notably, it creates a strain on the eden heap area, depleting it constantly and forcing several stop-the-world pauses, which hinder the normal execution of the program. Moreover, AspectJ increases the density of EACBs on average 10 times, which translates to slower executions. It is worth mentioning that AspectJ is a general-purpose AOP implementation for bytecode modification. The flexibility it provides is reflected in the amount of code automatically inserted to provide the higher-level features of the API.

Second, Section 5.1 showed one type of error obtained in the weaving process of an open-source library. We frequently found such errors while weaving non-trivial libraries with this thesis's aspects. Seven out of ten libraries presented similar errors, which discarded them as candidates for the side-to-side comparison. This is a potential problem, as it was not clear what caused the weaving process to fail and if the library cannot be weaved then it cannot be monitored.

Finally, this thesis's implementation revealed that the limitation of AspectJ to use parameters as join points resulted in an overcomplicated logic to identify and process each one inside the corresponding advice. This is an inherited limitation of BCEL, the bytecode library used by AspectJ. The lack of this feature prevents the implementation from distinguishing instances of the marker interface from basic or irrelevant

types, which then pushes the logic to run time instead of compile time. While the performance penalty of this condition may be irrelevant compared to the aforementioned limitations, it is a feature that would simplify the separation of concerns for parameters of methods and constructors.

5.4 Implementation differences

This section discusses additional details discovered during the analysis of the augmentation libraries discussed Sections 5.1 and 5.2 that are unrelated to the particular topics, but are relevant for the overall discussion on how this prototype's generated tests compare to those of ReCrash.

5.4.1 Test usability and EACB coverage

ReCrash-generated unit tests follow a common pattern. Listing 5.2 is an actual test case that follows this pattern, and can be summarized by the following steps:

1. Load a stack file during the `setup` method. This file should exist locally in the filesystem of the system executing the test. Shown in line 3.
2. Add one test method for each called method in the execution graph. Such as the one in line 6.
3. Load statically the object to be invoked. This contains the partial execution graph to reproduce the error. Applied in line 8.
4. Read the method's arguments following the same process mentioned in the previous step. Shown in lines 10 to 15;
5. Execute the actual method. Called in line 18.

Listing 5.2: Pattern followed by ReCrash to create tests

```
1 public class Recrash_reCrash5742130183052929655_trace_gz
    extends TestCase {
2   public void setUp() throws Exception {
3     TraceReader.readTrace("C:\\Users\\Giovanni\\AppData\\
        Local\\Temp\\reCrash5742130183052929655.trace.gz");
```

```
4    }
5
6    public void test_ca_uvic_rigi_aop_Internal2_call2_3()
        throws Throwable {
7      TraceReader.setMethodTraceItem(3);
8      ca.uvic.rigi.aop.Internal2 thisObject  = (ca.uvic.rigi
          .aop.Internal2) TraceReader.readObject(0);
9
10     // load arguments
11     // String arg_1 = empty;
12     String arg_1 = (String)TraceReader.readObject(1);
13     // ca.uvic.rigi.aop.HashCoder arg_2 = ca.uvic.rigi.aop
          .External@4f933fd1;
14     ca.uvic.rigi.aop.HashCoder arg_2 = (ca.uvic.rigi.aop.
          HashCoder)TraceReader.readObject(2);
15     int arg_3 = 99;
16
17     // Method invocation
18     thisObject.call2(arg_1, arg_2, arg_3);
19   }
20 }
```

Although the local stack file is convenient to manipulate tests and their values, it also imposes an external dependency that can potentially render the test unusable if the file is unavailable or corrupted. This dependency makes the test brittle and coupled to a resource outside of the system executing the test. AI makes each test method self-sufficient. It explicitly includes each expectation within the `@Before` method of the class. Then, the test method includes only a call that will execute the EACB that needs to be reproduced because it is suspected of failing. Listing 5.3 shows an example of this. Some differences with respect to RI are:

1. AI creates one test class per EACB while RI uses the same class for multiple related EACBs. While both approaches have advantages and disadvantages, having a single class per EACB limits the number of dependencies to only those effectively used, helping to identify clearly which interactions occurred during the monitoring phase.

