Replaying and Isolating Failure-Inducing Program Interactions

Martin Burger

Dissertation zur Erlangung des Grades des Doktors der Ingenierwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes
Saarbrücken, 2011

Summary

A program fails. What now? A programmer must debug the program to fix the problem, by completing two fundamental debugging tasks: first, the programmer has to *reproduce the failure*; second, she has to *find the failure cause*. Both tasks can result in a tedious, long-lasting, and boring work on the one hand, and can be a factor that significantly drives up costs and risks on the other hand. The field of automated debugging aims to ease the search for failure causes.

This work presents JINSI, taking a new twist on automated debugging that aims to combine ease of use with unprecedented effectiveness. Taking a single failing run, we automatically *record* and *minimize* the interactions between objects to the set of calls relevant for the failure. The result is a minimal unit test that *faithfully reproduces* the failure at will: "Out of these 14,628 calls, only 2 are required". In a study of 17 real-life bugs, JINSI reduced the search space to 13.7 % of the dynamic slice or 0.22 % of the source code, with only 1 to 12 calls left to examine—a precision not only significantly above the state of the art, but also at a level at which fault localization ceases to be a problem.

Moreover, by combining delta debugging, event slicing, and dynamic slicing, we are able to automatically compute failure reproductions along cause-effect chains eventually *leading to the defect*—both efficiently and effectively. JINSI provides minimal unit tests with related calls which pinpoints the circumstances under which the failure occurs at different abstraction levels. In that way, the approach discussed in this thesis ensures a high diagnostic quality and enables the programmer to concentrate on automatically selected parts of the program relevant to the failure.

Zusammenfassung

Ein Programm liefert ein falsches Ergebnis. Was nun? Ein Entwickler muss das Programm, mit dem Ziel den eigentlichen Defekt im Programmcode zu beheben, debuggen. Hierzu sind zwei fundamentale Schritte erforderlich: der Entwickler muss den Fehler zunächst *reproduzieren*, und er muss schließlich die genaue *Fehlerursache finden*. Diese Aufgabenstellung kann einerseits in einer mühsamen, langwierigen und auch langweiligen Suche münden, anderseits kann sie einen Faktor darstellen, der die Kosten der Entwicklung erheblich in die Höhe treibt und die damit verbundenen Risiken schwer kalkulierbar macht. Ziel des Gebietes der Automatischen Fehlersuche ist es, jene Suche nach den genauen Ursachen von Programmfehlern zu vereinfachen.

Die vorliegende Dissertation beschreibt JINSI, ein neuartiges Verfahren zur automatischen Fehlersuche, welches Anwenderfreundlichkeit mit beispielloser Leistungsfähigkeit vereint. Anhand eines einzelnen fehlschlagenden Programmlaufes *zeichnen wir Objektinteraktionen auf* und *vereinfachen diese*, bis lediglich jene Interaktionen verbleiben, die für den Fehler relevant sind. Das Ergebnis ist ein minimaler Unittest, der den Fehler *originalgetreu und nach Belieben reproduzieren* kann: "Von den 14.628 Methodenaufrufen werden nur 2 benötigt." In einer Untersuchung von 17 Fehlern aus der Praxis reduzierte JINSI den Suchraum auf 13,7 % des dynamischen Slices bzw. auf 0,22 % des Quelltextes. Dabei blieben 1 bis 12 Interaktionen übrig, die zum Auffinden der Fehlerursache untersucht werden müssen. Diese Präzision übertrifft nicht nur den bis dato letzten Stand der Forschung, sondern ist auch auf einem Niveau, auf dem das Problem der Fehlerursachenbestimmung aufhört zu bestehen.

Durch die Verknüpfung von Delta Debugging, Event Slicing und Dynami-

schem Slicing sind wir in der Lage Fehlerreproduktionen vollautomatisch entlang von Ursache-Wirkungsketten zu bestimmen – vom Symptom bis zur *Ursache des Fehlers*. JINSI bestimmt Unittests mit zusammenhängenden Interaktionen die genau festlegen, unter welchen Umständen ein Fehler auf verschiedenen Abstraktionsebenen innerhalb des Programmes auftritt. Auf diese Weise erreicht das in dieser Arbeit diskutierte Verfahren eine hohe Diagnosequalität und hilft dem Programmierer sich auf die Teile des Programmes zu konzentrieren, die für das Fehlschlagen relevant sind.

Acknowledgments

First and foremost, I thank my adviser Andreas Zeller for his patience and support over the years. A big thank you also goes to Sebastian Hack for being my second examiner.

Throughout the course of my PhD, I have had the pleasure to collaborate with excellent researchers. A big thank you to Alessandro Orso, as the key idea of minimizing object interaction was conceived together with him. Thanks to Christian Lindig who taught me a lot in the early years of my PhD. Thank you to Andrzej Wasylkowski for fruitful discussions and sharing his knowledge—especially at our retreats at Dagstuhl.

A big thank you goes to my colleagues at the chair for Software Engineering, especially to Kevin Streit, Clemens Hammacher, Gordon Fraser, Kim Herzig, and David Schuler who provided valuable feedback on this doctoral thesis.

A special thank you goes to my friend and colleague Valentin Dallmeier, my office mate through most of my PhD time. We have learned a lot together, and—not least—we also had a lot of fun.

Last but not least, I would like to give thanks to my mother for her incredible support, which enabled me to get an excellent education in the first place.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

When a program fails, a developer must *debug* it in order to fix the problem. Debugging consists of two essential steps. The first is *reproducing the failure.* Reproducing is essential because without being able to reproduce the failure, the developer will have trouble diagnosing the problem and eventually demonstrating that it has been fixed. Reproducing failures depends on the knowledge about the circumstances that lead to a failure; if these are little known or hard to recreate, reproducing can be a tough challenge. The second step in debugging is *finding the defect.* For this purpose, one must trace back the cause-effect chain that leads from defect to failure—a *search* across the program state and the program execution to identify the cause of the problem. The search space can easily involve millions of states, each consisting of thousands of variables. This enormous size not only makes debugging tedious, but also *risky,* as one cannot predict when a particular defect will be found.

While debugging often is a tedious, long-lasting, and boring work from the developer's point of view, seen from a different angle, debugging is one thing in particular: it is expensive as it drives up development costs. Furthermore, bugs can result in tremendous cost. There are many studies that support these observations—Zhivich and Cunningham present several examples from diverse critical systems [75]. For instance, the August 2003 blackout in the United States and Canada partially occurred because of a software failure [30]. The estimated costs associated with this blackout were between $7 and $10 billion [63]. Moreover, software bugs can be even deadly. In February 1991, for example, a Patriot system—the US Army's mobile surface-to-air missile defense system—failed to intercept an incoming Iraqi Scud missile in Saudi

Arabia. The software failure, caused by a rounding error, allowed the enemy missile to reach its target, killing 28 American soldiers [53]. These and other examples compiled by Zhivich and Cunningham show that software errors affect many critical systems, and in some cases result in fatal consequences. They argue that, despite of academic work in several disciplines, such as model checking, software verification, static analysis, and automated testing, software is inadequately tested for supporting critical systems reliably—in other words, that software still contains severe bugs that have to be debugged.

A study commissioned by the National Institute of Standards and Technology (NIST) concluded that software errors (usually also referred to as bugs) are so detrimental and so prevalent, that they cost the US economy an estimated 59 billion[1] every year—in spite of the fact that 50 % of software development budgets are spent for testing [61]. While that study was already conducted in 2002, bugs still pose a problem, and the financial—or, even injurious—implications are getting more serious as more and more people and business rely on software-based systems:

Tokyo Stock Exchange crashes. On November 1, 2005, exchange at Tokyo Stock Exchange (TSE) was incapacitated due to a bug in a newly installed transactions system for hours. Because of ongoing problems, many faulty securities orders were placed even during the following weeks. In one case, such an erroneous order caused a considerable financial damage of $350 million. As a consequence, Takuo Tsurushima, chief executive of TSE, and two other senior executives had to resign [29].

German debit card system fails. At the beginning of 2010, large numbers of bank customers[2] had problems withdrawing cash from cash points using EC cards and other debit/credit cards containing EMV chips. The

[1]That is about 0.6 % of the gross domestic product of the United States.

[2]About 20 million EC cards and around 3.5 million credit cards were affected. On the retail side, 200,000 terminals were affected.

> problem was caused by a certain type of chip which contained a software error in the processing of the year 2010. The problem finally was fixed by reconfiguring ATMs and point of sale terminals to work around this software error in the cards [26].

Because of the manifold negative consequences caused by bugs (see Figure 1.2 on page 11 for more examples, which cause image damage at the very least), many researchers have been devising new methods to *prevent* bugs from occurring [48]. These approaches include both technical and organizational improvements; for instance, programming languages like JAVA, which eliminates whole classes of errors [54], and software development methodologies like Extreme Programming, which relies, among other concepts, on early feedback in order to enforce that flaws are detected early [4]. However, as the above examples show, modern software still contains bugs, and we continue to need techniques that help us *debug* software systems.

The field of *automated debugging* aims to ease the search for failure causes. *Statistical debugging* [40, 74] determines features of the execution that correlate with failures, and thus gives hints on where to search first. *Delta debugging* narrows down failure causes in input [72], program state [68], or version histories [71] by means of automated experiments. As of today, all these techniques can substantially reduce the search space, but require *successful executions* to compare against—the higher the number of these executions and the more similar they are, the higher the chance to isolate a failure cause in the difference between failing and passing executions. In contrast, the alternative of *program slicing* [64] requires only the failing execution. It eliminates those parts of the program that could not have contributed to the failure; however, the remaining slice can still contain thousands of locations not all of which are relevant.

This work presents JINSI, taking a new twist on automated debugging that aims to combine ease of use with unprecedented effectiveness. JINSI treats an execution as a series of *object interactions*—e.g., method calls and returns—that eventually produce the failure. JINSI can record and replay these interactions at will, thus addressing the problem of reproducing failures. Taking a single failing run, JINSI *minimizes* its object interactions to the amount required for reproducing the failure, using a combination of delta debugging and slicing on method calls. The result is a *unit test* involving precisely those objects and interactions required to reproduce the failure. The reduction in search space is impressive: In an evaluation of 17 real-life bugs in JAVA programs, JINSI reduces the source code to an average of 13.7 % of the dynamic slice, or 0.22 % of the original source code, with only 1 to 12 calls left to examine. On top of that, the resulting unit test has a high *diagnostic quality,* explaining exactly how the failure came to be.

1.1 Contributions

In this work, we will make the following contributions:

1. We show how to *minimize object interactions* and thus executions, using a combination of delta debugging and slicing. In contrast to the state of the art, our approach
    a) requires only one *single failing run* (and easily integrates statistical approaches in the presence of multiple runs);
    b) has a *high diagnostic quality*, the result being a single unit test with related calls which pinpoints the circumstances under which the failure occurs;
    c) is *fully automatic*, not requiring any selection, annotation, or other interaction with the programmer.
2. We *evaluate* the approach on 17 bugs, demonstrating
    a) the *effectiveness* of the approach, namely a reduction in *debugging search space* to only 0.22 % of the source code, or 1 line out

of 450—a precision not only significantly above the state of the art, but also at a level at which fault localization ceases to be a problem;

b) a reduction to only 13.7 % of the dynamic slice, the relevant benchmark for having only a single failing run;

c) the *scaleability* of the approach, scaling to JAVA programs with 100,000 lines of code;

d) the *generality* of the approach, providing a full diagnosis on 16 out of 17 bugs examined.

JINSI is easy to apply; all it takes is *one single failing execution* of a JAVA program.

1.2 JINSI in a Nutshell

To show in a nutshell how JINSI[3] works, consider Listing 1.1 on page 6, which shows a piece of code that interacts with the JODA TIME library, a replacement for the JAVA date and time classes [17]; the example code, taken from the JODA TIME documentation, illustrates how to create a complex time zone. This code works just well when run in UTC. However, when run west of Greenwich, this code crashes JODA TIME 1.6, the latest release at the time of writing, and produces an exception.

This bug is hard to reproduce, as it depends on the current time zone. And it is hard to search for the defect: The 23 calls result in a trace containing no less than 484,745 executed lines, covering 1,528 out of 26,534 JODA TIME source code lines.

Here is how JINSI helps in both reproducing and simplifying the problem. The key idea of JINSI is sketched in Figure 1.1 on page 7. JINSI *wraps* around

[3]JINSI stands for “JINSI Isolates Noteworthy Software Interactions”. “Jinsi” is also the Swahili word for “method”, which is the most common interaction between objects.

```
DateTimeZone America_Los_Angeles = new DateTimeZoneBuilder()
  .addCutover(-2147483648, 'w', 1, 1, 0, false, 0)
  .setStandardOffset(-28378000)
  .setFixedSavings("LMT", 0)
  .addCutover(1883, 'w', 11, 18, 0, false, 43200000)
  .setStandardOffset(-28800000)
  .addRecurringSavings("PDT", 3600000, 1918,       1919, 'w',  3, ...)
  .addRecurringSavings("PST",       0, 1918,       1919, 'w', 10, ...)
  .addRecurringSavings("PWT", 3600000, 1942,       1942, 'w',  2, ...)
  .addRecurringSavings("PPT", 3600000, 1945,       1945, 'u',  8, ...)
  .addRecurringSavings("PST",       0, 1945,       1945, 'w',  9, ...)
  .addRecurringSavings("PDT", 3600000, 1948,       1948, 'w',  3, ...)
  .addRecurringSavings("PST",       0, 1949,       1949, 'w',  1, ...)
  .addRecurringSavings("PDT", 3600000, 1950,       1966, 'w',  4, ...)
  .addRecurringSavings("PST",       0, 1950,       1961, 'w',  9, ...)
  .addRecurringSavings("PST",       0, 1962,       1966, 'w', 10, ...)
  .addRecurringSavings("PST",       0, 1967, 2147483647, 'w', 10, ...)
  .addRecurringSavings("PDT", 3600000, 1967,       1973, 'w',  4, ...)
  .addRecurringSavings("PDT", 3600000, 1974,       1974, 'w',  1, ...)
  .addRecurringSavings("PDT", 3600000, 1975,       1975, 'w',  2, ...)
  .addRecurringSavings("PDT", 3600000, 1976,       1986, 'w',  4, ...)
  .addRecurringSavings("PDT", 3600000, 1987, 2147483647, 'w',  4, ...)
  .toDateTimeZone("America/Los_Angeles");
```

Listing 1.1: **Crashing JODA TIME.** This code, taken from the JODA TIME manual [17], crashes JODA TIME when run in the western hemisphere. Please note that we had to shorten the argument lists due to lack of space.

```
DateTimeZone America_Los_Angeles = new DateTimeZoneBuilder()
  .addRecurringSavings("PDT", 3600000, 1987, 2147483647, 'w',  4, ...)
  .toDateTimeZone("America/Los_Angeles");
```

Listing 1.2: **Minimized reproduction.** When fed with the execution in Listing 1.1, JINSI produces a unit test with just the relevant calls. Of the 23 calls, only three suffice to reproduce the failure.

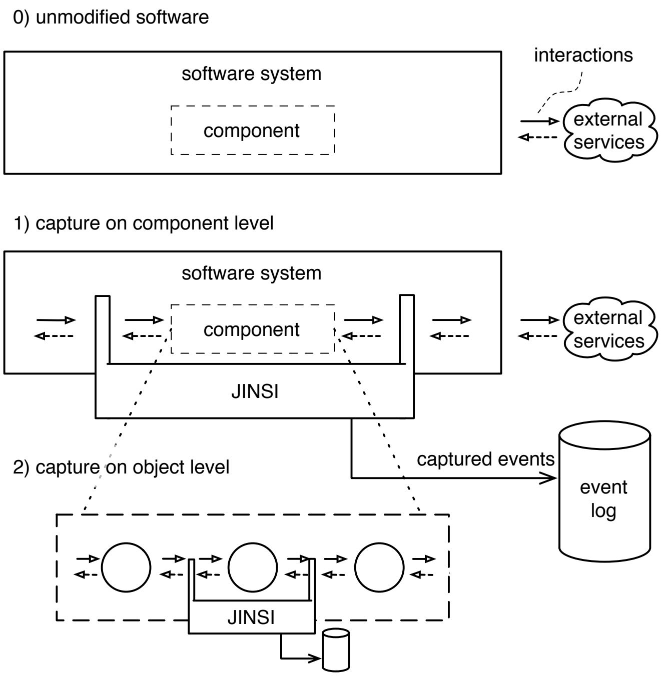


Figure 1.1: **Capturing interactions.** JINSI intercepts and records (1) the interactions between a component and its environment, and (2) all interactions of objects within that component.

a component, for instance the component building time zones, and *records* all interactions of the component with its environment and external services, like the operating system's time zone settings. JINSI then *replays* these captured interactions and thus reproduces the original failure. In the motivating example, JINSI records all constructor and method calls to the time zone builder and replays these interactions to reproduce the problem at will—in any time zone.

We are now able to *reproduce* executions. But which parts of the execution are actually relevant for the failure? Does the problem really depend on the exact sequence of calls on the time zone builder? Do we need these 23 calls altogether? Would it be sufficient to add only one recurring daylight saving time rule, instead of all 16 rules? Such questions can be answered *automatically*—by systematically simplifying the interactions between objects.

The basic idea of JINSI is to apply *delta debugging* to the captured interactions—more precisely, on the incoming method calls—to systematically narrow down the sequence of failure-inducing calls. Out of the 23 calls in Listing 1.1 on page 6, delta debugging would omit one call after another, and repeat execution again and again to check whether the failure persists. In the end, a *minimal subset* remains in which every call would be relevant to reproduce the failure. In our example above, this minimal subset consists of just three calls (Listing 1.2 on page 6): invoking the time zone constructor, adding one single recurring daylight saving time rule, and calling the converter `toDateTimeZone()`.

Applying JINSI to all objects of the failing call stack[4] yields a run that covers only 193 lines instead of 1,528. JINSI determines that out of these 193 lines, only 54 can possibly contribute to the failure. JINSI thus has narrowed down the search space to 54 lines—that is, 3.6 % of the executed lines and 0.2 % of the JODA TIME source.

But even within these 54 lines, not all have the same relevance. Of the three calls in Listing 1.2 on page 6, the first and last one form a *context:* We evidently need both the constructor and the crashing method to have the failure occur. However, the second method `addRecurringSavings()` is set *within* this context. Without it, the test simply passes, which makes it very in-

[4]The call stack can be easily obtained from the thrown exception, as in JAVA an exception contains a snapshot of the execution stack and allows programmatic access to the stack trace information. Figure 4.3 on page 75 shows the exception caused by the bug in JODA TIME.

teresting: Obviously, the problem is related to daylight savings rules—and our investigation would start right within this very method.

1.3 About this Thesis

The remainder of this thesis is structured as follows:

- We will present the state of the art in automated debugging, as applied to the motivating example, in Chapter 2, and discuss each approach's advantages and disadvantages.
- In Chapter 3, we will see how JINSI captures and replays object interactions within a faulty program run to reproduce an observed failure at will, allowing the programmer to investigate the problem in any environment.
- Chapter 4 will describe how JINSI applies different techniques to simplify the problem. These techniques include delta debugging and different slicing techniques. Delta debugging simplifies the recorded interactions until only the failure-inducing sequence remains; slicing can help make simplification both more efficient and effective.
- In Chapter 5, we will see how the JINSI approach can be applied to the bug in the motivating example. In that chapter, we will concentrate on the diagnostic quality of the minimal, failure-inducing sequence of interactions.
- JINSI's ability to automatically select the objects to be observed will be broadened and deepened in Chapter 6. In that way, JINSI is able to compute cause-effect chains that lead to the failure. Furthermore, we will see how JINSI can be extended to non-crashing bugs—which are notoriously difficult to debug.

- While the previous chapters address the general JINSI approach and how it can be applied to any object-oriented program, in Chapter 7 we will see how the approach is specifically implemented for the JAVA programming language, which involves a series of challenges. That chapter will focus on the most important challenges only; in the following Chapter 8, we will discuss more issues and their respective solutions to complete the discussion.

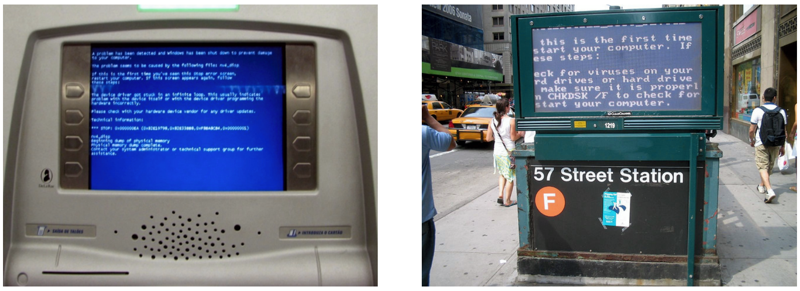
- In Chapter 9, we will investigate how well our approach works in practice by conducting an experimental evaluation on six different JAVA subjects, ranging from relatively small standard subjects to industrial-size projects. As we will see, JINSI is highly effective in reducing the debugging effort.

The thesis will conclude with a summary and ideas for future work in Chapter 10. The remainder of this chapter will define common terms used throughout the thesis, and will finally list publications related to this thesis.

1.4 Terminology

When talking about bugs and software failures, regularly various terms are used: error, defect, flaw, mistake, failure, or fault. Unfortunately, in different contexts these terms have different meanings. To avoid misunderstandings in this work, we will introduce the model of *defects*, *infections*, and *failures*. That widely accepted model for software failures was presented by Zeller in his book on systematic debugging [70]. The model defines how an unexpected or unwanted—or, more formally, incorrect—behavior of a software comes to be.

In the beginning, a programmer introduces a *defect* in the program code. If the program is executed, the incorrect code may cause an *infection*: the actual program state differs from the intended one. Finally, the infection in the program state will cause a *failure* if the effect is an externally observable error. Thus, the defect is the piece of code that is initially causative for the eventually observable failure; lying in between, there are one or more infections. The infection initially caused by the defect usually propagates through the program state, and can cause other infections—for the programmer seemingly unrelated—spread over the program run, which makes finding the root cause tedious and expensive.



(a) Crashed ATM (b) Crashed advertising panel

Figure 1.2: **Blue screen on ATM and advertising panel.** Crashed WINDOWS operating systems displaying a blue screen on publicly accessible systems. This failure not only affects desktop computers, but also systems many people rely on.[5]

For instance, a programmer may declare a condition-controlled loop, where the loop condition is never changed within the loop (the defect). If that loop is executed, the unintended condition will cause an infinite loop, but the pro-

[5]The image on the left (1.2a) was originally posted to Flickr by user hashashin, the one on the right (1.2b) by user thirdrail. Both are licensed under the terms of the Creative Commons license (cc-by-sa-2.0 and cc-by-nc-nd-2.0, respectively).

grammer's intention was to repeat the loop only a few times (the infection). A user who uses the program may observe that the user interface does not respond anymore (the failure), because of the never-ending loop. A well-known example of software failures is the so-called *Blue Screen of Death*, the error screen displayed by the MICROSOFT WINDOWS operating system family upon encountering a critical error that caused the operating system to crash. This failure does not affect desktop systems only, but publicly accessible systems like automated teller machines (ATM) and even electronic advertising panels as well, as shown in Figure 1.2 on page 11.

In the event of a failure, we say that the program *fails*. Furthermore, we speak of a *failing run*, in contrast to a *passing run*. To summarize the above model:

Defect. An incorrect program code.

Infection. An incorrect program state.

Failure. An observable incorrect program behavior.

Throughout this work, we will use the above terms. Furthermore, when we talk about bugs, we usually address defects.

As soon as we encounter a failure, we want to identify the defect in the program code in order to remove the defect. After we will have removed—or, fixed—the defect, the observed failure will no longer occur, and the program will behave correctly. This methodical process of finding and fixing defects in a computer program is called *debugging*. Zeller divides the debugging process into seven steps:[6]

1. **Track the problem.** Enter the issue in a ticket tracking system that manages software problems. This ensures that the defect will not be lost.

[6]Which subsume and refine, respectively, the steps "reproducing the failure" and "finding the defect" mentioned at the very beginning of this chapter.

Most tracking systems define a workflow that automates the lifecycle of the issue.

2. **Reproduce the failure.** Create a test case that reproduces the failure. This helps to verify the fix for the defect and to find regressions.

3. **Automate and simplify.** Concentrate on the relevant circumstances by leaving out the irrelevant. Ideally, use a method that automates this process.

4. **Find infection origins.** Every failure can be traced back to the defect via its infections. Find possible infection origins by going back this trail.

5. **Focus on likely origins.** If you find more than one possible origin, focus on the most likely (for example, so-called anomalies or code smells).

6. **Isolate the infection chain.** By transitively isolating the origins, create an infection chain from the defect to the failure.

7. **Correct the defect.** Remove the defect from the code, and verify the result; for instance, by running the test case created in step number two—that test case must not reproduce the failure anymore.

The subject of this work is mainly related to steps 2 to 7. Locating the defect is the most time consuming activity in the debugging process. Providing a good hint for the location of the defect—or even pinpointing the defect—the whole debugging process could be completed faster. In most cases, the cost of correcting the defect is negligible compared to the preceding steps—exactly those steps, that JINSI helps make much easier, as we will see in the evaluation (Chapter 9).

1.5 Publications

This thesis builds on the following publications (in chronological order):

- Alessandro Orso, Shrinivas Joshi, **Martin Burger**, Andreas Zeller. Isolating Relevant Component Interactions with JINSI. In *WODA '06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, pages 3–10, New York, NY, USA, 2006. ACM.
- **Martin Burger**. JINSI: Isolation fehlerrelevanter Interaktion in Produktivsystemen. In *Proceedings of the 9th Workshop Software Reengineering (WSR 2007)*, Bad Honnef, Germany, 2007. *Proceedings also appeared in Softwaretechnik-Trends (27:2), published by the Gesellschaft für Informatik (GI).*
- **Martin Burger**, Andreas Zeller. Replaying and Isolating Failing Multi-Object Interactions. In *WODA '08: Proceedings of the 2008 International Workshop on Dynamic Analysis*, pages 71–77, New York, NY, USA, 2008. ACM.
- **Martin Burger**, Andreas Zeller. Minimizing Reproduction of Software Failures. To appear in *ISSTA '11: Proceedings of the 2011 International Symposium on Software Testing and Analysis*, Toronto, ON, Canada, 2011. ACM.

2 State of the Art

How does JINSI compare to the state of the art in automated debugging? Using the motivating example in Listing 1.1 on page 6, let us discuss the current state of automated debugging and how JINSI improves on it.

2.1 Statistical Debugging

The basic idea of statistical debugging is to identify features of the program execution that *statistically correlate with failures.* This approach allows us to *focus on anomalies*: the differences between a multitude of faultless program runs and one single—or, several—faulty runs. The passing runs provide properties that are common across all regular runs; by contrast, the failing run specifically provides properties about the abnormal behavior. Having these properties at hand, we can *examine the differences*—differences that usually are likely to hint at the cause. One type of property that can be examined, for instance, is *code coverage* [40]: in the failing run, the defective code must be executed to trigger the failure; moreover, code that is executed in a failing run only is more likely to cause the failure, to *cause abnormal behavior*.[1] Further properties include *function return values* (i.e., erroneous function behavior tied to overall failures) [74], and *branches* (i.e., suspicious evaluation of conditional branches) [46].

[1]Similar approaches include extensions to coverage comparison like the nearest neighbor concept [55] and call sequence differences [22].

To obtain significant results, statistical approaches must consider thousands, or even tens of thousands of executions[2], either obtained from an extensive test suite [40], or sampled in the field [14]—this is in contrast to JINSI, which requires only *one* single failing run. But even given a large number of executions, the results may still be imprecise. The best statistical approach so far, the CP model by Zhang et al. [73] reduces the search space to 5 % or less for 50 % of the test cases examined.[3] While 5 % initially may sound impressive, it still means a huge absolute number of lines to examine. Also, these results are obtained from the so-called *Siemens suite* [36]—a frequently used benchmark in fault localization. The Siemens suite is not only very small and has artificial bugs, it also comes with an extremely extensive test suite, which is the main reason statistical approaches fare relatively well. As it minimizes object interaction, JINSI is not applicable to the (non object-oriented) Siemens suite.

For a simple comparison, we applied the TARANTULA [39] approach to the bug in Listing 1.1 on page 6, using JODA TIME's test suite with 3,496 individual tests. We chose TARANTULA for its simplicity and because it fared only marginally worse than CP [73]. Following down the ranking, the developer has to inspect the 522 most suspicious lines until she finds the defective line, or 2 % of the code. However, this is still almost *ten times* as much as the total 54 lines (0.2 %) identified by JINSI. Even modern approaches combining test case generation with statistical debugging [1] would be challenged by this precision. In addition, the locations returned by statistical approaches are scattered across the code, whereas JINSI's locations are related to each other through the single, minimal unit test.

A further disadvantage of statistical debugging is its runtime performance: applying TARANTULA took several hours. First of all, all 3,496 tests had to

[2]For instance, Zheng et al. assume that "sampled data from thousands of actual user runs" is available [74]; Liblit et al. require up to 21,000 runs to isolate all of the bug predictors [46].

[3]A further closely related work was published by Baah et al. [3]. Their approach reduces the search space, approximately, to a percentage between 3 % and 12 % for 50 % of the test cases examined, depending on whether best or worst case performance is taken into account.

be run individually, each single test run in a fresh instance of the JAVA virtual machine (because of the tool that was used to collect the coverage data); furthermore, more than 3 GB of coverage data had to be collected and analyzed. In contrast, applying JINSI is a matter of minutes (see Section 9.5).

*In contrast to statistical debugging, which must consider thousands, or even tens of thousands of executions, JINSI requires just **one single failing run**.*

2.2 Program Slicing

A further technique to focus on those parts of a program that could have contributed to the failure is *program slicing* [64]. This approach yields a subset of the program execution—called program slice—that is relevant for a specific state or behavior. Slices are based on *dependencies* between statements: A statement S_2 depends on a statement S_1, if S_1 can influence the program state accessed by S_2. Starting from a statement, the transitive closure over all dependencies forms a program slice (see Section 4.2 for details).

In debugging, computing the backward slice for a failing statement returns all statements that could have influenced the failure. An important distinction is made between static and dynamic slicing. While a *static* slice applies to all possible runs, and therefore is computed without making any assumptions about a concrete (failing) program run, a *dynamic* slice just applies to the failing run and thus is more precise.

Binkley and Harman studied static slice sizes in 43 C programs [5]. The average size was smaller than 30 % of the original program. In a basic evaluation on three C programs, Gyimóthy et al. compared static and dynamic slicing [33]. The static slice on average contains 58 % of the statements in the entire program, whereas the dynamic slice only contains 5 %. Thus, as for statistical debugging, for the programmer *the absolute number of suspicious statements may be still too large* to find the actual defect in a timely manner.

Just like JINSI, dynamic slicing requires only one failing run; it is thus the benchmark we compare against. As shown in our evaluation (Chapter 9), a dynamic slice applied to the motivating example contains only 512 suspicious lines out of 26,534 lines in total. Thus, only about 2 % remain, already producing a remarkable improvement for the programmer who has to debug this failure. However, applying JINSI reduces the number of suspicious lines further to only 193. Combining both techniques even results in only 54 lines—an amount that can be easily reviewed one by one. In contrast to the dynamic slice, the run minimized by JINSI is executable. Thus, the programmer can use her familiar debugger to further investigate the bug.

JINSI improves upon program slicing by ***systematic experiments****. Our approach is able to narrow down possible failure causes more precisely, and the minimized run is executable.*

2.3 Delta Debugging

Delta debugging is a technique to systematically narrow down failure causes by means of automated experiments. This method was devised by Zeller and Hildebrandt to simplify failure-inducing input [72]: delta debugging would repeat the failing program run while systematically suppressing parts of the input, eventually coming up with a simplified input of which every remaining part is relevant for producing the failure. In this way, they have been able to minimize an initial 896-line HTML document that caused the Mozilla browser to crash to one single `SELECT` tag (see Section 4.1 for details).

In addition to input, delta debugging was applied to further circumstances, like failure causes in version histories [71]. While generally effective, these techniques are not applicable to the motivating example, as we lack a working older version or a controllable external input; therefore, *hybrid approaches* combining delta debugging and slicing [32] are also excluded.

An interesting alternative could be to apply delta debugging to program states [16], isolating differences in the state and the behavior induced by the time zone change. However, manipulating states in JAVA programs is a daunting task, as the program state is not under direct control by the program. For instance, the JAVA run time system makes it hard to change private object attributes directly; the only unsanctioned method to manipulate objects is to invoke methods. It is therefore unclear whether this approach would be applicable to JAVA programs; on top, it again requires a (hopefully similar) passing run.[4]

Most of the precision of JINSI in fault localization comes from applying delta debugging to object interaction. This was first attempted for *generated* call sequences (i.e., generated test cases): Lei and Andrews [44] were the first to apply delta debugging to calls to minimize generated test cases; Leitner et al. [45] combine this with static slicing to speed up minimization. In contrast to these approaches, JINSI works on *recorded* calls, which is not only applicable to all sorts of failures (instead of only generated ones), but also more challenging: While applying delta debugging to *generated* call sequences simply means to twist the generator, applying it to *recorded* interaction means one will have to care about missing initializations, missing targets, or missing parameters whenever some object interaction is optimized away.

*JINSI is the first technique to apply delta debugging to **captured** method calls as recorded in **real-life** programs with actual bugs.*

2.4 Capture/Replay

A research field innately closely related to our work is *capture/replay*. In general, the approaches and tools in this area enable developers to first capture—or,

[4]That also excludes approaches that highly depend on both manipulating program states and comparing passing and failing states, like the work by Jeffrey et al. [38].

record—program runs, and to later replay these recorded runs *at will* for testing and debugging purposes. Most of the capture/replay techniques that have been presented to date fall into one of the two following categories. They represent different strategies to gather sufficient information in order to be able to replay the previously recorded runs:

Checkpointing. This strategy records information on *program state* at well-defined points in a program run. During replay, the recorded execution can be *re-run starting from these checkpoints*. Approaches using checkpoints to replay include the work by Xu et al. [67], and RECRASH, a tool developed by Artzi et al. [2]. RECRASH, for instance, performs a checkpoint at each method entry and maintains a shadow stack. In this way, the tool allows the developer to observe a run in several states before the actual crash. Because this strategy only captures program runs at certain points in execution, it may miss information that is crucial to reproduce the original failure faithfully, as we will see below.

Event based. In contrast to gathering information about the program state itself, and at certain points in time only, this strategy captures the information required for later replay as a sequence of *interactions that lead to a certain program state*; typically, the state where the failure becomes apparent. In this way, this strategy describes how the faulty state came to be, and thus can reveal how the faulty program state *evolves*. Typically, the captured interactions are written as events to some sort of trace or log file. JINSI captures *object interactions* in the form of events, and in this way is able to show how selected objects are constructed and eventually lead to a faulty state. Steven et al. presented JRAPTURE [60], a tool that uses events in a comparable way. In contrast to JINSI, their tool requires customization of the JAVA API for each version of the JAVA runtime environment and captures complete input/output information. Further approaches that use event based capture/replay include *test factoring* [57] and ADDA [15]. Like JINSI, test factoring captures at the method level, whereas ADDA records events at the level of C standard library and file operation functions.

JINSI had been initially inspired by the SCARPE prototype, presented for the first time by Orso and Kennedy [52], a further event based tool to capture/replay JAVA programs. JINSI uses capture/replay to *generate unit tests* that exercise only selected aspects of the original program run to help focus on relevant information. Similar work was presented in test factoring [57] and test carving [28], the latter of which extracts relevant parts of the *program state* to create unit tests. However, JINSI goes one step further and *simplifies these tests automatically* until only the minimized information remains. Like Leitner at al. [45], JINSI uses a combination of slicing and delta debugging to automatically minimize the sequence of failure-inducing interactions. However, JINSI is designed to isolate failure-inducing interactions in faulty runs of real-life applications, whereas Leitner et al. presented a technique that applies to randomly generated—thus, artificial—unit tests.

The true power of capture/replay comes as it is being combined with diagnostic features. As already mentioned above, tools using checkpoints, however, may miss crucial information and therefore may neither be able to reproduce the failure, nor to deduce a helpful diagnosis. RECRASH, for instance, may not be able to reproduce the crash at the crucial moment—exactly at the moment the fault is initially caused. RECRASH replays crashes starting from the individual frames on a stack trace that is produced when throwing an exception. This is very efficient, but assumes that the stack trace actually contains the code (and state) that caused the failure.

In our motivating example, an `ArithmeticException` is thrown (see Figure 4.3 on page 75). While RECRASH is able to reproduce this crash at the upper stack frames in all time zones, it fails to reproduce the crash starting at the most interesting frame where the failure is initially caused. Exactly there, the crucial time zone is obtained via a static method call. RECRASH serializes the program state at the individual checkpoints as it is reachable via direct object references.[5] However, these do not include state reachable via static method calls only. The reproduced run thus still relies on the system's current time zone and will not reproduce the run east of Greenwich; which renders the approach—in this particular example—unusable. By contrast, JINSI captures and replays *all* calls by design, and thus is able to replay and minimize the original failure.

A further advantage of JINSI over RECRASH is its ability to naturally capture/replay non-crashing bugs. While in this case the developer using RECRASH has to *provide proper checkpoints manually*, JINSI captures the required information without additional assistance.

By design, JINSI captures and replays ***all interactions*** *for an automatically selected set of objects—because any of these interactions may be relevant for the failure.*

2.5 Earlier Versions of JINSI

The current version of JINSI bases on earlier proof-of-concept prototypes [11, 51]. These prototype implementations already applied delta debugging to minimize *incoming* calls to some object, introducing and demonstrating the promise of delta debugging on method calls. However, JINSI previously required the programmer to manually select the object in question, providing a hint on where the fault might be, and to provide a *predicate*, which distinguishes a passing from a failing run; also, it was never demonstrated on more than a single real-life example (`COL-1` in this paper). In contrast, JINSI now is fully

[5]RECRASH provides several strategies to capture state, from copying only references to copying the entire state. Copying the entire state, RECRASH would loose its performance advantage and still may not catch the relevant information as in the motivating example.

automatic, requiring no hints by the programmer, and generates predicates automatically; it also works for non-crashing bugs, where a single initial predicate (typically, a test oracle) is needed to distinguish expected from observed behavior. Furthermore, JINSI's precision is greatly increased by including dynamic slicing both as a filter and a strategy guide. Finally, JINSI is demonstrated on a wide range of real bugs (see Chapter 9 for an evalutation), scaling up to problems of considerable complexity.

*JINSI is **fully automatic** as it both generates predicates and selects the objects to be observed automatically—even for non-crashing bugs.*

3 Reproducing Failures

In the next chapters, we shall walk through the individual steps JINSI takes to minimize failure reproduction. These include capture/replay to reproduce the problem (this very chapter), and applying delta debugging and slicing techniques to simplify the problem (Chapter 4). Figure 3.1 on page 26 shows a schematic overview of the whole process.

The very first task in debugging in general is to *reproduce the problem* in order to examine it, and eventually to check whether a fix is successful. To reproduce a problem, we have to recreate both

1. the **environment** in which the problem occurred, and
2. the **problem's history**; thus, the steps that lead to the problem.

Basically, there are two options to reproduce the *environment:* on-site with the user, or in the developer's laboratory. Usually, the second alternative is quite expensive and has modest prospects of success, while reproducing on-site is comparatively economical and more likely to succeed because the user was already able to (re-)produce the problem at least once. In our example above, it is very easy for the user who is in the right time zone to reproduce the failure, while a developer located in the wrong one would encounter a challenge. In such cases, JINSI can be applied on-site to produce a minimal reproduction of the failure which eventually can be sent to the developer who then is able to reproduce and investigate the original failure in question—even if she or he would not be able to reproduce that failure locally.

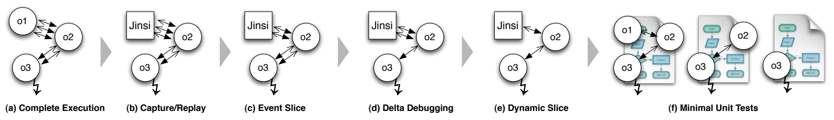


Figure 3.1: **How JINSI works.** JINSI captures and replays (b) a failing execution (a). Event Slices (c), Delta Debugging (d), and Dynamic Slices (e) all minimize the relevant object interaction. Repeating the process while gradually incrementing the set of observed objects, JINSI produces a set of unit tests at varying abstraction levels (f).

Either way, after having reproduced the environment, we have to reproduce the *program execution* (thus, the problem's history), basically by reproducing the *input*—a program's input in general predetermines its outcome[1]. Unfortunately, there are many different types of input: data, user interaction, schedules, and randomness, to name but a few examples. We could apply different techniques on each type, like tracing events on the operating system level, or capturing events in the graphical user interface. However, a better approach would be to find an *abstraction over all types of input*.

JINSI employs such an abstraction that is *inherently defined by the objects* used in an object-oriented program. This kind of abstraction "offers a mechanism that captures a model of the real world" [6]. Thus, as described by Bloch, classes and methods provide a natural abstraction, which further is easily understandable by developers. In a nutshell, JINSI captures and replays on the object level to reproduce the *environment,* and on the method level to reproduce the *problem's history*. In this sense, the JINSI approach in principle can be applied to all object oriented programs.[2]

[1]Even a non-deterministic algorithm uses input in the most general sense: user input, random values, or hardware errors.

[2]However, the current implementation is done for JAVA programs only.

In object-oriented programming, an object combines *data* (commonly called fields) with a set of *methods* for accessing and managing that data. This programming paradigm encourages programmers to place data where it is not directly accessible by the rest of the program; instead, the data is encapsulated within the object and can only be accessed by calling specially written functions, commonly called methods. An object-oriented program typically contains different types of objects, each one representing a particular kind of complex data to be managed. At a certain abstraction level, that complex data usually corresponds to real-world objects or concepts such as a bank account, computer systems, or time zones. Thus, those objects represent the state of particular aspects of the program's environment.

An object-oriented program may thus be viewed as a collection of *interacting* objects: each object is capable of receiving messages (namely, method calls), processing data, and sending messages (e.g., return values of such a method call) to other objects. In this way, different objects represent different aspects of a program's environment, and methods provide a way to manipulate the environment (to be exact, its representation). Obviously, a sequence of object interactions beginning with the initialization of a program, and ending in a program failure is a natural way to represent the problem's history. JINSI exactly takes advantage of these intrinsic properties of object-oriented programs—objects to represent certain aspects of the real world and the program's environment, respectively, and methods to manipulate that representation—to reproduce the problem that occurred within the program.

