Minimizing Reproduction of Software Failures

Martin Burger
Saarland University – Computer Science
Saarbrücken, Germany
mburger@cs.uni-saarland.de

Andreas Zeller
Saarland University – Computer Science
Saarbrücken, Germany
zeller@cs.uni-saarland.de

ABSTRACT

A program fails. What now? Taking a single failing run, we *record* and *minimize* the interaction 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, our JINSI prototype reduced the search space to 13.7 % of the dynamic slice or 0.22 % of the source code, with only 1–12 calls left to examine.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Algorithms

Keywords

Automated debugging, capture/replay

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.

ISSTA '11, July 17-21, 2011, Toronto, ON, Canada

The field of *automated debugging* aims to ease this search. *Statistical debugging* [20, 14] 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 [32], program state [31], or version histories [30] 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* [28] 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 paper 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 calls 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 1–12 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.

The remainder of the paper is organized as follows. After giving a motivating example (Section 2), we present the state of the art in automated debugging (Section 3) as applied to this example. We then make the following contributions:

1. We show how to *minimize object interactions* and thus executions, using a combination of delta debugging and slicing (Section 5). 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.

```
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, ...)
    .addRecurringSavings("PST",       0, 1918, ...)
    .addRecurringSavings("PWT", 3600000, 1942, ...)
    .addRecurringSavings("PPT", 3600000, 1945, ...)
    .addRecurringSavings("PST",       0, 1945, ...)
    .addRecurringSavings("PDT", 3600000, 1948, ...)
    .addRecurringSavings("PST",       0, 1949, ...)
    .addRecurringSavings("PDT", 3600000, 1950, ...)
    .addRecurringSavings("PST",       0, 1950, ...)
    .addRecurringSavings("PST",       0, 1962, ...)
    .addRecurringSavings("PST",       0, 1967, ...)
    .addRecurringSavings("PDT", 3600000, 1967, ...)
    .addRecurringSavings("PDT", 3600000, 1974, ...)
    .addRecurringSavings("PDT", 3600000, 1975, ...)
    .addRecurringSavings("PDT", 3600000, 1976, ...)
    .addRecurringSavings("PDT", 3600000, 1987, ...)
    .toDateTimeZone("America/Los_Angeles");
```

Figure 1: This code taken from the JODA TIME manual [6] crashes JODA TIME when run in the western hemisphere.

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

Figure 2: When fed with the execution in Figure 1, JINSI produces a unit test with just the relevant calls. Of the 23 calls, only three suffice to reproduce the failure.

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 a single failing execution of a JAVA program. Its implementation and the experimental data are publicly available.

2. JINSI IN A NUTSHELL

To show how JINSI[1] works, consider Figure 1, showing a piece of code that interacts with the JODA TIME library, a replacement for the JAVA date and time classes [6]; 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.

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 lines, covering 1,528 out of 26,534 JODA TIME source code lines.

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

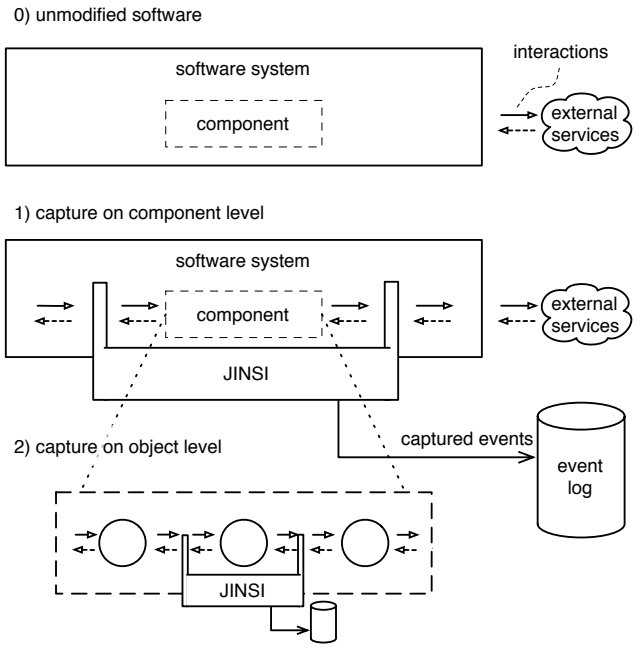

Figure 3: JINSI intercepts and records (1) the interactions between a component and its environment, and (2) all interactions of objects within that component.