2. Lines 4 to 17 of Listing 5.3 define the components that will be used during the test. This allows us to explicitly declare the direct dependencies of the EACB being reproduced. As mentioned previously, this allows direct identification of a EACB's dependencies. ReCrash does not have this in the test case, it is stored in the stack file.

3. Lines 19 to 34 of Listing 5.3 wire the expectations to the dependencies of the EACB. Similarly, this provides a thorough detail of the sequence of methods called and results obtained that ReCrash lacks.

4. Finally, Lines 36 to 39 of Listing 5.3 runs the EACB. ReCrash only supports public method invocations. In addition, object creation and class load for static methods are supported by AI.

Listing 5.3: Pattern followed by AI to create tests

```
1  @RunWith(PowerMockRunner.class)
2  @PrepareForTest({ Internal2.class, External.class })
3  public class ca_uvic_rigi_aop_Internal2__call2__Test {
4    @InjectMocks
5    private Internal2 target;
6
7    /**
8     * Local instance
9     */
10   @Mock
11   private External _External43;
12
13   /**
14    * Local instance
15    */
16   @Mock
17   private Alternate _Alternate66;
18
19   @Before
20   public void setUp() throws Exception {
```

```
21      PowerMockito.whenNew(External.class).withNoArguments()
           .thenReturn(_External43);
22      OngoingStubbing _External43Stub = PowerMockito.when(
           _External43.myHashCode()).thenReturn(1333938290);
23      _External43Stub.thenReturn(1468192631);
24      PowerMockito.mockStatic(External.class);
25      OngoingStubbing _External58Stub = PowerMockito.when(
           External.getSomeValue()).thenReturn(-13218909);
26      _External58Stub.thenReturn(-773065992);
27      _External58Stub.thenReturn(-1027996648);
28      PowerMockito.whenNew(Alternate.class).withNoArguments
           ().thenReturn(_Alternate66);
29      OngoingStubbing _Alternate66Stub = PowerMockito.when(
           _Alternate66.internalCall()).thenReturn("abc
           -782342615");
30      _Alternate66Stub.thenReturn("abc-633274710");
31      _Alternate66Stub.thenReturn("abc1831188303");
32      _Alternate66Stub.thenReturn("abc2067015901");
33      _Alternate66Stub.thenReturn("abc-306537181");
34    }
35
36    @Test
37    public void call2_test() {
38      target.call2("empty", _External43, 99);
39    }
40  }
```

5.4.2 Limitations on environment-dependent interactions

One of the motivations behind this thesis is to reproduce code blocks whose response depended on a particular state of the environment. This is something that differentiates AI from RI because the latter is unable to reproduce such invocations when the target instance is created on-demand. This is due to its stack definition, which does not store objects created within an EACB. Let us consider the code in Listing 5.4.

Listing 5.4: Environment-dependent EACB

```
public int getSomeValue() {
  return new Random().nextInt();
}
```

The `Random` instance created will not be tracked by ReCrash. Instead, it will create an instance of the containing class which in turn will create a new instance of the `Random` class on every invocation. This example, though elementary, highlights a serious limitation of ReCrash: it cannot reproduce instances created on-demand within the EACB. AI overcomes this limitation by recording every class instantiation and associating it with the previous boundary. By doing this, when the method call occurs, it effectively creates the link between the new instance created within the EACB and the method call.

5.4.3 EACB coverage differences

Each EACB is a possible point of failure in a program. As such, it becomes relevant to understand which EACBs are covered by each approach. This understanding is crucial to decide on the monitoring approach for a particular use case. Table 5.4 summarizes the support of each implementation for each EACB.

Table 5.4: EACB comparison between RI and AI

EACB	**RI**	**AI**
Constructors (public, protected, default)	No	Yes
Initialization blocks (instance, static)	No	Yes
Public methods	Yes	Yes
Protected, default access methods	No	Yes

Furthermore, AI retains context data and writes an isolated test on instances that are created within EACBs. RI does not cover this scenario, and it is unreliable if the error condition is randomized in interactions with the local instance.