3.1 Capture

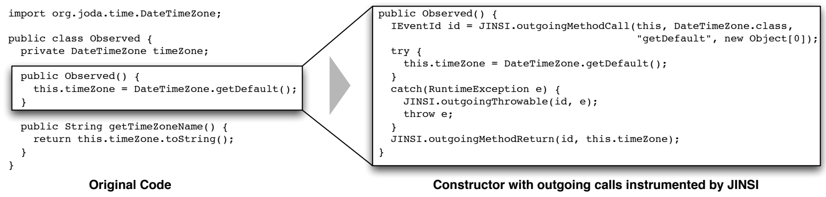
The very first step in reproducing a failure with JINSI is to observe and store (i.e., *capture*) suitable information on relevant interactions on the object level for later replay; thus, for the actual reproduction. In the following, we first outline the capture phase to give a brief overview on the related concepts. Hereinafter, we describe these concepts in greater detail, while staying on a conceptional level—independent of the actual programming language the program to be debugged is implemented in.[3] Chapter 7 finally provides explanations on how JINSI is applied to JAVA programs concretely and describes how JINSI deals with JAVA-specific language features.

JINSI's aim is to capture interactions between a suspicious component (defined by a set of classes, called *observed*) and its environment (called *unobserved*), as well as between all objects within that component (i.e., instances of the component's classes). For this purpose, JINSI's capture/replay technique identifies possible interactions between these objects, correspondingly instruments the program, and captures the relevant interactions at runtime (Figure 1.1 on page 7). In this way, JINSI captures (and replays) only *selected* parts of the program (see Section 3.1.1).

Interactions are captured as different *events:* constructor and method calls, return values, field accesses, and exception flow. For each of these, JINSI distinguishes *incoming events,* originating from the unobserved part, and *outgoing events,* originating from the observed component. Section 3.1.4 describes object interactions and their corresponding events in more detail.

JINSI uses *instrumentation techniques* to alter the program for the purpose of gathering data related to object interactions. Basically, JINSI introduces additional methods calls (so-called *probes*), which collect that data at proper points during the program execution. The concrete instrumentation depends on the event to be captured. For instance, to capture an outgoing method call `OMC` and its return value (including exception flow), JINSI

[3]This chapter also describes some topics that JINSI has to address due to properties specific for the JAVA programming language; essentially, we address exceptions, a primary feature of JAVA. However, we stay on a high level of abstraction, so that the concepts presented here can be adapted to other object-oriented languages as well. Chapter 7 contains more details on lessons learned with regard to the JAVA programming language. This includes handling of arrays: in JAVA, arrays are not implemented as full-fledged objects, but as a kind of special *container object* that holds a fixed number of values of a single type.



```
import org.joda.time.DateTimeZone;

public class Observed {
  private DateTimeZone timeZone;

  public Observed() {
    this.timeZone = DateTimeZone.getDefault();
  }

  public String getTimeZoneName() {
    return this.timeZone.toString();
  }
}
```

```
public Observed() {
  IEventId id = JINSI.outgoingMethodCall(this, DateTimeZone.class,
                                        "getDefault", new Object[0]);
  try {
    this.timeZone = DateTimeZone.getDefault();
  }
  catch(RuntimeException e) {
    JINSI.outgoingThrowable(id, e);
    throw e;
  }
  JINSI.outgoingMethodReturn(id, this.timeZone);
}
```

Figure 3.2: **Capturing outgoing method calls.** JINSI captures outgoing method calls by inserting several probes. In this way, JINSI records all information required for later replay of the call to `DateTimeZone.getDefault()`.

1. precedes OMC with a probe that records the *source* and the *target* of that call, the corresponding *method name,* and its actual *arguments*;

2. surrounds OMC with a try-catch-block (or extends existing blocks if any) to record possible *exceptions*; and

3. adds a probe directly after OMC that records the actual *return value*.

Figure 3.2 on page 29 shows how JINSI instruments an outgoing call which obtains the default time zone similarly as it happens in our motivating example. While capturing, JINSI records two events: one describing the outgoing method call itself, including attributes needed for replaying like the given arguments (their types and unique IDs, see Section 3.1.2); and one representing the returned time zone (or the exception, should one be thrown).

When capturing data, the type of information ranges from simple primitive values to complex objects like an object representing a time zone. While capturing primitive values is easy, gathering comprehensive information about entire composite objects would be much more expensive. To overcome this problem, JINSI records only a bare minimum of information required for replay (see Section 3.1.3 for details). In our approach, we

1. only have to capture those objects that **directly affect** the concrete computation, and

2. **incrementally capture** the information at runtime as soon as it is required.

Besides capturing events, all we require to replay a program run are unique object IDs, their types, and primitive values—no serialization as is involved in RECRASH, for instance. This way, JINSI can capture/replay *arbitrarily complex objects,* and dramatically reduce the space and time costs of the capture phase, as already evaluated by Joshi and Orso, who basically had used the same technique [41].

In our simplified example (Figure 3.2 on page 29), while observing an incoming constructor call to class `Observed`, JINSI captures the outgoing call to method `DateTimeZone.getDefault()` and maps the returned time zone instance to a unique object ID *id*. While observing a subsequent incoming call to method `getTimeZoneName()`, JINSI associates the callee—the time zone referenced by field `timeZone`—to the already known *id*. The returned time zone name is recorded depending on the actual time zone of the system where that code is run. If JINSI captures on-site with the user, *JINSI will capture exactly the time zone that is relevant for the failure*. In this way, JINSI is able to capture both the environment and all the steps that lead to the original failure in question.

In the following, we describe the fundamental concepts for capturing relevant information on object interactions in more detail: observing selected parts of a program (Section 3.1.1), using object IDs to clearly identify single objects (Section 3.1.2), saving the cost for expensive object serialization, or other means to persist program state, by capturing partial information incrementally (Section 3.1.3), using events to describe relevant object interactions

(Section 3.1.4), and eventually the actual capture of these events for future reference during replay (Section 3.1.5).

3.1.1 Selectivity: Observed and Unobserved Objects

One of the main characteristics of JINSI's capture/replay approach is its *selectivity*. Instead of manipulating the whole program and to capture complete executions, JINSI selects a subsystem (or component) of interest and captures all the interactions between this subsystem and the rest of the application. While this fundamental idea was first introduced by Orso and Kenedy [52] in order to (a) provide an efficient capture/replay technique practically applicable in the field, and (b) to overcome privacy issues, JINSI also takes advantage of the approach's selectivity to provide a step-wise diagnosis over *different abstraction levels* (see Chapter 6 for details).

For this purpose, JINSI divides the program to be debugged into two parts: the subsystem of interest—the part JINSI is to capture interactions for—is called *observed,* while the remaining part of the program is called *unobserved*. This name-giving distinction also applies to the respective entities contained in these two parts. The observed part of the program is defined by a given set of *observed classes,* or *observed set* in short. Instances of observed classes are called *observed objects*. *Observed constructors, observed methods*, and *observed fields* are constructors, methods, and fields in observed classes and objects, respectively. Similarly, we define *unobserved classes, unobserved set, unobserved objects, unobserved constructors, unobserved methods,* and *unobserved fields*. Figure 3.3 on page 32 gives a formal definition of these entities.

Figure 3.4 on page 33 illustrates the division of the program into two distinct parts. You can visualize that an imaginary boundary encloses all observed objects, while the unobserved objects are located outside of this boundary. Thus, the interactions of interest are those that either occur within that boundary (strictly between observed objects), or that cross that boundary (thus, either enter or leave the observed part of the program). This also illustrates the selec-

Observed and unobserved classes. Let C be the set of all classes of a program, and let C_o be the set of *observed classes*. Then, the set of *unobserved classes* C_u is defined by $C \setminus C_o$. C_o is also called *observed part* of the program, and C_u is accordingly called *unobserved part* of the program.

Observed and unobserved objects. An instance o_o of an observed class $c_o \in C_o$ is called *observed object,* an instance o_u of an unobserved class $c_u \in C_u$ is called *unobserved object*.

Observed and unobserved members. All members (methods and fields) of object o_o are called *observed members* (*observed methods* and *observed fields*), while members of object o_u are called *unobserved members* (*unobserved methods* and *unobserved fields*). This applies correspondingly to class members (fields and methods that belong to the class, rather than to an instance of the class) of c_o and c_u, respectively. Furthermore, constructors of class c_o are called *observed constructors,* while all other constructors are called *unobserved constructors*.

Figure 3.3: **The definition of observed and unobserved entities in a nutshell.** The property observed (and its *complement* attribute unobserved) divides the program, its classes, all class instances, and their members into two distinct sets.

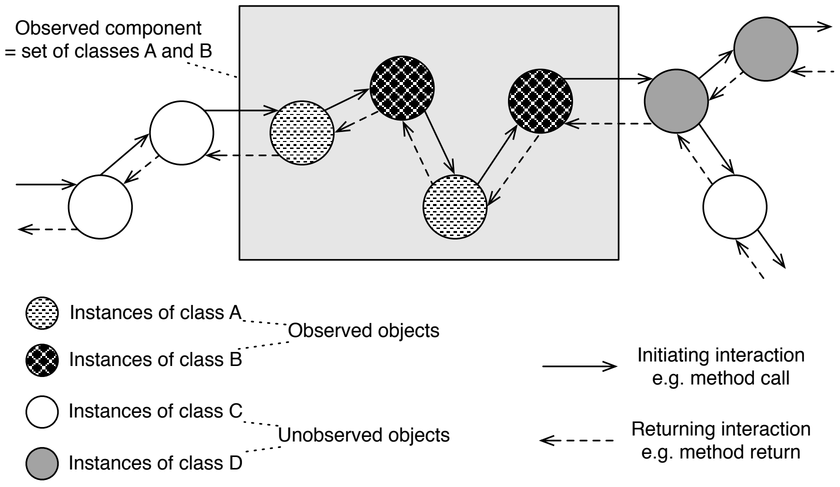


Figure 3.4: **Dividing the program into an observed and an unobserved part.** Interactions occur between objects both in *observed* and *unobserved* parts of the program. Observed objects are enclosed by an imaginary boundary. JINSI only captures interactions that either cross this boundary or occur within this boundary.

tivity of the approach: the boundary in a way selects the interactions of interest to be captured. However, it does not only select the interactions itself, but also defines the data to be captured as we will see in Section 3.1.3. Figure 3.5 on page 34 gives a formal definition of interactions of (no) interest.

To capture interactions of individual observed objects and to associate single interactions with their corresponding objects, JINSI has to identify objects uniquely. For this purpose, JINSI assigns unique IDs to all the objects involved in capture. We will see how these IDs are determined and assigned in the next

Let O_o be the set of *all observed objects,* and O_u the set of *all unobserved objects* of a program run. Furthermore, let (o_s, o_r) be an *interaction* between two objects o_s and o_r. Object o_s is called *sender,* and object o_r is called *receiver* of that interaction.

An interaction (o_s, o_r) with $o_s \in O_u \land o_r \in O_u$ (both unobserved) is *of no interest.* All other interactions, thus $o_s \in O_o \lor o_r \in O_o$ (at least one involved object is observed), are *of interest.*

The definition of *being of (no) interest* correspondingly applies to interactions between classes, and between classes and objects, respectively.

Figure 3.5: **The definition of (un)interesting interactions in a nutshell.** We classify all interactions within a program run either as being *of interest,* or as being *of no interest.* This property is derived from the character of the objects involved in that interaction: according to whether these objects are observed or unobserved.

section.

3.1.2 Object IDs

A further fundamental concept is the one of *object identifiers* (*object IDs* for short). An object ID is an ordered pair (`id`, `type`) whose first entry `id` is a positive numeric (i.e., `id`$\in \mathbb{N}_0$) unique identifier that bijectively names an object `obj` (e.g., an instance of class `TimeZone`), and whose second entry `type` is the actual type of that object (in that case class `TimeZone`).

To assign object IDs to individual objects, JINSI uses a numeric global counter which is initialized to zero when capture starts, and further maintains a global *object mapping* that associates object IDs with objects. JINSI populates the mapping incrementally: whenever the tool needs to determine an object's identity (see Section 3.1.3), it tries to look up the corresponding object ID in the mapping. If the mapping does not contain object `obj` already (first lookup for this object), JINSI will associate the object with both the current value of the global counter and its actual type by adding ((`id`,`type`),`obj`) as new entry to the object mapping, and by increasing the counter by one afterwards. Otherwise, JINSI will use the existing pair (`id`, `type`), which was assigned before and therefore is already contained in the mapping (subsequent lookups).

The second component, the object's actual type, can be easily derived from the object itself. While for objects the first entry `id` would be sufficient in order to identify all objects uniquely (by numbering them in order of occurrence), the second component `type` describes the object's runtime type, and allows JINSI to create proper instances during replay; for later replay, JINSI does not persist the complete mapping including the actual objects (which would involve techniques like object serialization including all their disadvantages, see Section 3.1.3), but only all ordered pairs (`id`, `type`). JINSI uses that structure, instead of simple numbers, for another reason: to identify classes involved in interactions.

3.1.2.1 Class Members

In addition to re-creation of objects during replay, the structure (`id`,`type`) enables JINSI to distinguish between different *class members*[4] involved in interactions; in object oriented programming, interactions can act on the class level rather than at the object level. In this case, a method call or a field access does not refer to a specific instance of the class, but to the class itself—in JAVA, these methods and fields are usually called *static*.

For this reason, next to IDs for individual objects, JINSI assigns IDs for individual classes as well if necessary. For static method calls and static field accesses, JINSI assigns a unique ID (-1,`type`) to the corresponding class involved in the interaction. While the `id` is fixed to -1, the `type` describes the

[4]In contrast to *instance members* which belong to an instance of a class, *class members* belong to the class.

actual class; for instance $(-1, \texttt{StringUtils})$[5]. In this way, JINSI can easily distinguish between individual objects and classes which are involved in interactions. Please note that in the following, an object ID may both refer to a true object (thus, a class *instance*), as well as to a class. Furthermore, when we talk about object interactions, this will include interactions that involve class members (where applicable).

Section 8.7 contains more details on objects IDs in general and on object IDs during capture in particular. In fact, for object mapping, JINSI uses different *map* implementations to map objects to their IDs (and vice versa) depending on whether JINSI is doing capture or replay. For instance, during capture JINSI ensures that the object mapping does not influence garbage collection.

3.1.3 Partial Information

When capturing data flowing across the boundary that separates the observed from the unobserved part (e.g., values assigned to a field), a major issue is that the types of such data range from simple scalar values to complex and composite objects. Whereas capturing scalar values can be done inexpensively, collecting object values is computation- and space-expensive. A straightforward approach that captures all values through the system (e.g., by serializing objects passed as parameters) would incur tremendous overhead, and therefore would render the approach impractical [52]. The key intuition to address this problem is that

1. we only need to capture the subsets of those objects that affect the computation, and

[5]A utility class defines a set of methods that perform common, often re-used functions. Most utility classes define these common methods under static scope; thus, define class member methods. In the example, this would be a class that provides static methods to deal with `String` objects.

```
boolean hasPositiveElement(Set set) {
  boolean found = false;
  Element e = null;
  Iterator it = set.iterator();
  while (it.hasNext()) {
    e = it.next();
    if (e.value > 0) {
      found = true;
      break;
    }
  }
  return found;
}
```

Listing 3.1: **Replaying interactions.** The `hasPositiveElement()` method. Replaying interactions requires only a minimum of recorded information: capturing partial information and only scalar values in particular suffice to replay the original run.

2. we can conservatively approximate such subset by capturing it incrementally and on demand, without using sophisticated static analyses.

As an example, consider the code shown in Listing 3.1 on page 37. Method `hasPositiveElement(Set set)` takes a set as a parameter and returns `true` if the set contains at least one positive element. Consider now a call to `hasPositiveElement(...)` in which the third element returned by the iterator is positive. In this case, even if `set` would contain millions of elements, we will only need to store the first three elements accessed in order to replay the call.

In fact, we do not need to store objects at all. Ultimately, what affects the computation are the scalar values stored in objects or returned by objects' methods. Therefore, as long as we can automatically identify and intercept accesses to those values, we can disregard the objects' state. For instance, in the example considered, the only data we need to store to replay the call to `hasPositiveElement(...)` are the boolean values returned by the calls to the iterator's method `hasNext()`, which determine the value of the `while` predicate, and the values associated with the three elements accessed.

Although it is in general not possible to identify in advance which subset of the information being passed to a method is relevant for a given call, we can conservatively approximate such subset by collecting it *incrementally.* To this end, when logging data that crosses the boundaries of the component, we record the actual value of the data only for scalar values. For objects, we only record their unique numeric identifier and type (their object ID in other words). With this approach, object IDs and scalar values are the only information required to replay executions, which can dramatically reduce the space and time costs of the capture phase.

3.1.4 Object Interactions and Events

As briefly mentioned above, JINSI captures (and replays) *object interactions* (*interaction* for short) in terms of different *events*. During capture, JINSI identifies and stores every relevant interaction either originating from, or targeting an observed object as an event with a set of attributes—thus, an event describes an interaction between objects during a program run. The events, together with their attributes, are finally recorded in an *event log*[6].

[6]This event log could be any mechanism that is able to store and load the events. Currently, JINSI uses a regular file and stores the events as XML data. While using XML at first sight could be regarded as a questionable choice because of its demanding memory requirements compared to other formats, like some customized binary format, this potential downside does not become important in our context: even the largest XML event log as occurred in doing our experiments

3.1.4.1 Events in General

Most generally, JINSI distinguishes between *incoming* and *outgoing* interactions and events, respectively—in doing so, the observed part's point of view defines the perspective:

Incoming An interaction that targets an observed object is called incoming. This also applies to its corresponding event.

Outgoing Analogously, an interaction that originates from an observed object is called outgoing. Again, the same applies to the corresponding event.

For each of the above two basic types of events, there is the same set of subclasses of events which describe the concrete type of interaction in terms of behavior (method call, field access, etc.). Depending on this type of interaction, JINSI stores a different set of attributes. Figure 3.6 on page 40 shows a greatly simplified class diagram that describes the basic hierarchy and attributes of the events as used by JINSI. It defines a set of abstract event classes that subsume the attributes that different concrete types of events have in common, and a set of concrete event classes each representing a concrete type. At the top of that hierarchy, JINSI defines an abstract event that shares the attributes *event ID* and *location* which all events have in common:

- An *event ID* is an ordered pair (`id,threadId`) of which both entries are positive numeric values. The first uniquely identifies the event; all events are numbered sequentially in order of occurrence. The second entry denotes the corresponding thread ID in which the event has occurred.

contains not more than 15 MB of textual data. After using `gzip` to compress this event log file, only 668 kB remain. This amount of output data is easy to manage even by actual laptop machines—as used for conducting the experiments.

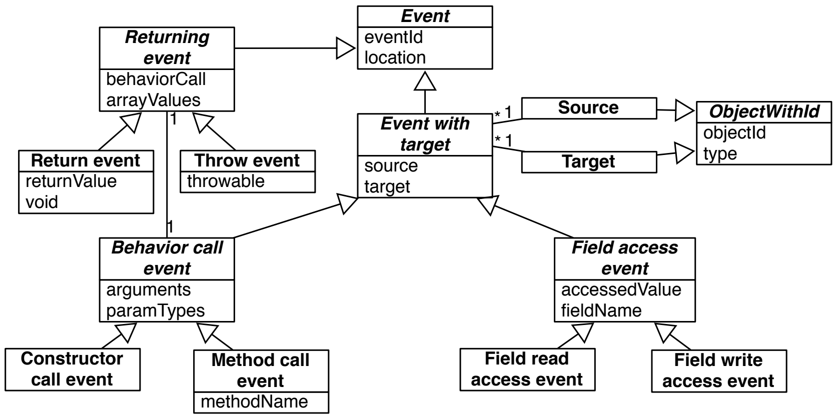


Figure 3.6: **Schematic overview of different events.** JINSI defines abstract events that subsume attributes common to several events, and concrete events for the different types of interactions (e.g., method calls). For simplicity, the diagram omits the distinction between observed and unobserved objects. Actually, there are individual concrete event classes for incoming constructor calls, outgoing field reads, and so on.

While the `id` already uniquely identifies the event, the `threadId` is used to be able to assign events to their respective threads.[7]

- A *location* has three attributes: (1) a file name, (2) a line number, and (3) a context. While the first two attributes define the location in the program's source code where the interaction is declared (e.g., a method call statement), the latter defines whether the interaction occurred within a class initializer, a constructor, or a method. This information is used for synchronization purposes during replay and minimization (see Section 4.1.2).

Whenever an interaction occurs, there must be two objects that are involved: the first one that initiates the interaction (sends a message), and the second one that perceives the effect (receives the message) of this very interaction.[8] Therefore, JINSI provides the corresponding events with a source and a target object:

Event with target This is the abstract superclass of all events that describe an interaction in which a sender and a receiver are involved: (1) constructor calls, (2) method calls, and (3) field accesses. Therefore, these events share a *source* and a *target* as their attributes. Each of these attributes contains an *unique object number* and the *type* of the corresponding object. Thus, source and target are equal to an object ID (see Section 3.1.2). In this way, JINSI is able (a) to clearly identify the two objects involved

[7] In the current implementation, JINSI replays all interactions regardless of the thread where they occurred, and in the same order as they were captured; thus, JINSI serializes the program in a way. In the evaluation (see Chapter 9), JINSI was mainly applied to single-threaded applications. In the COLUMBA example, JINSI was applied to a multi-threaded program; however, we captured interactions that occurred within the same thread only.

[8] It is possible that sender and receiver are the same object; e.g., whenever an object calls a method on itself. However, JINSI does not capture these self-directed interactions explicitly. Instead, these interactions re-occur naturally during replay.

in an interaction, and (b) to re-create these objects during replay (see Section 3.2.1).

In the following, we address in detail how concrete interactions are represented by specific events. As method calls are the most common type of interaction in object-oriented programs, we start with these.

3.1.4.2 Method and Constructor Calls

Since methods and constructors are very similar in nature[9] and describe a sort of *behavior* of an object, they share a set of common attributes: every constructor and method has a list of parameters specifying input by means of data types, and every call accordingly has a list of arguments[10]. JINSI therefore subsumes constructor and method calls under abstract, so-called behavior call events:

Behavior call event Such an abstract event represents a behavior call; i.e., either a constructor or a method call. They share two attributes: firstly, a list of parameter types defining the possible input variables, and secondly, a list of arguments describing the actual values as seen at runtime. If the passed value is scalar, JINSI will record the actual value. Otherwise, if it is complex, JINSI will record the object's object ID. In this way, JINSI will be able to properly re-create the captured arguments during replay (analogous to the source and target of an interaction).

In order to be able to distinguish between constructor and method calls, JINSI uses two different concrete events:

Constructor call event Represents a constructor call.

[9]A constructor is a special type of (static) method that does not have an explicit return type but implicitly returns an instance of the relevant class.

[10]*Parameters* refers to the list of input variables in a method's *declaration,* while *arguments* will be the actual values passed in when the method is *invoked*. When invoking the method, the arguments passed must match the declared parameters in order and type.

Method call event Analogously, represents a method call. Additionally, contains the method name to complete the method's signature; thus, to describe a method uniquely.

Behavior calls have two final important properties: First, they can *enclose further interactions*. For instance, a called constructor can access other object's fields during initialization. Second, they usually *have a return value*. In order to take these properties into account, JINSI defines an abstract returning event:

Returning event This event represents the return of a previously captured behavior call, and its actual result; it is captured whenever such an interaction exits. This type of event contains a reference to its corresponding behavior call event.

For returning events, there are two concrete subclasses that distinguish between (1) a regular return of a behavior call, and (2) an exceptional exit of such a call:

Return event This event describes a regular return of behavior calls; thus, when such a call returns control together with a value (unless its a *void* method) to the invoker. This type of event includes the attributes *returnValue* representing the actual return value (i.e., either a scalar value, or a complex object represented by its object ID), and the boolean flag *void* that indicates whether the called behavior has a return type. The latter makes it easier to distinguish between a method with some return type that returned `null`, and a void method without any return type.

Throw event A throw event describes the exceptional termination of a behavior call. Usually, an exception is caused either by an explicit `throw` statement, or an abnormal execution condition detected automatically (such as a division by zero). This event includes attribute *throwable,* which describes the actual exception thrown at runtime.

For the sake of simplicity, we omitted the concrete types of events that distinguish between incoming and outgoing events. These concrete events represent the different interactions as they can actually occur during a program run, for instance incoming method calls and outgoing constructor call returns.[11] However, the definition of these events is fairly straightforward and would not yield any new concept of further interest. In addition to behavior calls, JINSI captures field accesses as described in the following.

3.1.4.3 Field Accesses

Interactions between objects can also occur via accesses to fields. JINSI uses an abstract field access event to take into account both read and write accesses:

Field Access Event This specialization of an *event with target* describes a field access in general. It contains both the *accessed value* (either as scalar value, or as object ID for complex objects) either read from or assigned to the field, and the *name* of the field that had been accessed.

To be capable of distinguishing between read and write accesses, JINSI uses two corresponding, concrete events:

Field read access event This type of event is used whenever a field was read.

Field write access event Analogously, this type is used whenever a field was written.

Similar to behavior calls, we omitted the fact that JINSI actually uses four instead of two different events for field accesses: for each of the two access types, JINSI again uses one concrete type for incoming, and one for outgoing

[11] Actually, there are 10 different event types describing behavior calls and their corresponding events: from incoming constructor call events, to incoming method call returns and outgoing method calls, and finally to outgoing throw events.

accesses. In contrast to SCARPE [52], which does not capture incoming read accesses, JINSI captures all four different instances; hence, including incoming read accesses. At first sight there is no need of replaying this type of interaction because an incoming read access actually does not modify the state of the observed part of the program. However, JINSI requires to capture incoming read accesses as well. During capture, this mainly is for debugging reasons: it proofed to be helpful during debugging JINSI itself, especially when manually analyzing the captured data. However, during replay, incoming read accesses are used to add objects to the object pool if required (see Section 3.2.1).

3.1.5 Interception and Recording of Interactions

In the following, we shall address the actual interception and recording of object interactions. For the individual interactions, JINSI uses different appropriate techniques in order to identify, intercept, and process interactions; and eventually to record interactions of interest in terms of events. They all have in common: Before the program starts, JINSI identifies possible object interactions based on the set of observed classes, and accordingly modifies the program. This enables it to intercept interactions between observed and unobserved objects at runtime. The modifications are mainly done by inserting probes; i.e., by adding new code to the program. When the modified program runs, the modified parts and statements will transiently hand over control to JINSI, which then will process the current interaction and will decide whether an event has to be stored or not. In the former case, an event will be eventually written to the event log, in the latter the interaction will be ignored.

In this context, a major difference between SCARPE and JINSI comes into play: while SCARPE captures and replays interactions with respect to observed *code,* JINSI captures and replays *object* interactions. As an important possible consequence, SCARPE is not able to completely distinguish between different objects within an inheritance hierarchy [52][12]; furthermore, this crucial difference is especially important when capturing incoming constructor and method calls. Chapter 7 contains details on the various consequences for the instrumentation of JAVA programs; in this section, we concentrate on the universal concepts that generally hold true for object-oriented programs. Generally speaking, in contrast to SCARPE, JINSI has to check for the majority of interactions of *possible* interest *at runtime* whether each one actually *is* of interest. This is due to the fact that, in many situations and because of polymorphism and dynamic binding, the *dynamic* type of an object is determining whether an interaction is outgoing or incoming, or of no interest. Thus, it is often impossible to exactly decide during instrumentation—which is done statically—whether an interaction affects an observed or unobserved object; only at runtime JINSI can analyze the dynamic type. In summary it can therefore be said that JINSI works on a lower abstraction level (objects) than SCARPE (classes). Only because of that lower abstraction level, JINSI can use delta debugging together with slicing techniques, like event slicing and dynamic backward slicing, in order to efficiently minimize the faulty program behavior which, starting at the actual defect, across many different object interactions eventually leads to an observable failure (see Chapter 4 for details).

In the remainder of this section, we will address the different methods JINSI uses to capture the events described above. Please note that in addition to the actually relevant information (e.g., method call arguments), for each interaction in question, the inserted probes always pass information about the current *location* (i.e., file name, line number, and context of the corresponding statement, see above) to JINSI, and for returning events the unique event ID of the corresponding call event as well. For simplicity, we omit this information when describing the individual modifications. The actual instrumentation provides this information in all cases.

[12] When observing class c, all subclasses of c are observed automatically. This circumstance limits the approach's selectivity (compare Section 3.1.1).

3.1.5.1 Constructor Calls

Before an interaction between objects can occur, the involved objects will have to be constructed; hence, we start with this kind of interaction. Usually, constructor call declarations already define at compile time which concrete instance will be constructed—in contrast to the receiver of a method call (see below). Thus, JINSI can decide at instrumentation time whether the corresponding interaction-to-be will target an observed or unobserved object. However, whether the source is observed or unobserved can be decided at runtime only. In order to capture an *outgoing constructor call* OCC, JINSI

1. precedes OCC with a probe that will pass the actual *source object*, the statically known *target type* and *parameter types,* and its actual *arguments* to JINSI; and

2. adds a probe directly after OCC that will pass the recently *constructed object* to JINSI.

At runtime, JINSI can decide whether the intercepted interaction actually is an outgoing one, and can accordingly either store an event or discard it. For instance, when observing class c, JINSI instruments all constructor call statements declared within c. However, only at runtime JINSI can definitely decide on the actual type of the instance of c (thus, the source of that interaction): if the actual object is an instance of *unobserved subclass* c' of c, it is not an outgoing call, but rather a call originating in an unobserved object (however, it might be an incoming constructor call if the receiver is observed; see below). Thus, only if the *actual* dynamic type of the source object is c, it will be an outgoing call originating from an observed object.

The list of parameter types enables JINSI to distinguish between different constructors with different signatures. For the given list of arguments, JINSI either stores the actual scalar value, or a proper object ID in case of a complex object. All that information will be recorded as an outgoing constructor call event.

Right after the actual constructor call, the second probe added by JINSI will be called, and JINSI will be provided with the recently constructed object. This second probe enables JINSI to record the object ID of the newly created object. In a nutshell, JINSI adds two probes that capture (1) the actual outgoing constructor call, and (2) its corresponding return value—the constructed object. Capturing two events enables JINSI further to capture interleaving events: the call event marks the beginning of the constructor call that again can trigger further interactions; the return event eventually marks the end of the call. In languages that allow *exception handling,* like the JAVA programming language, an exception that has been thrown is another possible cause for a constructor (or method) call to abortively return control back to its caller. In fact, JINSI extends the above technique by exception handling, which will be described for the sake of clarity in an own section below (see page 53).

For *incoming constructor calls,* JINSI adds similar probes. At runtime, analogous checks will ensure that only actually incoming calls are captured. However, besides the direction of the interaction, there is a further main difference between capturing outgoing and incoming constructor calls: because the target of a constructor call is determined at compile-time (i.e., the programmer has to decide during coding which constructor will be called at run-time), for *outgoing* constructor calls, JINSI only has to instrument *observed* classes (and all their super- and subclasses if necessary, see Section 7.1), while for *incoming* constructor calls, it has to instrument *unobserved* classes as well. However, JINSI can rule out most[13] of the constructor calls declared within unobserved classes already during instrumentation: if a constructor call is declared within an unobserved class and its target type is not observed, which is known statically, this constructor call can be safely ignored—in contrast to potentially incoming method calls, which are subject to dynamic binding (see below).

[13]On the supposition that only a relatively small number of all classes are observed.

While capturing constructor calls seems to be straightforward, it poses several challenges when applied to JAVA programs, especially when it comes to incoming constructor calls. Section 7.1 describes in detail how JINSI instruments JAVA programs for capturing that kind of interaction.

3.1.5.2 Method Calls

While method calls are very similar to constructor calls at first glance (see Section 3.1.4.2), capturing them is a more complex and difficult task, in particular with respect to incoming method calls, as we will see shortly. Capturing *outgoing method calls* has been realized in a similar way as the above approach for outgoing constructor calls. However, in contrast to constructor calls, the actual receiver of a method call and its type are unknown at compile time. To capture an outgoing method call `OMC`, JINSI

1. precedes `OMC` with a probe that will pass the actual *source object* and *target object,* the statically known *method name* and *parameter types,* and its actual *arguments* to JINSI; and
2. adds a probe directly after `OMC` that will pass the recently *returned value* (if any) to JINSI.

Similar to outgoing constructor calls, JINSI checks at runtime whether an intercepted potentially outgoing method call actually is an outgoing one (depending on both the source and target object) and, if applicable, will record an appropriate event. The list of parameter types and the method name allow JINSI to clearly distinguish between different methods declared within one class.[14] The arguments are stored either as scalar value or object ID. After the called method has returned control back to its caller, the second probe allows JINSI to record

[14] And, together with the run-time type of the object the method was called on, JINSI can properly handle polymorphism, as inherently and frequently used in object-oriented programming, as well.

the return value, and the end of that interaction. In the case of methods without a return type, JINSI does not store any value but only the return event itself.

Compared to outgoing constructor calls, JINSI can not rule out as many possible calls because the actual receiver type is unknown. However, the set of observed classes considerably limits the number of possible locations: an outgoing call can only occur within an observed object. This reduces both the number of locations where JINSI has to insert probes (thus, to modify the program), and the number of run-time checks which slow down the execution—in contrast to *incoming method calls* which can occur virtually everywhere in the program.

This is due the fact that in object-oriented programming *polymorphism*[15] is an essential and therefore widely used concept: the ability of objects belonging to different types to respond to method calls (or field accesses), of the *same* name, but each one according to an appropriate, possibly *different,* type-specific behavior. Thus, the programmer (as well as both the program and JINSI) does not necessarily know the exact type of the object in advance (statically, which is equivalent to instrumentation-time), and so the exact behavior is determined only when the program is executed (dynamically). As a direct result, to capture incoming method calls in a similar fashion using probes as for outgoing ones and constructor calls, JINSI would have to instrument the program in much more places, because JINSI could not rule out most of the possible calls.[16] This would result in much more modifications of the program which might be error-prone (see Chapter 7 for the actual risk of modifying JAVA programs), and much more run-time checks which would slow down the program considerably. To capture incoming method calls, JINSI therefore uses a different approach than modifying the caller: it extends and modifies the observed

[15] To be precise, *subtype* polymorphism. However, in the context of object-oriented programming it is almost universally called just polymorphism.

[16] For instance, if an observed object implements a very common interface like `Comparable` in the JAVA language, JINSI would basically have to insert probes at all the places where this interface is used.

methods *themselves*. To capture an incoming method call `IMC` to method `m()`, JINSI

1. firstly replaces `m()` with a proxy method and an actual method. The *actual method* has the same body as `m()` (modulo some instrumentation), but has a modified name `m'`. The *proxy method,* conversely, has exactly the same name as `m()`, but a different implementation. The proxy method (1) creates and logs the appropriate call event, (2) calls the actual method `m'()` by specifying the modified name `m'`, (3) collects the value returned by the actual method (if any), logs a return event, and (4) returns to its caller the collected value (if any); and

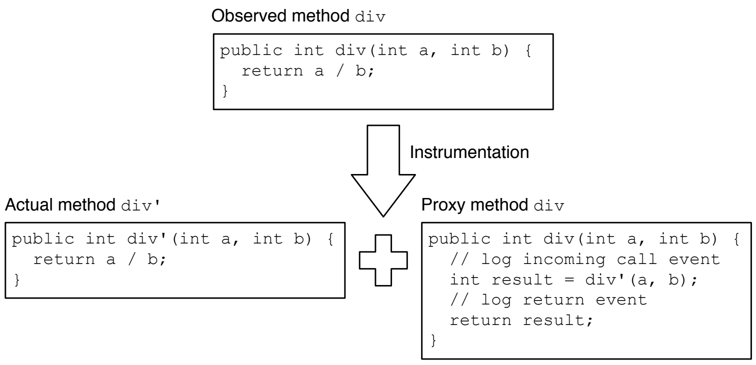
2. secondly, modifies all calls from observed methods to other observed methods by using the modified name `m'` mentioned above. In this way, we are guaranteed that calls that do not cross the boundaries of the observed part invoke the actual (and not the proxy) method and do not log any spurious incoming call or incoming return (these calls and returns occur naturally during replay).

Figure 3.7 on page 52 shows an example of a simple method `div` which divides two given numbers. For capturing incoming calls to `div`, JINSI introduces a proxy method under the name of `div`, and the actual method renamed to `div'` as described above. By using such proxy methods, JINSI is able to capture incoming method calls while minimizing both the number of locations where the program has to be modified and the number of possible run-time checks.

3.1.5.3 Field Accesses

Besides method calls, objects can interact with each other via field[17] accesses. Similar to methods, a field either can be shared with all class instances (static

[17] Also called member variable, or member field; thus, field for short.



```
public int div(int a, int b) {
  return a / b;
}
```

```
public int div'(int a, int b) {
  return a / b;
}
```

```
public int div(int a, int b) {
  // log incoming call event
  int result = div'(a, b);
  // log return event
  return result;
}
```

Figure 3.7: **Capturing incoming method calls.** JINSI replaces the original method `div` (top) with method `div′` which contains the actual body and runs the original computation, and proxy method `div` which passes all the relevant information about the incoming call to JINSI and forwards the call to the actual implementation (bottom).

field), or each instance of a class can have its own copy of the variable (instance variable). JINSI handles both cases by assigning proper object IDs. Field accesses differ from each other by being either a *read* or a *write* access; in our context we further have to differentiate between incoming and outgoing accesses. Capturing these four different event types is relatively straightforward, especially compared with (incoming) method calls. For instance, to capture a possibly *incoming field write* access `IFW`, JINSI precedes `IFW` with a probe that passes

1. the *source object* which writes the new value,
2. the *target object* the written field belongs to,

3. the *field name*, and
4. the *value* that is to be written.

Using a simple run-time check, JINSI determines whether the intercepted write access is an incoming one, and accordingly writes an event (if applicable). Both the source and target objects are recorded using their corresponding object ID; the written value is either stored as scalar value or as object ID. By recording the target object together with the field name, JINSI can handle shadowing[18] of fields.

For the other kinds of field accesses, JINSI uses closely related techniques. To capture *incoming field read* access, JINSI adds a similar probe; however, the probe is introduced right after the actual access. Instead of recording the value which is about to be written, JINSI records the read value. For *outgoing field write* and *outgoing field read* accesses, JINSI basically uses properly inverted run-time checks. Section 8.4 contains detailed information on how JINSI handles field accesses in practice and JAVA programs, respectively.

3.1.5.4 Exceptions

Many programming languages[19]—including JAVA—have built-in support for exceptions and exception handling to allow the program to report and to handle erroneous situations. An *exception* is an event[20], which occurs during the execution of a program, that disrupts the normal control flow: when an error occurs within a method, the method creates an *exception object*, which is provided with information about the circumstances of the error, and hands it off to

[18] When extending a class, you can *shadow* a field with the same name in the base class. In contrast to (method) overriding, the field in the base class is only hidden and is thus still accessible.

[19] Such as Ada, C++, Delphi, Erlang, Haskell, JAVA, OCaml, Perl, Python, Ruby, Smalltalk, and Visual Basic, to name but a few.

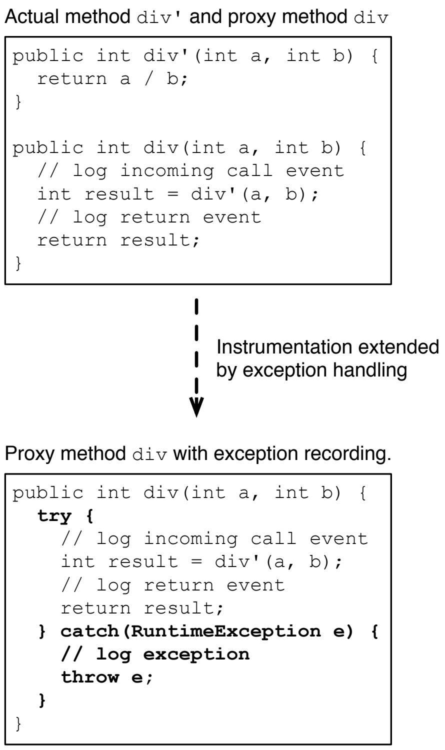
[20] Here not to be confused with an event in the sense of events as used by JINSI to capture interactions.

the runtime system. This procedure is called *throwing an exception*. The most important fact in our context is that the method in this case changes the normal flow of program execution—as one might say the method returns abortively but controlledly. Thus, throwing an exception is another way a method (or constructor) can terminate, in addition to returning a value explicitly, and therefore can also cause interactions to occur.

The possibility of an exception being thrown can be expressed explicitly, for instance by enclosing a method call within an exception handler.[21] However, an exception basically can happen to occur whenever a method is called, even if it is not expressed explicitly in the code; such exceptions include arithmetic exceptions, for instance when dividing by zero. In this case, if a method may throw an *unchecked* exception, the compiler would not complain about an unhandled exception. Furthermore, JINSI has to capture and replay *all* exceptions as they might occur in a program run; otherwise, replaying would result in a different behavior than originally happened and observed by the user. To cover all those cases where an exception might be thrown, JINSI extends the probes that capture constructor and method calls by exception handling—it basically wraps relevant behavior calls in appropriate `try-catch` blocks. If there is no explicit exception, JINSI introduces such a block. Otherwise, if there is already an exception handler, JINSI incorporates that handler. Figure 3.8 on page 55 shows how proxy method `div` (see Figure 3.7 on page 52) is extended in order to handle possible exceptions. Apparently, the actual method `div'` will throw an arithmetic exception if argument `b` is `0` (division by zero). The extended proxy method will record that exception if any. Chapter 7 contains detailed information on capturing exceptions as they might occur in JAVA programs.

In the next section, we shall see how JINSI uses the captured object interactions and events, respectively, to replay those events, and thus to reproduce program failures.

[21] In JAVA, for instance, using a `try-catch` block.