Here is how JINSI helps in both reproducing and simplifying the problem. The key idea of JINSI is sketched in Figure 3. JINSI *wraps* around a component, e.g., the component building time zones, and *records* its interactions 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 Figure 1, 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 (Figure 2): 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 stack trace 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 Figure 2, 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 call, the one to method `addRecurringSavings()`, is set *within* this context. Without it, the test simply passes, which makes it very interesting: Obviously, the problem is related to daylight savings rules—and our investigation would start right within this very method.

In the remainder of this paper, we will show how applying JINSI iteratively pinpoints the defect.

3. BACKGROUND

How does JINSI compare to the state of the art? Using the motivating example in Figure 1, let us discuss the state of automated debugging and how JINSI improves it.

3.1 Statistical Debugging

The basic idea of statistical debugging is to identify features of the program execution that *statistically correlate with failures.* Important features include *code coverage* (i.e., code that is executed only in a failing run is more likely to cause the failure), or *function return values* (i.e., erroneous function behavior tied to overall failures). To obtain significant results, statistical approaches must consider thousands of executions—either from an extensive test suite [20] or sampled in the field [14, 7]. 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. [33] reduces the search space to 5 % or less for 50 % of the test cases examined. 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*—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 [19] approach to the bug in Figure 1, 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 in [33]. 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.[2] However, this is still almost *ten times* as much than 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.

3.2 Program Slicing

A *program slice* [28] is a subset of the program execution 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. In debugging, computing the backward slice for a failing statement returns all statements that could have influenced the failure. While a *static* slice applies to all possible runs, a *dynamic* slice just applies to the failing run and thus is more precise.

Just like JINSI, dynamic slicing requires only one failing run; it is thus the benchmark we compare against. As shown in our evaluation (Section 8), 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 this bug.

3.3 Delta Debugging

Delta debugging is a technique to systematically narrow down failure causes by means of automated experiments. It had been previously used, for instance, to isolate failure-inducing changes [30] and to simplify failure-inducing input [32]. 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 [17] are also excluded.

An interesting alternative could be applying delta debugging to program states [5], 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 on JAVA programs; on top, it again requires a (hopefully similar) passing run.

Most of the precision of JINSI in fault localization comes from applying delta debugging on object interaction. This was first attempted for *generated* call sequences (i.e., generated test cases): Lei and Andrews [21] were the first to apply delta debugging to calls to minimize generated test cases; Leitner et al. [22] 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.

3.4 Capture/Replay

JINSI not only minimizes interactions, it also captures them from failing executions. By design, JINSI records *all* interactions for a given set of objects—because any of these interactions may be relevant for the failure. Such complete record/replay was already a feature of earlier prototypes [27, 25].

Test factoring [26] as well as *test carving* [16] also capture at the method level to extract unit tests specific to a task. ADDA [4] records events at the level of C standard library and file operation functions. While test carving captures *program state* to create unit tests, JINSI captures and minimizes *interactions* that reproduce failures.

The true power of capture/replay, however, comes as it is being combined with diagnostic features. The RECRASH tool [2] records executions by recording parts of the program state at each

[2] Applying TARANTULA took several hours; because of the tool we used to collect coverage data, all the tests had to be run individually in a fresh instance of the JVM, and more than 3 GB of coverage data had to be collected and analyzed. In contrast, applying JINSI is a matter of minutes (cf. Section 8.5).

method entry—namely those objects that are reachable via direct references. When the program crashes, it thus allows the developer to observe a run in several states before the actual crash. This is very efficient, but assumes that the stack trace actually contains the code (and state) that caused the failure. This is not the case in our motivating example, as the crucial time zone is obtained via a static method call; the reproduced run thus still relies on the system's current time zone and will not reproduce the run east of Greenwich. By capturing and replaying *all* interactions by design, JINSI can replay and minimize the original failure.