5.5 Chapter summary

This chapter compared this thesis's prototype with ReCrash. It analyzed augmented libraries in terms of binary size and execution time increases, and memory management. It concluded that monitoring applications using AspectJ introduced consider-

able overhead, and therefore should be used with caution. Finally, it demonstrated that generating test cases that rely on mock objects to simulate EACB interactions allows us to reproduce the context for calls whose response depended on a specific state of the environment produced from on-demand instances.

Chapter 6

Conclusions

This chapter summarizes by highlighting the motivations that led to this work, presenting the challenges identified in self-healing architectures through program repair, and the contributions of this thesis. Finally, it proposes areas of future work.

6.1 Thesis summary

IBM designed and proposed Autonomic Computing to cope with the increasing complexity of software systems. Autonomic Computing is composed of multiple dimensions, one of which is self-healing systems. Such systems can recover from failure conditions and adapt in such a way that allows them to handle known error conditions to avoid subsequent failures.

This thesis aims to move a step forward at self-healing of programmatic errors by proposing a model of application monitoring and failure condition reproducibility via an automatically-generated test case to fill the gap of automatic program repair tools, which require metrics to determine if a patch successfully adapted the source code. For this purpose, AOP was selected as the paradigm used to define mechanisms that monitor context-relevant parts of a program, identified as EACBs. This work selected AspectJ as the AOP implementation to use in Java applications, given its popularity and maturity.

Chapter 4 defined data structures to capture only sufficient context details to reproduce the exact failure conditions. Then, it identified a set of join points that allow to weave Java binaries for all EACBs in a selected package, and designated advices for each join point, intending to unfold the execution graph and gather appropriate

data that links the progression of EACBs during a specific execution. It proposes an algorithm to generate test cases from the textual representation of the program interactions obtained through monitoring.

During the discussion, Chapter 5 analyzed the aforementioned implementation alongside ReCrash—the only known alternative to the author for automatic test case generation of program failures. The binary-size growth analysis showed that AspectJ's implementation increased the binary size by a factor of 10, while ReCrash only incremented it by around 20%. The execution time exhibited a similar result, where ReCrash reduced only slightly the number of operations per second, while AspectJ's implementation decreased it by two orders of magnitude. The memory analysis revealed that the AspectJ implementation created at least seven new objects per instantiation, generating slow garbage collection cycles. ReCrash on the other hand, keeps references to the existing objects which increased dramatically the occupation of the Eden heap space. It outlined the major strengths of AspectJ's implementation over ReCrash's, which can be summarized as a better coverage of EACBs, as well as more atomic, self-sufficient, explicit test cases.

6.2 Contributions

This section reflects on the research questions that drove this thesis and links them to the contributions that answer them.

RQ1 Which Aspect Oriented Programming (AOP) elements can be used to monitor a program and reproduce its state in a test case for programmatic errors?

C1 Chapter 2 introduces relevant AOP concepts—namely join points, pointcuts, and advices—that are used in Chapter 4 to effectively monitor an application. Then, it shows how these concepts can be used to construct a test case that replicates the conditions under which a program failed.

RQ2 What are the limitations of AspectJ to effectively and comprehensively monitor Java programs?

C2 Chapter 5 describes the limitations and caveats observed through the construction of the program-monitor using AspectJ.

RQ3 Which data structures can be used to support the construction of an application-context monitor?

C3 Chapter 4 explains the anatomy of the data structures that are used to represent a program's execution context at a given point in time. It justifies how each data structure can be used to replicate a failure context, isolating it from external dependencies.

RQ4 What is the performance cost of monitoring Java applications using AOP and AspectJ compared to a purely instrumentation-based one, such as ReCrash?

C4 An AspectJ-based reference implementation of a monitor and test-generation component. Then, Chapter 5 evaluates the performance—in terms of binary size, execution time, and memory usage—of AspectJ's implementation listed in C2. ReCrash's implementation is evaluated under the same conditions and then a comparison is performed to clarify how both implementations operate.