Actual method div' and proxy method div
Instrumentation extended by exception handling
Proxy method div with exception recording.

```
public int div'(int a, int b) {
  return a / b;
}

public int div(int a, int b) {
  // log incoming call event
  int result = div'(a, b);
  // log return event
  return result;
}
```

```
public int div(int a, int b) {
  try {
    // log incoming call event
    int result = div'(a, b);
    // log return event
    return result;
  } catch(RuntimeException e) {
    // log exception
    throw e;
  }
}
```

Figure 3.8: **Capturing exception flow.** To capture possible exceptions, JINSI extends proxy method `div` by a custom exception handler that catches any thrown exception, reports the exception to JINSI, and finally re-throws it. In this way, JINSI ensures to capture exception flow while not altering the program flow.

3.2 Replay

To finally reproduce the original failure, JINSI replays the previously recorded interactions. Analogous to capture, it performs two steps before it runs the actual replay: JINSI identifies possible interactions between the observed component and the unobserved rest of the application, and accordingly instruments the application. For replay, *JINSI completely replaces the component's environment* (see Figure 3.9 on page 57). After instrumenting, the tool reads the previously captured events from the event log, processes them, and for each event, either triggers the incoming interaction on the observed objects[22], or consumes the outgoing interaction originating from the observed component and provides a proper return value. Interactions between observed objects are not intercepted[23]; they happen naturally as a result of the incoming interactions initiated by JINSI.

In this way, JINSI acts as both a driver and a stub during replay: it provides a *scaffolding that mimics* the behavior of the unobserved part as seen during capture. For incoming interactions, it passes both control and proper arguments (if any) to the observed objects which repeatably behave exactly in the same way as before. Whenever an observed object returns control to the unobserved part, JINSI takes over and provides a proper answer to that outgoing interaction. The tool will repeat this procedure until all the captured events are completely processed. In the end, JINSI reproduces both the environment and the problem's history, and therefore is able to reproduce the actual failure.

Similar to capture, in the following we first outline the replay phase to give a brief overview on the related concepts, and afterwards we discuss the fundamental ideas in detail while staying on a conceptual level. Again, Chapter 7 will provide details on replaying JAVA programs in practice.

Listing 3.2 on page 58 exemplifies an instrumented component on the basis

[22] JINSI uses the JAVA reflection API for that purpose; see Chapter 7 for details.
[23] Constructor calls constitute an exception, see Section 3.2.1.

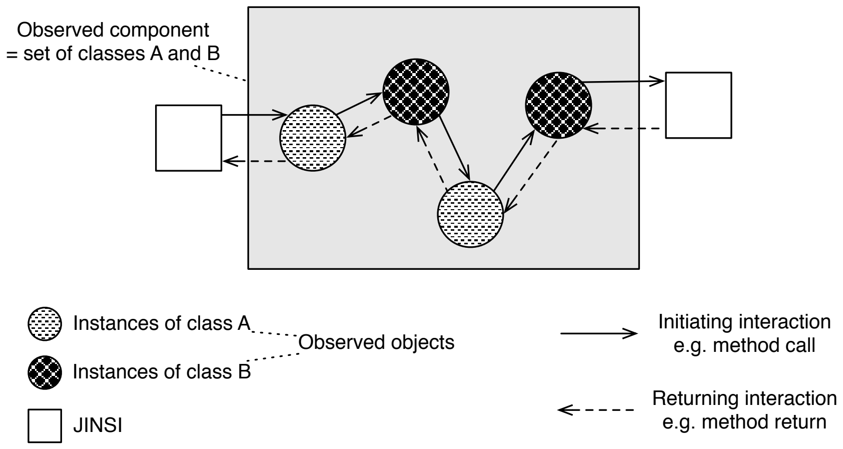


Figure 3.9: **JINSI completely replaces the component's environment.** JINSI acts both as a driver and a stub: it triggers incoming interactions and provides proper reactions for outgoing ones. Interactions between observed objects occur naturally; they happen directly and indirectly in consequence of the synthetically initiated incoming interactions.

of class `Observed` (see Figure 3.2 on page 29). JINSI has replaced the two invocations with calls to `JINSI.getReturnValue(...)` which returns a proper value depending on the concrete captured event. For instance, while replaying an incoming constructor call, JINSI consumes the replaced call to `getDefault()` and returns a *mocked* instance of `DateTimeZone`. Since JAVA does not allow to create instances without explicitly calling a constructor, JINSI returns a mock object that is just an empty—thus stateless—hull representing an instance of proper type. When a subsequent incoming call to

```java
public class Observed {
  private DateTimeZone timeZone;
  public Observed() {
    this.timeZone = (DateTimeZone)
      JINSI.getReturnValue(this, DateTimeZone.class,
                           "getDefault", new Object[0])
                               ;
  }
  public String getTimeZoneName() {
    return (String)
      JINSI.getReturnValue(this, this.timeZone,
                           "toString", new Object[0]);
  }
}
```

Listing 3.2: **Replaying outgoing method calls.** JINSI replaces outgoing method invocations with calls to itself and returns proper values depending on the captured event.

method `getTimeZoneName()` is replayed, JINSI again consumes the originally captured outgoing call via `getReturnValue(...)`, which returns the captured `String` representing the time zone at capture time.[24]

In our initial example, if the user captures on-site, the developer—having the captured events at hand—will be able to *reproduce the original problem in any time zone*. Other types of interaction, like constructor calls and field accesses, are captured and replayed in a similar fashion.

In the remainder of this chapter, we shall see how JINSI re-creates both observed and unobserved objects (Section 3.2.1), replays method calls (Sec-

[24]This is in contrast to tools like RECRASH [2], which capture only the objects on the heap, and which, in this example, return the current time rather than the captured time.

tion 3.2.2), field accesses (Section 3.2.3), and exceptions (Section 3.2.4) finally.

3.2.1 Object Re-Creation

Before JINSI can replay any object interaction, it has to instantiate the objects that were involved during capture. For this purpose, JINSI extracts object IDs (see Section 3.1.2) from the event log, and basically *re-creates* objects of the corresponding type. If an object is needed for the first time, JINSI will instantiate a new one; otherwise, it will retrieve the previously created instance from an *object pool*. In contrast to capture when an object mapping, which is similar to an object pool, mainly is used to map objects to their corresponding identifiers, during replay JINSI manages a pool that firstly *maps identifiers to objects* mainly, and that secondly establishes a set of initialized objects that are kept ready to use.[25]

Each time JINSI processes an event whose attributes contain an object ID (e.g., as receiver or argument of an interaction), it attempts to retrieve the corresponding object from the pool.[26] If the lookup is successful, it will use the retrieved object to reproduce the event. If the object was not contained in the pool, JINSI will either create a placeholder (mock) object if the object is unobserved, or will create a true object otherwise. In both cases, JINSI ensures that the re-created object is of the appropriate type. Finally, it will create a new entry in the pool that maps the object ID to the newly instantiated object.

While JINSI uses the same pool to manage both unobserved and observed objects, it uses different strategies to create appropriate instances. For *unobserved objects,* JINSI creates not an actual instance of the originally observed type, but

[25]This is for performance reasons; using an object pool saves the cost for instantiating the same object over and over again.

[26]Except for cases when not an actual class instance, but a class itself is required. For instance, if an outgoing interaction refers to a class member, JINSI will not require to use an object but the corresponding class.

a mock object that basically represents a placeholder: the mock object's type and identity are meaningful; however, the object itself has no state at all. The object's type actually is a sub-type of the unobserved object in order to support type comparison, and its identity is properly restored in order to support the comparison of object references.[27] Because the mock object does not hold any state, JINSI has to step in every time its state is accessed, which includes method and field accesses (we will see how JINSI replays these interactions below). Thus, one could say that these mock objects and JINSI itself, respectively, *mimic* the problem's environment.

The usage of mock objects instead of true instances is in contrast to SCARPE, which also creates true objects for unobserved ones. While that simpler method works well at first glance, it causes severe issues when applying the JINSI approach to JAVA programs. In brief, using mock objects instead of actual instances of unobserved types avoids the occurrence of unwanted interactions and side-effects while creating new objects (see Chapter 7 for details).

For *observed objects,* JINSI creates instances of the originally observed type; after all, we want to reproduce their original behavior by re-executing the code contained therein while providing a proper environment, so that mocked observed objects are out of the question. In contrast to the unobserved mock objects, which are created on demand, JINSI instantiates observed objects *proactively*. Whenever an incoming constructor call has to be processed, JINSI actively calls the corresponding constructor along with passing proper arguments, and finally adds the newly created object to the object pool, so that it can be retrieved later; e.g., when an incoming method call will be executed on that very instance.

Replaying constructor calls is similar to method calls, which we will discuss next. Please note that capture and replay have in common that JINSI has to check at runtime, where necessary, whether an interaction actually is outgoing,

[27] JAVA provides the type comparison operator `instanceof` to compare an object to a specified type. Object references are compared using the identity operator (==).

incoming, or of no interest at all. In the following, we do not discuss these runtime checks explicitly; however, JINSI checks for that property when intercepting interactions of potential interest, and either handles the interaction by itself (e.g., by providing proper return values to outgoing method calls), or just passes control back immediately (thus, if the interaction is of no interest; e.g., if an observed object accesses the field of another observed object).

3.2.2 Replaying Method Calls

For each captured *incoming method call,* JINSI extracts the corresponding receiver, the method called, and the arguments from the event log. In the event log, the receiver is represented by its object ID. If the method called is a class member, JINSI does not need to obtain the receiver from the pool and immediately calls the method on the appropriate class. Otherwise, JINSI uses the identifier to obtain the actual receiver (an observed object) from the object pool. Because all incoming method calls and field accesses on an observed object have to be preceded by an incoming constructor call (an object must have been created before further interactions can occur), it is guaranteed that the receiver object had been added before JINSI executed the incoming constructor call.[28]

Next to obtaining the receiver, JINSI has to pass proper *arguments* when calling the method. For these arguments, it distinguishes three cases:

Scalar values Scalar values are obtained from the event log. For instance, if integer value 11 was observed during capture, JINSI passes that simple numeric value as argument.

`null` references Similar to scalar values, JINSI passes a `null` reference as argument.

[28]Even if an observed object o_1 constructs another observed object o_2, JINSI will intercept this inter-observed call in order to add object o_2 to the object pool. In this way, it is guaranteed that all observed objects will be contained in the object pool, even if they are directly constructed by other observed objects.

Complex objects JINSI uses the object ID to obtain the corresponding object from the object pool. The actual object is either directly obtained from the pool (in case it was created before), or is created on demand as described above.

This way, JINSI is able to re-create all the arguments required for the actual method call. Technically, JINSI uses JAVA reflection in order to call the method while passing the previously retrieved arguments (see Chapter 7 for details). After JINSI has called the observed method, control will flow to the observed object, which will behave exactly as during capture. Note that passing a mock object (i.e., an object without any state) does not compromise the replay because *all* interactions of the observed objects with unobserved ones will be suitably identified, intercepted, and handled by JINSI—such as outgoing method calls (see below).

As soon as the called observed method finishes its execution and thus control flows back to JINSI, the tool once again takes over and consumes the return value (if any). It then, as the case may be, compares the returned value to the one stored in the corresponding incoming method return event. If they match, the failure reproduction continues with the next incoming interaction; otherwise, JINSI would report a warning because that could indicate an error within JINSI. If the return value is a complex object, it will be furthermore added to the object pool (together with its object ID).

Outgoing method calls are also consumed by the scaffolding. JINSI instruments all possible outgoing calls and replaces them with

1. a call to a specific method `getReturnValue(...)` in the scaffolding that will intercept the original call at runtime; and
2. an assignment that stores the returned value to the appropriate variable in the original code (if applicable).

Listing 3.2 on page 58 shows an example. Method `getReturnValue` passes information about both the caller and the callee, the method itself, as well as possible arguments to JINSI. Whenever that method is called, control flows to JINSI which retrieves the next outgoing method call event from the event log and compares the passed information with the captured one. If they match, replay continues with the next event. Otherwise (e.g., the next event is not an outgoing method call), JINSI will issue an error.

As soon as JINSI has to pass back control to the observed object, thus to the caller of method `getReturnValue`, it processes the information in the corresponding outgoing method return event, retrieves the captured return value in the usual way (i.e., depending on its type), and finally passes the value to the observed object, which continues computation using the captured value.

Due to the fact that constructor calls basically are specialized method calls, they are handled in a similar way. For *outgoing constructor calls,* JINSI replaces possible instances with a call to method `getNew(...)` and returns an object according to the recorded one during capture; thus, JINSI returns a mock object (see above). For *incoming constructor calls,* the tool replays them by calling the appropriate constructor while passing proper arguments as for incoming method calls.

3.2.3 Replaying Field Accesses

Field accesses in general are handled in an analogous way to method calls. For *incoming field accesses,* JINSI replaces the unobserved part and executes the actual field access. To replay an *incoming write access,* it first reads from the corresponding event (1) the receiver object[29] (the receiver class in case of class members), (2) the name of the field to be modified, and (3) the value to be written. As usual, the latter can either be a scalar value, or a complex object, and is retrieved in the usual way.[30] The observed receiver then is obtained

[29] As for incoming method calls, the observed receiver is guaranteed to be contained in the object pool already.

[30] To be more precise, the value to be written can be either (1) a scalar value, (2) a `null` reference, (3) an unobserved mock object, or (4) an observed true object.

from the object pool. Finally, JINSI simply writes the value to the field. While an *incoming read access* can not have any modifying effect on the observed part of the program—because such an access does not change its state—JINSI nevertheless executes these as well for the following reasons: firstly, the read value is compared to the captured one in order to check whether replay is still in sync (otherwise JINSI would again issue a warning), and secondly in order to add the read object to the object pool (if the read value is an unobserved mock object or observed true object). Apart from that, read accesses are processed in the same way as write accesses, save that the value is not written but read.

For *outgoing field accesses,* JINSI replaces all possible accesses by a call to a specific method in scaffolding, comparable to outgoing method calls. For *outgoing write accesses,* JINSI replaces the access with a call to method `handleWrite(...)`, which at runtime receives information about both the object that writes the value and the object whose field is written, the field name, and the value to be written. From the observed part's point of view, JINSI just consumes this value. However, as for incoming read accesses, it uses the written value to check for synchronicity, and to add the value to the object pool (if it is not a scalar value and if the object is not contained already). Eventually, JINSI just passes control back to the observed part as there is no need for returning any value; thus, there is no change in the state of the observed part. Finally, *outgoing read accesses* are handled in a very similar way—all possible outgoing read accesses are replaced with a call to method `getValue(...)`—except that JINSI does not only handle back control but also obtains the value as recorded during capture from the event log and eventually returns this value (thus, either a scalar value or an object reference) to the observed object. For the observed object, the effect of the artificial, introduced method call does not differ from the actual read access—in both cases, the observed object gets the requested value.

3.2.4 Replaying Exceptions

The remaining interactions to be replayed are thrown exceptions. They can occur in consequence of both outgoing and incoming method calls, which includes constructor calls in this context. For *outgoing calls,* exceptions are handled analogous to regular return events: instead of passing a return value, JINSI constructs a proper exception based on the captured information, which is retrieved from the event log, finally throws the exception, and thus handles control back to the observed part. There, the exception will be processed in the same way as during capture (e.g., by executing an exception handler which reacts to the exception with some resolution).

For *incoming calls,* JINSI consumes any exception by providing an exception handler that catches the thrown exception. During normal replay as discussed here, JINSI uses the caught exception to check whether replay is in sync. If the returning event that belongs to the incoming call is not an exception event, this would be a strong indicator for a failure caused by JINSI; consequently, the tool would issue an error. During delta debugging, which we shall discuss in the next chapter, the caught exception is used as predicate to distinguish between failing and unresolved runs.

As we have seen above, JINSI completely assumes the unobserved part's role and both triggers incoming interactions and provides exactly those reactions to outgoing interactions as they occurred during the observed, faulty program run, and thus reproduces the failure in question. After reproduction, the next task in debugging is to find out *which circumstances are relevant* for the failure. Irrelevant circumstances can be ignored; relevant ones must be investigated. JINSI's aim is thus to simplify the execution such that only relevant object interactions remain. For this purpose, it uses three techniques (Figure 3.1): *delta debugging, event slicing,* and *dynamic slicing*—which we shall all discuss in the following chapter.

4 Simplifying Interactions

After we are able to reproduce the failure in question, the next important task for us in debugging is to simplify the problem; thus, we want to find out *what circumstances are truly relevant*. A circumstance may be any aspect that influences the problem like input, scheduling—or, method calls. As during reproduction, these aspects are subsumed under the problem environment, and the steps in the problem history. By experimentation, we can eliminate those circumstances that are not relevant and thus isolate the failure-inducing ones. The benefits of this approach include cutting away irrelevant information; it also typically results in shorter program runs which cover less of the code (as we will see in Chapter 9). A well known technique that automates this procedure by means of automated experiments is *delta debugging.* In the following, we shall see how JINSI applies delta debugging to minimize the interactions which lead to the failure (Section 4.1). Additionally, as we will see in the second part of this chapter (Section 4.2), it applies different *slicing techniques* to reduce the initial sequence of interactions to be minimized on the one hand (Section 4.2.1), and to further reduce the code executed by the minimal sequence of interactions on the other hand (Section 4.2.2).

4.1 Delta Debugging

Zeller and Hildebrandt devised a method called delta debugging, an automatic technique that narrows down a failure cause by systematically running automated experiments [72]. Beginning with a set of possible circumstances, delta debugging isolates—or, simplifies—the *failure-inducing circumstances,* which negatively affect the function of a program. Delta debugging automates the *scientific method of debugging:* The basic idea is to first establish a *hypothesis* on why something does not work; for instance that specific method calls with particular arguments have to be called in a certain order to produce the problem. Afterwards, you *test* this hypothesis, and you refine or reject it depending on the test *outcome*. Applied to method calls, this basically would mean to first select an initial sequence of method calls, and to run this sequence to test whether the problem in question occurs. If the problem is (re-)produced (i.e., the test *fails*), you would refine the sequence: you could remove some method calls, for instance. Afterwards, you would run the test again to assess the outcome of the now smaller sequence—which could be a more precise fault description at the same time. Otherwise, if the problem in question does not occur anymore (e.g., the test *passes* now because you have removed the method that directly triggers the failure), you would reject your hypothesis; for instance, by starting over with a different initial sequence of method calls. To automate the whole process, delta debugging in general only requires a test which assesses a given set of circumstances—in our case a given *sequence of method calls*.

However, it would not be sufficient if that test could differentiate between two test outcomes (*passing* and *failing*) only. Instead, it has to be able to handle a third outcome called *unresolved* as well. Fundamentally, delta debugging in some sorts implements a binary search: it tests the first half of a given set of circumstances and depending on the outcome, continues either with the first half, or proceeds with the second half. If the first half produces the failure, the failure-inducing circumstance has to be included there. Otherwise, if the failure is not reproduced, that circumstance has to be in the second half. While this might work when applied to a set of unrelated circumstances (and, where only one single element is failure-inducing)[1], it is easy to construct an exam-

[1]For instance, a set of numbers as input in that one single particular element is sufficient to cause the program to crash. Then, you could identify that very failure-inducing number by applying

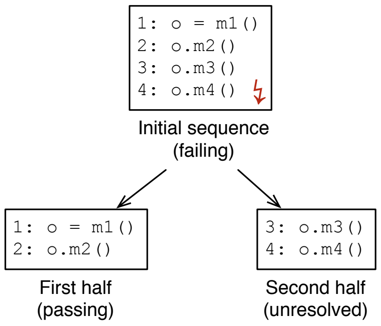


Figure 4.1: **Simple binary search fails for method calls.** If you would apply a binary search algorithm in order to simplify a failing sequence of method calls, you would get stuck because neither half would reproduce the original problem. The first half would just pass the test, while the second half could not be executed at all, because the receiver of the method calls to be assessed would not have been constructed; thus, nonexistent. The latter *unresolved* outcome is used by the delta debugging algorithm to extend binary search in order to provide a solution for that case.

ple where this will not work anymore. For instance, if you have a sequence of four method calls m_1, m_2, m_3, and m_4: constructor call m_1, which instantiates object o, followed by three further method calls m_2, m_3, and m_4, all executed on the initially instantiated object o. Furthermore, only executing the very last method call m_4 would crash the object. If you would test the first half of that sequence (thus, m_1 and m_2), the test would pass because m_4 would not be included. According to binary search, you would therefore test the second half

a binary search algorithm.

(m_3 and m_4). While this sequence would include the failure-inducing call m_4, you could not run it because you would have missed the crucial constructor call m_1 which would instantiate the receiver o in the first place. In this latter case, the test outcome would be *unresolved*—you can not clearly decide whether the outcome is passing or failing in the sense of the *original* failure; instead, a different error will occur. Hence, neither the first, nor the second half would reproduce the original failure in question (see Figure 4.1 on page 69 for an illustration). To deal with this problem, delta debugging actually extends binary search by splitting the set as soon as an unresolved outcome occurs, and thus performs a ternary, quaternary, or—generally—n-ary search if necessary. Figure 4.2 on page 71 shows the formal definition of the actual delta debugging algorithm that minimizes a given set of possible circumstances. For details and more definitions, see the work published by Zeller and Hildebrandt [72].

Zeller and Hildebrandt applied the algorithm on *program input:* delta debugging would repeat the failing program run while systematically suppressing parts of the input, eventually coming up with a simplified input where every remaining part is relevant for producing the failure. In this way, they have been able to minimize an initial 896-line HTML document that caused the Mozilla browser to crash to one single `SELECT` tag. Printing out this single line reproduced the failure in question, thus, crashed Mozilla. By simplifying input, delta debugging helps focus on suspicious code as well: having the minimized input at hand, the programmer can concentrate one those parts that are involved in printing that particular HTML tag, instead of going through all the statements that are related to the print feature in some way.

Unfortunately, applying delta debugging to input involves a number of non-trivial requirements. Firstly, we need a method to reproduce the run automatically. In the example above, this is a time-consuming task because the application has to be automated by simulating user interactions that load and attempt to print the simplified HTML input. Secondly, we need to control the input. In many cases, inputs are produced by other parts or external services. For instance, HTML pages typically are generated by servers. If Mozilla would

Let $\mathscr{C}$ be the set of all possible circumstances. Let $test : 2^{\mathscr{C}} \to \{✘, ✔, ?\}$ be a testing function that determines for a test case $c \subseteq \mathscr{C}$ whether some given failure occurs (✘) or not (✔) or whether the test is unresolved (?).
Now, let $test$ and $c_{✘}$ be given such that $test(\emptyset) = ✔ \wedge test(c_{✘}) = ✘$ hold.
The goal is to find $c'_{✘} = ddmin(c_{✘})$ such that $c'_{✘} \subseteq c_{✘}, test(c'_{✘}) = ✘$, and $c'_{✘}$ is 1-minimal.
The *Minimizing Delta Debugging algorithm ddmin(c)* is

$ddmin(c_{✘}) = ddmin_2(c_{✘}, 2)$ where

$$ddmin_2(c'_{✘}, n) = \begin{cases} ddmin_2(\Delta_i, 2) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\Delta_i) = ✘ \\ ddmin_2(\nabla_i, max(n-1, 2)) & \text{if } \exists i \in \{1, \ldots, n\} \cdot test(\nabla_i) = ✘ \\ ddmin_2(c'_{✘}, \min(|c'_{✘}|, 2n)) & \text{if } n < |c'_{✘}| \\ c'_{✘} & \text{otherwise} \end{cases}$$

where $\nabla_i = c'_{✘} - \Delta_i, c'_{✘} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n$, all Δ_i are pairwise disjoint,
and $\forall \Delta_i \cdot |\Delta_i| \approx |c'_{✘}|/n$ holds.
The recursion invariant (and thus precondition) for $ddmin_2$ is $test(c'_{✘}) = ✘ \wedge n \leq |c'_{✘}|$.

Figure 4.2: **In a nutshell: Delta debugging.** The minimizing delta debugging algorithm simplifies a given set of circumstances and thus yields a set of failure-inducing ones.

not had provided a method to load HTML data from a local file, we would have had to control a web server to systematically deliver parts of the initial data. Thirdly, we need knowledge about the input structure. While having knowledge about textual input may lead to performance improvements (for instance, by cutting down HTML data tag-by-tag instead of character-by-character), having no knowledge about the structure of non-textual data may render delta debugging practically useless, because this could easily trigger the delta debugging worst case complexity of $O(n^2)$ [72].

Next to input, delta debugging was applied on further circumstances, like failure causes in version histories [71] and program states [16]. However, all these methods cannot be applied on the failure that occurs in the motivating example: (1) the code producing the failure does not use any apparent input, (2) we do not know what previous version exactly would pass the test, let alone whether there is some version that would, and (3) compared to delta debugging on program states, *JINSI is much more robust*[2], and scales to applications with non-accessible state (e.g., native code or distributed applications).

Instead of the applications above, JINSI leverages delta debugging to *systematically suppressing parts of the program interactions* to narrow down the failure-inducing ones, as already exemplified in the motivating JODA TIME example (see Figure 1.1 on page 6 and Figure 1.2 on page 6). While delta debugging on input requires knowledge about the input structure and specific scaffolding to manipulate the input, *JINSI subsumes this as a special case,* since all input-processing methods are part of the interaction to be minimized. Delta debugging on version history requires this very history, whereas JINSI does not require any history at all. Finally, JINSI naturally utilizes the abstraction layers available in the program and thus does not depend on complex and large

[2]In 2004, Karsten Lehmann—as part of his diploma thesis—worked on DDSTATE, an ECLIPSE plug-in that applies delta debugging to program states [10], as had been described earlier in [68], and realized in the ASKIGOR debugging server [69]. While he was able to provide a rudimentary prototype, it was not possible to apply it on large programs. As it turned out, capturing, comparing, and especially transferring states of JAVA programs has been too tricky.

program states, as opposed to earlier work [10, 16].

In the remainder of this section about delta debugging, we will see how JINSI assesses individual delta debugging tests as either failing, passing, or unresolved. Furthermore, we will see how it handles outgoing interactions that are unexpected in the strict sense: because delta debugging tests *partial* sequences of recorded incoming interactions, program flow can be altered, and thus interactions different to the recorded ones can occur.

4.1.1 Predicates

As already mentioned above, the underlying delta debugging algorithm requires a testing function that determines for a given set of possible circumstances whether some given failure occurs or not, or whether the test is unresolved (see Figure 4.2 on page 71). In the context of JINSI, *a possible circumstance is an incoming interaction*: JINSI actively minimizes *incoming* interactions only, because it is only possible to appropriately suppress that kind of interaction—by omitting selected interactions exactly as the delta debugging algorithm does. In this sense, JINSI has control over the incoming interactions and is capable of either executing or excluding these interactions freely. As opposed to this, it is not possible to drop *outgoing* interactions directly—except for removing the corresponding statements from the code, which is beyond the scope of this work. JINSI rather has to provide a proper return value for each outgoing interaction, and hence makes use of the captured interactions for exactly this purpose.[3] Furthermore, interactions in general are inherently

[3]However, outgoing interactions *indirectly* caused by an incoming interaction are omitted automatically as soon as the corresponding incoming interaction is suppressed. Another way to influence the outgoing interactions would be to return different return values than the recorded ones; e.g., returning 0 instead of 13 to possibly break a loop earlier, or returning `false` instead of `true` to possibly change the predicate of conditional statements. Still, this would also be beyond the scope of this work.

well-ordered.[4] Hence, in this application of the algorithm, a set of possible circumstances and a sequence of incoming interactions are to be equatable. Delta debugging step by step simplifies that initial sequence—typically the complete sequence of captured incoming interactions—and eventually provides a minimal sequence which reproduces the original failure. JINSI provides a generally applicable testing function, which basically executes all the given incoming interactions sequentially, and finally returns whether the outcome was passing, failing, or unresolved. Depending on that return value, the delta debugging algorithm is able to decide how to proceed and when to terminate. The only thing that is missing to debug an individual failure is a *predicate* specially adapted for the failure in question. That predicate has to determine whether the failure in question is reproduced (*failing* outcome), or does not occur anymore (*passing* outcome); if something different happens (for instance, because JINSI suppressed a needed constructor call), the test will be classified as *unresolved*. Depending on the type of a bug's symptom—whether it is a crashing or a non-crashing bug[5]—JINSI uses different types of predicates.

4.1.1.1 Predicates for Crashing Bugs

For crashing bugs, JINSI *compares the exception thrown* to check whether the simplified run reproduces the original failure. In this we assume that a crashing bug causes an exception to be thrown, as it is the case for many computer languages that have built-in support for exceptions and exception handling, which includes the JAVA programming language. For other symptoms of crashing bugs, like segmentation faults, the same approach could be used as long as both a backtrace and some signal or similar distinct notification about the fault is available.

[4]Because the respective events are numbered consecutively using natural numbers, the interactions are well-ordered as a consequence.

[5]See Chapter 6 for details on the difference between crashing and non-crashing bugs.

```
Exception in thread "main" java.lang.ArithmeticException:
  Adding time zone offset caused overflow
at ZonedDurationField.getOffsetToAdd(ZonedChronology.java:357)
at ZonedDurationField.getDifference(ZonedChronology.java:339)
at BaseChronology.get(BaseChronology.java:260)
at BasePeriod.<init>(BasePeriod.java:100)
at Period.<init>(Period.java:441)
at PrecalculatedZone.create(DateTimeZoneBuilder.java:1439)
at toDateTimeZone(DateTimeZoneBuilder.java:398)
at JINSI.TestDriver.main(TestDriver.java:37)
```

Figure 4.3: **Exception while constructing new time zone.** Running the code from the API example (see Listing 1.1 on page 6), JODA TIME crashes with an `ArithmeticException` caused by an integer overflow.

In our motivating example, an `ArithmeticException` is thrown, because adding a time offset caused an integer overflow (Figure 4.3 on page 75 shows the actual exception). As long as a simplified, smaller subset reproduces the *same* exception, the failure in question is *reproduced* and delta debugging can continue to further minimize this set. Two exceptions are considered equal if they have the same type (thus, describe the same error or exceptional event), the same message, and the same location. In other words, the original failure's *context* has to be retained. Note that the stack trace data does not have to be the same, because JINSI replaces classes on the stack as needed during the debugging process (see Chapter 5 and Section 6.2.2), and therefore the backtrace may contain different classes than in the original, observed run.

A *different* exception results in an *unresolved* outcome, just as any other problem detected by JINSI during applying the delta debugging algorithm. For instance, a simplified sequence of incoming interactions might not contain an essential method call like a setter method which would initialize some field `f` on object `o`, and therefore a subsequent method call on `o` could cause a `NullPointerException` because `f` would not have been initialized; or, due to the altered (simplified) sequence the program flow within an observed object might be changed, and therefore an outgoing interaction could occur that would not have been captured before (JINSI could not provide a proper return value; Section 4.1.2 shows how it handles unknown program flow).

4.1.1.2 Predicates for Non-Crashing Bugs

For non-crashing bugs, where an exception in particular, and a backtrace in general are not available, JINSI uses alternative predicates both on further *externally observable properties* and *internal program state* (see Section 6.2 for details on how JINSI derives such predicates fully automatically). For instance, JINSI uses predicates on output on the console like "The output contains `"result: -1"`.", and properties on attributes of objects like "Attribute `name` of object with id 13 has value `"UTC"`." Similar to crashing bugs, as long as a smaller, simplified subsequence reproduces the *same* observable failure—this time characterized by a predicate on observable properties in general[6]—the originally observed failure is *reproduced* and delta debugging will continue to try to further minimize the incoming interactions which lead to that failure. This enables JINSI to also *debug failures other than crashes caused by exceptions*, as we will see in detail in Chapter 6.

4.1.2 Synchronization

While applying the delta debugging algorithm, JINSI has to handle outgoing interactions that are *unexpected* in the strict sense: because delta debugging tests *partial* sequences of recorded incoming interactions, both program state and control are likely to vary in the consequent program run, and thus outgoing interactions different to the recorded ones can occur. For instance, a

[6]In fact, an exception principally is a special case of an externally observable property that characterizes a failure. However, as we will see in Chapter 6, using an exception, if available, and its backtrace helps find the failure cause efficiently because stack traces tend to give hints about the actual failure cause.

loop could be executed less often, or the predicate of an if-then construct could evaluate differently. In both cases, the program would follow a different path in the control flow graph. Thus, it probably would execute either completely different statements, or a subsequence—the program run being simplified and the recorded sequence of interactions would diverge and would be out-of-sync eventually. If JINSI would respond with an unresolved test outcome in all such cases, delta debugging would require a higher number of tests, which could result in a significant longer processing time.[7] As we will see in this section, it solves this issue by providing *corresponding return values* if available, and tries to *re-synchronize* between the program run being simplified and the sequence of recorded events.

Basically, if the program follows a path in the control flow graph that is a shortcut (as it may be the case if a loop is executed less often) of the original one as occurred during capture, JINSI will search for the corresponding outgoing interactions in the event log by skipping those that would not occur during replay. Otherwise, if the actual path follows an unknown alternative (e.g., if the program follows an else branch which has not been executed at all during capture), JINSI will respond with an unresolved test outcome, because it would not know the requested return values.

Listing 4.1 on page 78 shows simple[8] JAVA class `NumberWriter` that writes a numerical sequence $1,2,\ldots n$ to a previously specified output file. If method `setNumbers(n)` is called, the given number n will be used as endpoint of that sequence; otherwise, no numbers will be written at all. Setter method `setFile()` specifies the output file; while a call to `setNumbers()` is optional, method `setFile()` must be called before producing any sequence, because only this method specifies the file where the numbers will

[7]If every test fails, the algorithm will converge in logarithmic time; if all tests are unresolved or passing, it will require quadratic time [72].

[8]For the sake of simplicity, we spare, for instance, error handling like both the usual try-catch-block that would surround all the calls to the internal `FileWriter` instance, and a better solution than just printing a warning if the output file has not been set.

```java
class NumberWriter {
  String file = null;
  int numbers = 0;

  NumberWriter() { /* empty constructor */ }

  void setFile(String file) { this.file = file; }

  void setNumbers(int num) { this.numbers = num; }

  void writeNumbers() {
    if (this.file == null) {
      System.err.println("Output␣has␣not␣been␣set.");
      return;
    }

    FileWriter writer = new FileWriter(this.file);
    for (int n = 1; n <= this.numbers; n++) {
      writer.write(n);
    }
    writer.close();
  }
}
```

Listing 4.1: **Example program for varying program flow.** Class `NumberWriter` writes a numerical sequence to an output file. If that file has not been set via method `setFile()`, the class will issue an error (conditional statement). Method `setNumbers()` sets the sequence's endpoint. Method `writeNumbers` writes the sequence to the output file. For this purpose, it uses a loop that counts from 1 to the endpoint of the sequence (for loop / iteration statement). Depending on the evaluation of the predicate and the loop counter, the program will follow a different execution path.

be written to. Method `writeNumbers()` finally writes the sequence to the specified file. For this, it first checks whether the output file was specified; if not, it will issue an error to the console and will return immediately. Otherwise, the method will create an instance of class `FileWriter`, which will be used to write the individual numbers to the output file. Then, a loop iterates over all numbers $1 \ldots n$ by counting, and writes each number to the file writer instance. Eventually, that writer will be closed properly, and method `writeNumbers()` returns. For example, when first calling both `setFile("output.txt")` and `setNumbers(2)` on an instance of `NumberWriter`, a subsequent call to method `writeNumbers()` will write numbers 1 and 2 to the output file.

When JINSI regards class `NumberWriter` as observed and captures the above incoming constructor call and the three subsequent method calls, JINSI will capture a sequence of interactions as shown in Figure 4.4 on page 80. It will accordingly capture eight events in total[9]; of these, the first four are incoming call events (events 1 to 4), and the remaining four are outgoing ones (events 5 to 8). While the first three incoming events do not contain any further outgoing interaction, the event representing the call to `writeNumbers()` involves four outgoing calls: the `FileWriter` object is created (event 5), numbers 1 and 2 are written (events 6 and 7), and eventually the writer is closed (event 8). When replaying the complete sequence of incoming interactions (thus, events 1 to 4), JINSI will intercept the four outgoing calls and will provide the captured return values.[10]

When simplifying this sequence of four incoming interactions[11] while ap-

[9]To simplify matters, we ignore the corresponding returning events of both the incoming and outgoing calls.

[10]If any; in this concrete example, JINSI will return a placeholder for the instance of class `FileWriter`, and will only receive the outgoing method calls to again return control to the observed caller without the need to provide any return value.

[11]For simplicity, we go without a concrete failure to minimize, as such a sequence can occur as subsequence of a longer sequence to be minimized.

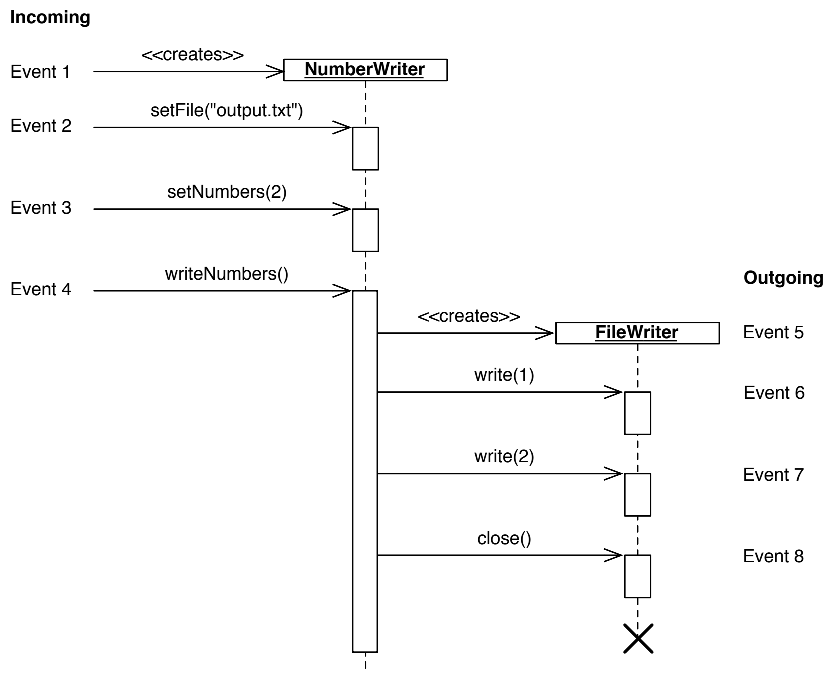


Figure 4.4: **Example interactions for varying program flow.** When running class `NumberWriter` in Listing 4.1 on page 78 by calling (event 1) its constructor, methods (2) `setFile()`, (3) `setNumbers()`, and (4) `writeNumbers()`, the sequence of interactions shown in the above sequence diagram will occur. The last call triggers four further outgoing interactions (5 to 8) to an instance of class `FileWriter`. Depending on whether the individual setter methods are called, a different sequence of interactions will occur.

plying delta debugging, the two—in this context—interesting subsequences $(1,\mathbf{3},4)$ and $(1,\mathbf{2},4)$ could be tested. In the first case (the call to method `setFile()` would be omitted while `setNumbers()` would be called), JINSI would have to assess the test with an *unresolved* outcome, because the predicate in the if-clause (line 14 in Listing 4.1 on page 78) would evaluate to true, and the instance of class `NumberWriter` would consequently follow the branch that would issue the error message. However, because JINSI does not have any information about possible interactions in this branch (as it was not executed during capture), it neither could return a proper placeholder object for the outgoing field access to `System.err`, nor could it handle the subsequent outgoing method call to `println()`. In the other case (method `setFile()` would be called, but method `setNumbers()` would not), JINSI would find the corresponding outgoing interactions by skipping those that occurred within the for-loop. Because method `setNumbers()` would not be called, field `this.numbers` would be set to 0, and therefore the instance of `NumberWriter` would not execute the for-loop at all. Instead, the call to the constructor of class `FileWriter` would be immediately followed by the call to method `close()`—the latter is known to JINSI, and JINSI would continue with this very interaction. In this way, JINSI is able to handle shortcuts of the original execution path as they frequently occur during delta debugging. Algorithm 1 on page 82 illustrates the algorithm as used by JINSI to *synchronize* outgoing method calls. The same fundamental idea is also applied to synchronize all other outgoing interactions, like filed accesses; the concrete algorithm is adapted to the different properties of each respective event.

During delta debugging and in contrast to replaying strictly sequentially[12], JINSI checks for every outgoing interaction whether replay *is still in sync*. For

[12]In addition to applying delta debugging, JINSI is able to replay a captured sequence of interactions in exactly the same order as the respective events occurred during capture. In this mode, any interaction that is out-of-sync would hint at a probable fault either in capture or replay. Therefore, when JINSI replays the given events strictly sequentially, it will not try to re-synchronize, but will issue an error instead.

```
let $s$ be the sequence $(i_0, \ldots, i_n)$ of all $n+1$ captured interactions
let $i_{inc}$ be the incoming interaction currently being executed
let $i_{ret}$ be the corresponding returning interaction of $i_{inc}$
// Thus, $0 \leq inc < ret \leq n$ holds, because $\mathrm{i}_{\mathrm{inc}}$ has to be followed by $\mathrm{i}_{\mathrm{ret}}$
let $m_{out}$ be the method's name of the currently requested outgoing method call
let $p_{out}$ be that method's parameter list
let $t_{out}$ be the type of that call's receiver
// We store the event last used for synchronization
if synchronizing for the first time then
    $i_{last} \leftarrow i_{inc}$
end if
for all interactions $i_{cur}$ in subsequence $(i_{last+1}, \ldots, i_{ret-1})$ do
    if $i_{cur}$ is not an outgoing method call then
        continue
    end if
    let $m_{cur}$ be the method name of $i_{cur}$
    let $p_{cur}$ be the parameter list of $i_{cur}$
    let $t_{cur}$ be the target type of $i_{cur}$
    if $m_{cur} = m_{out} \wedge p_{cur} = p_{out} \wedge t_{cur} = t_{out}$ then
        $i_{last} \leftarrow i_{cur}$
        return $i_{cur}$
    end if
end for
return unresolved
```

Algorithm 1: **Finding alternatives to synchronize outgoing interactions.** To synchronize between the sequence of recorded interactions and varying program flow during delta debugging, JINSI uses the above algorithm. While that algorithm is specialized on method calls, the same fundamental idea is applied to all possible types of outgoing interactions (e.g., field accesses). The algorithm ensures that an alternative event is found in the appropriate context, and that each one is utilized only once to avoid infinitive loops otherwise caused by synchronization.

this purpose, it checks the respective properties for each type of interaction: the tool compares the properties of the currently outgoing, intercepted interaction and the recored interaction that is next. If their properties match, replay is in sync; otherwise, JINSI tries to *re-synchronize* as exemplified above. To track which recorded event is next, JINSI uses an event counter—similar to a program counter—which indicates where replay is in its recorded event sequence.