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

3.5 Own Previous Work

The current version of JINSI bases on earlier proof-of-concept prototypes [24, 3], which again had been initially inspired by the SCARPE prototype originally presented by Orso and Kennedy [25]. JINSI 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 that the programmer select the *object in question*, providing a hint on where the fault might be, and that the programmer provide a *predicate* that determines whether a run is successful or not; also, it was never demonstrated on more than a single real-life example (`COL-1` in this paper). In contrast, JINSI now is fully 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 (Section 8), scaling up to problems of considerable complexity.

4. REPRODUCING FAILURES

In the next sections, we shall walk through the individual steps JINSI undertakes to minimize failure reproduction (Figure 4). The very first task in debugging is to *reproduce the problem* in order to examine it and eventually to check whether a fix is successful. To reproduce a problem, JINSI reproduces object interaction: It captures and replays on the object level to reproduce the *environment*, and on the method level to reproduce the *problem's history* (i.e., the program execution). The main advantage of this approach is that it abstracts over all kinds of input (say, data, user interaction, network events, or randomness) while keeping a uniform mechanism to minimize the reproduction.

JINSI uses state-of-the art mechanisms for capturing and replaying executions [24, 3, 25]; we thus just focus on the main concepts.

4.1 Capture

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 class files, and captures their interactions at runtime (Figure 3).

Figure 5 shows how JINSI instruments an outgoing call that 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 parameters (their types and unique IDs, see below); 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.

```
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]);
  }
}
```

Figure 6: JINSI replaces outgoing method calls with calls to itself and returns proper values depending on the captured event.

4.2 Replay

To reproduce the original failure, JINSI replays the previously recorded interactions. For replay, *JINSI completely replaces the component's environment*. After instrumenting, the tool processes the captured events and, for each event, either triggers some incoming interaction on the observed objects[3], or consumes some outgoing interaction originating from the observed component and provides a proper return value. Interactions between observed objects are not intercepted; they happen naturally as a result of the incoming interactions initiated by JINSI.

Figure 6 illustrates an instrumented component on the basis of class `Observed` (Figure 5). JINSI has replaced the two outgoing 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 instead, which is just an empty hull representing an instance of proper type. When a subsequent incoming call to method `getTimeZoneName()` is replayed, JINSI again processes the originally captured outgoing call, and now returns the captured `String` representing the time zone at capture time.[4]

Other types of interaction, like constructor calls and field accesses, are captured and replayed in a similar fashion. Unobserved objects, which, for instance, may be passed as arguments, are replaced by mock objects that behave exactly as observed at capture, while observed ones are re-created as soon as required [24]. JINSI thus sequentially replays all interactions exactly as during the capture phase, thus reproducing the original failure.

5. SIMPLIFYING INTERACTIONS

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 4): *delta debugging, event slicing,* and *dynamic slicing.*

[3] JINSI uses the JAVA reflection API for that purpose.

[4] 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.

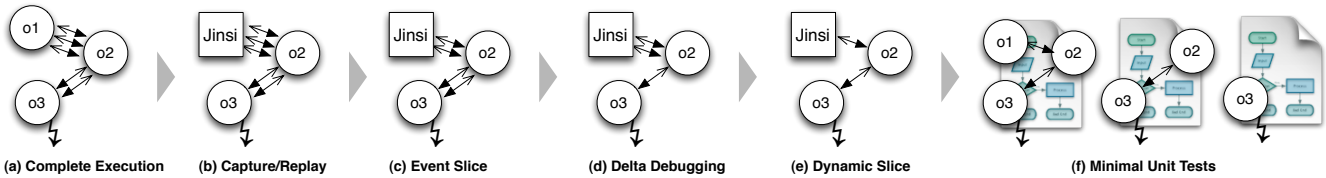

Figure 4: JINSI captures and replays (b) a failing execution (a). Event Slices (c), Delta Debugging (d), and Dynamic Slices (e) all minimize the relevant object interaction. Repeating the process while gradually incrementing the set of observed objects, JINSI produces a set of unit tests at varying abstraction levels (f).