6.3 Future work

This section proposes areas where the work embodied by this thesis can be further explored.

Classification failure types

The analysis of the implementation discussed throughout this thesis did not explore the feasibility of the AOP model for different failure types. To illustrate this point, let us consider a sporadic condition, such as a temporary network failure. In this case, self-healing is arguably not the best approach to improve the resiliency of the system. A possible area of study could be enumerating and classifying the types of failures that can occur and then evaluating the efficacy of the monitoring model defined in this thesis for each particular case.

Newer features of Java

In recent years, Java has gradually released modern features for the language that are of interest to context monitoring for failure replication. Let us consider default methods,[1] one relatively modern feature introduced in Java 8. In this case, the weaver would have to support bytecode mutation of interfaces to instrument a given default method, but this feature is not explicitly documented in AspectJ's reference

[1] https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

documents. It would be relevant to systematically analyze modern Java features and identify if the model exposed in this thesis is still applicable to those features, as the ability to replicate failures depends on the effective monitoring of EACBs.

Alternative to AspectJ

One of the biggest shortcomings of AspectJ is the number of lines of code that require to augment binaries, which is reflected in the final binary size and the performance hit that the target library receives. Since AspectJ is a multipurpose library, it requires weaving mechanisms that can handle a broad number of use cases. DiSL [27] is a Java bytecode instrumentation library that is lightweight compared to AspectJ while still supporting a similar, expressive syntax. Furthermore, AspectJ is limited in the bytecode areas that can be adviced. DiSL, on the other hand, allows selecting any bytecode area as a join point [28]. This can potentially improve the efficiency of method and constructor advices, since AspectJ does not support to advice each parameter individually, and the metadata provided by the API is not enough to capture the relevant context directly. Furthermore, Javed et al. constructed a tool to translate AspectJ's aspects into DiSL instrumentations [16], which could potentially speed up the conversion process.

Concurrency

The implementation of this thesis intentionally excluded concurrent programs, to focus solely on the construction of a model that capitalized AOP's paradigm to monitor Java applications and reproduce software failures; to measure the caveats of such a model using ReCrash as a reference. This thesis has shown how such a model can reproduce software failures and create test cases that embody the failure-specific context. One of the next steps could be to explore the challenges that concurrent and distributed systems face when the AOP model is applied to them. If the model can effectively gather the context of each concurrent entity, it could greatly aid in the task of diagnosing and recovering from deficiently-implemented synchronization mechanisms.

6.4 Final words

The work in this thesis allowed the author to explore ideas that have prevailed relevant to the time of writing. Self-healing is a concept that, in the author's opinion, should be embraced by the industry. As the author prepares to work in the industry, he is convinced that the knowledge acquired in this thesis will be a priority in all his endeavors.

References

[1] Mousa Al-zawi, Dhiya Al-Jumeily, Abir Hussain, and Azzelarabe Taleb-Bendiab. Autonomic computing: applications of self-healing systems. In *Proceedings of the 4th International Conference on Developments in E-Systems Engineering (DESE 2011)*, pages 381–386. IEEE, 2011. (page 6)

[2] Shay Artzi, Sunghun Kim, and Michael D. Ernst. ReCrash: making software failures reproducible by preserving object states. In *Proceedings of the 22nd European Conference of Object-Oriented Programming (ECOOP 2008)*, pages 542 – 565. Springer, 2008. (page 2, 22)

[3] Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced Java bytecode instrumentation. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ 2007)*, pages 135 – 144. ACM, 2007. (page 23)

[4] Valeria Cardellini, Emiliano Casalicchio, Vincenzo Grassi, Stefano Iannucci, Francesco Lo Presti, and Raffaela Mirandola. MOSES: A framework for QoS driven runtime adaptation of service-oriented systems. *IEEE Transactions on Software Engineering*, 38(5):1138–1159, 2012. (page 7)

[5] Antoine El-Hokayem, Yliès Falcone, and Mohamad Jaber. Modularizing behavioral and architectural crosscutting concerns in formal component-based systems - application to the behavior interaction priority framework. *Journal of Logical and Algebraic Methods in Programming*, 99:143 – 177, 2018. (page 11, 12)