For instance, JINSI checks for each outgoing method call whether the location in code, the method name, its parameter types, and the runtime type of the receiver match between the currently intercepted call and the recorded one. If both interactions match, JINSI basically will retrieve the recorded return event, will provide a proper return value (if any), and eventually will return control to the observed caller. If they do not match, JINSI will search for a proper event in the subsequence between the incoming interaction currently being executed and its corresponding returning event (every outgoing interaction is triggered by an incoming one). For this, the tool iterates over all the events contained in that subsequence, and again compares their properties as described above, with the exception that it does not check the location this time: already during implementation and early testing it turned out that comparing the exact location is too restrictive. In the vast majority of all cases JINSI was not able to find an alternative, proper event, because frequently program flow is changed in a way that JINSI could not relay on interactions recorded at the very same location.[13] Instead, JINSI tries to find an alternative event *within the same context*, which is defined by the circumscribing, triggering incoming interaction. Thus, if it is able to find a proper event that occurred within the same triggering interaction, it will use that one to obtain a proper return value. If that alternative value is completely wrong, the program will usually be totally out-of-sync—thus, JINSI will not be able anymore to find an alternative event—at a later point, and JINSI will assess the current test as unresolved. Otherwise, if the original failure can

[13]For instance, the predicate of an if-then statement evaluates to false instead of true during minimization, causing control to jump to the end of that if-then statement.

be reproduced while providing that alternative value, the choice will have been proper for our purpose.

In addition to the context, alternative events used for re-synchronization are subject to one further restriction: in order to avoid infinite loops, each alternative event is used only *once* during each delta debugging test. For instance, if the condition of a while loop uses an outgoing method call in its evaluation, and the alternative event repeatedly used for that call would return true, that loop condition would always evaluate to true, and thus the program may loop endlessly. However, because alternative events are used only once, it is guaranteed that such infinite loops—at least such caused by synchronization—can not occur.[14]

As we have seen, JINSI applies delta debugging to minimize the interactions which lead to the failure. That underlying algorithm isolates the failure-inducing interactions by means of automated experiments, and requires both a testing function that determines the test outcome for a given sequence of possibly failure-relevant interactions, and a predicate that is specially adapted for the failure in question. JINSI provides a suitable testing function as well as predicates that, depending on the type of a bug's symptom, classify the outcome. In order to be capable of handling outgoing interactions that are unexpected due to the fact that delta debugging tests parts of the whole sequence of captured interactions, JINSI provides a technique to re-synchronize between the program run being simplified and the sequence of recorded events.

In the second part of this chapter, we shall see how JINSI applies different slicing techniques to first reduce the initial sequence of interactions to be minimized, and to further minimize the statements executed by the minimal test driver as determined by applying delta debugging.

[14]Of course, infinite loops induced by other causes are not prevented by that technique.

4.2 Slicing Techniques

In automated debugging, *program slicing* [64] (or *slicing* for short) is a widely spread technique used to narrow down those source code statements that could possibly have contributed to a failure. Thus, slicing helps programmers—similar to delta debugging, which, however, is more universal—focus on those statements that are relevant for a failure. More generally, slicing uses an algorithm that yields for a given program and statement of interest c, the *slicing criterion,* a *slice* by following data and control dependencies; see Figure 4.5 on page 86 for formal definitions of data and control dependencies, and Figure 4.6 on page 87 for an illustration. That slice is the subset of all program statements that only contains those statements that either are influenced by c, or that influence c. The former slice is called *forward slice,* the latter one *backward slice*. Figure 4.7 on page 88 contains a formal definition of forward and backward slices.

While forward slicing is rarely used in automated debugging, backward slicing is more common: basically, forward slicing yields all statements that are influenced by c after executing c, and thus gives a diagnosis of *the future* after something has just happened; in contrast, backward slicing yields all statements that had an influence on c, and thus gives hints about possible failure causes in *the past*. Usually, in automated debugging we are interested in the latter: where does this incorrect value come from? Indeed, JINSI uses slicing techniques that are related to backward slicing: it implements a slicing algorithm that yields a backward slice on the recorded *events,* and directly takes advantage of *dynamic* backward slicing to filter the minimized program run.

Slicing can be applied either *statically* or *dynamically*. Static slicing applies to a given program and analyzes the source code of that program to make statements about the program in general—about all possible program runs. By contrast, dynamic slicing analyzes one concrete program run and its specific input; thus, it yields properties of a specific program execution and therefore is more precise. For this reason, *dynamic backward slicing* is well suited to nar-

Data dependency: A statement S_2 is *data dependent* on a statement S_1 if

- S_1 writes some part P of the program state that is being read by S_2, and
- there is at least one path in the control flow graph from S_1 to S_2 in which P is not being written by some other statement.

Thus, the outcome of S_1 can influence the data read by S_2.

Control dependency: A statement S_2 is *control dependent* on a statement S_1 if S_2's execution is controlled by S_1. Thus, the outcome of S_1 determines whether S_2 will be executed.

Dependence graph: The data and control dependencies form the so-called *program dependence graph* $G_P = (V, E_D, E_C)$ with

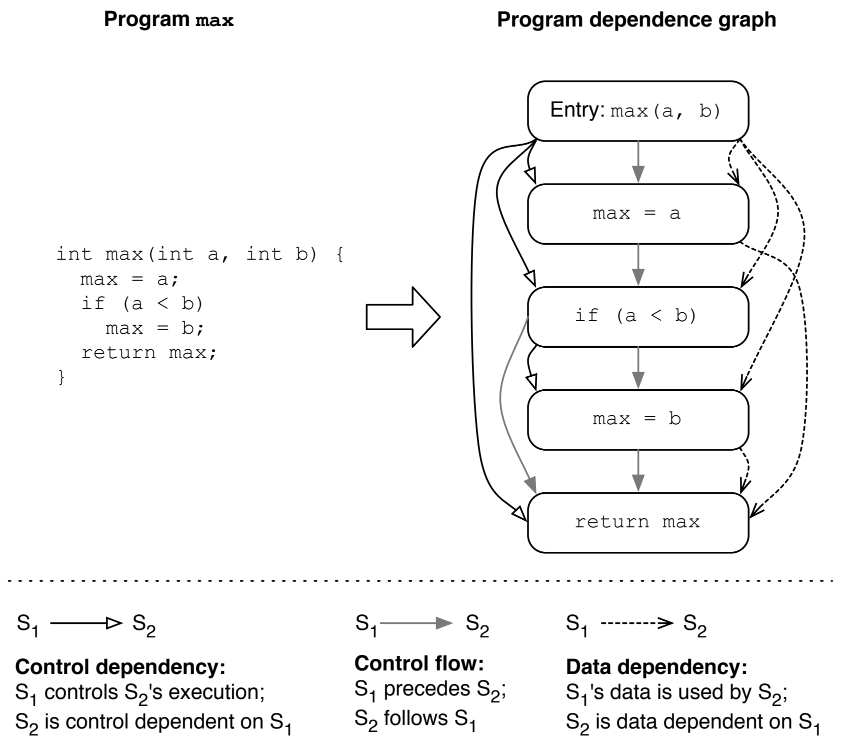
- $V = \{S \,|\, S \text{ is a statement with data or control dependency on}\}$, and
- $E_D = \{(S_1, S_2) \,|\, S_1, S_2 \in V \wedge S_2 \text{ is data dependent on } S_1\}$, and
- $E_C = \{(S_1, S_2) \,|\, S_1, S_2 \in V \wedge S_2 \text{ is control dependent on } S_1\}$.

Thus, the vertices V represent all the statements with some dependency on, and the edges E_D and E_C represent actual data and control dependencies, respectively; using two sets for the edges allows to distinguish between data and control dependencies. See Figure 4.6 on page 87 for an illustration of the program dependence graph.

Furthermore, the *dynamic dependence graph* is derived from one single program run. Thus, it does not contain possible, but actual dependencies as occurred in that concrete run.

The definitions of the data and control dependencies are freely adapted from Zeller's book on systematic debugging [70].

Figure 4.5: **In a nutshell: Data and control dependencies, and program dependence graph.** Statements in a program can depend on each other via data or control dependencies. These dependencies can be used to build the program dependence graph, which various program analysis techniques base on.



```
int max(int a, int b) {
  max = a;
  if (a < b)
    max = b;
  return max;
}
```

$S_1 \longrightarrow S_2$

Control dependency:
S_1 controls S_2's execution;
S_2 is control dependent on S_1

$S_1 \longrightarrow S_2$

Control flow:
S_1 precedes S_2;
S_2 follows S_1

$S_1 \dashrightarrow S_2$

Data dependency:
S_1's data is used by S_2;
S_2 is data dependent on S_1

Figure 4.6: **Program dependence graph.** Program `max` is exemplarily transformed into a program dependence graph. The illustration also contains the control flow which is used to derive the data dependencies (see definitions in Figure 4.5 on page 86). However, the control flow graph is not part of the program dependence graph proper, which includes data and control dependencies only.

Forward Slice. Given a program statement c, the forward slice S^F contains all those statements s whose read variables or execution could be influenced by slicing criterion c:

$$S^F(c) = \{s | c \rightarrow + s\}$$

Backward Slice. Given a program statement c, the backward slice S^B contains all those statements s that could influence the read variables or execution of slicing criterion c:

$$S^B(c) = \{s | s \rightarrow + c\}$$

The symbol $\rightarrow +$ stands for the transitive closure on the control and data dependencies.

Figure 4.7: **In a nutshell: Forward and Backward Slices.** The slice on slicing criterion c is defined by the transitive closure.

row down those statements that could possibly have had influence on a given program failure that manifests itself in a particular program run.

In the next section, we will see how JINSI applies a dynamic backward slicing technique to reduce the initial sequence of interactions to be minimized. In this way, it indirectly reduces the number of tests the delta debugging algorithm has to run to narrow down the failure-inducing interactions. As we will see, this preceding filtering can result in a significant speed up of the delta debugging part of the JINSI process (see Figure 3.1 on page 26), while—compared to the overall archived speed up—the time spent for slicing the events can be ignored.

4.2.1 Event Slicing

The computation time JINSI has to spent for applying delta debugging depends on the number of *unresolved* test outcomes: if every test fails, it converges in logarithmic time; if all tests are unresolved or passing, it requires quadratic time [72]. Thus, in the best case delta debugging behaves like a logarithmic binary search algorithm and executes in $O(\log n)$ time. However, with many tests having an unresolved outcome, it tends toward executing in $O(n^2)$ time.

Because JINSI observes the interactions of individual objects, and a large number of different instances can occur during one single observed program run, JINSI may capture a large number of different observed objects. Thus, the sequence of captured interactions will contain a correspondingly large number of incoming constructor calls. For instance, the recorded event log of test subject `COL-1` (see Chapter 9) contains 38,031 interactions in total which include 401 incoming constructor calls. While this number seams to be relatively small, even that amount can cause a very long computation time. Without event slicing, applying JINSI takes almost 4 hours[15] because of the large number of 4,578 tests—caused by many unresolved tests due to missing incoming constructor calls. Because the delta debugging algorithm is generally not aware of any structure of the circumstances to be tested, it does not know about the relations between the objects and interactions, respectively, involved in a subsequence to be tested. Consequently, the algorithm tests *arbitrary* compositions of interactions, and thus also produces subsequences that will result in an unresolved test outcome in any case, because some crucial interaction—e.g., an incoming constructor call—is missing from the start. Thus, in addition to this issue related to missing constructor calls, there are many more possible causes for unresolved outcomes. For instance, a method call that initializes or sets a value required for some computation may be missing. Therefore, it is best to filter the interactions to be minimized beforehand wherever possible.[16] In the above example, a total number of 401 observed objects is created.

[15]All experiments were conducted on a MacBook machine built in 2008 with a 2.1 GHz Intel Core 2 Duo processor and 4 GB memory.

[16]Note that it would be possible to extend the delta debugging algorithm by a so-called *splitter* that splits any given set of circumstances into n new sets [7]. While the default implementation of the algorithm splits the given set into *n equally sized* sets, a more sophisticated implemen-

However, only one is *directly* connected to the failure, the one whose called method throws the exception causing the crash. All the other 400 observed objects might be connected to the failure, but only if they are related by data dependencies; they might be totally unrelated just as well.

JINSI applies a *dynamic slicing technique* before it uses delta debugging (see Figure 3.1 on page 26), to filter out those captured events that could not have had any influence on the failure—it throws away all events that are not related to the failure for sure. In this way, it reduces the initial number of interactions to be minimized directly, and the number of tests to be carried out indirectly. JINSI thus reduces the number of potentially unresolved test outcomes and hereby speeds up the delta debugging process. In the example above, with event slicing applied before, the delta debugging algorithm has to run only 4 tests, and the whole process takes only 68 seconds—in this case event slicing yields a tremendous speed-up.

The slicing technique used for this purpose was inspired by the work on efficient test case minimization by Leitner et al. [45]. However, in contrast to applying static program slicing to the program code, *JINSI slices the captured sequence of interactions.* Hence, as already mentioned briefly before, JINSI applies dynamic slicing on the captured interactions of a specific, observed failure. Figure 4.8 on page 91 gives a formal definition of the so-called *event slice*[17]. The key idea of using slicing to filter interactions is straightforward. We start from the event of interest directly connected to the failure. By following back data dependencies, we establish the subset of the events (i.e., the event slice, or slice for short) that possibly could have influenced the initial event of

tation could take advantage of information on the used circumstances and their structure to produce a valid set of circumstances. For instance, we could ensure to include missing incoming constructor calls automatically. However, this would only solve the issues related to that kind of interactions, and other issues related to, for instance, missing setter methods would remain. Thus, event slicing is a more general approach, which might be complemented by a more sophisticated splitting of interaction sequences.

[17]Because that slicing technique works on the recorded *events,* we decided to call it event slicing, rather than interaction slicing.

Event Slice. Given a captured event of interest c called *slicing criterion,* the event slice S^E contains all those events e that could transitively influence the variables read by, or the execution of event c:

$$S^E(c) = \{e | e \to + c\}$$

Thus, similar to program slices, the event slice is described by the transitive closure on c. However, here the transitive dependencies are produced by *possible* data dependencies.

Figure 4.8: **In a nutshell: Event Slice.** Event slicing yields all interactions that had an influence on an interaction of interest c, and thus filters out all interactions and events, respectively, that could not have had any influence on c.

interest. Any event that is not in the slice can consequently be removed.

More concretely, by transitively following back possible data dependencies on the captured sequence of *incoming* interactions recorded as events, we establish a list of possibly relevant interactions that describes the *constructions and usages* of all observed objects that are involved in the captured sequence of interactions that reproduces the failure. We have to settle for *possible* data dependencies because we do not know for sure whether an event actually produces a data dependency or not: We only know whether an object's data *might* has been used by an interaction because we only see the outer shell of this interaction[18], but we do not know the interaction's effect on the receiver object in detail, as we lack information about the concrete respective computation and the object usage involved in that interaction. Thus, a (possible) data dependency on object o may be produced by event e and its corresponding

[18]For instance, for a method call we know the receiver, the passed arguments, and the returned value (if any).

interaction, respectively:

1. If o was *returned* by e. For instance, an incoming constructor call could create o. We also have to include method calls that return object o, because the method might have modified that object.

2. If o was passed as *argument* to e. That interaction could have modified the object's data.

3. If o was the actual *receiver* of e; for instance, o as callee of an incoming method call could be modified by this very call.

In these cases, we add e to the event slice and regard o as further object of interest. While we could use different static and dynamic analyses to find out whether an interaction in fact produces a data dependency, it turned out that the approach's uncertainty actually has not become important.[19]

For example, to slice the events captured while observing a crashing bug we start at the incoming method call that throws the exception causing the crash, and we put all objects involved in this interaction in an initial set. This includes the callee object, the returned object (if any), and all non-primitive arguments. We then transitively include all events (and again the objects involved as described above) where objects in the initial set are involved. We thus obtain those interactions that can actually affect the execution of the last interaction—the ones that make the program fail. Figure 2 on page 94 provides a high-level description of the event slicing algorithm. The algorithm shown is

[19]The obvious downside is that our event slice still contains events that are not relevant for the failure. For instance, calling a pure method whose return value is not involved in other interactions of interest can not be connected to the failure (because a pure method does not change the callee's state). However, the current approach is good enough; the downstream delta debugging identifies such method calls as irrelevant at the latest. In fact, we applied Dallmeier's tool for dynamic purity analysis for JAVA programs [21]. However, while applying a purity analysis indeed can omit some further events from the slice, during experimentation it turned out that applying purity analysis, all things considered, does not pay off.

a straightforward version which clarifies the basic principle of following possible data dependencies between events produced by objects shared by those events—JINSI actually uses optimizations that speed up the computation (see below, Section 4.2.1.1).

Figure 4.9 on page 95 exemplifies the above event slicing algorithm on a sequence of eight incoming interactions. Three different XML elements `x`, `r`, and `c` are created, and each one gets its name set. Furthermore, element `c` is added as child to element `r`. Eventually, method `getNumberOfChildren()` is called on element `r`. One would expect that this method call would return 1; however, in this example the method throws an exception and thus produces a crash. Starting at that last interaction, event slicing first adds corresponding event 8 to the slice and object `r` involved in that interaction to set `depends`, and then iterates backwards over all events. Events 6, 4, and 3 are added to the slice because they contain possible data-dependencies on object `r` which was initially involved in event 8. Event 5 is also added in the first iteration because it is related to previously added event 6: they both have object `c` in common and thus have a possible data-dependency. In the second iteration, the algorithm includes event 7 because now set `depends` contains object `c`, which was not regarded as of interest at this point in the first iteration; it was added to `depends` only when the algorithm iterated over event 6. In the third an last iteration, the algorithm does not find any event to be added to the slice and eventually terminates. Events 1 and 2 are not included in the final slice $\{3,4,5,6,7,8\}$ because they are not related at all to any event contained in the slice. On the assumption that setting the elements' name is not related to the failure—counting the children may fail even if all the elements do not have a name—the downstream delta debugging algorithm would further remove events 4 and 7, and will thus reduce the initial sequence of interactions to $(3,5,6,8)$.

```
let interacts be a sequence of incoming interactions
let criterion be the interaction of initial interest
slice ← {criterion}
// callee, arguments, and return value of criterion
// initiate set of possible data dependencies
depends ← get_objects(criterion)
repeat
    slice_size ← size(slice)
    for all events e in reverse(interacts) do
        for all objects o in depends do
            if o is (target ∨ argument ∨ return value ) of e then
                slice ← slice ∪ {e}
                depends ← depends ∪ get_objects(e)
            end if
        end for
    end for
until slice_size = size(slice)
return slice
```

Algorithm 2: **Event slicing.** Similar to dynamic backward slicing, event slicing yields all interactions that could have previously influenced a given interaction of interest c (called slicing criterion). Starting with c, the algorithm repeatedly scans all interactions backwards and joins those which have a possible data dependency on the slice established by then. A data dependency between two interactions will be given if the two use the same object; either as callee, argument, or return value. The slicing algorithm terminates as soon as a fixed point is reached.

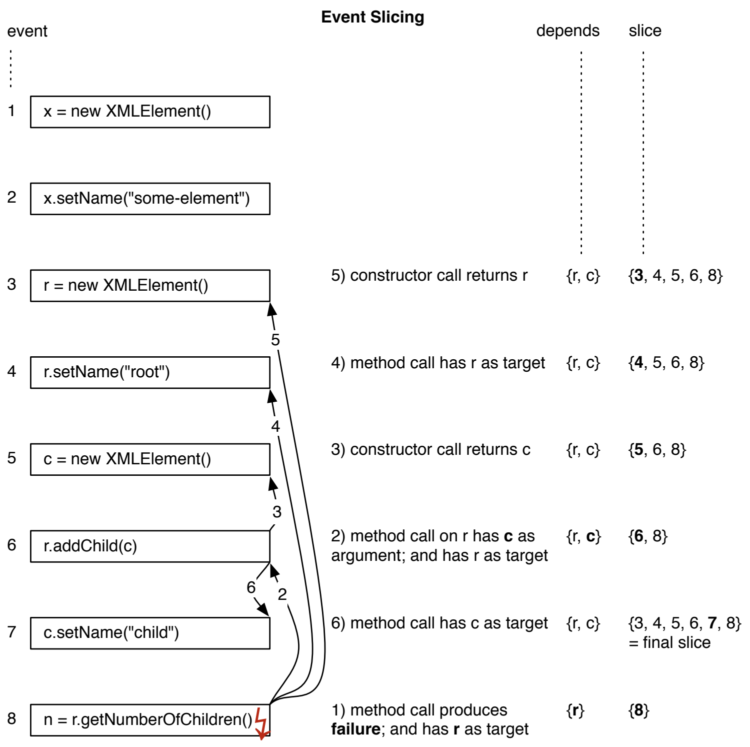

Event Slicing
event
depends
slice
1 x = new XMLElement()
2 x.setName("some-element")
3 r = new XMLElement()
4 r.setName("root")
5 c = new XMLElement()
6 r.addChild(c)
7 c.setName("child")
8 n = r.getNumberOfChildren()
5) constructor call returns r {r, c} {3, 4, 5, 6, 8}
4) method call has r as target {r, c} {4, 5, 6, 8}
3) constructor call returns c {r, c} {5, 6, 8}
2) method call on r has c as argument; and has r as target {r, c} {6, 8}
6) method call has c as target {r, c} {3, 4, 5, 6, 7, 8} = final slice
1) method call produces failure; and has r as target {r} {8}

Figure 4.9: **Example for event slicing.** Event slicing filters all events that are transitively involved in construction and usage of the objects directly used by an event of interest—usually the event directly involved in the failure. In this example, event 7 is only included in the slice eventually because event 6 was included before. Thus, it would not be sufficient to sequentially process over all events only once. In the end, events 1 and 2 can be dropped before applying delta debugging.

4.2.1.1 Efficient Implementation

Basically, instead of iterating over all events repeatedly until a fixed point is reached[20], JINSI first creates a directed graph that represents all inter-event dependencies, and then obtains the connected subgraph that contains the slicing criterion—this subgraph contains exactly the event slice. The graph can be constructed by iterating over all events only once; its vertices represent the events, its edges possible data dependencies. To determine the subgraph which represents the slice, JINSI transitively follows all edges (dependencies) starting at the slicing criterion by doing a breadth first search. The resulting subgraph's set of vertices contains exactly those events that are in the event slice.

While using a graph structure to represent all the dependencies and to compute the slice results in time complexity $O(|E|+|V|)$—since every vertex and every edge will be explored in the worst case—and thus, in theory, again in $O(n^2)$ as $|E| = O(|V|^2)$, it turned out during experimentation that this approach outclasses the above naive one in practice. Using the naive approach, we were not able to compute the event slice at reasonable cost, especially in terms of computing time. As opposed to this, we were able to compute the event slice within seconds to a few minutes using the graph based approach. However, that latter approach entails an issue: the built graph easily contains hundreds of millions of edges, and thus involves memory problems. To deal with this issue, JINSI uses an *adjacency matrix* consisting of bit arrays[21] to represent which

[20]Thus, the basic slicing algorithm in Figure 2 on page 94 is in $O(n^2)$. Having sequences of captured events with several tens of thousands of elements, this poses a serious issue regarding the runtime of the slice computation. JINSI initially implemented that naive algorithm; as it turned out quickly during early experimentation, that approach was computationally too inefficient.

[21]Also known as a bitset. We actually use uncompressed bitsets as provided by the JAVA runtime environment. Because uncompressed bitsets are wasteful data structures for sparse sets, we also tried different implementations of compressed bitsets, like word-aligned hybrid compressed bitsets. However, as it has turned out, the spatial advantages were outweighed by runtime disadvantages.

vertices (events) of the used graph are adjacent to (depend on) which other vertices: because each entry in the adjacency matrix requires only one bit, they can be represented in a very compact way; besides avoiding wasted space, this compactness also encourages locality of reference which enables the computer to cache the data more efficiently.

Using a graph represented by an adjacency matrix enabled us to apply event slicing to event logs containing tens of thousands of elements on a laptop computer with only 4 GB memory; thus, no especially powerful machine is required to run the analysis in a reasonable amount of time. Furthermore, applying event slicing in general helps reduce the number of tests the delta debugging algorithm has to execute, and thus reduces the total runtime of applying JINSI on a failure.

In the motivating example, applying the event slice to the 14,629 incoming interactions takes 24 seconds and 1,940 interactions remain. The downstream delta debugging needs 38 seconds and 39 tests to minimize these interactions until the two failure-inducing ones remain (see Listing 5.1 on page 106). Without event slicing, delta debugging would have taken 102 tests and two minutes. With event slicing, JINSI needs less than one minute for the entire process. While it is not guaranteed that applying event slicing yields a significant speed up, we observed some cases where event slicing was able to filter out unrelated events for the most part (see Chapter 9 for details), and JINSI was able only in this way to minimize the initially captured sequence of interactions within minutes, instead of hours.

In the remainder of this section, we will see how JINSI applies dynamic backwards slicing on the minimized program run to further reduce the amount of program statements that are relevant to the failure.

4.2.2 Dynamic Slicing

After applying event slicing and delta debugging to the captured program run, JINSI finally applies *dynamic program slicing* using Hammacher's dynamic slicer for JAVA [34]. While delta debugging computes the minimal sequence of interactions that reproduce the failure, the subsequently applied dynamic backward slice computes all statements that possibly could have *influenced the failure within the minimal run*. In this way, JINSI reduces the number of lines to be inspected by the developer still more. For instance, in the motivating example, the minimized run covers 193 lines, whereas the slice within this run contains only 54 lines.

JINSI applies delta debugging on the captured interactions and therefore minimizes on the *method level* at that phase. That granularity consequently provides a diagnosis not finer than on the method level: it basically provides a set of methods[22] that are related to the failure—to reproduce the failure, one has to call these methods. Additionally, that set also provides a set of source code statements that have contributed to the failure: exactly those statements that are executed by the minimized method calls. However, these method calls usually also cover lines that are not strictly related to the failure, but that are technically required to replay the minimized program run. For instance, a failure-inducing constructor call may initialize fields that are not strictly required to reproduce the failure—it may initialize a cached hash code[23]—but by calling that constructor we automatically execute all its statement[24]. However, we can neither selectively execute nor minimize single statements within a method because JINSI applies delta debugging exactly on the method level.

In contrast, dynamic program slicing yields all statements that could have contributed to the failure. Thus, it provides a diagnosis on the *statement level,*

[22]One can easily derive that set from the minimized sequence. Strictly speaking, that set further contains both method and constructor calls, as well as field accesses.

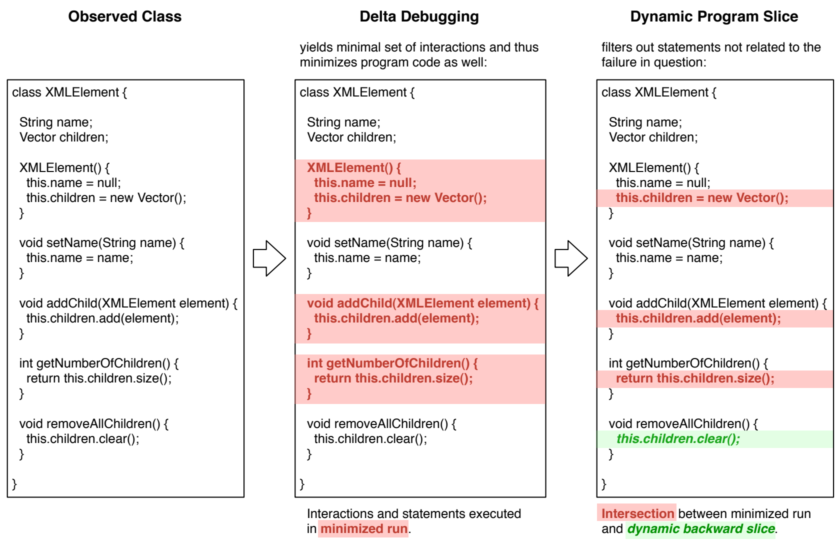
[23]For complex objects meant to be put into a collection, caching the probably expensive computation of the hash code usually is a good strategy to speed up the provision of these objects by the collection.

[24]At least the statements covered by the call triggered by JINSI; there might be statements not covered, for instance the statements within an else-branch while the corresponding then-branch gets executed.

an abstraction level lower than the method level and thus with a finer granularity. However, it typically yields more statements than strictly required to reproduce the failure, too. For instance, the dynamic slice may contain statements in methods executed in the observed faulty run, but not required by the minimized run to reproduce the failure. Fortunately, the set of statements provided by delta debugging and the one provided by dynamic backward slicing are not congruent: they both typically contain different statements that are not strictly required, so they may not be contained in the respective other set at the same time.

JINSI combines advantages of both worlds and basically computes the *intersection* between the statements provided by delta debugging, and the ones given by dynamic backward slicing. More concretely, it applies dynamic backward slicing on the minimized program run, and thus is able to provide a set of statements that could have influenced the failure *within* the already minimized run. Figure 4.10 on page 100 illustrates this approach. The left column contains simplified class `XMLElement` that provides methods to set an element's name, and to manage its child elements. We assume that a faulty program run executes all five methods; thus, all methods and lines would be covered by that run. The middle column shows the methods and statements executed by the simplified run as minimized by delta debugging. We assume that the minimal run is similar to the one in the above example (see Section 4.2.1 on event slicing); thus, setting a name is not required to trigger the assumed failure in method `getNumberOfChildren()`, and method `removeAllChildren()` must not be called[25]. The statements executed by the minimal sequence contain the technically required statement that initializes the element's name in the constructor, but that does not strictly influence the failure: the name is not connected to the failure at all. Thus, while we obtained a minimal sequence or set of interactions, respectively, by applying delta de-

[25]Otherwise, the effect of calling `addChild()` would be reverted and therefore `addChild()` could not be contained in the minimal sequence.


Observed Class
Delta Debugging
yields minimal set of interactions and thus minimizes program code as well:
Dynamic Program Slice
filters out statements not related to the failure in question:
Interactions and statements executed in minimized run.
Intersection between minimized run and dynamic backward slice.

```
class XMLElement {

  String name;
  Vector children;

  XMLElement() {
    this.name = null;
    this.children = new Vector();
  }

  void setName(String name) {
    this.name = name;
  }

  void addChild(XMLElement element) {
    this.children.add(element);
  }

  int getNumberOfChildren() {
    return this.children.size();
  }

  void removeAllChildren() {
    this.children.clear();
  }

}
```

Figure 4.10: **Combining delta debugging with dynamic slicing.** Comparison of applying delta debugging alone and combining it with dynamic backward slicing. The dynamic backward slice contains statements that are not executed by the minimized run (`this.children.clear()`). Thus, the combination of delta debugging and dynamic slicing can result in smaller sets than applying dynamic slicing alone.

bugging, we still are faced with statements not required to understand the failure. The right column finally shows both the statements in the dynamic slice of the original, non-simplified program run, and the statements contained in the dynamic slice applied to the minimized run. While the initialization of the element's name is not contained in both slices as expected, the slice of the original run would contain the statement in method `removeAllChildren()` which actually is not required to be called to reproduce the failure. Thus, applying dynamic backward slicing on the minimized run helps identify statements that do not have any data dependency on the slicing criterion, and thus usually reduces the minimized run even more.

As we have seen, JINSI uses two dynamic slicing techniques to speed up the delta debugging phase, and to further reduce the set of executed statements that are relevant to the failure. In this way, JINSI is able to provide a diagnostically conclusive result both efficiently and effectively. Chapter 9 provides an empirical study that supports this hypothesis. In the next chapter, we shall see how JINSI is able to automatically pinpoint a defect in practice. For this, we apply the tool to the JODA TIME bug discussed in the introductory example of a crashing bug that is hard to debug manually.

5 Pinpointing the Defect

In the previous chapter, we have seen how JINSI successively applies three different techniques—event slicing, delta debugging, and dynamic program slicing—to simplify a faulty program run to finally provide a minimal sequence of interactions that reproduces a given failure. Algorithm 3 on page 104 recapitulates this approach: this algorithm computes both the minimal *sequence of interactions* that reproduce the original failure, and a *set of statements* that could have possibly contributed to the failure within the minimized run. While the first diagnosis—failure-inducing interactions—is suitable for replaying the original failure at will, and thus for examining the simplified run, the latter—relevant source code statements—-helps one to directly focus on program statements that contributed to the failure on the source code level.

In this chapter, we will focus on the diagnostic quality of the minimal *sequence* of interactions. For this, we will see how the JINSI approach applies to the crashing bug in the motivating JODA TIME example[1], and how the minimized run helps pinpoint the defect. Chapter 9 finally contains an experimental evaluation that helps us investigate how well our approach works in practice on a broader scale.[2]

In order to assist the developer to debug a failure and finally fix the defect, JINSI requires a suspicious set of classes to start with. If a program crashes, the tool by default starts with the topmost class on the stack trace, and *automat-*

[1]As a reminder, the code in Listing 1.1 on page 6 crashes JODA TIME when run in the western hemisphere

[2]For this, we will compare against dynamic slicing and therefore will then exploit the set of possibly failure-inducing *statements* as computed by JINSI.

```
let $captured$ be the sequence of all captured interactions
let $predicate$ be the predicate assessing delta debugging test outcomes
**procedure** MIN($captured$, $predicate$)
    $prefiltered \leftarrow$ EVENT_SLICE($captured$)
    $min_{interactions} \leftarrow$ DD_MIN($prefiltered$, $predicate$)
    $min_{statements} \leftarrow$ DYNAMIC_SLICE($min_{interactions}$)
    **return** ($min_{interactions}$, $min_{statements}$)
**end procedure**
```

Algorithm 3: **Minimizing interactions.** For a given sequence of captured interactions and a predicate, JINSI combines event slicing (see Section 4.2.1 for details), delta debugging (see Section 4.1), and dynamic program slicing (Section 4.2.2) to compute a minimal sequence of interactions that reproduce the original failure, and a set of statements that could have possibly contributed to the failure within the minimized run.

ically follows all the objects on the stack; one after another, the corresponding classes are added to the set of observed classes. This iterative process stops when the bottommost stack element is reached, providing the developer with failure reproductions at various *abstraction levels*. The lowest abstraction level contains the topmost class only, and thus directly describes the location of the crash and its closest context, respectively. By contrast, the highest level contains all classes on the stack, and thus provides a failure description on the outermost hull: Listing 1.1 on page 6, for instance, shows the JODA TIME failure at the highest abstraction level, which is useful to *understand* the failure in its general context. However, to actually fix the failure, it is wiser to *start at the lowest abstraction level*, the topmost class on the stack. In our example (see Figure 4.3 on page 75 for the full stack trace and the exception thrown), this is class `ZonedDurationField`. Following this, one proceeds along the various abstraction levels towards the bottom by adding more and more involved classes. In our example, this successively includes classes `ISOChronology`[3], `Period`, and so on.

This iteration strategy will pinpoint the defect in JODA TIME after only three iterations.[4] Examining the minimized unit test after adding the `Period` class shows that only two interactions are required to reproduce the failure (see Listing 5.1 on page 106 for the minimal unit test): the `Period` constructor fails on the given parameters. Why does this happen? Because the given parameters stand for a time that *does not exist* in a time zone west of Greenwich. And how can a time given in seconds not exist? Because when daylight savings time ends, local time shifts by one hour—in North America, 1:59 a.m. is followed by 3 a.m., for instance. On this day, 2:30 a.m. is indeed an illegal time.

[3]`ISOChronology` is the concrete instance at location `BaseChronology.java:260`.

[4]Although there are five elements on the stack trace starting at class `Period`, there are only three different objects involved: frames `Period.init` and `BasePeriod.init` belong to the same instance of class `Period`, frame `BaseChronology.get` belongs to the instance of class `ISOChronology`, and the uppermost two frames belong to the same instance of class `ZonedDurationField`.

```java
PeriodType o89 = (PeriodType) JINSI.getMock(89);
new Period(-9223372036854775808L, -2717640422000L, o89);
```

Listing 5.1: **JODA TIME: Minimized test driver.** After only three iterations, JINSI isolates the crucial interaction that pinpoints the defect: the `Period` constructor fails on the given parameters, because they stand for a time that *does not exist* in a time zone west of Greenwich.

To finally fix the defect, we must call the `Period` constructor with legal values—namely, force the local time zone to UTC. Figure 5.1 on page 107 shows how the JODA TIME developers fixed the bug: by passing a fourth argument to the `Period` constructor that fixes the chronology to UTC.

To debug the issue with JINSI, the developer would only have to examine three minimized unit tests until she has reproduced the original failure *and* executed the defective code. Each of these minimized unit tests consists of two lines or less, and executes only 54 lines overall. JINSI cannot fix the bug on its own; but the amount by which the search space is reduced considerably eases debugging.

This example shows two further benefits—in addition to leading the developer to the defect—of the diagnosis as it is yielded by applying JINSI:

Mocked Environment. While JINSI actually creates an instance of observed class `Period`, it by contrast mocks the required parameter object `o89` of type `PeriodType`. Thus, the programmer can concentrate on those living, observed objects that are indeed relevant to the failure. JINSI replaces all other objects by mock objects that behave just like observed during capture. Apart from that, the programmer could—if desired—add so far unobserved classes to the observed ones in order to examine their actual behavior. However, in the case above, calling the constructor

```
class DateTimeZoneBuilder$PrecalculatedZone {
  static PrecalculatedZone create(...) {
    // ..
    p = new Period(trans[i], trans[i + 1],
-       PeriodType.yearMonthDay());
+       PeriodType.yearMonthDay(),
+       ISOChronology.getInstanceUTC()); // <- FIX
    // ..
```

Figure 5.1: **JODA TIME: Fix for crashing bug.** The above patch was applied by the developers of JODA TIME to fix the crashing bug in our motivating example: the time zone is set *explicitly*. Thus, the code does not depend on the system's time zone anymore, which explains why the former code fragment caused a crash in some time zones, while it worked as expected in others. After this fix has been applied to JODA TIME, the minimized test driver as computed by JINSI (see Listing 5.1 on page 106) succeeds as well.

of class `Period` already pinpoints the defect.

Executable Unit Test. JINSI converts the minimal sequence of failure-inducing interactions into a unit test which enables the developer to run these interactions as a JAVA program (JINSI provides the proper scaffolding which handles the outgoing interactions). In this way, the developer is able to examine the defect within her familiar environment (e.g., her debugger). Furthermore, the form of common JAVA source code makes it easier for the developer to read and understand the interactions, especially in contrast to the event log which stores the individual events as XML data.

In the following chapter, we will go into details about how JINSI automatically selects the observed objects whose interactions are to be minimized. As we will see, depending on the type of the bug—whether it is a crashing or a non-crashing bug—JINSI uses a different strategy to determine these objects. In both cases, the tool in this way provides failure reproductions at different abstraction levels—along *cause-effect chains*—as seen above. For non-crashing bugs, we will further see how JINSI generates proper predicates also automatically.

6 Selecting Observed Objects

In the previous chapters, we have seen how JINSI in its entirety captures and replays objects interactions to reproduce failures, and how it applies different techniques to eventually minimize those failure reproductions. Finally, in the preceding chapter, we have seen in concrete terms how JINSI is able to provide failure reproductions at different abstraction levels by following all the objects on the exception's stack trace, and, in this way, to pinpoint the defect causing a crash. In this very chapter, we will expand on JINSI's ability to automatically select objects of interest—a selection process that starts with a failure and eventually leads to its causing defect.

As already broached, a central feature of JINSI is its ability to *automatically select the objects to be observed*—and thus, the objects whose interaction is to be minimized. JINSI starts with the object in whose context the failure occurs, and gradually extends the set of objects along *cause-effect chains* [68] that lead to the failure: first, this happened, therefore, that happened, and therefore the program failed. In the context of JINSI, the cause-effect chain includes objects and their corresponding interactions; it involves the *constructions and usages* of all observed objects that are involved in the interactions that reproduce the failure. This way, JINSI ensures that both the eventually failing object, as well as the object that initially causes the failure are included in the diagnosis. This chapter discusses the two general strategies used by JINSI to select observed objects for *both crashing and non-crashing bugs*.

In debugging, software bugs can be classified into two categories: crashing and non-crashing bugs. The former category refers to bugs that cause the program to crash: the program execution gets interrupted by an exception or signal.[1] The latter refers to bugs that do not cause crashes, but different symptoms like unexpected output, or wrong results in general. One main difference between crashing and non-crashing bugs is the availability of a backtrace: while crashing bugs often cause the underlying system to provide a call stack of the crashing function, non-crashing bugs usually manifest themselves by more subtle symptoms, and therefore are more difficult to debug. In the following, we will first recap how JINSI takes advantage of a crashing bug's stack trace to select the objects to be observed. In the remaining main part of this chapter, we will finally see how JINSI handles non-crashing bugs, where an exception and thus a stack trace is not available.

6.1 Crashing Bugs

In a *crashing bug,* an exception is thrown from the topmost object on the call stack; such bugs tend to be easier to be debugged because the stack trace frequently provides good hints about the defect location [58]. For crashing bugs, JINSI uses the stack trace, and hence the *control-flow* information contained therein, as a cause-effect chain:

1. It starts observing the topmost object and minimizes its incoming interaction using the presence (or non-presence) of the same crash as a predicate for minimization; as long as the same crash occurs, the failure in question is reproduced.

2. In the following steps, the set of objects is gradually expanded to include more and more callers.

Algorithm 4 on page 111 formalizes this strategy.

[1]In JAVA programs, crashing bugs cause an *exception*—shorthand for the phrase "exceptional event"—to be thrown. In other programming languages, crashes may cause other symptoms, like *signals* in Unix applications.

```
let $exception$ be the thrown exception
// predicated is derived from type of exception, its message, and its location
$predicate \leftarrow$ DERIVE_PREDICATE($exception$)
// set of observed classes
$observed \leftarrow \emptyset$
// following down the stack
for all objects $obj$ on stack of $exception$ do
    $class \leftarrow$ CLASS_OF($obj$)
    $observed \leftarrow observed \cup \{class\}$
    $captured \leftarrow$ CAPTURE($observed$)
    // for MIN, see Figure 3 on page 104
    $min \leftarrow$ MIN($captured, predicate$)
    print UNIT_TEST($min$)
end for
```

Algorithm 4: **Iteration strategy on objects for crashing bugs.** JINSI takes advantage of the *control-flow information* contained in the stack trace to derive observed objects and classes, respectively. Starting with the *topmost* element on the stack, JINSI follows *down* the stack while it successively minimizes the corresponding object interactions and outputs the minimal unit test. The predicate initially derived from the exception stays the same all along the stack.