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

public class Observed {
  private DateTimeZone timeZone;

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

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

Original Code

```
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);
}
```

Constructor with outgoing calls instrumented by JINSI

Figure 5: 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()`.

5.1 Delta Debugging

JINSI applies delta debugging [32] to automatically isolate the failure-inducing interactions by systematically testing subsets of the initial sequence—until a set is found where every remaining interaction is relevant for reproducing the failure. During this process, JINSI systematically *suppresses subsets of incoming interactions and assesses the result*: if the original failure occurs, the test fails (meaning that the suppressed interaction was not relevant); if it does not, the test succeeds (meaning that the suppressed interaction was relevant). This process is repeated until only the relevant interactions remain.

To assess the result of a test, JINSI needs a *predicate* that determines 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 an actually indispensable constructor call), the test is classified as *unresolved*. To check whether the simplified run reproduces the original failure, JINSI *compares the exception thrown.* In our example, an `ArithmeticException` is thrown because adding a time offset caused an integer overflow (see Figure 7). As long as a simplified, smaller subset reproduces the same exception[5], the failure in question is reproduced and delta debugging can continue to further minimize this set. A different exception results in an unresolved outcome.

For non-crashing failures, JINSI provides alternate predicates on both further externally observable properties and internal program state: for instance, output on the console, and properties on attributes of objects like "Attribute `name` of object with id 13 has value `"UTC"`." This enables JINSI to also *debug failures other than crashes caused by exceptions*, as we will see in Section 7.