[6] David Flanagan. *Java In A Nutshell, 5th Edition.* O'Reilly Media, Inc., 2005. (page 9)

[7] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Sympo-*

sium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE 2011), page 416–419. ACM, 2011. (page 21)

[8] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Steenkiste Peter. Rainbow : Architecture- Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004. (page 7)

[9] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*, pages 933–944, 2018. (page 21)

[10] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Deep reinforcement learning for syntactic error repair in student programs. In *Proceedings of the 33rd Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence (AAAI 2019)*, pages 930–937, 2019. (page 2, 21)

[11] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. Deepfix: Fixing common C language errors by deep learning. In *Proceedings of the 31st Association for the Advancement of Artificial Intelligence Conference on Artificial Intelligence (AAAI 2016)*, pages 1345–1351, 2016. (page 21)

[12] Jacob A. Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Peter Chin. Learning to repair software vulnerabilities with generative adversarial networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS 2018)*, page 7944–7954. Curran Associates Inc., 2018. (page 21)

[13] Petr Jan Horn. Autonomic computing: IBM's perspective on the state of information technology. IBM, 2001. (page 6)

[14] IBM. *An architectural blueprint for autonomic computing.* 4th edition, 2006. (page 6)

[15] David Janzen and Hossein Saiedian. Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50, 2005. (page 21)

[16] Omar Javed, Yudi Zheng, Andrea Rosà, Haiyang Sun, and Walter Binder. Extended code coverage for AspectJ-based runtime verification tools. In Yliès Falcone and César Sánchez, editors, *Runtime Verification*, pages 219–234. Springer, 2016. (page 65)

[17] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*, pages 298–309, 2018. (page 2, 21)

[18] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011. (page 20)

[19] Jeffrey Kephart and David Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. (page 6)

[20] Manju Khari, Prabhat Kumar, Daniel Burgos, and Rubén González Crespo. Optimized test suites for automated testing using different optimization techniques. *Soft Computing*, 22(24):8341–8352, 2018. (page 17)

[21] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 49–58, 2005. (page 13)

[22] Ramnivas Laddad. *AspectJ in action*. Manning Publications, 2nd edition, 2009. (page x, 11, 14)

[23] Jeff Langr, Andrew Hunt, and David Thomas. *Pragmatic Unit Testing in Java 8 with JUnit*. Pragmatic Bookshelf, 2nd edition, 2015. (page 17)

[24] Xuan Bach D. Le, David Lo, and Claire Le Goues. History driven program repair. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)*, pages 213–224. IEEE, 2016. (page 21)

[25] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE 2015)*, pages 166–178, 2015. (page 21, 24)

[26] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, number 1, pages 298–312, 2016. (page 2, 21, 24)

[27] Lukáa Marek, Yudi Zheng, Danilo Ansaloni, Lubomír Bulej, Aibek Sarimbekov, Walter Binder, and Zhengwei Qi. Introduction to dynamic program analysis with DiSL. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering (ICPE 2013)*, pages 429–430, 2013. (page 65)

[28] Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Aibek Sarimbekov, Walter Binder, Petr Tůma, and Zhengwei Qi. Java bytecode instrumentation made easy: The DiSL framework for dynamic program analysis. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, pages 256–263. Springer, 2012. (page 65)

[29] Daniel Menasce, Hassan Gomaa, Sam Malek, and Joao Sousa. SASSY: A framework for self-architecting service-oriented systems. *IEEE Software*, 28(6):78–85, 2011. (page 7)

[30] Shaikh Mostafa and Xiaoyin Wang. An empirical study on the usage of mocking frameworks in software testing. In *Proceedings of the 14th International Conference on Quality Software (QSIC 2014)*, pages 127–132, 2014. (page 18)

[31] Thomas Mullen. Writing code for other people: Cognitive psychology and the fundamentals of good software design principles. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, (OOPSLA 2009), page 481–492, 2009. (page 17)