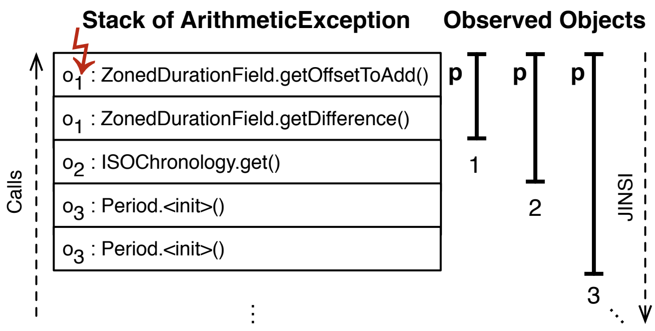

Stack of ArithmeticException
Observed Objects
o1 : ZonedDurationField.getOffsetToAdd()
o1 : ZonedDurationField.getDifference()
o2 : ISOChronology.get()
o3 : Period.<init>()
o3 : Period.<init>()
Calls
p
p
p
1
2
3
JINSI

Figure 6.1: **Selecting observed objects for crashing bugs.** For crashing bugs like the one in JODA TIME, JINSI starts with observing the topmost target object (o_1) on the call stack, successively expanding the set of objects from caller to caller (o_2, o_3, ...). Due to calls of o_1 and o_3 on themselves, there are only three different objects but five methods on the stack. The minimization predicate p derived from the exception stays in the topmost method, identifying the crash, and thus minimizing the interaction that leads to the crash. The result is a *cause-effect chain of interactions along the control-flow* in which every chain element is minimized. Algorithm 4 on page 111 formalizes this strategy.

Figure 6.1 on page 112 illustrates this strategy by the JODA TIME crash—as already exemplified and discussed in detail in the previous chapter. JINSI minimizes the interactions into the failing object (o_1), then the additional interactions into its caller (o_2), then those into the caller of the caller (o_3), and so on. The result is a cause-effect chain of interactions along the control-flow in which every chain element is minimized. The main advantage of this strategy is that it takes advantage of the *crash early principle*[2] of exceptions: the distance in execution time between the defect being executed and its final manifestation as crash is relatively short; thus, the cause-effect chain is short as well.

However, this strategy—as the case may be—has a weak point: it will work particularly well in those cases if and when the stack trace contains the defective class; thus, if the trace *directly* involves an instance of that class with the defect in its code. Only then, JINSI would eventually select the defective class as observed, and thus would include it in the minimizing reproduction. If not, JINSI will nevertheless be able to compute minimal failure reproductions at all the individual abstraction levels and stack frames, respectively. Yet, the diagnosis would neither contain the defective class which would have to be manually defined as observed for this purpose, nor would it execute the defective code as the defective class will be mocked. Thus, if we want to include the defective class into the diagnosis automatically, we would have to use a different strategy to select objects to be observed. Indeed, JINSI provides a further strategy that does not only work in such cases where the defective class is not on the stack, but that also works for non-crashing bugs. In the following, we will see the alternative strategy used by JINSI to effectively and automatically debug non-crashing bugs as well—as we will see moreover, this strategy is a generalization of the one for crashing bugs.

[2]The crash early principle tells that whenever an error occurs in your software you should crash *immediately,* and not later in the program run, because then it would be hard to find the cause for the error (thus, the defect). This principle and its maxim “A dead program normally does a lot less damage than a crippled one” are described in detail in the book by Hunt and Thomas [35].

6.2 Non-Crashing Bugs

In a *non-crashing bug,* a failure comes to be as some incorrect output—or, more generally, as an unexpected observable behavior caused by an *infection* of the program state. Such infections are usually much harder to debug than a crash, because the infection itself is caused by a defect *early* in the program execution, but often discovered only *at the end* of the execution once the defect manifests itself as the externally observable failure.[3] In such a situation, the programmer must identify the source of the infection chain, progressing backwards along the origins of values.[4]

This is where a dynamic backward slice is most helpful, as it contains precisely those parts of the program that could have influenced a specific set of variables at some point. In particular, this technique can be applied to the variables that are obviously connected to the observed failure—like the variables that are used when printing some incorrect output—yielding only those parts of the program that are connected to the failure via data dependencies (see Section 4.2 for details on program slicing).

6.2.1 Object Dependencies

While dynamic slicing according to the standard definition yields a *set of statements* only, Hammacher's slicer for JAVA [34] additionally provides a *dynamic dependence graph* (see Figure 4.5 on page 86 for definitions of data and control dependencies, and of the program dependence graph). Similar to a dynamic

[3]The *infection* usually affects the innermost scope of the program state, which is hard to be observed by the programer. Usually, the programmer has to cope with a debugger to find and examine such infections, which often is a tedious work. By contrast, the failure appears on the observable surface of the system and can therefore be noticed without taking any particular measures.

[4]Such a situation can also happen with crashing bugs, although this was not the case in the bugs we observed. On the contrary, cause-effect chains along the stack trace tend to be much shorter than the ones in non-crashing bugs—in accordance with the crash early principle of exceptions.

slice, that graph is derived from one single program run, and therefore does not contain possible, but actual data and control dependencies as they occurred in the concrete run.

The dynamic dependence graph enables JINSI to deduce information on dependencies between objects, and to determine subslices to be minimized: For instance, if a statement y executed within object Y depends on a statement x executed within object X, JINSI deduces that object Y depends on object X; there is an *object dependency*[5] between Y and X. In other words, object Y gets some required information from object X, either represented by a data or by a control dependency. Thus, these object dependencies represent an ordered set of objects on the dynamic program slice. Having a long chain of these object dependencies, a *subslice* is defined by a set of connected chain links. Furthermore, while dynamic program dependencies in fact result in graphs but not in linear chains (and, by implication, in object dependence graphs rather than simple chains), in the following we assume, for the sake of simplicity, that we work on linear structures. However, all concepts and strategies presented in this section can be easily[6] adapted to graph structures as well—in reality, JINSI applies these to the occurring graphs. Figure 6.2 on page 116 shows a definition of both object dependencies and the thereof derived object dependence graph.

For non-crashing bugs, JINSI takes advantage of the information on dynamic program and object dependencies, respectively, to track back the infection chain, and to minimize individual subslices and their corresponding object interactions to eventually obtain a sequence of minimized unit tests along the cause-effect chain similar to crashing bugs.[7] JINSI basically starts with a given

[5]JINSI always works on one single (faulty) program run. Therefore, in the context of JINSI, we consider *dynamic* dependencies only.

[6]By doing a breadth-first search on the graph structure instead of simply following the links in a strictly serial chain.

[7]Because the dynamic dependence graph includes both data and control dependencies, it contains the control dependencies of a possible stack trace as well. Consequently, the approach for

Object dependency: An object instance O_2 of class C_2 is dependent on an object instance O_1 of class C_1 if there is either

- a statement s_2 in C_2 that is *data dependent* on a statement s_1 in C_1; or
- a statement s_2 in C_2 that is *control dependent* on a statement s_1 in C_1.

Thus, object O_2 gets some required information from object O_1, either represented by a data or by a control dependency: either O_2 reads some data previously written by O_1, or O_1 had determined that the program's execution continued in O_2. In the context of this work, we consider *dynamic* dependencies only; therefore, object dependencies are derived from a single program run, and in fact are *dynamic* object dependencies—object dependencies for short.

Object dependence graph: The object dependencies form the so-called *object dependence graph* $G_O = (V, E)$ with

- $V = \{O \,|\, O \,\text{is an object with an object dependency on}\}$; and
- $E = \{(O_1, O_2) \,|\, O_1, O_2 \in V \wedge O_2 \,\text{is object dependent on}\, O_1\}$.

Thus, the vertices V represent all the objects with some dependency on, and the edges E represent actual object dependencies.

Because that graph is derived from (dynamic) object dependencies, the object dependence graph also contains actual dependencies as they occurred within a single program run.

Figure 6.2: **In a nutshell: Object dependencies and the thereof derived graph.** Object dependencies and the corresponding dependence graph are defined on top of—and, similar to—data and control dependencies (see Figure 4.5 on page 86).

predicate on externally observable state (typically, a failing test oracle) and determines the dynamic backward slice from that object, and the corresponding dynamic dependence graph. It then progresses along the deduced object dependencies using a *sliding window* approach: First, the interaction into the last object on the slice (o_1) is minimized. Then, the interaction into the objects o_1 gets data from (o_2) is minimized. Then, the interaction into the next level of dependency (o_3) is minimized, and so on.

However, non-crashing bugs pose a special challenge to minimization, as the set of involved objects can potentially grow very large. If we were to apply a similar strategy as for crashing bugs (i.e., including more and more caller objects), we would eventually include all objects on the dynamic slice, which results in a huge number of interactions, and finally in a diagnosis which would be less conclusive. Instead, we go for a *stepwise minimization:* rather than minimizing the entire set on the dynamic slice, we basically minimize each element on the chain *in stages*—that is, for every object, only the object itself as well as the objects it *directly depends* upon. Though, the window size is not fixed (e.g., a window size of 2); JINSI rather adjusts the window size, as we will see in the following example of a crashing bug.

Listing 6.1 on page 118 shows the incorrect output caused by a non-crashing bug: NANOXML, an XML parser for JAVA (see issue `NAN-6` in Section 9.1 for details), reads a given XML file and pretty prints the parsed data to the console. Unfortunately, the output contains an artifact which is not contained in the input. Thus, *somewhen* between reading the file and printing the output *somewhere* a defect had been executed that had caused the initial infection and eventually led to the faulty output.

To minimize the object interactions along the object dependencies—which are in principle derived from the dynamic slice, and thus along the cause-effect chain—JINSI starts with object o_1, an instance of `XMLWriter`, that writes XML

non-crashing bugs is a generalization of the one applied by JINSI to crashing bugs. However, following the control dependencies directly given by the stack trace of a crashing bug, JINSI can exactly follow the cause-effect chain as represented by that stack.

```
1  <FOO>
2    <BAR/>
3    <BAR x="1"/>
4    <BAR x="1" y="2"/>
5    <BAR>
6        <BAR/>
7        <BARBAR/>
8        <BA/>
9        <FOO/>
10   </BAR>
11   <BAR>blah</BAR>
12   <BAR x="1">blah</BAR>
13   <BAR>&lt;&amp;&gt;&apos;&quot;</BAR>
14   <BAR>abc</BAR>
15   &lt;&amp;#38;&gt;&quot;&apos;!:&#x2030;
16 </FOO>
```

Listing 6.1: **Defective output as issued to the console by NANOXML.** The content of root element `FOO` should not contain the artifact `#38;` (Line 15). The expected, correct content would be `&lt;&amp;&gt;&quot;&apos;!:&#x2030;`.

```
public class XMLEntityResolver {
  public XMLEntityResolver() {
    // ..
-    this.entities.put("amp", "&#38;#38;");
+    this.entities.put("amp", "&#38;");
    // ..
```

Figure 6.3: **Actual fix in NANOXML.** The resolver was not initialized correctly. Instead of “`&#38;`”, the entity “`amp`” was initialized with “`&#38;#38;`”—precisely the artifact being present in the faulty output.

element o_2 to the console, and follows along objects o_2 and o_3, of which the latter builds the written XML element, and so on. Eventually, the cause-effect chain ends at an instance of `EntityResolver` which executes the actual defect as shown in Figure 6.3 on page 118. Thus, JINSI had minimized along the data and control dependencies of the dynamic slice. This strategy finally results in the initial link of the cause-effect chain—thus, the actual defect. Figure 6.4 on page 120 illustrates the strategy on object dependencies as used by JINSI.

While the above strategy enables JINSI to proceed along the cause-effect chain as given by the dynamic slice, it results in a further challenge: the sliding window approach requires JINSI to drop the initially observed object—o_1, the instance of class `XMLWriter` in the above example—from the set of observed objects. Thus, it looses the initial predicate that is defined on this very object. In the remaining part of this chapter, we will see how JINSI generates new predicates fully automatically in order to still be able to decide on success or failure during delta debugging.

6.2.2 Generating Predicates

Removing the initial failing object (thus, o_1 in the above examples) from the set of observed objects brings another challenge: we lose the original predicate deciding on success or failure, and thus have no predicate anymore to minimize against. For instance, because the above predicate is defined on the output written to the console, and object o_1 exactly writes that output, without o_1 being observed and thus being substituted by JINSI, no output would be written at all—the predicate would be of no use.

JINSI therefore generates *alternative predicates* (shown as $p\prime$ and $p\prime\prime$ in Figure 6.4 on page 120). A predicate needs to distinguish passing and failing runs; to derive such predicates automatically, we could compare known passing and failing runs to deduce distinctive features which would identify a run either as passing, or as failing—for instance, we could use statistical debugging for this purpose. However, since we start with a single failing run only, where do we

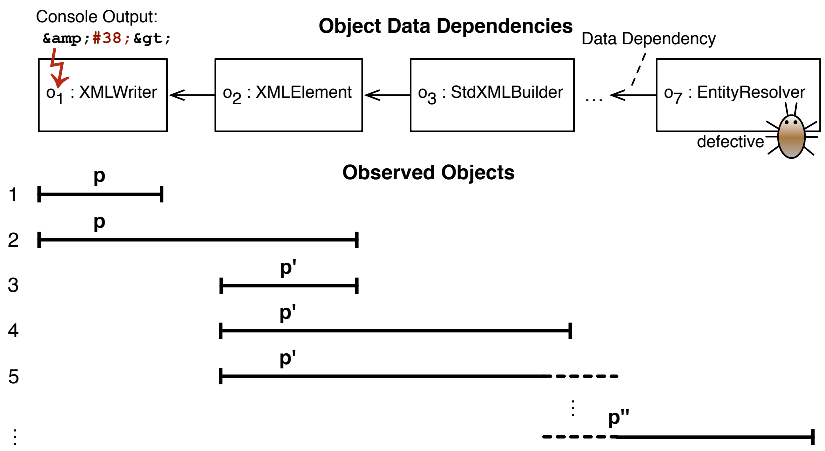


Figure 6.4: **Selecting observed objects for non-crashing bugs.** For non-crashing bugs, JINSI progresses by stages along object dependencies basically derived from the dynamic backward slice, minimizing object interactions in individual subslices to finally obtain minimized unit tests along the cause-effect chain. Because JINSI removes the initial failing object o_1, it loses the original predicate to minimize against. JINSI therefore generates alternative predicates ($p\prime$ and $p\prime\prime$). For simplicity, we omitted the actual graph structure of the underlying dynamic dependence graph.

get the passing runs from? The answer is simple: *delta debugging yields similar passing runs* as a by-product.[8] These synthetic passing runs are the base for the subsequent predicate generation.

Similar to the work of Zeller [68], JINSI extracts *memory graphs* for both the single failing run and all the produced passing runs, computes *differences* on objects and their attributes using these graphs, and finally derives proper predicates from these differences. In this way, JINSI automatically obtains predicates like "In the failing run, Attribute `name` of object with id 13 has value `"UTC"`.", or "The list with id 13 contains 5 (instead of 4) elements." These predicates then are used to minimize the interactions against.

The concept of memory graphs has been exhaustively discussed in the work on visualizing memory graphs by Zimmermann and Zeller [76]. Therefore, we briefly and succinctly present the most important aspects only. Basically, memory graphs are a means to capture and explore program states. They give a comprehensive view of all data structures of a program; the therein contained data items are related by operations like dereferencing, indexing, or member access. Furthermore, what is of especial interest in the context of JINSI, two memory graphs can be used to easily compute the differences between the corresponding program states.

A memory graph $G_M = (V, E, root)$ models memory and a program state, respectively, by a set V of values (the graph's vertices), a set E of references between these values (the graph's edges), and a dedicated vertex *root*. The memory graph encompasses the entire program state and basically gets extracted and entirely unfolded by automatically dereferencing pointers (e.g., in

[8]This is due the *1-minimality* of the delta debugging algorithm—no single interaction can be removed without removing the failure [72]. For instance, if JINSI computes a sequence of one constructor call and three subsequent method calls, we would know for sure that removing that constructor call or any of the three method calls would result in either a passing or unresolved run. Furthermore, it is guaranteed that at least one interaction is directly failure-inducing: removing this interaction from the minimal set would result in a passing run; the other interactions could all be, by contrast, interactions that would set up the proper context for the failure to re-occur (a constructor call is a typical example for that latter kind).

C programs) and references (e.g., in JAVA programs) starting at *root*. One main benefit of the abstraction from concrete values yielded by using the above graph structure is that comparing program states becomes a rather simple graph operation: detecting the greatest common subgraph is sufficient to determine differences in the graphs; thus, differences in the program states.

For more details, refer to the afore mentioned work [76] as it contains, for instance, definitions on the formal structure of memory graphs, their construction, and how to compare them to yield differences between program states. While that work applies the presented techniques to C programs and their data structures, we adapted these techniques for our needs: JINSI applies them to JAVA programs and their corresponding data structures like arrays and lists, and thus is capable of computing differences between states of JAVA programs, and finally of deriving alternative predicates from those differences.

In the above NANOXML example, JINSI starts with a predicate[9] on the externally observable output: as long as the output contains the artifact, the run is classified as failing. In this way, JINSI is able to minimize both the interactions into the writing object o_1, as well as the interactions captured for o_1 and XML element o_2 (stages 1 and 2 in Figure 6.4 on page 120). While minimizing the incoming interactions into the latter two observed objects, delta debugging yields a passing run which is produced by removing the incoming method call `o2.setContent(...)` from the minimal sequence of interactions. Logically, as soon as the (faulty) content is not propagated to the later written XML element anymore, there will be no infection, the writer will not obtain this content from the written element, and, in the end, the output will not longer contain the artifact. By comparing the failing and passing XML elements, JINSI finds the crucial difference in their content, and deduces from this very difference the predicate "Field `content` of XML element o_2 must contain `#38;`." which exactly describes the infection in the written XML element (shown as $p\prime$ in Fig-

[9]Typically, given by a failing test oracle. In this example, the programmer could define the initial predicate which specifies the unwanted output on the console.

ure 6.4 on page 120). Now, having a predicate on XML element o_2, the initially observed writer o_1 can be dropped (stage 3 in Figure 6.4 on page 120). In the following 4th stage, JINSI includes the object on which XML element o_2 directly depends: the `StdXMLBuilder` instance o_3 which again builds the infectious XML element—and actually calls method `o2.setContent(...)`.

Now, an interesting case occurs: although builder o_3 propagates the infection to the XML element, there is no difference in the builder's state between failing and passing runs. This is because, in this context, the builder merely is a static and thus stateless utility class which obtains information about the element being constructed and immediately discards this information as soon as the element was finished. To resolve this issue, JINSI includes the object on which builder o_3 depends (thus, o_4). This is repeated until a usable difference and a new predicate, respectively, is found. Thus, instead of using a sliding window approach with a fixed window size of 2, JINSI actually uses a *flexible window size* which increases until a predicate on a newly observed object is found; then, JINSI resets the window size and starts over.

6.2.3 Algorithm for Non-Crashing Bugs

Algorithm 5 on page 125 shows the basic algorithm used by JINSI to debug non-crashing bugs.[10] The algorithm combines the above findings for non-crashing bugs, and thus takes advantage

1. of the *data- and control-flow information* provided by the slicer to select observed objects (Section 6.2.1);

2. and of the *1-minimality* of the delta debugging algorithm to generate predicates (Section 6.2.2).

[10]Please note: For simplicity, we only show how the algorithm would work on *linear* data dependencies. In fact, the slicer yields a dynamic dependence *graph*. JINSI therefore extends the above algorithm by a *breadth-first search*. Furthermore, we omitted the upstream event slice and the downstream dynamic slice (see Algorithm 3 on page 104).

JINSI actually extends this algorithm by a breadth-first search in order to minimize objects on subslices on the dynamic dependence graph. In this way, JINSI is able to provide minimal unit tests along the cause-effect chain of non-crashing bugs which faithfully reproduce the different infection stages—going from the externally observable failure to the actual site of infection; thus, the defect to be fixed.

As for crashing bugs, applying JINSI to the above non-crashing bug results in a considerable reduction of the search space. Applying dynamic slicing to the statement that prints to the console yields a set of 313 lines—out of 3,295 lines of source code, which is 9.5 % of the source code. JINSI minimizes object interactions along the object dependencies, which start at the object that prints to the console, and in this way computes a minimal set of relevant statements of only 75 lines, or 2.3 % of the source code. The evaluation in Chapter 9 contains results for further non-crashing bugs.

In this very chapter, we have seen how JINSI is able to *automatically select the objects to be observed*, and thus how the tool is able to compute minimal tests that faithfully reproduce the observed failure along *cause-effect chains*, eventually leading to the defect—regardless of whether the defect causes a crash, or a non-crashing failure. After having covered all central aspects and strategies of the JINSI approach in general terms, in the following chapter we will see how this approach is specifically applied to JAVA programs; we will see how JINSI deals with the practical challenges mainly caused by features specific to the JAVA programming language. This discussion enables other people to re-implement the JINSI approach for JAVA programs, or programs written in a different—but still similar to JAVA—language.

```
let $dyn\_slice$ be the dynamic slice of the failing run
let $pred$ be the initial predicate given by test oracle
let $obs\_init$ be the initial observed object given by test oracle
$observed \leftarrow \{obs\_init\}$
while $obs\_next \leftarrow next\_by\_distance(dyn\_slice)$ do
    $observed \leftarrow observed \cup \{obs\_next\}$
    $captured \leftarrow capture(observed)$
    $1\_min \leftarrow dd(captured, pred)$
    $state\_f \leftarrow mem\_graph(1\_min, obs\_next)$
    let $pred$ be an empty predicate
    for all interactions $int$ in $1\_min$ do
        $subset \leftarrow 1\_min \setminus \{int\}$
        if $outcome(subset)$ is passing then
            $state\_p \leftarrow mem\_graph(subset, obs\_next)$
            $diff \leftarrow mem\_diff(state\_p, state\_f)$
            if $diff$ is not empty then
                $pred\_on\_diff \leftarrow predicate(diff)$
                $pred \leftarrow pred \wedge pred\_on\_diff$
            end if
        end if
    end for
    if $pred$ is not empty then
        $observed \leftarrow \{obs\_next\}$
    end if
end while
```

Algorithm 5: **Selecting objects and generating predicates for non-crashing bugs.** JINSI takes advantage of the *data- and control-flow information* provided by the slicer to select observed objects, and of the *1-minimality* of the delta debugging algorithm to generate predicates. JINSI extends the set of observed objects according to their distance on the dynamic slice, and computes alternative predicates on newly observed objects by comparing program states. As soon as a new predicate is found, JINSI can drop the previously observed objects and continues with the newly observed ones and the new predicate.

7 Implementation Overview

To capture, replay, and minimize interactions in JAVA programs, JINSI has to extensively modify[1] a given program, for example by inserting probes and method proxies to capture and replay interactions as described on a conceptional level in Chapter 3. Applying all the techniques described in the previous chapters to JAVA programs involves a series of challenges. In this chapter, we will discuss the most important ones only, while Chapter 8 will cover further issues to complete the discussion. Both chapters together will provide solutions for all main challenges encountered while applying JINSI to JAVA programs, ranging from small toy examples to industrial-sized projects. The findings in these chapters will enable other people to understand how JINSI is applied to JAVA programs and its language features; moreover, the knowledge contained therein enables other people to re-implement the JINSI approach for JAVA programs. The challenges and solutions presented in this chapter will include:

- **Capture constructor calls.** Basically, the first statement in a constructor's body has to be either `super(...)` or `this(...)`. Thus, it is not possible to obtain information about `this`, the object being constructed, until all the super and alternate constructors, respectively, will have been executed. However, while these constructor calls are being executed, other interactions interwoven with the initial one to be captured are likely to occur as well. In Section 7.1, we will see how JINSI deals

[1] JINSI modifies given JAVA programs on the byte code level using JAVASSIST, a class library for editing byte code. See Section 8.1 for more details.

with pitfalls related to these and other issues related to constructor calls as they are implemented in the JAVA programming language.

- **Capture incoming method calls.** JAVA uses dynamic method lookup for (non-static) methods. Thus, at compile time it is not known which concrete method implementation will be invoked later at run-time. To capture incoming method calls effectively and efficiently, JINSI replaces observed methods by a logging proxy method, as we will see in Section 7.2.

- **Who calls whom?** To know whether a call to an observed object is actually incoming from the perspective of the observed object, you have to know its caller. However, in JAVA it is not possible to dynamically obtain the caller of a method. You only have access to `this` (thus, the callee) and the passed arguments. Unfortunately, instrumenting the call side instead—where you would have access to a reference to both the caller and the callee—is not an option in practice. As we will see in Section 7.2.1, JINSI instead has to register all callers of potentially incoming method calls at the respective call side in order to obtain their runtime type and to decide on whether that call is incoming or not.

- **Inheritance.** If you want to observe class `B` that extends unobserved class `A` that again defines method `calc(...)`, you will want to observe all calls to method `calc(...)` on instances of `B` only—however, not on instances of `A`. In Section 7.3, we will see how JINSI handles inheritance by a combination of instrumenting both observed classes and their—in most cases—unobserved superclasses, and proper runtime checks to ensure that all interactions of interest are captured properly.

In the following, we will address the above challenges in more detail. All the language features discussed here (and, in Chapter 8) are described in the official JAVA Language Specification by Gosling et al. [31]; for the sake of simplicity, we will not cite that book for every single feature discussed in this work individually. We will instead refer to the book of Gosling et al. only at passages where it will be of peculiar interest.

7.1 Capture Incoming Constructor Calls

To replay the construction of observed objects, JINSI has to first intercept and record all incoming calls to constructors in observed classes. A straightforward approach would be to insert *probes*—artificial method calls to JINSI that inform JINSI about interactions—at the beginning and at the end of each observed constructor: The probe at the constructor's beginning would record the incoming call including its given arguments, the one at the end would record the return from this call. All interactions originating from the object being constructed in between, for example outgoing method calls, could be linked to this specific object.[2]

However, as we will see in the following sections, inserting probes in constructor declarations of observed classes, and thus capturing incoming calls to these constructors from *within*, involves many issues that are not fully resolvable. In addition to the principal difficulties, JAVASSIST contained several bugs related to instrumenting constructors when JINSI's capture feature had been developed.[3] In Section 7.1.6 we will see how JINSI avoids all the issues by capturing incoming constructor calls from the *call side* instead, but first we will start with the discussion of manipulating constructor declarations directly as the initial idea implies.

[2]This would be similar to the proxy methods that capture incoming method calls. These also record respective call and return events, while the interactions caused by the observed method itself are recorded in between (see Section 7.2 for details).

[3]JINSI uses JAVASSIST to instrument JAVA programs for its purposes; see Section 8.1 for details. For instance, JAVASSIST had problems with calls to `super` and `this`; see bug `JASSIST-53` [37].

Observed class Observed in Listing 7.1 on page 130 provides a basic example of a constructor declaration. The class' single-argument constructor Observed(Unobserved init) assigns boolean field enabled to the value obtained by calling method isEnabled() on the given unobserved argument init.

```
public class Observed {

    private boolean enabled;

    public Observed(Unobserved init) {
        this.enabled = init.isEnabled();
    }

}
```

Listing 7.1: A simple observed class providing a constructor for initialization.

The instrumented version of constructor Observed(...) with probes inserted according to the above idea would look similar to the one in Listing 7.2 on page 130. The single statement to assign the field would basically be surrounded by calls to JINSI to capture the start and the end of all incoming constructor calls.[4]

```
    public Observed(Unobserved init) {
        JINSI.recordIncomingConstructorCall(...);
        this.enabled = init.isEnabled();
        JINSI.recordIncomingConstructorReturn(...);
    }
```

[4]This instrumented version is not complete for the sake of simplicity: the constructor as shown in Listing 7.2 on page 130 would contain a surrounding try-catch-block to capture exceptions, and further calls to JINSI to capture outgoing calls, like the one to init.isEnabled().

Listing 7.2: Instrumented version of `Observed`'s constructor with probes inserted at the beginning and at the end.

In practice, JINSI would further instrument the constructor to capture the outgoing call to `init.isEnabled()`. In this way, the outgoing call could be associated to the very instance under construction (by assigning proper object-IDs as described in Section 8.2).

7.1.1 `super()` is Called Automatically

Unfortunately, the first statement in a constructor has to be a call to one of its superclass constructors `super(...)`. Even if a constructor does not explicitly invoke a superclass constructor, the JAVA compiler automatically inserts a call to the *no-argument* constructor `super()` of the superclass.[5] If the superclass does not provide a no-argument constructor, the compiler would issue a compile-time error as the programer would have to explicitly choose which one should be called.

Because a subclass constructor always invokes a constructor of its superclass[6], either explicitly or implicitly, there will be a whole chain of constructors called, all the way back to the constructor of the primordial class `Object`. This mechanism is called *constructor chaining,* and we need to be aware of it as soon as we want to capture interactions of classes involved in any inheritance hierarchy. Consequently, the instrumented constructor looks like the one in Listing 7.3 on page 132: `super()` is necessarily called first—thus, even before the incoming call would be recorded by JINSI.

[5] In this case, the compiler inserts a call to `super()`, an invocation of the constructor of its direct superclass that takes no arguments. See §8.8.7 "Constructor Body" in the JAVA language specification [31].

[6] More precisely, either `super` or `this` (call to an alternate constructor in the same class, see Section 7.1.4) has to be called. However, in both cases, eventually a chain of constructors ending in `Object()` will be executed.

```
    public Observed(Unobserved init) {
        super(); // has to be the first statement;
        // added automatically to byte code if missing
        JINSI.recordIncomingConstructorCall(...);
        this.enabled = init.isEnabled();
        JINSI.recordIncomingConstructorReturn(...);
    }
```

Listing 7.3: Calling `super()` has to precede the inserted probe.

In this simple example, JINSI could record all required information anyway, because `Observed(...)` calls `super()` as defined in class `Object`[7]; its only no-argument constructor `Object()` is an empty constructor that does not contain any statement. Thus, no interaction could occur before JINSI would record the incoming call to `Observed(...)`. However, as soon as there is some outgoing interaction in superclasses of observed ones, inserting such probes would not be sufficient anymore, as we will see in the following example that involves a inheritance hierarchy of depth three.

A more complex example involving both inheritance and statements that trigger interactions in super constructors can be found in Listings 7.4 and 7.5: Class `SuperClass` in Listing 7.4 on page 132 defines a constructor that logs its calls to the console. The log message contains both the class' name and the individual instance's hash code. The class name enables us to see which constructor gets called, the hash code allows us to identify the concrete instance that issues the log message. Its extending subclass `SubClass` in Listing 7.5 on page 133 overrides this constructor and logs all calls with its own name instead of the one of its superclass. Note that neither the superclass, nor the subclass calls `super()` explicitly.

```
public class SuperClass {
```

[7]Because class `Observed` *directly* subclasses class `Object`.

```java

    public SuperClass() {
        // super() in class Object is called implicitly
            .
        System.out.println("SuperClass␣-␣hashCode:␣" +
            this.hashCode());
    }

}
```

Listing 7.4: Class `SuperClass`: logs its construction to the console.

```java
public class SubClass extends SuperClass {

    public SubClass() {
        // super() in class SuperClass is called
            implicitly.
        System.out.println("SubClass␣-␣hashCode:␣" +
            this.hashCode());
    }

    public static void main(String[] args) {
        new SuperClass();
        new SubClass();
    }

}
```

Listing 7.5: Class `SubClass`: overrides its super constructor and logs its construction to the console. The class name in the log message is adapted.

In addition to overriding its super constructor, `SubClass` provides method `main()` to run a small sequence of constructor calls that demonstrates the issues caused by inheritance relating to constructor calls. That method first creates an instance of `SuperClass`, and afterwards an instance of `SubClass`. Without already knowing that `super()` is called automatically, you might expect only two lines of output: The first line would contain the log message from the constructor in `SuperClass`, the second line would contain the log message from `SubClass`. Though, the actual output is different and consists of three lines instead, as shown in Listing 7.6 on page 134.

```
SuperClass - hashCode: 1965484127
SuperClass - hashCode: 405223709
SubClass - hashCode: 405223709
```

Listing 7.6: Sample output of constructing one instance of `SuperClass` and one instance of `SubClass`.

As you can see by this example, constructor `SuperClass()` is called *twice*: The first time explicitly in method `main()`, the second time implicitly via `super()` in `SubClass()` which was called explicitly in `main()`—the JAVA compiler had introduced that call in order to comply the JAVA language specification. This involves a problem for JINSI: When calling constructor `SubClass()`, an outgoing interaction from the instance currently being constructed to method `println()` already occurs in `SuperClass()` *before* JINSI's probe in `SubClass()` would be called.

Apparently, an extended approach would be to insert a probe in the constructor `Object()` that is called by all classes in any case. Then, it would be sufficient to insert the probe for the incoming call in class `Object`, the one for the corresponding constructor return at the end of all constructors in observed classes.

7.1.2 Constructor `Object()` Cannot Be Instrumented

However, it is very error-prone to instrument class `Object`. It easily results in various errors, like segmentation faults in the JAVA virtual machine[8] or unexpected exceptions at run-time[9]. This is mainly caused by optimizations done by the JAVA virtual machine, which assumes some properties like number and order of fields in `Object`. Furthermore, some internal library code assumes a specific depth of the stack when objects are constructed.

Another disadvantage would be performance: JINSI would have to check for *virtually every* single object—there could easily be millions—whether an incoming constructor call goes to an observed object or to an unobserved one. In the first case, JINSI would have to record the call, in the second case it would have to discard the event. This could only be achieved by executing a run-time check at every constructor call. Furthermore, with the current—as of this writing—version of JAVASSIST it is not possible to instrument class `Object` because of bug `JASSIST-88` [9]. Therefore, instrumenting `Object()` to capture incoming constructor calls is not an option at all.

7.1.3 Instrumenting Constructors in the Inheritance Hierarchy

Another option would be to instrument all the constructors along the inheritance hierarchy in a similar way as described above—but just not in class `Object` itself. Instead, we could insert the probe for incoming calls in all `Object`'s *direct* subclasses that are superclass of an observed class. Thus, the constructors in an observed class would still get probes that capture the constructor return, but its superclass that is the direct child of `Object` would be instrumented for the incoming call. Basically, the probe for incoming calls would be pushed down from class `Object` to its direct descendants. Because

[8]`http://mail-archive.ow2.org/asm/2007-05/msg00027.html`
[9]`http://forums.sun.com/thread.jspa?threadID=5119309&tstart=0`

constructor `Object()` does not contain any code, we know for sure that there would be no interaction triggered by this constructor. Thus, calling `super()` in `Object`'s direct descendant before JINSI is notified about the incoming call would be safe.

In the previous (rather simple) example, we would instrument both the constructors in classes `SuperClass` and `SubClass`. If `SubClass` would be observed but not `SuperClass`, JINSI would have to decide at runtime whether a potentially incoming call actually goes to an observed instance (object of type `SubClass`) or to an unobserved one (instance of `SuperClass` but not `SubClass`).

However, in large projects with more complex class hierarchies, this scenario usually gets more complicated. If both classes are observed, we would have to insert probes for the constructor *return* in both of them: an incoming call to `SuperClass()` as well as to `SubClass()` would have to be recorded as a proper event. The probe for the return at the end of `SuperClass()` would then have to check whether the incoming call is constructing an instance of `SuperClass` or `SubClass`. In the first case, the probe in `SuperClass()` would have to record the constructor return, otherwise it would have to ignore it because instantiation would continue in `SubClass()`. In this latter case, the instance of `SubClass()` would have to record the constructor return in the end.

Despite the complexity of such an approach, it would get even more difficult to be implemented as soon as you want to observe several classes in a more complex inheritance hierarchy. Furthermore, constructors may call other constructors; this results in complex interleaving of constructor calls of observed and unobserved classes. To solve this issue, JINSI would have to maintain its own stack of constructor calls, so as to track incoming calls and their relations with each other.

7.1.4 Calling an Alternate Constructor

Another related topic that has not been covered so far is calling `this()`. In contrast to `super`, which references to the super class, `this` references to the object *currently* being instantiated; thus, the object whose constructor is being called. From within a constructor, this keyword calls an alternate constructor in the same class. Just like for `super`, the invocation of an alternate constructor must be the first statement in the constructor. In fact, the first statement of a constructor body has to be either an explicit invocation of another constructor of the same class (call to `this()`), or of the direct superclass (call to `super()`). Thus, either `this()` or `super()` has to be called first.

Similar to `super()`, `this()` calls for a very sophisticated and thus complicated instrumentation in observed and unobserved constructors as well. During implementing JINSI for JAVA programs, experience has shown that pursuing the idea to capture incoming constructor calls *within* the object being instantiated fails for many cases of application.

7.1.5 Capturing Exceptions

Besides capturing regular constructor returns, JINSI has to track exceptions as well; when calling a constructor, an exception may occur. In addition to the two probes for the incoming call and its return, the instrumentation has to surround the constructor with a try-catch-block. This incorporates even more complexity into the byte code that has to be added to capture incoming constructor calls—and again would increase the error-proneness of the approach.

In the following, we will see how JINSI avoids all the various issues involved in manipulating constructor calls. Instead of inserting probes in observed constructors, and thus extensively manipulating the called constructors, JINSI captures incoming constructor calls from the *outside*—thus, from the *call side*.

7.1.6 Capturing Constructors from the Call Side

In JAVA, a constructor of a class is invoked by using the special function name `new`. This operator instantiates a class by first allocating memory for a new object, and returning a reference to that memory afterwards. The `new` operator also invokes the object constructor. Its single postfix argument is the call to a constructor; the name of this constructor determines the name of the concrete class to instantiate. Thus, it is defined already at *compile-time* which particular constructor is called when constructing a new object.

Therefore, JINSI can statically determine the called constructor already during instrumentation, and therefore knows exactly which class will be instantiated. JINSI does not require any run-time check or any other sophisticated analysis and can simply surround the `new` statement with probes that capture the construction of observed objects. Consequently, constructors of unobserved classes can be left untouched safely.

Listing 7.7 on page 138 contains a basic example of a call side. First, an unobserved object is instantiated, afterwards this object is passed as argument while calling `new` to instantiate an observed object.

```
public class CallSide {

  public void someMethod() {
    Unobserved unobserved = new Unobserved();
    Observed observed = new Observed(unobserved);
  }

}
```

Listing 7.7: Call side of an incoming constructor call.

Because the call to `new` specifies the name of the class to be instantiated, JINSI already knows during instrumentation that the first call goes to an unobserved constructor, whereas the second call goes to an observed one. Thus, JINSI only surrounds the second instantiation with probes (1) to capture the incoming call that is going to occur, (2) to capture the newly instantiated observed object, and (3) to capture possible exceptions which may have been thrown during the observed instantiation.

A simplified instrumented version of the call side can be found in Listing 7.8 on page 140. Capturing in that example works as follows:

1. JINSI captures all information that is required to replay the call (target type, parameter types, and actual arguments) and stores the event's id in a local variable. Later, this id is used to link the corresponding return event with the initial call event.

2. The call to `new` is surrounded by a new try-catch-block to capture possible exceptions.

3. The actual instantiation is executed; thus, `new` is called.

4. The very next statement after the successful instantiation is a call to JINSI to capture the newly created object. The parameters include the event id stored before, as well as the created object. JINSI cannot identify the object before it is constructed, therefore JINSI cannot assign an object id in the first step. Instead, this object id is obtained after constructing the object[10] and stored in the return event. Because the return event is linked with the call event via the stored event id, the call event in return is linked with the object id.

5. The closing catch block handles all *unchecked* exceptions, which can be thrown during the normal operation of the JAVA virtual machine. A method is not required to declare any unchecked exception that might be thrown during the execution of the method but not caught. Thus, by

[10]To be more precise, the object id may be obtained already at the first outgoing call during the instantiation.

catching all run-time exceptions, JINSI ensures that all possible exceptions are captured. The handler itself records the run-time exception and again uses the event id to link the throw event with the call event.[11]

```java
public void someMethod() {
  Unobserved unobserved = new Unobserved();
  EventId eventId = JINSI.
      recordIncomingConstructorCall(...);
  Observed observed = null;
  try {
    observed = new Observed(unobserved);
  } catch (RuntimeException e) {
    JINSI.recordIncomingThrowable(eventId, e, ...);
    throw e;
  }
  JINSI.recordIncomingConstructorReturn(eventId,
      observed, ...);
}
```

Listing 7.8: Capturing incoming constructor calls by surrounding the `new` statement with probes.

Using this relatively simple approach—compared to the various trials described before—JINSI is able to capture all incoming constructor calls. We can avoid all the pitfalls like calls to `super` or `this`, inheritance, and interleaving of different constructor calls. From JINSI's perspective, calls to `super` or `this` happen in a totally transparent and natural way.

However, this approach involves a downside in principle. As long as we

[11] If the method would declare any *checked* exception in its `throws` clause, JINSI would add this exception to the surrounding try-catch-block and thus could capture this type of exception as well (see Listing 8.4 on page 159 for an example).

observe custom JAVA code[12] only, this approach works as expected. Constructors declared in custom code can be called in the very same code only. There is no way to explicitly call the constructor of a custom class in the core JAVA libraries, for example, as that constructor would have to be already known. Consequently, to capture incoming constructor calls to custom code, JINSI has to instrument custom code only. However, if we want to observe core JAVA classes like `Vector`, we would have to instrument all the call sides of constructors declared in class `Vector`. Unfortunately, classes like `Vector` are instantiated in very crucial parts of the core JAVA library, and it is very error prone—as already discusses above—to instrument these parts. Therefore, with the this approach it is virtually impossible to observe all instantiations of classes like `Vector` as JINSI would miss constructor invocations in classes that cannot be instrumented for technical reasons. To solve this issue, JINSI handles classes like `Vector` similar to class `StringBuilder` (see Section 8.6.2 for details).

JINSI captures incoming constructor calls at their call side. In this way, we can avoid all the pitfalls like calls to `super` *or* `this`*, inheritance, and interleaving of different constructor calls.*

7.2 Capture Incoming Method Calls

While the approach to capture incoming calls by surrounding them with probes at their *call side* is well suited for observed constructors as shown above, it is in general not applicable to method calls. In contrast to constructor invocations, JAVA uses *dynamic method lookup* for (non-static) method calls, because *overriding* may occur. For such method invocations, an instance method is to

[12]Code developed by first-party developers and thus not contained in the core JAVA libraries. Usually, we are interested in observing that part of a program as we are interested in failures and defects, respectively, in the custom part of a program.

be invoked, and there is a target reference to a target object. This target object is looked up at *run-time* only.[13] Therefore, JINSI cannot decide in advance whether a possibly observed method will be invoked on an actual observed or on an unobserved object—whereas for constructor calls this is known at compile time. This issues becomes especially important as soon as we want to observe classes that implement interfaces.