5.2 Event Slicing

The effort for delta debugging depends on the number of unresolved outcomes. If every test fails, it converges in logarithmic time; if all tests are unresolved or passing, it requires quadratic time [32].

```
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 7: Running the code from the API example (Figure 1), JODA TIME crashes with an `ArithmeticException` caused by an integer overflow.

To speed up the delta debugging process, *JINSI applies a slicing technique before it uses delta debugging* (see Figure 4), a technique inspired by the work of Leitner et al. [22]. However, instead of applying slicing to the program code, *JINSI slices the captured sequence of interactions.* Basically, by following back data dependencies on the captured sequence of interactions recorded as events, we establish a list of possibly relevant interactions that describes the constructions and usages of all objects that are involved in the interactions that reproduce the failure. For example, starting at the incoming method call that throws the exception causing a crash, we put all objects involved in this interaction in an initial set. This includes the callee object and all non-primitive arguments. We then transitively include all events 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.

In the motivating example, applying the event slice to the 14,629 incoming interactions takes 24 seconds[6] and 1,940 interactions remain. The downstream delta debugging needs 38 seconds and 39 tests to minimize these interactions until the two failure-inducing

[5]We consider two exceptions to be the "same" if they have the same type, message, and location.

[6]All times were measured on a MacBook machine with a 2.1 GHz Intel Core 2 Duo processor and 4 GB memory.

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

Figure 8: After three iterations, JINSI isolates an interaction that pinpoints the defect.

ones remain (see Figure 8). 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.

5.3 Dynamic Slicing

After applying event slicing and delta debugging to the captured program run, JINSI eventually applies *dynamic slicing* using Hammacher's dynamic slicer for JAVA [18]. While delta debugging computes the minimal sequence of interactions that reproduce the failure, the subsequently applied dynamic 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.

6. PINPOINTING THE DEFECT

In order to help the developer debug a failure and finally fix the defect, JINSI requires a suspicious set of classes to start with. If a program crashes, JINSI by default starts with the topmost class on the stack and *automatically follows all the objects on that 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. Figure 1, for instance, shows the JODA TIME failure at the highest abstraction level, which is useful to *understand* the failure. To actually fix the failure, it is wiser to start at the lowest abstraction level—that is, the topmost class on the stack (`ZonedDurationField` in Figure 7)—and adding more and more involved classes (`ISOChronology`[7], `Period`, ...) as one proceeds towards the bottom.

This iteration strategy will pinpoint the defect in JODA TIME after only three iterations[8]. Examining the minimized unit test after adding the `Period` class shows that only two interactions are required to reproduce the failure (Figure 8): 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, 1:59am is followed by 3am, for instance. On this day, 2:30am is indeed an illegal time.

To fix the defect, we must call the `Period` constructor with legal values—namely, force the local time zone to UTC. Figure 9 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 had reproduced the original failure *and* execute the defective code. Each of these minimized unit tests consists of two lines or less, and execute only 54 lines overall. JINSI cannot fix the bug on its own, which is left to the programmer; but the amount by which the search space is reduced considerably eases debugging.

[7] `ISOChronology` is the concrete instance at location `BaseChronology.java:260` (Figure 7).

[8] 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`.

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

Figure 9: To fix JODA TIME, the time zone is set explicitly; thus, the code does not depend on the system's time zone anymore.

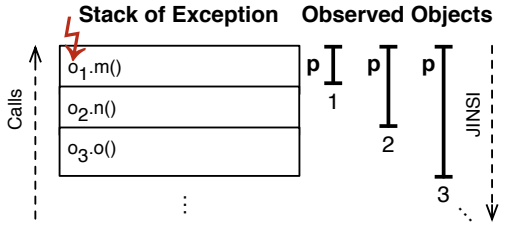