[32] Hausi Müller, Holger Kienle, and Ulrike Stege. Autonomic computing now you see it, now you don't. In Andrea De Lucia and Filomena Ferrucci, editors, *Software Engineering*, pages 32–54. Springer, 2009. (page 6)

[33] Sangeeta Neti. *Quality criteria and an analysis framework for self-healing systems.* Master thesis, University of Victoria, 2007. (page 6)

[34] Scott Oaks. *Java Performance: The Definitive Guide.* O'Reilly Media, Inc., 1st edition, 2014. (page 20)

[35] Carlos Pacheco and Michael Ernst. Randoop: Feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA 2007)*, page 815–816, 2007. (page 21)

[36] Martin Rinard and Fan Long. Staged Program Repair in SPR. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE 2015)*, pages 166–178, 2015. (page 2, 21)

[37] Eddie Antonio Santos, Joshua Charles Campbell, Dhvani Patel, Abram Hindle, and Jose Nelson Amaral. Syntax and sensibility: Using language models to detect and correct syntax errors. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2018)*, pages 311–322, 2018. (page 21)

[38] Taeksu Kim, Chanjin Park, and Chisu Wu. Mock object models for test driven development. In *Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications (SERA 2006)*, pages 221–228, 2006. (page 18)

[39] Gabriel Tamura, Norha Villegas, Hausi Müller, Laurence Duchien, and Lionel Seinturier. Improving context-awareness in self-adaptation using the DYNAMICO reference model. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2013)*, pages 153–162, 2013. (page 8)

[40] Warren Teitelman. *PILOT: A Step Toward Man-Computer Symbiosis*. Doctoral Dissertation, Massachusetts Institute of Technology, 1966. (page 12)

[41] Alex Villazón, Walter Binder, and Danilo Ansaloni. MAJOR: An aspect weaver with full coverage support. *Investigación & Desarrollo*, 1(11):122–136, 2011. (page 23, 35)

[42] Norha Villegas, Gabriel Tamura, Hausi Müller, Laurence Duchien, and Rubby Casallas. DYNAMICO: A reference model for governing control objectives and context relevance in self-adaptive software systems. In Rogério de Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, pages 265–293. Springer, 2013. (page 7)

[43] Juho Vuori. A component-based architecture for self-healing systems, 2007. `https://www.cs.helsinki.fi/u/niklande/opetus/SemK07/paper/vuori.ps`. (page 8)

[44] Ke Wang, Rishabh Singh, and Zhendong Su. Dynamic neural program embedding for program repair. In *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, pages 1–11, 2018. (page 21)

Appendix A

AspectJ vs ReCrash script source code

The following bash code processes all `.jar` files in a directory and creates the directories `original` (unmodified code), `weaved` (AspectJ-instrumented code) and `recrashed` (ReCrash-instrumented code). Then, extracts each library to allow a file-by-file comparison.

```
#/bin/bash -e

base_dir=#the directoy where the source JAR's are to be
   read from
recrashed_dir="$base_dir/recrashed"
weaved_dir="$base_dir/weaved"
original_dir="$base_dir/original"
aspects_jar=#the library with the aspects to be woven
recrash_jar=#ReCrash's JAR

rm -rf "$original_dir"
rm -rf "$recrashed_dir"
rm -rf "$weaved_dir"
mkdir -p "$original_dir"
mkdir -p "$recrashed_dir"
mkdir -p "$weaved_dir"
```

```
cd "$base_dir"
for f in /mnt/hgfs/Thesis/libraries/*.jar; do
  original_file="$(basename -- $f .jar)"
  recrashed_file="$original_file-recrashed.jar"
  weaved_file="$original_file-weaved.jar"

  echo "$original_file"
  unzip "$original_file.jar" -d "$original_dir/
     $original_file"

  echo "$weaved_file"
  ajc -inpath "$f" -aspectpath "$aspects_jar" -outjar "
     $weaved_file" -XnotReweavable
  mv "$weaved_file" "$weaved_dir"
  unzip "$weaved_dir/$weaved_file" -d "$weaved_dir/
     $original_file"

  echo "$recrashed_file"
  java -jar "$recrash_jar" "$f" "$recrashed_file"
  mv "$recrashed_file" "$recrashed_dir"
  unzip "$recrashed_dir/$recrashed_file" -d "
     $recrashed_dir/$original_file"
done
```

Appendix B

AspectJ-woven empty constructor

Listing B.1 shows the empty constructor that was woven using AspectJ, while Listing B.2 shows the result of weaving the empty constructor. Note that there are places where the weaver inserted a try/catch block for an empty code block, which is wasteful.