For example, an observed class may implement a very common interface, like `Comparable`: this interface defines method `compareTo(...)` and imposes a total ordering on the objects of each class that implements it. That interface is defined by the JAVA Collections Framework and, for example, is used in sorted maps and sorted sets. If we would capture incoming method calls at the call side, we would have to surround every single call to method `compareTo(...)` with probes—in the whole program at places where this method *might* be called at runtime, including JAVA libraries and other third-party components that use `Comparable`. For each call, JINSI would have to check at run-time whether the actual target instance of `Comparable` is an instance of an observed class (thus, an incoming call), or an instance of an unobserved one that implements `Comparable` as well (thus, not an incoming call). Such interfaces play a crucial part in the JAVA Collections Framework and therefore are used frequently in JAVA programs. Thus, the issues imposed by dynamic method lookup are the rule rather than the exception.

Another issue would be imposed by methods in class `Object`, especially very common ones like `equals(...)` and `toString()`. Again, if we would capture incoming method calls at the call side, JINSI would have to surround every single call to these methods in the whole program. This would result in many locations where JINSI would have (a) to instrument the byte code, and (b) to check at run-time if the target object is observed actually—resulting in an in practice inapplicable approach. Therefore, JINSI implements a different

[13]See §15.12.4 "Runtime Evaluation of Method Invocation" in the JAVA language specification [31] for more details about how methods to invoke are located.

approach (see Section 3.1.5 for a detailed description on the conceptual level) and replaces all observed methods `m` by logging proxy methods `p` (having exactly the same signature as `m`), which basically capture the incoming call, and delegate the incoming call to the original method `m`.

Whenever an actual observed method `m` is dynamically looked up, JAVA will invoke the newly introduced logging proxy `p` instead. Listing 7.9 on page 143 contains a simple example of an observed method `m`. This method takes one parameter and returns a `String` value.

```java
public class Observed {

  public String m(Unobserved u) {
    return u.toString();
  }

}
```

Listing 7.9: Simple observed method that takes one parameter and returns String value.

Listing 7.10 on page 143 shows the JAVA code after being instrumented by JINSI: Method `m` (logging proxy `p`) captures the incoming call and delegates the call to method `mOriginal` (original method `m`); finally, the return value is returned to the caller.[14] For the caller it is completely transparent that it actually called a proxy method instead of the original one—the signature stays the same and eventually decides on the concrete implementation that is called by the JAVA virtual machine.

```java
  private String mOriginal(Unobserved u) {
    // instrumentation for outgoing call disregarded
```

[14]This is very similar to the approach described in Section 8.3 to capture incoming constructor calls within `this` by replacing it with a factory method.

```
    return u.toString();
  }

  public String m(Unobserved u) {
    EventId eventId = JINSI.recordIncomingMethodCall
        (...);
    try {
      // obtaining return value from original method
      String returnValue = this.mOriginal(u);
      JINSI.recordIncomingMethodReturn(eventId,
          returnValue, ...);
      return returnValue;
    } catch (RuntimeException e) {
      JINSI.recordIncomingThrowable(eventId, e, ...);
      throw e;
    }
  }
```

Listing 7.10: The original method `m` was replaced by a logging proxy with the same signature.

By introducing these proxy methods, JINSI is able to efficiently and effectively capture incoming method calls. However, we still have to resolve the following issue: What if an observed object calls method `m`, and therefore the proxy, on itself by invoking `this.m(...)`? With the approach as described above, JINSI would record this as an incoming call—although the interaction does not leave the instance at all. A possible approach would be to replace all calls to the proxy `p` by a call to the original method `m'`: in our example, `this.m(...)` would become `this.mOriginal(...)`. Then, instances of class `Observed` would invoke the non-logging original implementation `mOriginal` of `m` rather than the logging proxy. Still, another similar issue turns up: what if an instance of `Observed` calls method `m` on a different instance? What about inheritance? We will see how JINSI solves this issue in the following section. For now, we can record the intermediate result:

JINSI introduces logging proxy methods to capture incoming method calls. Thus, in contrast to incoming constructor calls, the probes are not introduced at the call side, but in the observed class itself.

7.2.1 Knowing the Caller of Observed Methods

As indicated above, introducing a logging proxy to capture incoming method calls is not sufficient. Because JINSI needs to know the caller of an incoming method call for later analyses, it has to record not only the called object (target, or callee), but the caller (source object) as well. Unfortunately, in JAVA it is not possible to dynamically obtain the caller of a method. Within a method, you only have access to `this` (callee) and the passed arguments.[15] Therefore, JINSI cannot avoid to instrument the call side of observed methods as well. Nevertheless, the required instrumentation is much more light-weighted than using probes at the call side.

JINSI implements class `CallRegistry` that keeps track of all callers of possibly observed methods. JINSI registers the actual caller at the call side right before a possibly observed method is called. A possibly observed method is either (1) a method declared in an observed class, (2) a method declared in a superclass of an observed class, or (3) a method declared in an interface implemented by an observed class.

By default, JINSI instruments calls only to those methods declared in superclasses of observed ones (see (2) above) that are either not declared in class

[15] Method `Thread.currentThread().getStackTrace()` returns an array of stack trace elements representing the stack dump of a thread. However, each element only contains the name of the class that *declares* that method. If subclass `B` of superclass `A` calls method `C.c()` in `A.a()` (method `a` is implemented in `A` but not overridden in `B`.), the visible caller of `c()` will be `A`, but not `B`. Thus, it is not possible to obtain the actual runtime type of a caller via this method.

Object, or that are declared in Object but overridden in some subclass at the same time. For example, if method Object.equals(...) is not overridden in an observed class or in one of its superclasses, calls to that method will not be registered and therefore the call side will not be instrumented: methods implemented in Object like toString, hashCode, and equals do not contain any interaction of interest and therefore can be omitted. In this way, the number of places where JINSI has to instrument the call side is kept minimal—especially, JINSI avoids to instrument crucial core JAVA classes.

Listing 7.11 on page 146 shows class Caller that calls a method declared in an observed class, before it calls method equals, which is not overridden by that observed class. Thus, JINSI registers the caller only at the first call, as shown in Listing 7.12 on page 146.

```java
public class Caller {

  public void method(Observed o) {
    o.observedMethod();
    o.equals(new Object());
  }

}
```

Listing 7.11: Call side of an observed class. Firstly, a method declared in that observed class, secondly the not overridden method equals is called.

```java
  public void method(Observed o) {
    JINSI.registerCaller(this, o, ...);
    o.observedMethod();
    // equals(...) not overridden in class Observed
    // -> call not registered
    o.equals(new Object());
    }
```

Listing 7.12: JINSI instruments only the first call at the call side.

As seen in Section 7.2, it is not desirable to instrument all call sides—on the other hand JINSI requires the caller. However, tracking callers is a good compromise. Instead of doing a runtime check at every single call to a possibly observed method, JINSI just keeps a reference to the caller. Only if an observed method is called (the dynamic lookup makes a preliminary selection), JINSI uses the stored reference to check if the call is an incoming one. Furthermore, introducing the call to the registry (a single method call without any conditional statement) is less complex than inserting a probe that does the actual runtime check. Therefore, this kind of instrumentation is less likely to fail—as seen in practice while implementing JINSI for JAVA programs and conducting the experiments.

Implementing the approach described above, JINSI is able to obtain information about the actual caller of observed methods. However, JINSI still might miss some callers. For instance, if an observed class implements an interface like `Comparable` (see Section 7.2), not all call sides are instrumented: crucial classes in `java.lang` or in the JAVA Collections framework are not instrumented because doing so is very error prone (as described in Section 7.1). If an observed method is called from such a class, JINSI will not have any information about the caller; or, the registered caller still refers to a method invoked before. If JINSI would use this outdated reference, the captured information would be incorrect and could lead to faulty replays.

To identify such situations, JINSI uses the reference to a caller only once in the method that captures the call (called by the logging proxy method, see above) and that does the runtime check. As soon as a reference is obtained from the registry, it is automatically marked as *tainted*. If the same reference would be queried a second time, an exception would be thrown indicating that the caller of an incoming call was missed. Furthermore, while registering a caller, JINSI checks if it was accessed before. If not and it was an incoming call, an exception is thrown indicating that a caller of an actual incoming call was registered but not used while recording a call via the proxy. By using these double checks, JINSI is able to control the complex interplay between the caller registry and the probes in the proxy methods, and can early detect possible errors in the instrumentation. Anyhow, while conducting the experiments, that error has never occurred.

JINSI combines logging proxy methods and tracking of callers to capture incoming method calls. It is not possible to completely avoid instrumenting the call side of methods, but compared to inserting probes, this combination provides a light-weighted and error-robust solution.

7.3 Handling Inheritance

We want to observe *objects* and their respective interactions. Therefore, JINSI has to instrument *superclasses* of observed classes as well. If JINSI would only instrument those classes that are directly declared as observed, it could miss interactions of interest: For example, let `A` be the superclass of `B`, and `B` again the superclass of `C`. Furthermore, let class `B` be observed, but not `A` and `C`. If JINSI would instrument for outgoing interactions in explicitly observed class `B` only, JINSI would miss all outgoing interactions originating in methods and constructors declared in class `A` but not overridden in `B`. Therefore, JINSI has to instrument for possible outgoing interactions in these superclasses, too.[16]

Because we do not know the runtime type during instrumentation, JINSI has to add runtime checks. For example, methods declared in class `A` that are instrumented for outgoing interactions, among others, can be invoked at runtime on instances of `A`, `B`, or even `C`. If such a method is invoked on an instance of

[16]The same holds for incoming interactions as well. For example, for incoming method calls, logging proxy methods and tracking of callers are used both in observed classes *and* their superclasses.

A or C, outgoing interactions will have to be omitted because these objects will be unobserved. On the other hand, if the method is invoked on an observed instance of B, outgoing interactions will have to be captured.

To keep the instrumentation as simple as possible—especially to avoid possible bugs introduced by JAVASSIST rather than to speed up the capture phase—JINSI does not introduce these runtime checks in the instrumented code itself. Otherwise, JINSI would have to introduce additional conditional statements to the probes. Instead, the probes pass source and target objects to JINSI, and the tool itself executes the runtime checks. Depending on their outcome, the initially intercepted interactions are either recorded to the log file, or omitted.

To be capable of capturing all interactions of interest, JINSI has to instrument both observed classes and their—possibly—unobserved superclasses. Runtime checks are used to decide during program execution whether intercepted interactions are incoming or outgoing and therefore to be recorded, or whether they can be omitted.

The above challenges and respective solutions are only a small sample from all the issues encountered while implementing the JINSI approach for JAVA programs. While the above findings represent some of the most important ones, the following Chapter 8 will complete the discussion.

8 Implementation Details

In this Chapter, we will complement the findings in Chapter 7; while the previous chapter focused on the most important challenges only, in the following we will discuss more issues and their respective solutions to complete the discussion:

- **Bytecode instrumentation.** JINSI has to modify the JAVA program to be debugged in order to intercept interactions during capture, and to trigger interactions during replay. The canonical approach for JAVA programs is to instrument their bytecode. Different tools and frameworks are available for this purpose—tools with different advantages and disadvantages. We will see which framework JINSI uses; this choice has different effects on the implementation.

- **Partial object information.** JINSI has to record data (e.g., arguments of method calls) that ranges from simple scalar values to complex and composite objects. We will see how JINSI is able to capture/replay interactions where complex external systems like databases are involved—without the necessity to have the database available during replay.

- **Calls within `super(...)` or `this(...)`.** The first statement in a constructor has to be a call either to `super(...)` or to `this(...)`. Therefore, further calls enclosed within such a call, like `this(new Observed())`, have to be handled in a special way. For instance, it is not possible to directly surround the incoming constructor call `new`

`Observed()` with probes—this incoming call instead has to be replaced by a static factory method that records the interaction.

- **Field accesses.** While capturing both read and write field accesses can be implemented by calling probes introduced next to the accesses, for replay JINSI instead has to replace the field accesses.

- **Interactions with arrays.** In the JAVA programming language, arrays are objects. However, they have some special properties which distinguish them from regular objects: arrays contain components, they are dynamically created, it is not possible to subclass them, and there is no way to manipulate their bytecode—as there is none. Observed objects may interact with arrays in different ways; therefore, during replay in general and minimizing in particular, the special characteristics of arrays have to be taken into account.

- **Strings and StringBuffers.** In JAVA, strings are represented by instances of class `String`. However, similar to arrays, class `String` has special characteristics. Due to these properties, JINSI has to treat strings like literal values. Moreover, the JAVA compiler automatically introduces instances of class `StringBuffer` to make string concatenation more efficient. This transparent, automatic transformation has to be taken into account during capture/replay.

- **Managing objects.** Both during capture and replay, JINSI has to identify and manage objects that are involved in the observed interactions. For this purpose, JINSI uses different object pool implementations: to look up object IDs for given objects during capture, and to look up objects for given IDs during replay.

- **Mock objects.** To replay interactions on unobserved objects, JINSI has to create objects that *mock* the unobserved objects' behavior as previously recorded. As we will see, especially the instantiation of such—in principle stateless—mock objects is not straightforward.

8.1 Bytecode Instrumentation

To implement capture/replay for JAVA programs, JINSI has to manipulate parts of the program under investigation. A straightforward way to perform the required manipulation would be to modify the program's source code. However, for programs implemented in the JAVA programming language, it is much easier to manipulate the bytecode[1] generated by the JAVA compiler, than to modify the relatively unstructured source code. Manipulating the bytecode has at least two advantages: first, we do not need the original source code, and second because JAVA is statically typed, the bytecode innately provides type information, which would be hard to obtain from the source code.[2]

For manipulating and transforming bytecode, several libraries and frameworks are available. Notable are at least the following ones:

- **ASM.** ASM is a JAVA bytecode manipulation and analysis framework [8]. Compared to other similar frameworks, it claims to be focused on simplicity of use and performance. Instead of using an object representation of the bytecode (as SERP and BCEL, two further bytecode manipulation libraries, do), ASM uses the Visitor and Adapter design patterns [43]. However, we still would have to take care of bytecode specific issues, as instructions are introduced directly at bytecode level.

[1] JAVA bytecode—in this work, bytecode in short—is the form of machine instructions that the JAVA virtual machine executes. The virtual machine in itself knows nothing of the JAVA programming language [47].

[2] Another option would be to use the JAVA Virtual Machine Tool Interface, which allows for native libraries to capture events and control the virtual machine. However, compared to bytecode manipulation, this method is relatively slow, as the JAVA virtual machine communicates with the native libraries via callbacks, while the manipulated bytecode is optimized by the virtual machine in the usual way.

```
MethodCall.replace("{$1 = 0; $_ = $proceed($$);}");
```

Listing 8.1: **JAVASSIST: Replacing a method call.** JAVASSIST provides a powerful source level API that makes it easy to manipulate method calls, for instance.

- **ASPECTJ.** ASPECTJ is an aspect-oriented extension to the JAVA programming language [62]. Using so-called pointcuts and advices, it is possible to introduce code that is run at well-defined moments in the execution of a program. A very early prototype of JINSI's capture feature used ASPECTJ to intercept method calls. However, it turned out that the provided pointcuts were not sufficiently precise and expressive to define all required locations where JINSI has to intercept interactions.[3]

- **JAVASSIST.** JAVASSIST is a unique class library for editing bytecode in JAVA [12]. In contrast to other similar bytecode editors, JAVASSIST provides two levels of API: bytecode level—as all other libraries do—and *source level* [13]. It uses a custom compiler that enables its user to directly apply JAVA source code to manipulate the bytecode. For example, using JAVASSIST it is relatively effortless to replace a method's body, a feature JINSI depends on.

The key feature of JAVASSIST is its powerful source level API. This makes the instrumentation much easier than, for instance, with ASM. JAVASSIST provides a special syntax like `$1` to access parameters, and `$proceed` to access a called method. For example, the simple snippet in Listing 8.1 on page 154 replaces the first argument of a method call with value 0.

[3]JINSI used ASPECTJ in version 1.5.0. Version 1.6.x, a major revision, has been available since April 2008 and may provide more required features that were not available in 1.5.0.

We decided to use JAVASSIST for the reasons outlined above. However, during implementation of JINSI, we stumbled on several severe bugs in JAVASSIST. Because of these bugs, some design decision had to be taken, like how to capture observed constructors (see Section 7.1 for details).

JINSI utilizes JAVASSIST to instrument the JAVA bytecode because of its easy-to-use and poweful source level API. However, JAVASSIST contained and still does contain some severe bugs, which implied decisions concerning the design of the instrumentation.

8.2 Partial Capture of Object Information

The type of data that flows through the boundary between the observed and the unobserved part of the application to be debugged ranges from primitive scalar values (for instance, integer value 5) to complex objects (for instance, a database abstraction). When replaying interactions that involve such data, JINSI has to provide proper arguments and proper values. Instead of capturing the entire information about objects, JINSI captures only the subsets of those objects that actually affect the computation, and approximates such subsets by capturing incrementally and on demand at run-time. Therefore, JINSI neither requires a sophisticated static analysis, nor requires to serialize complete object graphs (see Section 3.1.3 for details).

For example, consider a call to method `sumAllOrders(...)` defined by the observed class `OrderStatistics` in Listing 8.2 on page 156. The method queries table "orders" via the given SQL database statement and sums up all values in row "ordertotal". Briefly, it sums up all orders. Because the method does not open a connection to the database, the given statement has to point to a connection that has been opened already. Thus, the statement object must contain references to objects related to querying the database and the database itself. Capturing all this data, for example by serialization as mentioned above, would not only be expensive, but also impossible in this case. Usually, the natively implemented JDBC driver is a barrier that cannot be crossed from the JAVA part of the program. Moreover, capturing the whole database obviously would not be an option.

```java
package partialinformation;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class OrderStatistics {

  // ...

  public float sumAllOrders(Statement stmt) throws
      SQLException {
    float sum = 0.0f;

    // execute query against database
    ResultSet rs = stmt.executeQuery("SELECT␣*␣FROM␣
        orders");

    // iterate over all rows
    while (rs.next()) {
      // obtain actual order total and sum it up
      sum = sum + rs.getFloat("ordertotal");
    }

    return sum;
  }

}
```

Listing 8.2: Querying a database in an observed class.

Instead, for that call, all we have to capture are the dynamic type of the single argument `stmt`, the obtained result set `rs`, and the returned scalar values of methods `next()` and `getFloat(...)` called on this set. We even do not have to capture the whole result set: what affects the computation of the method's return value `sum` are only the scalar values returned by the two outgoing method calls to the set. Thus, the only data we need to store to replay are the boolean values returned by the calls to the result set's method `next()`, and the float values returned by the calls to its method `getFloat(...)`. For instance, if the result set would contain 100 orders, all we would have to store would be 100 boolean and 100 float values.

In general, we do not need to capture objects at all. Whenever an object is accessed, we just need to capture its dynamic type and its unique *object ID* that is automatically assigned by JINSI.[4] The object ID is used to differentiate between individual instances of the same class. In this way, we can collect all required information incrementally at *run-time*. In general, it is not possible to identify in advance (thus, statically) which subset of the information is relevant for later replay. We have to store the actual data only for scalar values, as seen in the above example. Thus, collecting object IDs, dynamic type names and scalar values is all we need to know about objects that cross the border of the observed part of the program. This minimal information is sufficient for later replay, and dramatically reduces the cost of the capture phase.

To replay data crossing the border between observed and unobserved parts, JINSI captures nothing but object IDs, type names, and scalar values. All required data is collected incrementally and at run-time.

[4]JINSI uses a hash map with *weak* references. Thus, JAVA's garbage collection is not hindered by references pointing from object references to IDs. See Section 8.7.1 for details.

8.3 Calls Within `super` or `this`

While Section 7.1 addressed most of the issues involved in instrumenting for incoming constructor calls, one further issue arises from alternate constructor (`this`) and superclass constructor (`super`) invocations.

As we have seen in Section 7.1, the first statement in a constructor body has to be either an explicit invocation of another constructor of the same class (`this`), or of the direct superclass (`super`). However, it is possible to call another constructor or a static method to instantly pass an argument to `this` and `super`, respectively. For example, a call like `this(new Observed())` is a legal invocation. Unfortunately, in this case JINSI cannot instrument for incoming constructor calls as described in Section 7.1.6. JINSI would have to capture the call and to assign the event ID before `this` would be invoked, resulting in invalid JAVA bytecode. Furthermore, JINSI uses the approach as described in Section 7.1.6 analogously to capture *outgoing* constructor calls as well. Thus, the same issue holds for outgoing constructor calls, as, for instance, in `super(new Unobserved()).`

Listing 8.3 on page 158 contains an example for incoming constructor calls that illustrates the above issue. Constructor `Unobserved(Object)` invokes alternate constructor `Unobserved(Observed).`

```
public class Unobserved {

  public Unobserved(Object obj) throws Exception {
    this(new Observed(obj));
  }

  public Unobserved(Observed observed) throws Exception
       {
    // ...
  }

}
```

Listing 8.3: Instantiating an observed class within `this`.

To capture the incoming call `Observed(obj)`, JINSI has to replace this call with a static method. The introduced method has to

1. take and pass the given parameter `obj`;
2. log the corresponding events;
3. instantiate the observed object;
4. return the created instance.

To implement this technique, JINSI does, in principle, an "extract method" refactoring and in this way introduces a private factory method for this specific constructor call in class `Unobserved`. Listing 8.4 on page 159 shows the refactored constructor, as well as the introduced static factory method. The factory method captures the events exactly as described in Section 7.1.6. This time, JINSI adds a catch-block for `Exception` because the constructors in class `Unobserved` explicitly declare this type in their `throws` clause. Additionally, the factory method has to return the instantiated object to its caller.

```
  public Unobserved(Object obj) throws Exception {
    this(factoryJINSI(obj));
  }

  private static Observed factoryJINSI(Object obj) {
    EventId eventId = JINSI.
        recordIncomingConstructorCall(...);
    Observed observed = null;
    try {
      observed = new Observed(obj);
    } catch (RuntimeException e) {
      JINSI.recordIncomingThrowable(eventId, e, ...);
      throw e;
    } catch (Exception e) {
      JINSI.recordIncomingThrowable(eventId, e, ...);
      throw e;
    }
    JINSI.recordIncomingConstructorReturn(eventId,
        observed, ...);
    return observed;
  }
```

Listing 8.4: The incoming constructor call was replaced by a capturing static factory method.

The same technique is analogously used to capture outgoing constructor calls like `super(new Unobserved())` in an observed constructor, as well as to capture incoming or outgoing calls in `this` and `super` to static methods.

JINSI extracts custom factory methods to capture incoming and outgoing calls within invocations of `this` *and* `super`*.*

8.4 Field Accesses

Next to constructor and method calls, JINSI has to capture and replay field accesses as well. To capture, JINSI *inserts* probes *next to* an access. By contrast, to enable replay JINSI has to *replace* the access.

8.4.1 Capture Field Accesses

Listing 8.5 on page 161 shows class `Observed` that executes an outgoing read access in method `readField`, and an outgoing write access in method `writeField`. To capture these accesses, JINSI has to insert probes that record the corresponding interactions.

```
public class Observed {

  private Object myValue;

  public void readField(Unobserved u) {
    this.myValue = u.value;
  }

  public void writeField(Unobserved u) {
    u.value = this.myValue;
  }

}
```

Listing 8.5: Observed class with outgoing read and write access to an unobserved field.

In Listing 8.6 on page 161, you can see the instrumented version of class `Observed`. JINSI has inserted probes that record the information required for later replay of the outgoing interactions. Compared to constructor (see Section 7.1.6) and method (see Section 7.2) calls, only one probe is sufficient to capture all required information; in this probe, JINSI tracks source and target objects (`this` and argument `u`), the accessed field's name (`value`), and the accessed value (`this.myValue` after and before the access, respectively).

```
  public void readField(Unobserved u) {
    this.myValue = u.value;
    JINSI.recordOutgoingFieldRead(this, u, "value",
        this.myValue, ...);
  }
```

```java

  public void writeField(Unobserved u) {
    JINSI.recordOutgoingFieldWrite(this, u, "value",
        this.myValue, ...);
    u.value = this.myValue;
  }
```

Listing 8.6: Instrumented class `Observed` with probes to record field interaction.

In the above example, we have seen how JINSI captures outgoing field accesses. To capture incoming accesses, JINSI applies an analogous instrumentation. For this purpose, JINSI has to instrument both observed and unobserved classes. While unobserved objects can only trigger an incoming access, observed ones can trigger both an incoming and an outgoing access within the same interaction: If unobserved object `o1` accesses a field in another, different observed instance `o2`, this will be both an outgoing and an incoming access. We will see in the following Section 8.4.2 how JINSI handles this situation during replay.

Like for method calls, JINSI has to instrument field accesses in superclasses of observed classes as well (see Section 7.3). However, in JAVA, there is a major difference between method calls and field accesses regarding *binding*: fields in JAVA are only hidden and not overridden—binding for fields is always *static*.[5] JINSI considers this fact when instrumenting field accesses.

To capture field accesses, JINSI inserts probes next to the access; the access itself is not modified. JINSI considers that, in JAVA, fields are hidden instead of overridden.

[5] ORACLE's JAVA tutorial writes on that matter: "Within a class, a field that has the same name as a field in the superclass hides the superclass's field, even if their types are different." ([50])

8.4.2 Replay Field Accesses

To replay field accesses, JINSI has to *replace* them:

- for outgoing accesses, JINSI replaces the field accesses with substitutive method calls to itself. Thus, JINSI instruments the JAVA bytecode of observed classes to *intercept* the outgoing field access;

- for incoming accesses, JINSI uses JAVA reflection to read or write the captured value.

In the following, we will concentrate on *outgoing* field accesses, as this case is more interesting.

If the access is an outgoing read, JINSI will receive the introduced method call and will query its event log for the capture value; this value is returned to the caller. For the observed object, it makes no difference whether it does an actual field access or calls the substitutive method. In both cases, the accesses value is obtained. If the access is an outgoing write, JINSI will receive the value to be written via the substitutive method call. However, the value is not written, but consumed by JINSI.

Listing 8.7 on page 163 shows class `Observed` as instrumented for replay. In addition to the substitutive method calls, the introduced code contains runtime checks that test if a field access is in fact outgoing. If not, the access is not redirected to JINSI but targets the actual object—for example, if an instance accesses a field on itself.[6]

```
public void readField(Unobserved u) {
  if (JINSI.isOutgoingFieldAccess(this, u, "value"))
      {
```

[6]This is a very common interaction in method `equals(Object)`. In this method, (private) fields of the given object are accessed to indicate whether the given object is equal to `this`.

```
      this.myValue =  JINSI.getFieldValue(this, u, "
          value", ...);
    } else {
      this.myValue = u.value;
    }
  }

  public void writeField(Unobserved u) {
    if (JINSI.isOutgoingFieldAccess(this, u, "value"))
        {
      JINSI.handleFieldWrite(this, u, "value", this.
          myValue, ...);
    } else {
      u.value = this.myValue;
    }
  }
```

Listing 8.7: For replay instrumented class `Observed`. Both field accesses will be redirected to JINSI if the access is outgoing.

As noted above, an observed object can access another, different observed instance, resulting in an outgoing *and* an incoming access, depending on the perspective of the interaction. During replay, this has to be taken into account: these two interactions belong together and JINSI has to handle them as one single unit. Furthermore, the outgoing access has to be received by the actual target object. If the interaction is a read access, the object doing the access has to read the value from the target object; if it is a write access, the value has to be written to the target.

If an outgoing access addresses another observed object, JINSI will intercept this access just like a sole outgoing access targeting an unobserved object. However, instead of querying the event log for the captured value or consuming the write access, JINSI uses JAVA reflection to forward the access again to the original receiver.

To replay field accesses, JINSI redirects outgoing accesses to itself by replacing these accesses with method calls to itself. While intercepting the access, JINSI either uses the event log to handle the captured value (outgoing access targets unobserved object), or again forwards the access to the actual target object (access targets different observed object.)

8.5 Arrays

In JAVA, arrays are objects in principle. However, they have some special properties and therefore take on a special position:

- arrays contain components;
- they are created dynamically;
- it is not possible to subclass them;
- and there is no way to manipulate their bytecode—there is none, because arrays are implemented natively.

The consequent main issue is that JINSI *cannot use arrays as observed objects*. While it is relatively easy to capture and *sequentially* replay outgoing interactions targeting arrays, *minimization* becomes very tricky. While minimizing, JINSI mainly has to replay *subsets* of the captured interactions, instead of the whole sequence; thus, interactions with arrays may be skipped. If we do not serialize whole array structures, it will be very difficult to reflect these skipped array accesses in a meaningful way in all cases—*without breaking causality*.

8.5.1 Capturing Array Accesses

Before JINSI can replay interactions that target arrays, JINSI has to capture those interactions. As we will see, JINSI can capture/replay *outgoing* array accesses only, as arrays are *invariably unobserved* objects. Thus, in the following, we will see how JINSI captures outgoing array accesses.

Listing 8.8 on page 166 shows an excerpt from an observed class that uses an array of type `byte[]`, referenced by its field `values`. The class' method `setValues(...)` can be used to set this field to a given array, method `transform()` iterates over the internal array and adds value `1` to every element, and method `init` creates an array of length 2 and initializes it with the given arguments `a` and `b`. In this short example, we can see how JINSI handles (1) arrays as parameter values, (2) read and (3) write accesses to arrays, and finally (4) array creation and initialization.

```
  private byte[] values;

  public void setValues(byte[] values) {
    this.values = values;
  }

  public void transform() {
    for (int i = 0; i < this.values.length; i++) {
      this.values[i] = (byte) (this.values[i] + 1);
    }
  }

  public void init(byte a, byte b) {
    this.values = new byte[] { a, b };
  }
```

Listing 8.8: Observed methods that use a byte array in field `values`. At construction time, the array is initialized with `null`. `setValues` uses an array as parameter, `transform` does a read access followed by a write access, and `init` creates and initializes the array.

When capturing incoming calls to method `setValues(...)`, JINSI captures minimal information about the given array: its object id, its type, and its length. Thus, in comparison to other objects, JINSI additionally captures the array's length. By default, JINSI does not capture any information about the contained elements. JINSI also is able to capture partial object information (object id and type / scalar values) about every element referenced by the array. Though, for large arrays (or, even multidimensional arrays) this can get very expensive—in JAVA, arrays can hold up to $2^{31} - 1$ elements—especially if an array would again contain other arrays (nested arrays). In the latter case, JINSI has to descend the arrays until a component type that is not an array type will be reached.

In method `transform`, JINSI *replaces* all read and write accesses to elements with method calls to JINSI. In contrast to other interactions (especially to field accesses, see Section 8.4), these accesses cannot be surrounded with probes. This is due to the functionality provided by JAVASSIST: it is not possible to surround array accesses with new JAVA code; it's only possible to replace array accesses completely.[7] At runtime, JINSI receives the redirected array access, captures the information about the access, and returns the value read from the array and writes the given value to the array—via JAVA reflection in both cases—respectively. Listing 8.9 on page 167 shows the instrumented version of method `transform`.

[7]See method `replaceArrayAccess(...)` in class `javassist.CodeConverter`.

```java
public void transform() {
  for (int i = 0; i < this.values.length; i++) {
    byte element =
      JINSI.captureArrayReadByte(this, this.values, i
          , ...) + 1;
    JINSI.captureArrayWriteByte(this, this.values, i,
        element, ...);
  }
}
```

Listing 8.9: Instrumented method `transform`. Array accesses are replaced by method calls to JINSI.

The array creation in method `init` uses a shortcut syntax to create and initialize an array. The length of the array is determined by the number of values provided between { and }, the elements are assigned in the given order. In fact, at the bytecode level, this creation looks similar to the JAVA code in Listing 8.10. JINSI again instruments the array write accesses by replacing them with method calls. The creation is not instrumented at all because there is no interaction of interest, as it is not necessary to explicitly replay the creation of arrays, as we will see in Section 8.5.2.

```java
private void init(byte a, byte b) {
  // was: this.values = new byte[] { a, b };
  this.values = new byte[2];
  this.values[0] = a;
  this.values[1] = b;
}
```

Listing 8.10: Unfolded array creation in method `init`.

If we would want to observe arrays, thus, to capture array accesses as *incoming* interactions as well, we would have to instrument virtually all array accesses in the whole program unless we would apply some sophisticated static analysis. Because we cannot instrument arrays for incoming interactions as for methods, we would have to instrument all call sides of possible array accesses. For some widely used methods like `arraycopy` in `java.lang.System`[8] this is not possible, because this method is implemented natively.[9] Therefore, arrays always are unobserved objects; it is not possible to include arrays in the set of observed classes.

JINSI captures array read and write accesses by replacing them with method calls to JINSI. The creation of arrays is not captured. Arrays always are unobserved objects, they cannot be treated as observed.

8.5.2 Replaying Array Accesses

Replaying the previously captured array accesses is relatively straightforward. Similar to capturing array accesses (and similar to replaying field accesses, see Section 8.4), JINSI replaces the array accesses as shown in Listing 8.11 on page 169.

```
public void transform() {
  for (int i = 0; i < this.values.length; i++) {
    byte element =
      JINSI.replayArrayReadByte(this, this.values, i,
           ...) + 1;
    JINSI.replayArrayWriteByte(this, this.values, i,
        element, ...);
  }
```

[8]This methods provides a native implementation of copying the contents of arrays. It is much faster than iterating over all elements in JAVA code and therefore is preferred over this manual copy.

[9]We could replace this call to a custom method that provides the same functionality but is not native. However, then we would have to redirect all respective calls—including in core JAVA classes, which cannot be instrumented for technical reasons.

```
    }
```

Listing 8.11: For replay instrumented method `transform`. Analogously to capturing, array accesses are replaced by method calls to JINSI.

This time, instead of recording information about the access, the information passed to JINSI is used to query the event log for the originally captured value. Thus, when `replayArrayReadByte` is called, JINSI uses the passed information to get the originally accessed byte value from the event log. Afterwards, the obtained value is returned to the caller and replaces the array read access. For the instrumented observed object, this happens completely transparently. The call to `replayArrayWriteByte` is used for synchronization purposes only. Because we intercept *all* array accesses, subsequent read accesses will not target the actual array but will be redirected to `replayArrayReadByte` again. In fact, JINSI does not write the passed value to the array at all.

You may have noticed that JINSI does not instrument the access to the array's field `length`[10], neither during capture nor during replay. When replaying method `setValues`, JINSI uses the captured information to create an empty array of proper length and type, and passes this empty array while invoking the method. If the observed object reads the array's `length`, the array itself will return the proper value. Because we redirect all accesses to the array's elements, the observed object does not access the empty components, but calls JINSI instead. For read accesses, JINSI again returns the captured value. From the observed object's perspective of view, it makes no difference whether the object accesses the array, or calls the appropriate methods on JINSI instead.

Method `init` is instrumented in a similar way. The two write accesses are replaced by method calls. As for capture, the creation of the array is not instrumented because there is no interaction of interest.

[10]In fact, currently with JAVASSIST it is not possible to manipulate accesses to field `length` of arrays. In JAVA bytecode, this field access is represented by a special bytecode instruction and therefore differs from field accesses on regular objects.

```
Observed o = new Observed(); // o.values: null
o.setValues(new byte[] { 1, 2 }); // o.values: { 1, 2 }
o.init((byte) 7, (byte) 8); // o.values: { 7, 8 }
o.transform(); // o.values: { 8, 9 }
```

Listing 8.12: Example for incoming calls on observed object that uses an array, see Listing 8.8 for its source code.

Listing 8.12 on page 171 shows a short sequence of incoming calls to the observed object that uses an array as declared in Listing 8.8 on page 166. After creating the observed object `o`, method `setValues` is called to set the array to elements {`1, 2`}. Immediately afterwards, method `init` is used to overwrite these values with {`7, 8`}. Finally, method `transform` changes these values to {`8, 9`}. When replaying these interactions sequentially, the following happens:

1. JINSI creates the observed object `o`. Thus, we have object `o` and its field `values` refers to `null`.
2. JINSI calls method `setValues` with an *empty* array of component type `byte` and length `2`. Field `o.values` now refers to this empty array.
3. JINSI calls method `init` and passes byte values `7` and `8`. Again, field `o.values` is assigned to a *new* empty array of length `2`. Because the given values are not written to the array, it still refers to an empty one.
4. While method `transform` is running, object `o` accesses field `length` on the empty array and obtains its actual length `2`. Now, instead of accessing the empty array, the read accesses are redirected to JINSI. JINSI queries the event log for each loop iteration, finds value `7` and `8`, respectively, and returns this value. Thus, although the referenced array is empty, the proper values are used within the `for` loop. The computation results in correct values `8` and `9`, respectively.

On the basis of this short example, we have seen how different array accesses are replayed. While this example may appear artificial, it is quite brief and nevertheless demonstrates important aspects regarding capture/replay of array accesses. Furthermore, as we will see in the following section, even this simple example causes fundamental challenges during minimization.

JINSI replays array read and write accesses analogously to capturing. Array read and write accesses are redirected to JINSI; for read accesses, JINSI queries its event log and returns the captured value. It is not necessary to explicitly replay the creation of arrays.

8.5.3 Minimizing Array Accesses

While sequential replay as described above is straightforward, minimizing is much more problematic and thus requires a more sophisticated handling of array accesses. As we have seen, we can *sequentially* replay the interactions in Listing 8.12 on page 171 without changing the semantic of the program run. However, when replaying *subsets* during minimization, several problems occur. For example, if we replay interactions 1, 3, and 4, replay will still work. However, if we replay interactions 1, 2, and 4, causality will be broken:

Interaction 1. JINSI creates instance `o` and `o.values` refers to `null`.

Interaction 2. JINSI calls method `setValues` and passes an empty array of length `2`; `o.values` refers to this empty array.

Interaction 4. In method `transform`, `o` accesses the `length` of the array instance previously given via `setValues`. The proper value `2` is obtained. However, this time the redirected read accesses return wrong values: During capturing, JINSI stored the accessed values `7` and `8` set by `init`. Now, `init` is not called at all and values `1` and `2` should have been set by `setValues` instead. Thus, in the `for` loop wrong values

7 and 8 are used. Furthermore, the computation leads to wrong values 8 and 9, instead of the expected values 2 and 3.

If we assume that one of these values, 8 and 9, is relevant for causing a failure in the observed instance, the minimization would classify the call `o.init((byte) 7, (byte) 8)` as irrelevant, while the actually irrelevant call `o.setValues(new byte[] { 1, 2 })` would be classified as relevant. Thus, *causality would broken*. In the following, we will see how JINSI deals with this fundamental issue.

8.5.3.1 Taking Object IDs into Account

JINSI could detect the above mismatch by comparing the object IDs of the accessed arrays. Then, JINSI would detect that the array access in `transform` reads from the wrong array instance: `init` creates a new array and the accesses in `transform` are expected to target this array, but not the one set by calling `setValues`.[11] However, the same issue would still hold in other cases. Listing 8.13 on page 173 shows method `add` as declared by the observed class used in the array examples. It adds given values to the array starting at index 0 until the array is filled up.

```
private int index = 0;

public int addValue(byte val) {
  if (this.index < this.values.length) {
    this.values[this.index] = val;
    this.index++;
  }
  return this.index;
}
```

[11]Delta Debugging would treat this situation as an *unresolved* outcome.

Listing 8.13: Method `add` adds given byte value to the array.

We can replace the call `o.init((byte) 7, (byte) 8)` with two corresponding calls to `add` as shown in Listing 8.14 on page 174. Because method `add` does not create a new array (in contrast to method `init`) the very same array as passed by calling `setValues` is used to write the given values to. Therefore, comparing object IDs of arrays would not help resolving that issue at all.

```
1 Observed o = new Observed(); // o.values: null
2 o.setValues(new byte[] { 1, 2 }); // o.values: { 1, 2 }
3 o.addValue((byte) 7); // o.values: { 7, 2 }
4 o.addValue((byte) 8); // o.values: { 7, 8 }
5 o.transform(); // o.values: { 8, 9 }
```

Listing 8.14: Instead of calling `init` as in Listing 8.12 on page 171, `add` is called twice.

Again, the interactions in lines 1, 2, and 5 could reproduce the original failure, although the calls to add values `7` and `8` would be missing, just as before. This is caused by the way how JINSI handles array accesses. Because—by default— it records no information about the actual elements, JINSI in that case can use only empty arrays while passing them as arguments to incoming calls. Furthermore, because JINSI passes empty arrays only, read accesses have to be redirected to JINSI. As mentioned before (see Section 8.5.2), capturing information about the individual elements may be very expensive. However, let's see whether we can fix causality by *capturing array component values*.

8.5.3.2 Recording Information About Individual Elements

If JINSI would record the individual elements as well (actual values for primitive types and partial object information for complex types, see Section 8.2.), it would not be necessary anymore to replace the array accesses with calls to JINSI. While replaying, the observed objects would *access the elements directly*, both for read and write accesses. The minimization of the examples in Listings 8.12 on page 171 and 8.14 on page 174 would result in proper minimal sets because the actual and up-to-date elements would be used for computation.

However, capturing array elements still may not be sufficient, as we will see in the following example. Listing 8.15 on page 175 shows a further observed method that interacts with array `this.values` (see Listing 8.8 on page 166): Method `readValues` uses a `FileInputStream` to read bytes from a given file and to store these values into the array referenced by field `values`. The array is passed to void method `is.read`—provided by the stream used to read from the given file—which reads bytes of data from the input stream into the given array of bytes. Thus, array `this.values` is changed *outside of the observed object's scope*.

```
  public void readValues(File file) throws IOException
      {
    InputStream is = new FileInputStream(file);
    is.read(this.values);
  }
```

Listing 8.15: Method `readValues` copies byte values from a given file to the array.

Having method `readValues`, we can replace the call to `init` in Listing 8.12 on page 171, as well as the two calls to `add` in Listing 8.14 on page 174. Listing 8.16 on page 175 shows the resulting sequence of calls.

```
Observed o = new Observed(); // o.values: null
o.setValues(new byte[] { 1, 2 }); // o.values: { 1, 2 }
o.readValues(new File("file.txt")); // o.values: { 7, 8
     }
```

```
o.transform(); // o.values: { 8, 9 }
```

Listing 8.16: Instead of calling `init` as in Listing 8.12 on page 171, `readValues` is called.