Figure 10: For crashing bugs, JINSI starts with observing the topmost target object (o_1) on the call stack, successively expanding the set of objects from caller to caller. Predicate p stays in the topmost method, identifying the crash, and thus minimizing the interaction that leads to the crash.

7. SELECTING OBSERVED OBJECTS

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 which the failure occurs, and gradually extends the set of objects along *cause-effect chains* that lead to the failure. This way, JINSI ensures that both the failing object as well as the object that causes the failure is included in the diagnosis. This section discusses the two general strategies used by JINSI.

7.1 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. For crashing bugs, JINSI uses the stack trace as a cause-effect chain; 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 (Figure 10). In the following steps, the set of objects is gradually expanded to include more and more callers. JINSI thus minimizes the interactions into the failing object (o_1), then the interactions into its caller (o_2), then the interactions into the caller of the caller (o_3), and so on. The result is a cause-effect chain of interactions in which every chain element is minimized.

7.2 Non-Crashing Bugs

In a *non-crashing bug,* a failure comes to be as some incorrect output—or, more generally, an incorrect or "infected" program state. Such infections are usually much harder to debug because the infection is discovered only at the end of the execution. In such a situation, the programmer must identify the source of the infection chain, progressing backwards along the origins of values.[9]

[9] Such a situation can also happen with crashing bugs, although this

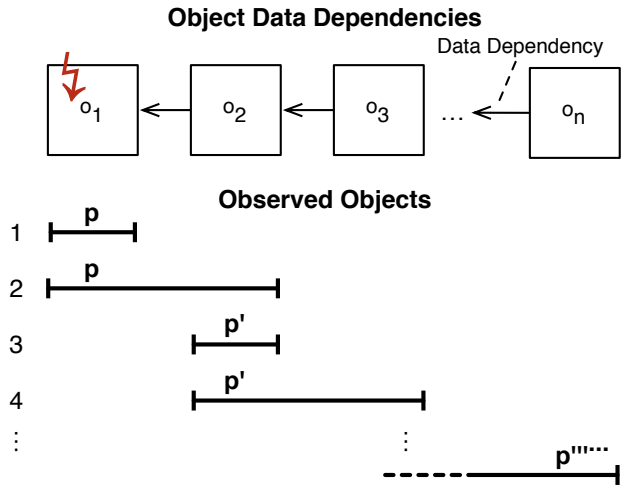

Figure 11: For non-crashing bugs, JINSI progresses along the dynamic backward slice, minimizing individual subslices of object interaction to obtain a minimized cause-effect chain.

This is where a dynamic backward slice is most helpful, as it contains precisely those values that could have influenced the failure; and this is also what JINSI uses to track back the infection chain. JINSI starts with a given predicate on object state (typically, a failing test oracle) and determines the dynamic backward slice from that object. It then progresses along the backward slice using a *sliding window* approach (Figure 11): First, the interaction into the last object on the slice (o_1) is minimized. Then, the interaction into the objects o_1 depends upon (o_2) is minimized. Then, the interaction into the next level of dependency (o_3) is minimized, and so on. Again, we obtain a cause-effect chain of interactions along the data dependencies in which every chain element is minimized.

7.3 Generating Predicates

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. Instead, we go for a *stepwise minimization*: Rather than minimizing the entire set on the dynamic slice, we minimize each element on the chain—that is, for every object, only the object itself as well as the objects it directly *depends* upon. Our observed set thus would contain, for instance, o_2 and o_3, but not o_1; as well as o_3 and o_4, but not o_1 or o_2.

Removing the initial failing object from the set of observed objects brings another challenge, though: We lose the original predicate deciding on success or failure, and thus have no predicate anymore to minimize against. JINSI therefore generates *alternative predicates* (shown as $p\prime$, $p\prime\prime$, and so on in Figure 11).

A predicate needs to distinguish passing and failing runs. Since we start with a single failing run, where do we get the passing runs from? The answer is simple: *Delta debugging yields similar passing runs* as a by-product.[10] These synthetic passing runs are the base for the subsequent predicate generation.

As in [31], JINSI extracts *memory graphs* for both the single failing run and all the produced passing runs, and afterwards computes *differences* using these graphs. Finally, JINSI derives proper predicates from these differences. In this way, JINSI automatically derives predicates like "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 being minimized against, and again, the end result is a cause-effect chain along the dependencies, with the predicates as inbetween oracles. Figure 12 formalizes the algorithm used by JINSI to debug non-crashing bugs.