Listing B.1: Empty constructor to be woven

```
public CompareToBuilder() {
}
```

Listing B.2: The result of weaving an empty constructor using AspectJ

```
public CompareToBuilder() {
  try {
    BoundariesAspect.aspectOf().ajc$before$com_aop_lib_BoundariesAspect$6$8bdb7eee(
       this, ajc$tjp_61);
```

```
try {
  try {
    BoundariesAspect.aspectOf().
       ajc$before$com_aop_lib_BoundariesAspect$6$8bdb7eee(this, ajc$tjp_1);

    try {
      ;
    } catch (Throwable var14) {
      CallTrackerAspect.aspectOf().
         ajc$afterThrowing$com_aop_lib_CallTrackerAspect$8$142837e9(this, var14);
      throw var14;
    }
  } catch (Throwable var15) {
    BoundariesAspect.aspectOf().
       ajc$after$com_aop_lib_BoundariesAspect$7$8bdb7eee(this);
    throw var15;
  }

  BoundariesAspect.aspectOf().ajc$after$com_aop_lib_BoundariesAspect$7$8bdb7eee(
     this);
  CallTrackerAspect.
     ajc$interFieldInit$com_aop_lib_CallTrackerAspect$com_aop_lib_CallTrackerMarker$objectUses
     (this);
```

```
CallTrackerAspect.
   ajc$interFieldInit$com_aop_lib_CallTrackerAspect$com_aop_lib_CallTrackerMarker$currentBou
   (this);
CallTrackerAspect.
   ajc$interFieldInit$com_aop_lib_CallTrackerAspect$com_aop_lib_CallTrackerMarker$boundaryOb
   (this);
CallTrackerAspect.
   ajc$interFieldInit$com_aop_lib_CallTrackerAspect$com_aop_lib_CallTrackerMarker$fieldObjec
   (this);
CallTrackerAspect.
   ajc$interFieldInit$com_aop_lib_CallTrackerAspect$com_aop_lib_CallTrackerMarker$staticVarO
   (this);
CallTrackerAspect.
   ajc$interFieldInit$com_aop_lib_CallTrackerAspect$com_aop_lib_CallTrackerMarker$instanceOb
   (this);
CallTrackerAspect.
   ajc$interFieldInit$com_aop_lib_CallTrackerAspect$com_aop_lib_CallTrackerMarker$argObjectU
   (this);
CallTrackerAspect.
   ajc$interFieldInit$com_aop_lib_CallTrackerAspect$com_aop_lib_CallTrackerMarker$returnedOb
   (this);

try {
  ;
```

```
} catch (Throwable var13) {
  CallTrackerAspect.aspectOf().
     ajc$afterThrowing$com_aop_lib_CallTrackerAspect$8$142837e9(this, var13);
  throw var13;
}

try {
  BoundariesAspect.aspectOf().
     ajc$before$com_aop_lib_BoundariesAspect$6$8bdb7eee(this, ajc$tjp_2);
} catch (Throwable var12) {
  BoundariesAspect.aspectOf().
     ajc$after$com_aop_lib_BoundariesAspect$7$8bdb7eee(this);
  throw var12;
}

BoundariesAspect.aspectOf().ajc$after$com_aop_lib_BoundariesAspect$7$8bdb7eee(
   this);
JoinPoint var1 = Factory.makeJP(ajc$tjp_0, this, this);

try {
  BoundariesAspect.aspectOf().
     ajc$before$com_aop_lib_BoundariesAspect$1$557212b1(this);

  try {
```

```
        CallTrackerAspect.aspectOf().
           ajc$before$com_aop_lib_CallTrackerAspect$1$35767e98(this, var1);
          this.comparison = 0;
      } catch (Throwable var10) {
        CallTrackerAspect.aspectOf().
           ajc$afterThrowing$com_aop_lib_CallTrackerAspect$8$142837e9(this, var10);
        throw var10;
      }
    } catch (Throwable var11) {
      BoundariesAspect.aspectOf().
         ajc$after$com_aop_lib_BoundariesAspect$2$35767e98(this);
      throw var11;
    }

    BoundariesAspect.aspectOf().ajc$after$com_aop_lib_BoundariesAspect$2$35767e98(
       this);
  } catch (Throwable var16) {
    CallTrackerAspect.aspectOf().
       ajc$afterThrowing$com_aop_lib_CallTrackerAspect$8$142837e9(this, var16);
    throw var16;
  }
} catch (Throwable var17) {
  BoundariesAspect.aspectOf().ajc$after$com_aop_lib_BoundariesAspect$7$8bdb7eee(
     this);
```

```
    throw var17;
  }

  BoundariesAspect.aspectOf().ajc$after$com_aop_lib_BoundariesAspect$7$8bdb7eee(this
     );
}
```