If we call `o.setValues(new byte[] { 1, 2 })` followed by a call to `o.readValues(...)`, the observed object would at first initialize the array with {`1, 2`}; afterwards, it would store the given file's first two bytes in the array. If these would be {`7, 8`} again, a subsequent call to method `o.transform` would in turn result in values {`8, 9`}—as before. If JINSI would capture these interactions including actual array values, regarding the array it would record:

- For the call to method `setValues` the actual array {`1, 2`} *including* the scalar values as argument.
- For the call to `readFile`, only partial information about the passed file object and about the outgoing calls. JINSI would record {`1, 2`} as argument of the outgoing call to `is.read(...)`. However, the array is changed *within this unobserved* method—outside of the observed object's scope.
- In method `transform`, no information about the array accesses would be recorded at all, because the observed object would directly interact with the array.

Especially, in method `readFile` JINSI could not observe the change to the array's elements because this change happens in the unobserved part of the program. If we would replay these interactions *sequentially,* JINSI would trigger the following interactions:

- JINSI would call `setValues` with the recorded elements {`1, 2`}; this time, `o.values` would point to the array {`1, 2`}, instead of an empty one.

- While replaying `readValues`, JINSI would receive the outgoing call to `is.read` and would do—nothing, as *void* method `is.read` changes the given array as a *side effect*. Thus, `o.values` would still refer to {`1, 2`} instead of the values {`7, 8`} as in the original run.

- In `transform`, the observed object would access the actual array elements: {`1, 2`}. However, in the original run these were {`7, 8`}. Therefore, the computation would be wrong.

In this small example, for arrays accessed *directly* by the observed object, JINSI did not intercept accesses to arrays, but the actual array elements were used for replay. However, then even the basic[12] sequential replay does not work anymore. Thus, we have to extend the approach of capturing whole array structures by handling of side effects in unobserved methods.

A possible next step to solve this issue would be to include so far unobserved class `FileInputStream` into the set of observed classes. Then, the instance of that class used to fill the array would not be mocked and thus replaced by JINSI, but would in fact fill the array. However, this class uses private *native* method `readBytes(byte b[], ...)` to do the actual read from the file—which cannot be observed in principle. Thus, the same issue holds as before: JINSI could not properly capture/replay some crucial interactions.

As mentioned above, JINSI does not capture array component values *by default*. However, in such cases as described above where capturing only the array accesses would not be sufficient, JINSI indeed records whole arrays and their respective values by serializing[13] array structures:

[12]In fact, sequential replay is fundamental. Delta debugging tests both the empty and the complete sequence of interactions. It expects that the empty sequence results in a passing run, the complete set in a failing run; no circumstance at all must not reproduce the failure, while taking into account all circumstances has to reproduce the failure.

[13]JINSI captures partial information about objects contained in arrays, and serializes the actual values of arrays of primitive values, as byte arrays used in the examples.

Arrays passed to observed methods. If an array is passed as argument to an observed method, like to method `setValues(...)` in Listing 8.8 on page 166, JINSI will record the array values during capture. During replay, it will re-create the array and its values, and pass it to the method as argument. The observed method will access the actual array values.

Arrays returned by outgoing method calls. Analogous to the previous situation, if an array is returned to an observed object, JINSI will capture the array values, and return the re-created array with actual values to the observed object.

Arrays passed to unobserved methods. If an array is passed to an unobserved method that might change the array outside of the observed object's scope as *side effect*, like method `is.read` in Listing 8.15 on page 175 does, JINSI will capture the whole array *right after* the unobserved method returns control to the observed object. Thus, JINSI captures the array's state after the outgoing call. During replay, JINSI will analogously fill the passed array with the recorded values in order to preserve the state as seen during capture. The observed object will access the—possibly—changed values.

When JINSI records array structures and replays array accesses, it does not replace these accesses by method calls, but observed objects actually interact with the arrays, which in that case indeed contain the actual values. Please note that field accesses that involve arrays are handled in a similar way.

While this fallback solution may result in performance losses—because potentially large arrays have to be serialized—it is required in some cases to preserve causality both during sequential replay and minimization.

When actual array values are not recorded, causality may be broken while replaying incoming interactions that involve accesses to arrays. To solve this issue, JINSI captures whole array structures as fallback solution.

8.6 Strings

Like values of a primitive type and `null`, strings are *literals* in JAVA: strings can be represented as a fixed value in source code, for example `"I am a string"`. At the same time, a string is an instance of class `String`, which represents a sequence of Unicode characters. String literals are references to instances of class `String`. Thus, strings are objects in JAVA. `String` objects have a constant value, they always refer to the same instance. More generally, `String` objects that are the values of *constant expressions*, like `"I am " + "a string"`, are *interned* so as to share *unique instances*.[14]

It is important to distinguish between literal strings, strings computed by concatenation at run time, and strings computed at compile time:

Literal Strings. *Literal* strings within the same class or different classes represent references to the *same* `String` object.

Compile time. Strings computed by constant expressions, as in the above example, are computed at *compile time*. These strings are treated as if they were literals. Thus, they are *interned*.

Run time. By contrast, strings computed by concatenation at *run time*[15] are newly created and therefore *distinct*.

The latter can be explicitly interned by calling method `intern()` on the computed `String` instance; after interning a computed string, it is the *same* one as any literal string with the same contents.

As other literal values and their wrapper classes, respectively, strings are *immutable*. The only way to change a string is to create a new instance and point to that. For example, methods like `toLowerCase()` do not change

[14]Using the *native* method `String.intern()`. This method returns a canonical representation and maintains a pool of strings.

[15]Like `"I am " + aString`, whereas `aString` is a non-constant variable.

the `String` object the method is called on, but rather return a new instance. In typical JAVA programs, Strings are *omnipresent* and their immutability has some concrete advantages:

Multithreading. If strings would be mutable, they could not be shared between threads without expensive locking. Different threads can work on a shared string at the same time without any possibility of conflict.

Security. It is not possible, in contrast to a mutable `Set` instance for example, to pass a string and then to change it unnoticeably afterwards. This is particularly important in high security applications.

Memory. Because strings are interned automatically, duplicates point to a single instance, saving memory.

Performance. Substrings can be created without any copying. For instance, JAVA's implementation of `substring` is very fast because a substring just points into its source string that is guaranteed to never change.

Strings are immutable. Literal strings and strings computed at compile time are interned. Only strings computed by concatenation at run time are newly created.

8.6.1 Capture and Replay of Strings

During capture/replay, JINSI treats instances of class `String` in a special way: it treats them *as literal values*, they therefore behave in their *natural* way. Methods called on strings, like `contains(...)`, are not intercepted, neither during capture, nor during replay. This approach solves many issues that would occur otherwise:

Observing strings. To preserve the actual behavior of strings during replay, we would have to include class `String` as observed. Otherwise, methods like `contains(...)` would return captured return values in all cases, regardless of the actual string value. However, some computation involving strings could change due to simplification, resulting in different string values. In this case, causality may be broken.

Incoming interactions. If we would have to treat `String` instances as observed, we would have to instrument virtually *all* call sides of methods declared in that class. However, as `String` is omnipresent, this would be practically impossible to implement.

Instrumenting `String`. We also would have to instrument class `String` itself, which is not possible for technical reasons, as the JAVA virtual machine make assumptions about the class' internal structure. Furthermore, crucial method `String.intern()` is native, and therefore cannot be instrumented using bytecode manipulation.

As opposed to objects in general, JINSI therefore does not only capture type and object ID of a string that crosses the border between the unobserved and the observed part, but its actual *content* as well. Furthermore, JINSI passes real strings as arguments of incoming method calls and returns real strings as return values of outgoing interactions. Because class `String` is a direct child of `Object` and it is final, it can be decided at instrumentation time if a call will target a string at runtime. Calls to methods defined in `String` therefore can be completely ignored during instrumentation; at runtime, calls to `String` instances will target the actual string. Thus, calls on `String` objects like `contains(...)` target the actual string value; the returned boolean value indicates whether the given substring is in fact contained within that string or not.

For instances of class `String`, JINSI uses real values. Thus, JINSI treats string values as primitive values. Respective method calls are completely ignored during replay/capture, and target the actual `String` objects.

8.6.2 StringBuffer and StringBuilder

In the context of how to handle instances of class `String`, it seems natural to discuss classes `StringBuffer` and `StringBuilder` as well. While `String` represents an immutable sequence of characters, `StringBuffer` and `StringBuilder` represent *mutable* sequences. The JAVA compiler takes advantage of this fact to optimize string concatenation.[16]

If JAVA would use instances of `String` to concatenate n strings via the + operator, it would have to create $n-1$ intermediate `String` objects during concatenation, because `String` is immutable. To avoid dispensable strings, which would have to be removed anyway by the garbage collection afterwards, JAVA uses mutable instances of `StringBuffer` and `StringBuilder`, respectively.[17] In the following discussion, we go into detail regarding class `StringBuilder` only; because class `StringBuffer` provides the same API and functionality except for thread-safety, this discussion applies to this class just as well.

Listing 8.17 on page 183 shows two example methods that concatenate strings. While method `concatenationPlus()` utilizes the + operator, method `concatenationBuilder()` uses `StringBuilder` explicitly. The call to method `toString()` at the end returns the concatenated `String`

[16]See §15.18.1 "String Concatenation Operator +" in the JAVA language specification [31].

[17]Prior to JAVA 5, thread-safe `StringBuffer` was used. Since JAVA 5, `StringBuilder` is used instead. The latter provides a faster drop-in replacement for `StringBuffer` that does not use synchronization. Thus, if a sequence of characters is used by a single thread (as is the case of this optimization), no synchronization is needed and therefore the faster `StringBuilder` can be used without any possibility of risk.

instance. As mentioned above, if JAVA would use `String` instances for concatenation, it would have to create two dispensable, immutable instances in the first method. However, as you can see in Listings 8.18 on page 183 and 8.19 on page 184, JAVA uses mutable instances of `StringBuilder` in *both* cases.

```
public static void concatenationPlus() {
  String comma = ",␣";
  String s = "Hello" + comma +  "world!";
}

public static void concatenationBuilder() {
  String comma = ",␣";
  String s = new StringBuilder().append("Hello").
      append(comma).append("world!").toString();
}
```

Listing 8.17: Methods that concatenate strings by using the + operator and a mutable instance of `StringBuilder`, respectively.

```
  NEW java/lang/StringBuilder
  DUP
  LDC "Hello"
  INVOKESPECIAL java/lang/StringBuilder.<init>(Ljava/
      lang/String;)V
  ALOAD 0
  INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/
      lang/String;)Ljava/lang/StringBuilder;
  LDC "world!"
  INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/
      lang/String;)Ljava/lang/StringBuilder;
  INVOKEVIRTUAL java/lang/StringBuilder.toString()
      Ljava/lang/String;
```

Listing 8.18: Decompiled method `concatenationPlus` (excerpt). Behind the scenes, JAVA uses a mutable instance of `StringBuilder` to concatenate the strings.

```
1       NEW java/lang/StringBuilder
2       DUP
3       INVOKESPECIAL java/lang/StringBuilder.<init>()V
4       LDC "Hello"
5       INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/
           lang/String;)Ljava/lang/StringBuilder;
6       ALOAD 0
7       INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/
           lang/String;)Ljava/lang/StringBuilder;
8       LDC "world!"
9       INVOKEVIRTUAL java/lang/StringBuilder.append(Ljava/
           lang/String;)Ljava/lang/StringBuilder;
10      INVOKEVIRTUAL java/lang/StringBuilder.toString()
           Ljava/lang/String;
```

Listing 8.19: Decompiled method `concatenationBuilder` (excerpt). Except for the constructor call in line 3 (the empty one is called), a mutable instance of `StringBuilder` is used in the same way as in Listing 8.18 on page 183.

As a direct consequence of this automatic optimization, JINSI has to treat instances of class `StringBuilder` in a similar way as instances of class `String`. To minimize interactions that involve outgoing calls to instances of `StringBuilder`, they would have either to be observed, or to behave in their natural way—interactions to these instances must not be answered by JINSI but by the actual instance. Otherwise, JINSI would always return the captured value if an observed object would call `toString()` on a `StringBuilder` instance. Because JAVA replaces string concatenations with these instances automatically and thereby makes `StringBuilder` omnipresent, it is virtually certain to encounter this issue—as seen during experimentation.

Similar to class `String`, it is practically not possible to instrument class `StringBuilder` to observe all its interactions. A better and much faster approach would be to let interactions between observed objects and instances of `StringBuilder` simply untouched—as for `String` instances. Then, calls from an observed object would target a `StringBuilder` instance directly, and this object could concatenate strings without any troublesome side effect. However, as shown in Listing 8.20 on page 185, JINSI would have to analyze which interactions have to stay untouched and which ones have to be intercepted. It is not sufficient to just let pass all interactions. In method `getAsHeading()`, the observed object uses a `StringBuilder` only *internally,* reference `b` is not passed to any other object. Thus, these interactions must not be intercepted by JINSI but target the actual builder. In contrast, in method `append(...)` a given builder passed from *outside* the object is used. Thus, if this builder would be created and passed from an unobserved object—it would have to be mocked therefore—the interactions would have to be intercepted. In sum, whenever a builder is used internally only, the interactions must stay untouched; whenever a (mocked) builder is passed from the unobserved to the observed part, the interactions must be intercepted by JINSI.[18]

```
public class Observed {

  public String getAsHeading() {
    // return "<h1>" + this.toString() + "</h1>";
    StringBuilder b = new StringBuilder();
    b.append("<h1>");
```

[18]For `String` instances, JINSI does not need to differentiate between the two cases as for `StringBuilder`, because *actual* strings are always used when passed as argument.

```
    b.append(this.toString());
    b.append("</h1>");
    return b.toString();
  }

  public String append(StringBuilder b) {
    b.append(this.toString());
    return b.toString();
  }

}
```

Listing 8.20: Observed class that uses `StringBuilder` instances both only internally and visibly to the outside.

To solve this issue, JINSI, treats `StringBuilder` in a special way:

- In observed objects, JINSI leaves object instantiations of—thus, constructor calls to—class `StringBuilder` *untouched* during instrumentation. At runtime, actual instances instead of mocked ones will be constructed. Furthermore, subsequent calls to these actual instances will not be intercepted. Consequently, such *internally* created instances behave in their natural way, similar to instances of class `String`.

- Whenever an instance of class `StringBuilder` that was mocked before is passed as argument to an incoming call, JINSI will detect this case and will intercept and handle subsequent outgoing calls targeting this argument in the usual way.

- To handle further usages of class `StringBuilder`, JINSI provides a custom implementation providing the same API, which can be observed without causing any (technical) issues. This custom implementation can be used as drop-in replacement if necessary.

In this way, JINSI's instrumentation does not interfere with the optimization done by the JAVA compiler.

Not only `String` *instances have to be handled in a special way, but instances of* `StringBuilder` *as well. Indeed, JINSI ensures that internally used instances behave in their natural way, while externally constructed instances will be mocked.*

8.7 Managing Objects

Both during capture and replay, JINSI maintains an *object pool*[19] that associates numeric unique identifiers (IDs) with objects:

Capture: JINSI uses a pool to look up IDs for given objects, so as to store object information in the event log.

Replay: JINSI uses a pool to look up objects passed as arguments or objects that are receivers of incoming interactions, specified by their IDs in the event log.

While the requirements for both cases appear to be very similar at a first glance, in each case JINSI uses a different pool implementation with different properties and features.

8.7.1 Pooling of Recorded Objects

During capture, JINSI uses an object pool implementation that maps objects to IDs (`Object` $\longmapsto$ `number`). This implementation enables JINSI to look up IDs for given objects. The pool implementation itself is backed up by a `ReferenceIdentityMap`, a `java.util.Map` implementation that first

[19]See Section 3.1.2 for a discussion on the conceptual level.

allows mappings to be removed by the JAVA garbage collector, and second matches keys and values based on == instead of `equals()` [18]. This implementation therefore has the following two key advantages:

Garbage collection. The first property ensures that JINSI does not retain references to objects that are not used by the program under investigation anymore; otherwise the garbage collector could not clean up these objects. For this purpose, the type of reference used for keys (thus, objects) is weak, the one for values (thus, IDs) is hard. As soon as the garbage collection determines that an object is weakly reachable, it will clear all weak references to that object.[20] Thus, these week references used by the pool do not prevent the garbage collector from reclaiming memory used by recorded objects. Furthermore, as soon as a key is garbage collected, the map will automatically purge the respective value.

Comparing by ==. The second property ensures that keys in this map are compared using ==. This violates the detail of various `Map` and map view contracts. However, the pool has to guarantee that different objects get different IDs, even if they are equal according to their `equals()` method.

As soon as an object is about to be recorded (see Section 8.2), JINSI queries the object pool for the corresponding ID. If the given ID was requested before, the pool will return the previously assigned ID as stored in its map. Otherwise, the pool assigns a new unique ID, associates the object and its ID in its map, and eventually returns the ID. In this way, different objects get different IDs, whereas the same object gets the same ID. In this way, JINSI is able to *unambiguously* assign interactions to their corresponding objects.

[20]For details, see the API documentation of package `java.lang.ref` in general, and the one of `java.lang.ref.WeakReference` in particular.

During capture, JINSI uses an object pool that maps objects to their unique IDs. The pool is backed up by a map that allows mappings to be garbage collected and compares keys using ==.

8.7.2 Pooling of Objects Involved in Replay

During replay, the requirements for the object pool are different. First, instead of mapping objects to their IDs, this object pool maps IDs to their corresponding objects (`number` $\longmapsto$ `Object`). While during capture JINSI has to look up IDs for given objects, for replay it has to look up objects for given IDs that are listed in the event log—thus, the mapping works the other way around. Second, JINSI cannot use weak references because otherwise JAVA's garbage collector would clean up objects prematurely.

To replay interactions, JINSI has to keep references to the objects that are involved in these interactions. For example, when replaying an incoming method call, JINSI has to have available the object that will receive the call, as well as the objects passed as arguments, if any. For outgoing interactions that returned an object, JINSI has to have available the proper object, in order to return it in place of of the unobserved target object as during capture. Obviously, before JINSI can look up an object, it has to be added to the pool.

Whenever JINSI replays an incoming or outgoing constructor call, it adds the constructed object to the pool; the corresponding ID can be found in the event log. In case of an incoming constructor call, JINSI uses reflection to call the object's constructor and afterwards puts the newly constructed object into the pool. If it is an outgoing constructor call, JINSI creates a mock object (see Section 8.8) and associates the ID with the mocked object. The same will happen if a required argument is not already contained in the pool, or if an outgoing method call or field access targets an object that is not already contained in the pool (an outgoing interaction may target an object that was constructed in the unobserved part, and whose construction was not recorded therefore).

In general, whenever an unobserved object is required, JINSI looks it up in the pool. If it is requested for the first time and thus not contained in the pool, a proper mock object is created and afterwards put into the pool. Subsequent lookups will find the associated mock object. For observed objects, there has to be an incoming constructor call first, otherwise JINSI would have missed such a call (at least during sequential replay, see below). As already mentioned above, JINSI adds the observed object to the pool after calling the constructor. For subsequent incoming interactions, JINSI will find the previously pooled observed object.

8.7.2.1 Missing Objects During Minimization

During minimization, it may happen that the delta debugging algorithm will test a subset that contains an incoming method call or field write, but that does not contain the corresponding incoming constructor call. In this case, JINSI will detect that the interaction that would create the object is missing (simply by the fact that the requested target object is not contained in the pool) and will throw a proper exception. The algorithm will consequently classify this test run as unresolved. The alternative would be to create and pool a mock object. However, then the incoming constructor call would be classified as irrelevant. Obviously, as soon as an incoming interaction is relevant for a failure to occur, the corresponding construction of this very object is relevant as well. In principle, the same applies to outgoing interactions: whenever JINSI encounters an outgoing interaction that targets an unobserved object that was not recorded at all during capture, JINSI throws an exception indicating that this interaction cannot be replayed by JINSI. For example, this may happen if a tested subset of interactions takes a different branch in the program. Again, the alternative would be to create a mock object. However, in this case JINSI could not return values that behave in a proper way—JINSI would have to serve as an oracle to replay interactions that are unknown to JINSI.

Next to handling unknown branches (or, more generally code that was not executed before), JINSI has to handle missing initialization of objects that are target of an outgoing interaction. An outgoing method call or field access could target an unobserved object that was not obtained before because the corresponding interaction is not contained in the tested subset. In this case, the required receiver incorrectly would be `null`.

For example, observed object o could obtain an unobserved object u within incoming method call mc_1 on method m_1 by reading an unobserved field. Within the same call, u could be stored in an field of o, like $o.f$. During a subsequent incoming call mc_2 on a different method m_2 of o, an outgoing method call on u stored in $o.f$ may occur. During sequential replay, both incoming method calls would be replayed and therefore it would be guaranteed that the unobserved object u would be obtained (via mc_1) before the outgoing method call would be executed (via mc_2). On the other hand, during minimization it could happen that a tested subset contains the second method call mc_2 but not the first one mc_1. Then, the unobserved object u would not be obtained and field $o.f$ would point to a `null` reference instead. In this case, JINSI would detect that the actual target reference is `null`, but the captured return event would not be a throw event. If this reference would have been `null` during capture, the corresponding returning event would have to represent a thrown `NullPointerException`. JINSI would react to this mismatch with an unresolved test result. Thus, the first incoming method call mc_1 would have to be in the minimal subset.

The above scenario can be extended in a way that a wrong unobserved object would be replayed. If the first method m_1 would be called twice (calls $mc_{1,1}$ and $mc_{1,2}$) and at the second time a different unobserved object u' would be obtained and stored in field $o.f$, the last method call mc_2 on m_2 would execute the outgoing method call on the unobserved object obtained at the second call $mc_{1,2}$; thus, on u' instead of u. However, the minimization could classify the first call $mc_{1,1}$ and the last one mc_2 as relevant, whereas the second one $mc_{1,2}$ would be irrelevant instead. If JINSI would not differentiate between the two different receivers, the first call $mc_{1,1}$ to m_1 that obtains a different object than accessed in the last one mc_2 could be classified as relevant, instead of the actually relevant second one $mc_{1,2}$.

To solve this issue, JINSI compares the current receiver's object ID with the one at capture time. For this purpose, JINSI has to lookup the object's ID—although the beforehand mentioned pool maps IDs to objects. Therefore, during replay, JINSI uses a *second* mapping that maps the objects to their IDs (`Object` $\longmapsto$ `number`).

8.7.2.2 Hard References Instead of Week Ones

As mentioned above, JINSI cannot use weak references for pooled objects during replay. This is mainly because of the observed objects. Because we make alive only the observed part of the program during replay, unobserved objects that referenced to an observed object in the original run do not exist anymore. Thus, there are no other references to observed objects than those in the object pool and in other observed objects. If there are no observed objects that reference to a particular observed object, the reference in the pool will be the only one. Furthermore, if this would be a weak reference as in the case during capture, it would be very likely that a previously constructed observed object would be garbage collected before it would be targeted by successive incoming interactions. In this case, JINSI would not have available the object to execute the interaction on. Constructing the object de novo is not an option because this would mix up the order of the interactions, especially if other observed objects would be involved.

On the other hand, this constitutes a potential disadvantage. Since JINSI currently holds hard references to all objects in the pool, JAVA's garbage collector has no chance to reclaim memory used by pooled objects—even if an object is not needed anymore in the remaining interactions to be replayed. An option would be to extend JINSI by a custom garbage collector. However, in contrast to capture when the whole program and all related instances are alive, during replay only the observed objects and mock objects representing the involved unobserved objects are constructed. Usually, compared to capture, the amount of constructed objects is small during replay. The maximum number of objects is equal to the sum of observed objects and mocked unobserved ones; the mocked objects are just hulls without any state and therefore do not consume much memory. During minimization, the object pool has to be cleared anyway after each test run[21] and consequently the number of objects will decrease as soon as the tested sets will get smaller. At the present time, the extension by a custom garbage collector seems to be disproportionate to the effort necessary to implement it.

During replay, JINSI uses a pool to manage observed objects and mocked unobserved ones. The pool is backed up by a map that maps IDs to objects. This internal map uses hard references, and hence prevents JAVA's garbage collector from removing objects that are seemingly not needed anymore. The pool is also used to detect subsets of interactions that miss essential interactions like incoming constructor calls. To extend this check to outgoing interactions, JINSI uses a second pool that maps the observed objects to their IDs.

8.8 Mock Objects

To replay interactions on unobserved objects, JINSI has to create objects that *act* like the unobserved instances during capture. Constructing an actual instance in general is not possible since instantiation may trigger further interactions or may depend on conditions that are not available during replay. An example can be found in Listing 8.2 on page 156. Observed class `OrderStatistics`

[21]To ensure that objects that are not involved in the tested subset of interactions are not present. Otherwise, JINSI could not detect that an incoming constructor call is missing in this set, for example.

queries an unobserved `Statement` instance. The latter class is not a concrete class but an interface used for executing SQL statements and returning the results. The concrete instance that occurs during capture depends on the used database server. During replay, this very server may not be available, or you may not want to run queries against it during minimization. In general, a failure that is to be debugged by JINSI may depend on non-deterministic results (for example, sensor input), or be caused by states that are difficult to reproduce (for example, network errors). Thus, usually it is either not wanted, impractical, or even just not possible to create instances of unobserved classes as they occurred during capture.

The very same issue occurs in unit testing. For example, if you want to run automatic tests on class `OrderStatistics` autonomously (i.e., without a database), you would again have to create a proper `Statement` object that can be queried. In unit tests, so-called *mock objects* are used for this purpose: a mock object, or in short just mock, is a simulated object that *mimics* the behavior of a real object in a controlled way. Usually, the programmer who writes the unit test creates a mock object programmatically; she explicitly defines the expected behavior in the program code. For example, to test `OrderStatistics` you could create a mocked `Statement` that returns a mocked `ResultSet` that again returns once `true` and once `false` when method `next()` is called. The call to method `getFloat()` could be mocked to return float value 1.3, for instance. Having these mock objects, it is possible to test class `OrderStatistics` without having an actual database.

For JAVA, several mock frameworks are available [59]. While these frameworks provide features to create mock objects, for our purpose it is impractical to make use of these frameworks. This is mainly due to the fact that the mocked behavior has to be hard coded. While this may be sufficient for sequential replay (JINSI could create the proper code), this would be too inflexible during minimization when arbitrary subsets are replayed. Therefore, JINSI uses the *concept* of mock objects, but provides its *own implementation*.

JINSI uses the concept of mock objects to mimic captured object behavior during replay. While there are frameworks available, these are too inflexible and therefore JINSI provides its own implementation.

8.8.1 Creating Mocked Unobserved Classes

The first requirement on mock objects is the feasibility to *create an instance*

1. without triggering any other interaction;
2. and without the necessity to provide further objects that may be required as arguments at construction time (if any).

If we want to replay an incoming call to method `sumAllOrders()` in Listing 8.2 on page 156, for instance, first we would have to create a mocked version of the unobserved `Statement` object passed as argument to this method. Thus, the provided mock has to provide the same interface as the *concrete* `Statement` instance occurred during capture, and must not interact with or manipulate other parts of the program state. Of course, before we can create a mock object, we have to create the respective *concrete* class.

If you run, for example, `OrderStatistics` against a MYSQL database, the concrete type of the passed `java.sql.Statement` instance will be `com.mysql.jdbc.StatementImpl`[22] (in the following abbreviated to `StatementImpl`). This implementation

1. extends class `Object`,
2. implements interface `com.mysql.jdbc.Statement`,
3. interface `java.sql.Statement` and

[22]This implementation is provided by MYSQL Connector/J, the official JDBC driver for MYSQL.

4. interface `java.sql.Wrapper`; further

5. it declares `StatementImpl(ConnectionImpl, String)` as single constructor;

6. finally, it provides its own interface consisting of its own type and those methods that are neither declared in `Object`, nor specified in the three interfaces listed above.

Constructor `StatementImpl(ConnectionImpl, String)` expects an open connection to a MYSQL server and the database name in use. The statement being constructed will instantly use the passed connection to query several properties from the database, like the encoding used for communication. If we want to create an instance of this class, we would have to provide proper arguments. However, as mentioned above, when we replay, we neither want to depend on an open connection, nor on a database server.

Furthermore, a mock object has to behave in the same way when used with the `instanceof` operator, and it has to be possible to cast the mock to the original concrete type. In the above example, it has to be possible to cast the passed `Statement` instance into an instance of `StatementImpl`. If the mocked statement does not provide the *identical* interface as provided by concrete class `StatementImpl`, it would not be possible to use it in the same way. Therefore, a mocked instance of `StatementImpl` has to

1. extend `Object` (see item 1 above),

2. extend `StatementImpl` itself (see 6),

3. implement all three interfaces (see 2 to 4), and

4. be instantiatable *without* calling the constructor (see 5).

In this context, a common technique are so-called *dynamic proxy classes*: A dynamic proxy class (simply referred to as a proxy class in the following) is a class that implements a list of interfaces specified at runtime when the class is created. Each proxy instance has an associated *invocation handler* object. A method invocation on a proxy instance through one of its proxy interfaces will be dispatched to the instance's invocation handler. The invocation handler processes the dispatched method invocation as appropriate, and the result that it returns will be returned as the result of the method invocation on the proxy instance.

JAVA itself provides this functionality as part of its Reflection API. Class `java.lang.reflect.Proxy` can create the `Class` object for a proxy class specified by a given array of interfaces. The dynamically created proxy class has one constructor that takes one argument, an implementation of the interface `java.lang.reflect.InvocationHandler`. This argument is used to set the invocation handler for the proxy instance to create. Thus, it is possible to create an instance of a proxy class for the specified interfaces that dispatches method invocations to the specified invocation handler.

In respect of the example above, we can use JAVA's `Proxy` class to create a statement instance that

1. extends `Object` (see item 1 above),
2. implements all three interfaces (see 2 to 4), and
3. that can be instantiated *without* calling the constructor declared in class `StatementImpl` (see 5).

However, the last property is achieved only by the automatic omission of class `StatementImpl` in the proxy class' inheritance hierarchy. The proxy class only implements `StatementImpl`'s interfaces that are implemented *explicitly,* but does not extend `StatementImpl` itself. Because this implementation provides further methods that are not defined by any of the three explicit interfaces, the proxy class would have to extend it to be fully interchangeable with the original class. Furthermore, the proxy would behave differently in `instanceof` and in casts, because it is not an instance of `StatementImpl`. Unfortunately, JAVA's proxy implementation does not provide the creation of proxy classes of *concrete* classes, but only of explicitly defined interfaces. Thus, JAVA's proxy functionality does not meet JINSI's requirements, as it lacks support for proxies of concrete classes.

Fortunately, JAVASSIST provides dynamic proxy classes that do support proxies of concrete classes. Class `ProxyFactory`[23] provides the same functionality as `java.lang.reflect.Proxy`, and additionally allows to specify the *superclass* of a proxy class. Thus, in this way we can create a proxy class of `StatementImpl` that

1. extends `Object` (see item 1 above),
2. extends `StatementImpl` itself (see 6), and
3. implements all three interfaces (see 2 to 4).

In comparison to JAVA's `Proxy` implementation, JAVASSIST's implementation has the advantage that it can create proxy classes of concrete classes by extending it. However, at the same time it has the *disadvantage* that it extends the concrete class: we cannot easily instantiate an object of this proxy class. To do so, we would have to call the constructor provided by `StatementImpl` because it is automatically added to the proxy class. As we know from Section 7.1.1, a subclass has to call one of its superclass constructors, either implicitly or explicitly. What may, at first glance, appear to be a contradiction is a fundamental issue: we cannot create a mock object that is an instance of the concrete class without extending the concrete class. Without this inheritance, the mock would behave differently in `instanceof` and in casts. Thus, the

[23]To be precise: `javassist.util.proxy.ProxyFactory`.

mock and the proxy class, respectively, have to extend the concrete original class. Hence, JAVA's implementation drops out, and we have to find a way to instantiate a mock instance without calling a constructor.

Extending the class to be mocked has another minor downside: JAVASSIST cannot create proxy classes of *final* classes. A final class cannot be subclasses, or extended, in any way. To bypass this disadvantage, JINSI automatically removes the final modifier of all classes.

JINSI uses JAVASSIST to create dynamic proxy classes. This proxy extends a given concrete class. Therefore, it provides the same interface as the original class and behaves in the same way in `instanceof` *operations and can be properly casted. Calls to a proxy object are dispatched to its invocation handler. Because the proxy class extends the original class, it has to call a super constructor. Thus, you would have to provide proper arguments and unwanted interactions may occur during instantiation. To create proxy classes of final ones, JINSI just removes their final modifier.*

8.8.2 Instantiation of Mock Objects

A possibility to create an instance of a class without triggering further unwanted interactions would be to enclose all constructors—including static initialization blocks—with an on/off switch. This switch would consist of an `if-then` statement where the `if` clause would test if the constructor would be disabled and the `then` clause would contain the actual constructor statements. However, this would have similar disadvantages as described in Section 7.1, first of all the issues when instrumenting classes vital to the operation of the JAVA interpreter.

Fortunately, class `sun.reflect.ReflectionFactory`[24] provides—although undocumented and therefore not officially supported—functionality

[24]This is the master factory for all reflective objects, including those in package `java.lang.reflect`.

to instantiate a class without the necessity to call one of its constructor. Its method `newConstructorForSerialization()` returns a special so-called *munged* constructor.[25] Using this special constructor, it is possible to instantiate a new object by bypassing the actual constructors. Thus, code in the constructors will never be executed and parameters do not have to be known. Furthermore, it is guaranteed that there will be no side effects and that no exception will be thrown.

The disadvantage here, however, is that this mechanism relies on internal ORACLE code that may not be present on all JVMs. While JINSI only uses this ORACLE-dependant mechanism, there is a JAVA library named “Objenesis” that provides the same functionality for other JVMs as well [65]. Using this library, JINSI could accordingly bypass the actual constructors in other JVMs; thus, JINSI could be independent from a specific JVM vendor as ORACLE in this case. However, for our experiments this dependency on ORACLE’s JVM is currently acceptable.

While it is possible to instantiate JAVASSIST’s proxy classes by bypassing their actual constructors, this is, of course, possible for the original class as well. JINSI nevertheless uses these proxy classes to ensure that outgoing interactions are intercepted and do not target the actual object by mistake, as we will see in Section 8.8.3.

JINSI uses munged constructors to instantiate proxies by bypassing their declared constructors. Thus, code in the constructors will never be executed and arguments do not have to be known. There will be no side effects; no interaction will occur while instantiating a proxy object via a munged constructor.

[25] Normally, this special constructor is used by ORACLE’s serialization classes—as the method’s name indicates.

8.8.3 Mocking Unobserved Object Behavior

In addition to the feasibility to create mock objects of unobserved classes (i.e., their sheer existence), JINSI has to mock the original object's *behavior*—JINSI has to replay outgoing interactions by intercepting them and returning proper values if necessary. Thus, JINSI has to handle outgoing method calls, field accesses, and constructor calls that target mock objects. Listing 8.21 on page 201 shows an example of observed class `Observed` that interacts with three other unobserved classes `Unobserved`, `Rational` and `Result`. Its method `divide(...)` exemplifies all three types of outgoing interactions.

```java
public class Observed {

  public Result divide(Unobserved u) {
    Rational rat = u.rational;
    float f = rat.getNumerator() / rat.getDenominator()
        ;
    Result r = new Result(f);
    return r;
  }

}
```

Listing 8.21: Observed class `Observed` that uses field accesses, method and constructor calls to interact with unobserved classes `Unobserved`, `Rational` and `Result`.

After calling observed method `divide(...)`, the first interaction is an outgoing field access to the passed unobserved object `u`. It reads the value of field `u.rational` and assigns it to local variable `rat` of type `Rational`. Next, there are two outgoing method calls to get the numerator and the denominator from rationale `rat`. After dividing the obtained values, there is an outgoing constructor call to create an instance of `Result` named `r` to store the computed value. Finally, result `r` is returned to the caller.

During replay, JINSI has to intercept these very interactions and to provide proper return values. As briefly mentioned above, JINSI does not use the proxy and mock objects, respectively, to handle the outgoing interactions, but manipulates the bytecode to *redirect* the interactions to itself as needed. Listing 8.22 on page 202 shows the instrumented version of class `Observed` as processed by JINSI for replay.

```
public Result divide(Unobserved u) {

  Rational rat;
  if (JINSI.isOutgoingFieldAccess(this, u, "rational"
      )) {
    rat = JINSI.getFieldValue(this, u, "rational",
        ...);
  } else {
    rat = u.rational;
  }

  int numerator;
  if (JINSI.isOutgoingMethodCall(this, u, "
      getNumerator")) {
    numerator = ((Integer) JINSI.getReturnValue(this,
         u, "getNumerator", ...)).intValue();
  } else {
    numerator = rat.getNumerator();
  }

  int denominator;
  if (JINSI.isOutgoingMethodCall(this, u, "
      getDenominator")) {
    denominator = ((Integer) JINSI.getReturnValue(
        this, u, "getDenominator", ...)).intValue();
    } else {
      denominator = rat.getDenominator();
    }

    float f = numerator / denominator;

    Result r = (Result)
      JINSI.getNewObject(this, u, "Result", new Class {
            float.class }, new Object[] { new Float(f) },
            ...);

    return r;
  }
```

Listing 8.22: JINSI has instrumented class `Observed` for replay. All outgoing interactions will either be redirected to JINSI (outgoing interaction), or furthermore target the original object.

Given that we have an observed object `o` of type `Observed`, and that furthermore all other objects are unobserved, replaying an incoming call to observed method `divide(...)` proceeds as follows. As previously described in Sections 8.8.1 and 8.8.2, JINSI at first creates a mock object of class `Unobserved`. Afterwards, JINSI calls the method on `o` and passes the mock as parameter `u`.

8.8.3.1 Field Accesses

Next, `o` reads field `rational` from the given argument `u`. This outgoing interaction is replayed as described in Section 8.4.2 and shown exemplarily in Listing 8.7 on page 163. If it is an outgoing access, it will be redirected to JINSI. If it would be an access from the object to itself (`this`), it would target the actual object unchangedly. Note that it would be difficult to replay field accesses by using the mock itself. Redirecting the access to one of JINSI's methods allows us to run arbitrary code, while a field access cannot contain other code than the access itself. While it would be possible to set a field to a particular value when constructing the mock, and this value could be read once from the mock directly, it would be hard to set the value for successive accesses. Using a method instead is much more flexible. In this way, JINSI can query its log file and can do further runtime checks, like to check if replay is still synchronous. Because objects of type `Rational` are unobserved, JINSI returns a mocked version of `Rational`. The returned mock is stored in local variable `rat`. In summary, the outgoing field read is replaced by a method call to JINSI that returns a mocked `Rational` object.

8.8.3.2 Method Calls

The next two interactions, the outgoing method calls to `getNumerator()` and `getDenominator()`, are both handled in the same way. In the following, we will explain it in more detail based on method `getNumerator()`. One possibility to replay the outgoing call would be to use the invocation handler that is associated with mock `rat`. However, as you can see in Listing 8.22 on page 202, JINSI manipulates the bytecode analogously to the field access. In fact, JINSI replays outgoing method calls analogously to field accesses. If it is an outgoing call, the call will be redirected to JINSI (and will not target the mock and its invocation handler, respectively); otherwise, the call will target the original object (a call on `this`). You may wonder why JINSI does not use the invocation handler; all calls to the mock would be dispatched to this handler anyway. There are several reasons. The main one is that we can pass more information in this way, mostly caller `this`. As mentioned in Section 7.2.1, to know the caller we either have to register it at the call side (see Listing 7.12 on page 146), or we have to pass it like done via method `getReturnValue()` in Listing 8.22 on page 202. By using the latter option, we can guarantee that we pass the actual caller and that we do not access one that is possibly out of date. Furthermore, due to historic reasons. An early version of JINSI used

EASYMOCK, a mock framework, instead of its own implementation. Because we would had had to provide the mock's behavior programmatically, JINSI used it only to instantiate proper mock objects, but not to mock the observed behavior. Thus, it was required to redirect the call to JINSI instead to let it pass to the mock.[26] While it would be possible to replace `getReturnValue()` by registering the caller similar to capturing incoming method calls (see Section 7.2.1) and by using the invocation handler, there is no real benefit and this transition would be fraught with risk.

Still, you may wonder why JINSI uses mocks at all. As already mentioned briefly in Section 8.8.2, we could instantiate the original classes in the same way as the mocks. However, using mocks has several advantages: JINSI can use the invocation handler to check that all outgoing calls are intercepted properly. Further, JINSI can internally put the mocks into collections that use methods `hashCode()` and `equals()`; furthermore, method `toString()` returns a readable string representation of the mock:

- If method `toString()` is called, the invocation handler returns a custom string that describes the mock object (its unique object id, for example). This comes in handy while debugging JINSI itself.

- Methods `hashCode()` and `equals()` are implemented properly by JINSI's invocation handler so that the mocks can be put into collections.

- If any other method is called, the invocation handler will throw an exception to signal that the instrumentation missed an outgoing call. This double check helped several times while implementing the instrumentation for outgoing calls and ensures that bugs related to them are detected early.

[26]This is another (historic) reason why JINSI does not replay field accesses via mocks. EASYMOCK cannot mock field accesses but only method calls.

In summary, the outgoing method call is replaced by a method call to JINSI. In the example, JINSI returns a proper primitive value of type `int`. If the method's return type would be a complex type, JINSI would return a mock as for the outgoing field read.

8.8.3.3 Constructor Calls

The final interaction in Listing 8.22 on page 202 is an outgoing constructor call to unobserved class `Result`. In comparison to field accesses and method calls, no runtime check is needed this time. Because we execute code in observed objects only (the unobserved ones are mocked), all constructor calls are outgoing, regardless of whether the object to be initialized is observed or unobserved. If the outgoing constructor call goes to an unobserved class as in our example, JINSI will create a new mock object and will add it to the object pool. If the call would target an observed class, JINSI would call the actual constructor and add the observed object to the pool (see Section 8.7.2).