```
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 ← {obs_init}
while obs_next ← next_by_distance(dyn_slice) do
    observed ← observed ∪ {obs_next}
    captured ← capture(observed)
    1_min ← dd(captured, pred)
    state_f ← mem_graph(1_min, obs_next)
    let pred be an empty predicate
    for all interactions int in 1_min do
        subset ← 1_min \ {int}
        if outcome(subset) is passing then
            state_p ← mem_graph(subset, obs_next)
            diff ← mem_diff(state_p, state_f)
            if diff is not empty then
                pred_on_diff ← predicate(diff)
                pred ← pred ∧ pred_on_diff
            end if
        end if
    end for
    if pred is not empty then
        observed ← {obs_next}
    end if
end while
```

Figure 12: For non-crashing bugs, JINSI takes advantage of the *data-flow information* provided by the slicer to select observed objects, and of the *1-minimality* of the delta debugging algorithm to generate predicates. For simplicity, we only show how the algorithm would work on *linear* data dependencies. In fact, the slicer yields a data dependence *graph*, and JINSI extends the algorithm by a *breadth-first search*.

8. EXPERIMENTAL EVALUATION

In this section, 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.[11] The dynamic slice contains all lines that possibly could have contributed to the failure; it is this set that we want to minimize.

8.1 Subjects

To evaluate the effectiveness of JINSI, we have applied it to six different JAVA subjects listed in Table 1. The subjects are divided into three groups according to their context:

Standard subjects. This set consists of two subjects used by other researchers before: **BST** by Artzi et al. in evaluating RECRASH [2] and by Csallner and Smaragdakis in evaluating

was not the case in the bugs we observed.

[10]This is due the 1-minimality of the algorithm [32]—no single interaction can be removed without removing the failure.

[11]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 1: Subjects used in the case studies.

Subject	Description
BST	Subject used by Artzi et al. [2] and Csallner et al. [8].
NANOXML	XML parser; part of the SIR Repository [12]; used by [10, 9, 15]
COLUMBA	Feature-rich email client with graphical user interface as used before in [3].
VENDING MACHINE	Proof of concept that demonstrated JINSI's feasibility in [24].
JAXEN	Universal JAVA XPATH engine. Used by SUN in several products [23].
JODA TIME	Replacement for the JAVA date and time classes. More than 100k downloads [6].

Check 'n' Crash [8]. **NANOXML** is a common subject as part of the Software-artifact Infrastructure Repository [12] (SIR) and used by many research groups for evaluation purposes [10, 9, 15].

Previous work. We use two examples from earlier work on JINSI: **VENDING MACHINE** [24], a proof-of-concept program, and **COLUMBA** [3], a complex email client consisting of several components using a graphical user interface—in contrast to the other subjects.

Industrial-size subjects. We applied JINSI on two *industrial-size subjects*. Both **JAXEN** and **JODA TIME** are large JAVA libraries actively maintained and used by a large number of users.[12]

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 requires an object-oriented execution model. These circumstances constrain the available subjects. For example, JINSI is not applicable to subjects like the non-object-oriented Siemens test suite.

Based on the above subjects, we applied JINSI on a total of 17 individual issues (Table 2), consisting of 14 crashing and three non-crashing bugs, which we selected as follows: BST contains three individual crashing bugs used both by Artzi et al. as well as Csallner and 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). Finally, we added three non-crashing bugs we found (`NAN-3`, `NAN-5`, and `NAN-6`) to explore JINSI's performance on non-crashing bugs.

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 criteria above; for NANOXML, we have selected issues randomly from four different versions as stored in SIR. All this ensures a selection independent from expected results.

[12]JODA TIME was downloaded 126,536 times as of 2010-07-30 [6]; JAXEN is used by SUN in several products [23].

8.2 Experiment Setup

To measure the effectiveness of our approach, we applied JINSI on each of the 17 issues listed in Table 2. 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 [29], shortened SLOC. To count the number of lines executed by the original failing run, we applied COBERTURA [13], an open-source coverage tool. Hammacher's dynamic slicer for JAVA [18] provided the lines in the dynamic slice. As slicing criterion, we chose the location where the exception had been thrown. 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. These numbers can be found in the left part of Table 3.

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 8.6). Secondly, for issue `COL-1`, we were not able to obtain a dynamic slice due to 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.

8.3 Search Space Reduction

For our motivating example (`JOD-3`), the total number of source code lines is 26,534[13] (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 contains 54 lines (0.20 %) only. 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 3. 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[14]. 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 our evaluation, JINSI reduces the search space to **3.1 %** of the executed lines, or **0.22 %** of the source code.*

8.4 Size of Resulting Unit Tests

Further results concern the number of levels of abstraction to be examined by the developer, the total number of interactions and the

[13]The SLOC varies between the individual issues of the very same project as each issue is related to a different version.

[14]We omit `COL-1` from the total as the slicer crashed; hence the subtrahends.

Table 2: Issues used in the case studies.