Appendix C

Time and memory comparison details

Listing ?? shows the source code used to test `org.apache.commons.lang3.ArrayUtils.addAll(...)` of the `commons-lang` library. It is intended to serve as a reference on how to reproduce or test other methods. Listing

Listing C.1: JMH source code to compare the instrumentation modes

```
package benchmark;

import org.apache.commons.lang3.ArrayUtils;
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Fork;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.Scope;
import org.openjdk.jmh.annotations.State;
import org.openjdk.jmh.profile.GCProfiler;
import org.openjdk.jmh.results.format.ResultFormatType;
import org.openjdk.jmh.runner.Runner;
import org.openjdk.jmh.runner.options.OptionsBuilder;

public class JmhTester {

  @Benchmark
```

```
  @Fork(value = 1, warmups = 1)
  @BenchmarkMode({Mode.Throughput, Mode.AverageTime})
  public Number[] init(Params params) {
    return ArrayUtils.addAll(params.first, params.second);
  }

  public static void main(String[] args) throws Exception
     {
    org.openjdk.jmh.runner.options.Options opt = new
       OptionsBuilder()
      .include(JmhTester.class.getSimpleName())
      .shouldDoGC(true)
      .resultFormat(ResultFormatType.JSON)
      .result(System.currentTimeMillis() + ".json")
      .addProfiler(GCProfiler.class)
      .jvmArgsAppend("-Djmh.stack.period=1")
      .warmupIterations(5)
      .measurementIterations(20)
      .forks(1)
      .build();

    new Runner(opt).run();
  }

  @State(Scope.Benchmark)
  public static class Params {
    public Number[] first = new Number[]{Integer.valueOf
       (1)};
    public Long[] second = new Long[]{Long.valueOf(2)};
  }

}
```

Listing C.2: Source code for the org.apache.commons.lang3.ArrayUtils.addAll(...)

method

```
...
public static <T> T[] addAll(T[] array1, T... array2) {
  if (array1 == null) {
    return clone(array2);
  } else if (array2 == null) {
    return clone(array1);
  } else {
    Class<?> type1 = array1.getClass().getComponentType();
    T[] joinedArray = (Object[])((Object[])Array.
       newInstance(type1, array1.length + array2.length));
    System.arraycopy(array1, 0, joinedArray, 0, array1.
       length);

    try {
      System.arraycopy(array2, 0, joinedArray, array1.
         length, array2.length);
      return joinedArray;
    } catch (ArrayStoreException var6) {
      Class<?> type2 = array2.getClass().getComponentType
         ();
      if (!type1.isAssignableFrom(type2)) {
        throw new IllegalArgumentException("Cannot store "
            + type2.getName() + " in an array of " + type1
          .getName(), var6);
      } else {
        throw var6;
      }
    }
  }
}
...
```