8.8.3.4 Comparing References

Finally, mocks are used when comparing references. Because JINSI assigns all objects to unique object ids, JINSI can lookup objects during replay in its object pool (see Section 8.7.2). If an object occurs more than once in a program run (as it is usually the case), JINSI will use the same mock object for these multiple occurrences during replay. This ensures that comparing references (==) evaluates to the same result as during capture. An example can be found in Listing 8.23 on page 207 in the following section.

JINSI redirects all outgoing interactions to itself; it does not use the mock's invocation handler. Instead, the handler is used for internal purposes. The mocks themselves are used as arguments for method and constructor calls, in `instanceof` *statements, and when comparing references.*

8.8.4 Class Loading

When replaying outgoing interactions using mock objects, there is one final issue that has to be taken into account: *class loading*. In contrast to languages such as C and C++, JAVA does not use a linker to combine compiled artifacts into a single executable program. Instead, JAVA uses a class loader that dynamically loads JAVA classes into the JAVA Virtual Machine at runtime. Given the name of a class, a class loader usually locates or generates data that constitutes a definition for the class. Next to obtaining a class object from a class loader, class `Object` defines method `getClass()` that returns the runtime class of an object (the object's corresponding class definition).

Listing 8.23 on page 207 shows observed class `CheckClassObjects`. Its single method `check()` obtains two class objects and checks if they are the same by comparing their references. Firstly, in line 4, it calls method `getClass()` on itself (`this`). This call returns the class object that represents the class definition that was used by the JAVA virtual machine to construct this very object. Secondly, in line 8, it calls method `loadClass()` on the current thread's class loader to load its own class definition. For this purpose, method `check()` obtains the currently executing thread at first (line 6). Next, it obtains the context class loader from this thread (line 7). This loader is used by code running in the thread when loading classes. Finally, it loads its own class definition by passing its class name to the `loadClass()` method.

```
1 public class CheckClassObjects {
2
3   public void check() throws ClassNotFoundException {
4     Class runtimeClass = this.getClass();
5
6     Thread currentThread = Thread.currentThread();
7     ClassLoader classLoader = currentThread.
          getContextClassLoader();
8     Class loadedClass = classLoader.loadClass(
          runtimeClass.getName());

      if (runtimeClass == loadedClass) {
        System.out.println("Class␣objects␣are␣identical."
            );
      } else {
        System.err.println("Class␣objects␣are␣different."
            );
      }
    }

}
```

Listing 8.23: Observed class `CheckClassObjects` uses methods `getClass()` and `loadClass()` to obtain the same class object in different ways. Both calls return the object's own class definition.

If we assume that method `check()` is called within the same thread that initially loaded class `CheckClassObjects` (as it is the case usually), we will expect that both class objects are the same. Thus, the test in the `if-then` statement usually evaluates to `true`. However, when replaying the call to `check()`, the test evaluates to `false`: While the call to `getClass()` is a call to `this` and therefore is not intercepted by JINSI, the other call to `loadClass()` is an outgoing call and therefore is redirected to and answered by JINSI. In the first case, the original class object is returned, in the second case JINSI returns a mocked class object. Thus, these are different objects and consequently the test evaluates to `false` during replay.

For this reason, JINSI must intercept calls to `getClass()`, even though it is not an outgoing call in all cases. JINSI returns a mocked class object and assigns the mock to its corresponding object ID. When intercepting the call to method `loadClass()` afterwards, JINSI lookups the class object on the basis if its ID and eventually returns the same mock. Thus, the test evaluates to `true`, as in the original run or during capture. Because class loaders are vital to the virtual machine, this approach is preferable to instrumenting class loaders as observed classes.

JINSI treats calls to `getClass()` *as outgoing in any event. In this way, class objects are always mocked and their object references are always the same, regardless of whether* `getClass()` *or* `loadClass()` *was used to obtain the class object. If different class objects were returned, JINSI will return different mocks. This ensures that comparing the references during replay evaluates to the same result as during capture.*

In the following experimental evaluation, we will see how JINSI performs in practice when applied to a range of different JAVA programs—from small standard subjects to industrial-size subjects containing up to nearly 100,000 lines of code.

9 Experimental Evaluation

In this chapter, we investigate how well our approach works in practice. Our measure for success is the search space for the defect, as expressed by lines of source code to examine. The benchmark we compare against is *dynamic slicing,* since it is the one other technique which *requires only one single failing run* like JINSI.[1] The dynamic slice contains all lines that possibly could have contributed to the failure; it is this set of possibly failure-relevant source code lines that we want to minimize and compare against, respectively.

9.1 Subjects

To evaluate the effectiveness of JINSI, we have applied it to six different JAVA subjects summarized in Table 9.1 on page 212. The subjects are divided into three groups according to their context:

1. **Standard subjects:** This set consists of two subjects other researchers had used before, and may be considered as standard toolkits in evaluating debugging techniques. We have chosen these subjects to show how JINSI works on *accepted* subjects.

   (1) **BST** provides an implementation of binary search trees and was used by Artzi et al. in evaluating RECRASH [2], as well as by Csallner

[1]If additional runs are available, *statistical debugging* can be used on the statements isolated by JINSI, thus providing an additional *ranking* of the few remaining statements and increasing precision further. This straight-forward extension is part of our future work.

Table 9.1: **Subjects used in the case studies.** The subjects are divided into three groups: (1) subjects commonly used by other research groups, (2) subjects used in earlier work, and (3) further industrial-size subjects with large code base and large number of users.

Subject	Description
BST	Subject used by Artzi et al. [2] and Csallner et al. [19].
NANOXML	XML parser; part of SIR [24]; used by [20, 22, 27]
COLUMBA	Feature-rich email client with GUI as used before in [11].
VENDING MACHINE	Used in proof of concept that demonstrated JINSI's feasibility in [51].
JAXEN	Universal JAVA XPATH engine. Used by ORACLE in several products [49].
JODA TIME	Replacement for the JAVA date and time classes. More than 165,000 downloads [17].

and Smaragdakis in evaluating Check 'n' Crash [19]. This subject was also chosen because it had been specifically used to evaluate RECRASH, which generates unit tests that reproduce a given program failure, as JINSI does.

(2) **NANOXML** is an XML parser for JAVA and a common subject as part of the Software-artifact Infrastructure Repository [24] (SIR). It is used by many research groups for evaluation purposes [20, 22, 27]. This subject shows how JINSI performs on a subject that is frequently used to evaluate debugging aids.

2. **Earlier work:** We use two examples from work on prototype versions of JINSI that had been published earlier.

(3) **VENDING MACHINE**, an artificial proof-of-concept program. It was straightforward enough to explain our approach, yet not entirely trivial. Moreover, it is a pre-existent example that was not created by any of the authors and had been used in a number of papers [51].

(4) **COLUMBA** [11] is a complex email client consisting of several components using a graphical user interface—in contrast to the other subjects. It also shows how JINSI scales on large programs as it consists of almost 100,000 lines of code.

3. **Industrial-size subjects:** We applied JINSI to two further industrial-size subjects that show how JINSI performs on realistic, real-life programs with actual bugs. These subjects do not contain seeded bugs, but bugs like they occur *in reality*.

   (5) Both **JAXEN**, an XPath library written in JAVA, and

   (6) **JODA TIME**, a replacement for the date and time library provided by the JAVA runtime environment, are large JAVA libraries actively maintained and used by a large number of users.[2]

All subjects are object-oriented programs implemented in JAVA. While the JINSI approach is applicable to all object-oriented languages, its basic premise of minimizing object interaction consequently requires an object-oriented execution model. These circumstances constrain the available subjects. For instance, JINSI is not applicable to subjects like the non-object-oriented Siemens test suite [36].

Based on the above subjects, we applied JINSI to a total of 17 individual issues (Table 9.2 on page 214 gives a detailed description), consisting of 14 crashing and three non-crashing bugs, which we selected as follows: BST contains three individual crashing bugs used by both Artzi et al. and Csallner and

[2] JODA TIME had been downloaded 165,095 times as of 2011-04-04 [17]; JAXEN is used by ORACLE in several products [49].

Table 9.2: **Issues used in the case studies.** We applied JINSI to a total of 17 individual issues, consisting of 14 crashing (C) and three non-crashing bugs (N).

Issue	Project-Specific Issue ID	Type	Version	Description
`BST-1`	n/a	C	n/a	`BSTNode.setData(...)` crashes on given `char` array.
`BST-2`	n/a	C	n/a	`BSTNode.setData(...)` crashes on given `Object` instance.
`BST-3`	n/a	C	n/a	`StringCoding.encode(...)` fails on given charset name.
`NAN-1`	SP_HD_1	C	1	Parsing XML document fails.
`NAN-2`	XE_HD_2	C	2	Parsing XML document fails.
`NAN-3`	XEL_HD_2	N	3	Obtaining named children from XML tree fails.
`NAN-4`	XEL_HD_6	C	3	Removing child from XML tree fails.
`NAN-5`	CR_HD_2	N	5	XML entities are not handled correctly.
`NAN-6`	XER_HD_1	N	5	Output contains unexpected artifact.
`COL-1`	n/a	C	1.4	Importing an address book fails with nondescript error dialog.
`VME-1`	n/a	C	n/a	Vending machine erroneously stays in enabled state.
`JAX-1`	29	C	r375	XPATH function `normalize-space` on empty argument `''` fails.
`JAX-2`	111	C	r1170	Selecting node on empty document crashes.
`JAX-3`	156	C	r1216	Changing a node-set does not update the position or node size.
`JOD-1`	1788282	C	r1256	Parsing valid French date using French locale fails.
`JOD-2`	2487417	C	r1377	Converting date in Brazilian time zone fails.
`JOD-3`	2889499	C	r1493	Building complex time zone does not work on Western hemisphere.

`BST` = BST, `NAN` = NANOXML, `COL` = COLUMBA, `VME` = VENDING MACHINE, `JAX` = JAXEN, `JOD` = JODA TIME.
C = crashing, N = non-crashing bug.

Smaragdakis. COLUMBA and VENDING MACHINE were used to demonstrate JINSI on one crashing bug in each subject. These bugs therefore predefine five issues for our evaluation. For the remaining nine crashing issues, we chose three bugs from each JAXEN, JODA TIME, and NANOXML that all met the following criteria:

1. The issue had to be caused by a defect in JAVA code (and not in the build system, for instance);
2. the bug's symptom had to be a crash, i.e. a thrown exception;
3. the issue had to be reported by a user (because we wanted real, post-release failures);
4. we had to be able to reproduce the error (which is not the case for many JAXEN and JODA TIME bug reports, which do not include version information).

Given these constraints, we sequentially checked the issue trackers of JAXEN and JODA TIME and applied JINSI to the first available three issues that met the above criteria; for NANOXML, we have selected issues randomly from four different versions as stored in SIR. All this ensures a selection independent from expected results.

Finally, we added three non-crashing bugs we found (issues `NAN-3`, `NAN-5`, and `NAN-6`) to explore JINSI's performance on non-crashing bugs. In Table 9.2 on page 214, crashing-bugs are marked by the letter C, non-crashing ones by the letter N.

9.2 Experiment Setup

To measure the effectiveness of our approach, we applied JINSI to each of the above 17 issues. The basic idea is that the less code is executed, the smaller the search space and the easier it is for the developer to understand and fix the problem. To measure the total size of each corresponding program, we first counted the physical source lines of code [66], shortened SLOC. To count the number of lines executed by the original failing run, we applied COBERTURA [25], an open-source coverage tool. Hammacher's dynamic slicer for JAVA [34] provided the lines in the dynamic slice. In the case of crashing bugs, we chose the location where the exception had been thrown as slicing criterion. For non-crashing bugs, we chose the location where the faulty output had been issued; as in a non-crashing bug, a failure comes to be as some incorrect output (see Section 6.2 for details). For the minimized run, we again applied COBERTURA to get the number of executed lines. Finally, we intersected the dynamic slice with the line coverage of the minimized run to get the suspicious lines the programmer would be interested in most.[3] These numbers can be found in Table 9.3 on page 217.

For two issues, we can not provide a complete set of numbers. Firstly, we could not apply JINSI to issue `BST-3` because of current limitations of our approach (see Section 9.6 for details on limitations of the current implementation). Secondly, for issue `COL-1`, we were not able to obtain a dynamic slice due to technical limitations in the dynamic slicer that cause a crash of the program being sliced. Therefore, we can neither compare the dynamic slice to the minimized run, nor intersect the two of them.

[3]Technically, due to implementation details Hammacher's slicer can not be applied to the minimized program run directly, because both extensively instrument the JAVA program's byte code. Unfortunately, in many cases the two tools would interfere with each other. Thus, we instead intersect the dynamic slice of the original program run and the set of statements executed by the minimized run. Note that this workaround does *not* improve our results artificially. On the contrary, that intersection may be less precise than applying dynamic backward slicing to the minimized run directly. Although that procedure is to our disadvantage, it still yields conclusive results showing our approach's capacity.

Table 9.3: **Results of the experimental evaluation:** reduction in search space.

		Original Run		Minimized Run	
Issue	SLOC	Line Coverage	Dynamic Slice	Line Coverage	Inter-section
`BST-1`	38	5	3	2	1
`BST-2`	38	14	3	11	1
`BST-3`	38	1	1	n/a	1
`NAN-1`	1,891	130	43	42	42
`NAN-2`	2,540	98	15	13	2
`NAN-3`	3,118	498	199	18	8
`NAN-4`	3,101	501	252	31	18
`NAN-5`	3,278	584	304	218	79
`NAN-6`	3,295	594	313	154	75
`COL-1`	94,863	6,318	crash	5	5
`VME-1`	185	121	63	23	10
`JAX-1`	13,226	2,533	1,224	37	19
`JAX-2`	12,983	861	255	7	3
`JAX-3`	12,957	16	4	12	4
`JOD-1`	25,748	1,613	150	946	48
`JOD-2`	25,810	1,455	373	248	145
`JOD-3`	26,534	1,528	512	193	54
Total	229,643	16,870	3,714	1,960	515

SLOC = physical source lines of code [66]. In totals, missing values ("crash" or "n/a") are counted as zero.

9.3 Search Space Reduction

For our motivating JODA TIME example (`JOD-3`), the total number of source code lines is 26,534[4] (100 %), whereof 1,528 lines (5.76 %) are executed during the failing run; the dynamic slice contains 512 lines (1.93 %), already producing a remarkable reduction of the search space. However, the minimized run as computed by JINSI accesses only 193 lines (0.73 %)—moreover, the intersection between the minimized run and the dynamic slice only contains 54 lines (0.20 %). The numbers for the other issues are similar; in all but one case (`JAX-3`), the final intersection is smaller than the dynamic slice alone. In 12 cases, the minimized run executes less lines than contained in the dynamic slice; in these cases, JINSI outperforms dynamic slicing even without being intersected with the slice itself.

To quantify the overall search space reduction, let us take a look at the *totals* shown in the last row in Table 9.3 on page 217. Simple line coverage narrows down the number of relevant lines to $16{,}870/229{,}643 = 7.3\,\%$ of the source code. A dynamic slice reduces these covered lines to $3{,}714/(16{,}870 - 6{,}318) = 35.2\,\%$, or $3{,}714/(229{,}643 - 94{,}863) = 2.8\,\%$ of the source code.[5] Intersecting the dynamic slice with the minimized run produced by JINSI reduces the set of relevant lines to $(515 - 5)/3{,}714 = 13.7\,\%$ of the dynamic slice alone—in total, 3.1 % of the executed lines, or 0.22 % of the source code. JINSI thus reduces the search space to a handful of code lines.

In all but one cases, the minimized run contains the location where the developer had fixed the bug (see Section 9.6 for details). Thus, JINSI in fact helps finding the defect.

*In our evaluation, JINSI reduces the search space to **3.1 %** of the executed lines, or **0.22 %** of the source code.*

[4]The SLOC varies between the individual issues of the very same project as each issue is related to a different version.
[5]We omit `COL-1` from the total as the slicer crashed; hence the subtrahends.

9.4 Size of Resulting Unit Tests

Further results concern the number of levels of abstraction to be examined by the developer[6], the total number of interactions and the number of the relevant ones per level, as well as the runtime behavior of the whole debugging process. These numbers can be found in Table 9.4 on page 220. For the motivating example (`JOD-3`), the total number of abstraction levels is 6, whereof 3 have to be examined by the developer until the bug is found. The number of interactions denotes the number of incoming interactions on the individual levels and therefore the size of the generated minimized test driver. For instance, in `JOD-3` on the last level, 14,628 incoming interactions are captured and reproduce the original failure—but only two are actually relevant to reproduce the failure (see Listing 5.1 on page 106). In contrast to the captured but not minimized test drivers that contain up to 14,628 interactions, none of the 39 levels in total to be examined produces a test driver containing more than twelve interactions (and frequently much less). Paired with its faithful reproduction of failures, this reduction of search space makes JINSI highly effective in reducing the debugging effort.

In our evaluation, test drivers produced by JINSI contained at most ***twelve interactions*** *that faithfully reproduce the failure; for crashing bugs, the maximum number is* ***eight interactions.***

9.5 Performance

The last two columns in Table 9.4 on page 220 show the runtime behavior of JINSI. The second last column shows the total runtime of the debugging process.

[6]For crashing bugs, an abstraction level is to be equatable with a frame in the stack trace of the thrown exception; successive frames having the same current object are subsumed under one level (see Chapter 5 for an example). For non-crashing bugs, an abstraction level is given by the distance in the dependency graph yielded by the dynamic slicer (see Section 6.2.1 for details).

Table 9.4: **Results of the experimental evaluation:** size of resulting unit tests and runtime behavior.

Issue	Levels		Interactions		Runtime	
	Total	To examine	Total	Relevant	Time [min:s]	Tests
BST-1	2	1	2	2	0:10	4
BST-2	2	1	2	2	0:10	4
BST-3	3	n/a	n/a	n/a	n/a	n/a
NAN-1	3	2	5–6	4–5	0:50	38
NAN-2	2	1	5	2	0:26	14
NAN-3	9	3	1–98	1–4	0:57	217
NAN-4	3	1	90	8	0:29	151
NAN-5	11	4	2–99	2–12	2:21	508
NAN-6	11	7	2–566	2–12	37:16	2707
COL-1	29	1	14,008	2	1:08	4
VME-1	2	1	32	7	0:15	68
JAX-1	6	3	6–1,919	2–3	3:40	416
JAX-2	7	2	3–4	2–3	0:53	20
JAX-3	2	2	5–6	3–4	0:26	39
JOD-1	3	3	4–8	3–5	1:15	60
JOD-2	4	4	80–123	2–3	4:29	190
JOD-3	6	3	45–14,628	1–2	4:58	243

This includes capture and minimization, as well as the upstream instrumentation.[7] In the last column, the total number of tests run by delta debugging is shown. For the motivating example, JINSI reduces the 26,534 lines in total to only 54, and requires only 4 minutes and 58 seconds for the whole process. In all cases but one, JINSI needs only a few minutes to compute the minimal test driver. For `NAN-6`, the process takes long because at one minimization stage, the number of unresolved test outcomes is high, triggering the delta debugging worst case complexity of $O(n^2)$ [72].

An interesting instance is issue `COL-1` where the process *without the upstream event slicing* takes 3 hours and 46 minutes. In this case, the delta debugging algorithm would have to run 4,578 tests instead of only 4 that execute up to 14,008 incoming interactions to compute the minimal set. However, as we have seen in Section 4.2.1, JINSI applies event slicing before delta debugging. In this example, the event slice contains only two interactions—the two relevant ones. The downstream delta debugging can not minimize further and runs 4 tests, taking 3 seconds. In total, including computing the event slice, the whole process takes 68 seconds—instead of almost 4 hours. Like Leitner et al. [45], we found that while delta debugging consistently produces the best results, a preprocessing with slicing can dramatically improve performance.

In our evaluation, JINSI needs at most ***37 minutes*** *to compute the minimal test drivers; for crashing bugs, it needs at most* ***five minutes.***

9.6 Limitations and Threats to Validity

While conducting the experiments, we encountered some limitations of our approach:

[7]We assume the availability of the dynamic slice, whose computation takes 30 s to 11 min. Computing the intersection is a relatively cheap operation that takes only a few seconds.

JAVA reflection: NANOXML uses *factory methods* to create instances of various (hard coded) classes. Because JINSI currently does not support automatic instrumentation of reflection, we manually replaced these instances by direct constructor calls. This can easily be addressed in a future revision of JINSI.

`String` as primitive: Due to hard-coded assumptions in the JAVA virtual machine, JINSI can not instrument the class `String` as it would be needed for capture / replay. As a workaround, JINSI thus treats strings like primitive values. In issue `BST-3`, a `String` instance was on the stack and JINSI would have had to include this object as observed. Because of the restrictions mentioned above, it was not possible to apply JINSI to this issue. Again, future revisions of JINSI may implement specific workarounds for `String` classes.

Fixed location not in minimized run: In issue `JAX-3`, the location where the actual fix was applied is not executed by the minimized run as computed by JINSI—although it is executed by the original run. While JINSI provides an alternative way to reproduce the failure in question, the bug was actually fixed by adding functionality not to the code executed by the minimized run, but to a method omitted during minimization. Interestingly enough, the dynamic slice does not contain this location either. Generally speaking, when a fix contains new statements only, the programmer has a large choice of locations that may or may not be included in a minimized execution, and it is hard to exactly predict this location.

Object orientation: JINSI leverages the abstraction levels naturally defined by object-oriented programming. Therefore, JINSI currently can be applied to object-oriented programs only. However, JINSI may also exploit different means of encapsulation. For instance, JINSI could use the abstraction levels given by the logical boundaries in modular programming. However, because abstraction when using modules usually is not as finegrained as when using objects, the result of the minimization process may not be as precise as when applying JINSI to object-oriented programs.

All these limitations pose threats to the *external validity* of our results: There likely are programs or bugs on which JINSI will not perform as well as in our sample, and any generalization from the results of our experimental evaluation is to be taken with a grain of salt. Our sample size is small; in total we investigated seven subjects with 17 different issues—it is time-consuming to find real bugs by manually analyzing bug reports, to download and to compile old versions, and finally to reproduce the original failure. Our selection process creates a bias towards well documented and publicly available issues; also, it is obviously slanted towards crashes of object-oriented systems. However, this evaluation documents every single problem we have applied JINSI upon, with consistent and promising results. Furthermore, by randomly choosing various issues instead of applying JINSI to many issues of one single subject, we increased the heterogeneity of our sample set. We concentrated on defects that cause an exception as error indication. JINSI can also be applied to non-crashing bugs using other types of predicates as shown for the three issues `NAN-3`, `NAN-5`, and `NAN-6`; future experiments should include more types of bugs.

Other possible threats include *construct validity,* which concerns the appropriateness of our measures; here, the size of the program should well correlate with the effort needed for its examination—we assume the developer is able to fix a bug the more efficiently the less code remains and the shorter the execution becomes. Finally, *internal validity* may be threatened by defects in JINSI or our evaluation setting; we have addressed this threat through careful testing[8].

[8]For instance, as soon as some changed code is submitted to JINSI's code repository, a continuous integration tool [42] re-runs experiments automatically, and compares current with previous results.

10 Conclusions and Future Work

Debugging a program can be a tedious, long-lasting, and boring work on the one hand, and a factor that drives costs up significantly on the other hand. It would be best, of course, if we did not have to debug programs at all, but if we were able to prevent software errors in the first place. Unfortunately, despite all efforts and the scientific progress made, modern software still contains bugs that not only cause pure inconveniences, but also have a negative impact on economy. Thus, we still need techniques that help us debug software systems.

Debugging consists of two essential steps: the first is reproducing the failure, the second is finding the defect. Both reproducing and finding can be a tough and risky challenge. The field of automated debugging in general and the field of statistical debugging in particular aim to ease the search for failure causes. However, as of today, most of these techniques can substantially reduce the search space, but require successful executions to compare against. By contrast, the alternative of program slicing requires only the failing execution, and likewise is able to eliminate those parts of the program that could not have contributed to the failure; however, the remaining slice can still contain thousands of locations not all of which are relevant.

This work presented JINSI, taking a new twist on automated debugging that aims to combine ease of use with unprecedented effectiveness. During debugging, JINSI brings lots of benefits at little cost. By reducing the search space considerably and providing minimal unit tests, which describe exactly how the failure comes to be, the approach helps reproduce the failure and find the defect. Applied in the field, JINSI could both relax programmers' nerves and reduce development costs caused by software failures as they frequently occur in software systems to this day. The contributions of this work in a nutshell are as follows:

Failure-inducing interactions: We showed how to *minimize object interactions* and thus executions, using a combination of delta debugging and slicing. Our approach reduces the set of executed statements until only the relevant ones remain.

Requirements: Our approach's requirements are trivial, all it takes is one *single failing run* that is observed by JINSI. Furthermore, the tool is *fully automatic*, does not require any selection, annotation, or other interaction with the programmer.

High diagnostic quality: The resulting *minimized unit test* is easy to understand, and encompasses all steps that are relevant to reproduce the failure.

Effectiveness: In terms of precision, JINSI combines best-of-breed techniques to dramatically improve upon the state of the art. By reducing the search space to a *handful of code lines*—0.22 % of the source code, or 1 line out of 450—JINSI eases debugging to the point where it ceases to be a problem.

Scaleability: JINSI scales to JAVA programs with *100,000 lines of code*.

Generality: We were able to provide a *full diagnosis* on 16 out of 17 bugs examined, ranging from relatively small standard subjects to industrial-size projects.

10.1 Future Work

Besides general improvements to stability, efficiency, and usability, our future work will focus on the following topics:

Universal strategies: Right now, JINSI provides a general strategy along the dynamic slice for non-crashing bugs and a special (but very common) strategy along the stack for crashing bugs. At a higher abstraction level, all bugs are equal; and consequently, we are working into merging the two strategies into one.

Ranking locations: In the presence of passing runs, one could leverage *statistical debugging* in order to identify the most defect-prone locations even in the set minimized by JINSI. Where runs are missing, they could be generated automatically [1].

Automatic fixes: Rather than just simplifying interaction, JINSI could also use delta debugging to isolate the *difference* between a passing and a failing run. Such differences would improve diagnostic quality even further; in fact they could even be turned into candidates for *automatic fixes* [23].

Inferring Failure Conditions: Another technique to identify failure circumstances automatically was recently presented by Rößler et al. While JINSI isolates failure-inducing object interactions, they isolate failure conditions like “The program fails whenever the credit card number begins with 6, 5, and a non-zero digit.” [56]. JINSI can help reduce the search space, as it already computes a minimal run, which precisely describes how the failure comes to be.

10.2 Outlook

As of today, debugging is a laborious business and, on top of that, *poses a risk*: because you never know when you will find the defect, you are not able to schedule this task properly. By contrast, finding a needle in a haystack is a walk in the park. The programmer could be able to complete debugging by the afternoon, or only at the crack of dawn. The product manager does not know when she will be able to launch a new, groundbreaking product; either on schedule to play a leading role in the market, or right after a competitor has presented a working alternative that satisfies customer needs.

JINSI aims to minimize the search effort required to fix a defect. Moreover, the results of our study show that JINSI reduces the search space to the point where debugging ceases to be a problem. Thus, JINSI helps make software development in general and debugging in particular *predictable*. Applied in the field, JINSI ensures that developers are at home in time for dinner with their family, and that product managers are able to plan software releases more predictably and reliably. This is an important contribution not only to the scientific community, but also to software development in general.

Bibliography

[1] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization", in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10, New York, NY, USA: ACM, 2010, pp. 49–60, ISBN: 978-1-60558-823-0. DOI: 10.1145/1831708.1831715.

[2] S. Artzi, S. Kim, and M. D. Ernst, "ReCrash: making software failures reproducible by preserving object states", in *Proceedings of the 22nd European conference on Object-Oriented Programming*, ser. ECOOP '08, Berlin, Heidelberg: Springer-Verlag, 2008, pp. 542–565, ISBN: 978-3-540-70591-8. DOI: 10.1007/978-3-540-70592-5_23.

[3] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis", *IEEE Trans. Softw. Eng.*, vol. 36, pp. 528–545, 4 Jul. 2010, ISSN: 0098-5589. DOI: 10.1109/TSE.2009.87.

[4] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley Professional, Nov. 2004, ISBN: 978-0321278654.

[5] D. Binkley and M. Harman, "A large-scale empirical study of forward and backward static slice size and context sensitivity", in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '03, Washington, DC, USA: IEEE Computer Society, 2003, pp. 44–53, ISBN: 0-7695-1905-9. DOI: 10.1109/ICSM.2003.1235405.

[6] G. Booch, “Object-oriented development”, *IEEE Trans. Software Eng.*, vol. 12, no. 2, pp. 211–221, 1986.

[7] P. Bouillon, M. Burger, and A. Zeller, “Automated debugging in Eclipse: (at the touch of not even a button)”, in *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, ser. eclipse ’03, New York, NY, USA: ACM, 2003, pp. 1–5. DOI: `10.1145/965660.965661`.

[8] E. Bruneton, R. Forax, E. Kuleshov, and A. Loskutov. (Dec. 21, 2010). ASM, [Online]. Available: `http://asm.ow2.org/` (visited on 04/15/2011).

[9] M. Burger. (Jul. 16, 2009). Javassist Issue JASSIST-88, JBoss Community, [Online]. Available: `https://jira.jboss.org/jira/browse/JASSIST-88` (visited on 04/08/2011).

[10] M. Burger, K. Lehmann, and A. Zeller, “Automated debugging in Eclipse”, in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA ’05, New York, NY, USA: ACM, 2005, pp. 184–185, ISBN: 1-59593-193-7. DOI: `10.1145/1094855.1094926`.

[11] M. Burger and A. Zeller, “Replaying and isolating failing multi-object interactions”, in *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, ser. WODA ’08, New York, NY, USA: ACM, 2008, pp. 71–77, ISBN: 978-1-60558-054-8. DOI: `10.1145/1401827.1401842`.

[12] S. Chiba. (Oct. 5, 2010). Javassist, [Online]. Available: `http://www.csg.is.titech.ac.jp/~chiba/javassist/` (visited on 04/15/2011).

[13] S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient Java bytecode translators", in *Proceedings of the 2nd intl. conference on Generative programming and component engineering*, ser. GPCE '03, New York, NY, USA: Springer-Verlag New York, Inc., 2003, pp. 364–376, ISBN: 3-540-20102-5.

[14] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: effective statistical debugging via efficient path profiling", in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 34–44, ISBN: 978-1-4244-3453-4. DOI: `10.1109/ICSE.2009.5070506`.

[15] J. Clause and A. Orso, "A technique for enabling and supporting debugging of field failures", in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07, Washington, DC, USA: IEEE Computer Society, 2007, pp. 261–270, ISBN: 0-7695-2828-7. DOI: `10.1109/ICSE.2007.10`.

[16] H. Cleve and A. Zeller, "Locating causes of program failures", in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05, New York, NY, USA: ACM, 2005, pp. 342–351, ISBN: 1-58113-963-2. DOI: `10.1145/1062455.1062522`.

[17] S. Colebourne. (Sep. 14, 2010). Joda Time, Java date and time API, [Online]. Available: `http://joda-time.sourceforge.net/` (visited on 04/11/2011).

[18] Commons Documentation Team. (Apr. 15, 2008). The Apache Commons-Collections project, [Online]. Available: `http://commons.apache.org/collections/` (visited on 04/17/2011).

[19] C. Csallner and Y. Smaragdakis, "Check 'n' crash: combining static checking and testing", in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05, New York, NY, USA:

ACM, 2005, pp. 422–431, ISBN: 1-58113-963-2. DOI: 10.1145/1062455.1062533.

[20] —, "DSD-crasher: a hybrid analysis tool for bug finding", in *Proceedings of the 2006 international symposium on Software testing and analysis*, ser. ISSTA '06, New York, NY, USA: ACM, 2006, pp. 245–254, ISBN: 1-59593-263-1. DOI: 10.1145/1146238.1146267.

[21] V. Dallmeier. (Apr. 28, 2008). Dynamic Purity Analysis for Java Programs, Software Engineering Group at Saarland University, [Online]. Available: http://www.st.cs.uni-saarland.de/models/jpure/ (visited on 04/11/2011).

[22] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight defect localization for Java", in *ECOOP '05: Proceedings of 19th European Conference on Object-Oriented Programming*, ser. Lecture Notes in Computer Science, vol. 3586, Glasgow, UK: Springer, Jul. 2005, pp. 528–550, ISBN: 3-540-27992-X.

[23] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies", in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 550–554, ISBN: 978-0-7695-3891-4. DOI: 10.1109/ASE.2009.15.

[24] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact", *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005, ISSN: 1382-3256.

[25] M. Doliner. (Mar. 12, 2010). Cobertura, [Online]. Available: http://cobertura.sourceforge.net/ (visited on 04/11/2011).

[26] C. von Eitzen. (Jan. 6, 2010). EC card problem persists, The H Security, [Online]. Available: http://h-online.com/-896549 (visited on 04/07/2011).

[27] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases", in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. SIGSOFT '06/FSE-14, New York, NY, USA: ACM, 2006, pp. 253–264, ISBN: 1-59593-468-5. DOI: 10.1145/1181775.1181806.

[28] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases", *IEEE Transactions on Software Engineering*, vol. 35, no. 1, pp. 29–45, 2009, ISSN: 0098-5589.

[29] M. Fackler. (Dec. 20, 2005). Tokyo Exchange President Resigns Over Trading Loss, New York Times, [Online]. Available: http://www.nytimes.com/2005/12/20/business/worldbusiness/20cnd-glitch.html (visited on 04/07/2011).

[30] "Final report on the August 14, 2003 blackout in the United States and Canada: causes and recommendations", Unites States-Canada Power System Outage Task Force, Tech. Rep., Apr. 2004.

[31] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification*, 3rd ed. Addison Wesley, Jun. 2005, ISBN: 978-0321246783.

[32] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops", in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05, New York, NY, USA: ACM, 2005, pp. 263–272, ISBN: 1-58113-993-4. DOI: 10.1145/1101908.1101948.

[33] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging", in *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, ser.

ESEC/FSE-7, London, UK: Springer-Verlag, 1999, pp. 303–321, ISBN: 3-540-66538-2. DOI: 10.1145/318773.319248.

[34] C. Hammacher. (Nov. 28, 2010). JavaSlicer, an efficient dynamic slicer for Java, Software Engineering Group at Saarland University, [Online]. Available: http://www.st.cs.uni-sb.de/javaslicer/ (visited on 04/11/2011).

[35] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., Oct. 1999, ISBN: 978-0201616224.

[36] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria", in *Proceedings of the 16th international conference on Software engineering*, ser. ICSE '94, Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 191–200, ISBN: 0-8186-5855-X.

[37] Y. Inghelbrecht. (May 8, 2008). Javassist Issue JASSIST-53, JBoss Community, [Online]. Available: https://jira.jboss.org/jira/browse/JASSIST-53 (visited on 04/08/2011).

[38] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement", in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA '08, New York, NY, USA: ACM, 2008, pp. 167–178, ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390652.

[39] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique", in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05, New York, NY, USA: ACM, 2005, pp. 273–282, ISBN: 1-58113-993-4. DOI: 10.1145/1101908.1101949.

[40] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization", in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02, New York, NY, USA: ACM, 2002, pp. 467–477, ISBN: 1-58113-472-X. DOI: 10.1145/581339.581397.

[41] S. Joshi and A. Orso, "SCARPE: a technique and tool for selective record and replay of program executions", in *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, Paris, France: IEEE, Oct. 2007, pp. 234–243.

[42] K. Kawaguchi. (Apr. 12, 2011). Jenkins CI, [Online]. Available: http://jenkins-ci.org/ (visited on 04/12/2011).

[43] E. Kuleshov, "Using the ASM framework to implement common Java bytecode transformation patterns", presented at the 6th Intl. Conference on Aspect-Oriented Software Development (AOSD 2007), Vancouver, Canada, Mar. 2007.

[44] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases", in *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, Washington, DC, USA: IEEE Computer Society, 2005, pp. 267–276, ISBN: 0-7695-2482-6.

[45] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization", in *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, ser. ASE '07, New York, NY, USA: ACM, 2007, pp. 417–420, ISBN: 978-1-59593-882-4. DOI: 10.1145/1321631.1321698.

[46] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation", in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05, New York, NY, USA: ACM, 2005, pp. 15–26, ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065014.

[47] T. Lindholm, *The Java Virtual Machine Specification*. Boston: Addison-Wesley, 1999, ISBN: 9780201432947.

[48] M. McDonald, R. Musson, and R. Smith, *The practical guide to defect prevention*, 1st ed. Redmond, WA, USA: Microsoft Press, Mar. 2007, ISBN: 9780735622531.

[49] B. McWhirter. (Jun. 17, 2010). JAXEN — Universal Java XPath engine, Codehaus, [Online]. Available: `http://jaxen.codehaus.org/` (visited on 04/11/2011).

[50] Oracle. (Apr. 15, 2011). Java Tutorial: Learning the Java Language, [Online]. Available: `http://download.oracle.com/javase/tutorial/java/index.html` (visited on 04/15/2011).

[51] A. Orso, S. Joshi, M. Burger, and A. Zeller, "Isolating relevant component interactions with JINSI", in *Proceedings of the 2006 international workshop on Dynamic systems analysis*, ser. WODA '06, New York, NY, USA: ACM, 2006, pp. 3–10, ISBN: 1-59593-400-6. DOI: `10.1145/1138912.1138915`.

[52] A. Orso and B. Kennedy, "Selective capture and replay of program executions", in *Proceedings of the third international workshop on Dynamic analysis*, ser. WODA '05, New York, NY, USA: ACM, May 2005, pp. 29–35, ISBN: 1-59593-126-0. DOI: `10.1145/1082983.1083251`.

[53] "Patriot Missile Defense – software problem led to system failure at Dhahran, Saudi Arabia", U.S. Government Accountability Office, Tech. Rep. IMTEC-92-26, Feb. 1992.

[54] G. Phipps, "Comparing observed bug and productivity rates for Java and C++", *Software - Practice and Experience (SPE)*, vol. 29, pp. 345–358, 4 Apr. 1999, ISSN: 0038-0644.

[55] M. Renieris and S. P. Reiss, “Fault localization with nearest neighbor queries”, in *Proceedings of the 18th IEEE intl. Conference on Automated Software Engineering*, ser. ASE ’03, Los Alamitos, CA, USA: IEEE Computer Society, 2003, pp. 30–39, ISBN: 0-7695-2035-9. DOI: 10.1109/ASE.2003.1240292.

[56] J. Rößler, A. Orso, and A. Zeller, “When does my program fail?”, presented at the CSTVA 2011: 3rd Workshop on Constraints in Software Testing, Verification, and Analysis, Berlin, Germany, Mar. 2011.

[57] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, “Automatic test factoring for Java”, in *Proceedings of the 20th IEEE/ACM intl. Conference on Automated software engineering*, ser. ASE ’05, New York, NY, USA: ACM, 2005, pp. 114–123, ISBN: 1-58113-993-4. DOI: 10.1145/1101908.1101927.

[58] A. Schröter, N. Bettenburg, and R. Premraj, “Do stack traces help developers fix bugs?”, in *Proceedings of the 7th International Working Conference on Mining Software Repositories (MSR 2010)*, IEEE, 2010, pp. 118–121, ISBN: 978-1-4244-6803-4.

[59] V. Sizov. (Mar. 31, 2009). Java Mock Frameworks Comparison, [Online]. Available: http://bit.ly/GXpqT (visited on 04/17/2011).

[60] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, “jRapture: a capture/replay tool for observation-based testing”, in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA ’00, New York, NY, USA: ACM, 2000, pp. 158–167, ISBN: 1-58113-266-2. DOI: 10.1145/347324.348993.

[61] G. Tassey, “The economic impacts of inadequate infrastructure for software testing”, National Institute of Standards and Technology (NIST), Gaithersburg, MD, USA, Planning Report 02-3, May 2002.

[62] The Eclipse Foundation. (Mar. 15, 2011). The AspectJ Project, [Online]. Available: http://eclipse.org/aspectj/ (visited on 04/15/2011).

[63] "The economic cost of the blackout: an issue paper on the northeastern blackout", ICF Consulting, Tech. Rep., Aug. 2003.

[64] F. Tip, "A survey of program slicing techniques", *Journal of programming languages*, vol. 3, no. 3, pp. 121–189, 1995.

[65] H. Tremblay. (Aug. 26, 2009). Objenesis, [Online]. Available: http://objenesis.googlecode.com/svn/docs/index.html (visited on 04/17/2011).

[66] D. A. Wheeler. (Mar. 28, 2010). SLOCCount — count source lines of code (SLOC), [Online]. Available: http://www.dwheeler.com/sloccount/ (visited on 04/11/2011).

[67] G. Xu, A. Rountev, Y. Tang, and F. Qin, "Efficient checkpointing of Java software using context-sensitive capture and replay", in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07, New York, NY, USA: ACM, 2007, pp. 85–94, ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287638.

[68] A. Zeller, "Isolating cause-effect chains from computer programs", in *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, New York, NY, USA: ACM Press, Nov. 2002, pp. 1–10, ISBN: 1581135149.

[69] —, "Isolating cause-effect chains with AskIgor", in *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, ser. IWPC '03, Washington, DC, USA: IEEE Computer Society, 2003, pp. 296–297, ISBN: 0-7695-1883-4.

[70] —, *Why Programs Fail: A Guide to Systematic Debugging (Second Edition)*. Morgan Kaufmann, Jun. 2009, ISBN: 978-0123745156.

[71] —, "Yesterday, my program worked. Today, it does not. Why?", in *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, ser. Lecture Notes in Computer Science, vol. 1687, London, UK: Springer, 1999, pp. 253–267, ISBN: 3-540-66538-2.

[72] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input", *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002, ISSN: 0098-5589.

[73] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states", in *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, New York, NY, USA: ACM, 2009, pp. 43–52, ISBN: 978-1-60558-001-2.

[74] A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken, "Statistical debugging of sampled programs", in *Advances in Neural Information Processing Systems 16*, Cambridge, MA: MIT Press, 2003, pp. 9–11, ISBN: 0-262-20152-6.

[75] M. Zhivich and R. K. Cunningham, "The real cost of software errors", *IEEE Security & Privacy*, vol. 7, no. 2, pp. 87–90, 2009, ISSN: 1521-9615.

[76] T. Zimmermann and A. Zeller, "Visualizing memory graphs", in *Revised Lectures on Software Visualization, International Seminar*, ser. Lecture Notes in Computer Science, vol. 2269, London, UK: Springer, 2002, pp. 191–204, ISBN: 3-540-43323-6.