Issue	Project-Specific Issue ID	Version	Description
BST-1	n/a	n/a	`BSTNode.setData(...)` crashes on given `char` array.
BST-2	n/a	n/a	`BSTNode.setData(...)` crashes on given `Object` instance.
BST-3	n/a	n/a	`StringCoding.encode(...)` fails on given charset name.
NAN-1	SP_HD_1	1	Parsing XML document fails.
NAN-2	XE_HD_2	2	Parsing XML document fails.
NAN-3	XEL_HD_2	3	Obtaining named children from XML tree fails.
NAN-4	XEL_HD_6	3	Removing child from XML tree fails.
NAN-5	CR_HD_2	5	XML entities are not handled correctly.
NAN-6	XER_HD_1	5	Output contains unexpected artifact.
COL-1	n/a	1.4	Importing an address book fails with nondescript error dialog.
VME-1	n/a	n/a	Vending machine erroneously stays in enabled state.
JAX-1	29	r375	XPATH function `normalize-space` on empty argument `''` fails.
JAX-2	111	r1170	Selecting node on empty document crashes.
JAX-3	156	r1216	Changing a node-set does not update the position or node size.
JOD-1	1788282	r1256	Parsing valid French date using French locale fails.
JOD-2	2487417	r1377	Converting date in Brazilian time zone fails.
JOD-3	2889499	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.

number of the relevant ones per level, as well as the runtime behavior of the whole debugging process. These numbers can be found in the right part of Table 3. 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 (Figure 8). 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.***

8.5 Performance

The last two columns show the runtime behavior of JINSI. The second last column shows the total runtime of the debugging process. This includes capture and minimization, as well as the upstream instrumentation.[15] 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 $O(n^2)$ [32].

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 5.2, 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. [22], we thus 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.***

8.6 Limitations and Threats to Validity

While conducting the experiments, we encountered some limitations of our approach:

JAVA reflection. NANOXML instantiates some (hard coded) classes using *factory methods*. 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 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—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 by the delta debugging algorithm. Interestingly enough, the dynamic slice does not contain this

[15] 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.

Table 3: Results of the experimental evaluation.

Issue	Type	SLOC	Original Run		Minimized Run		Levels		Interactions		Runtime	
			Line Coverage	Dynamic Slice	Line Coverage	Intersection	Total	To examine	Total	Relevant	Time [min:s]	Tests
BST-1	C	38	5	3	2	1	2	1	2	2	0:10	4
BST-2	C	38	14	3	11	1	2	1	2	2	0:10	4
BST-3	C	38	1	1	n/a	1	3	n/a	n/a	n/a	n/a	n/a
NAN-1	C	1,891	130	43	42	42	3	2	5–6	4–5	0:50	38
NAN-2	C	2,540	98	15	13	2	2	1	5	2	0:26	14
NAN-3	N	3,118	498	199	18	8	9	3	1–98	1–4	0:57	217
NAN-4	C	3,101	501	252	31	18	3	1	90	8	0:29	151
NAN-5	N	3,278	584	304	218	79	11	4	2–99	2–12	2:21	508
NAN-6	N	3,295	594	313	154	75	11	7	2–566	2–12	37:16	2707
COL-1	C	94,863	6,318	crash	5	5	29	1	14,008	2	1:08	4
VME-1	C	185	121	63	23	10	2	1	32	7	0:15	68
JAX-1	C	13,226	2,533	1,224	37	19	6	3	6–1,919	2–3	3:40	416
JAX-2	C	12,983	861	255	7	3	7	2	3–4	2–3	0:53	20
JAX-3	C	12,957	16	4	12	4	2	2	5–6	3–4	0:26	39
JOD-1	C	25,748	1,613	150	946	48	3	3	4–8	3–5	1:15	60
JOD-2	C	25,810	1,455	373	248	145	4	4	80–123	2–3	4:29	190
JOD-3	C	26,534	1,528	512	193	54	6	3	45–14,628	1–2	4:58	243
Total		229,643	16,870	3,714	1,960	515						

SLOC = physical source lines of code [29]. In totals, missing values ("crash" or "n/a") are counted as zero. C = crashing, N = non-crashing.

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 as 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 fine-grained 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 also can use 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 and further counter it by making JINSI and all the experimental data publicly available.

9. CONCLUSIONS AND CONSEQUENCES

During debugging, JINSI brings lots of benefits at little cost. Its requirements are trivial: all it takes is one single failing run that is observed by JINSI. The resulting minimized unit test is easy to understand, and encompasses all steps that are relevant to reproduce the failure. 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, JINSI eases debugging to the point where it ceases to be a problem.

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 take advantage of *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 interactions, 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* [11].

JINSI and all material required for reproducing the experiments are available for download. For details, see

`http://www.st.cs.uni-saarland.de/jinsi/`

Acknowledgments. This work was supported by grants Ze-509/2-1 and Ze509/4-1 from Deutsche Forschungsgemeinschaft. The key idea of minimizing object interaction was conceived together with Alessandro Orso, who provided valuable feedback all throughout the project. Valentin Dallmeier, Gordon Fraser, Clemens Hammacher, David Schuler, Kevin Streit, and Andrzej Wasylkowski provided helpful comments on earlier revisions of this paper.

10. REFERENCES

[1] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *ISSTA '10*, pages 49–59. ACM, 2010.

[2] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *ECOOP '08*, pages 542–565, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] M. Burger and A. Zeller. Replaying and isolating failing multi-object interactions. In *WODA '08*, pages 71–77, New York, NY, USA, 2008. ACM.

[4] J. Clause and A. Orso. A technique for enabling and supporting debugging of field failures. In *ICSE '07*, pages 261–270, Washington, DC, USA, 2007. IEEE Computer Society.

[5] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05*, pages 342–351, New York, NY, USA, 2005. ACM.

[6] S. Colebourne. Joda Time—Java date and time API. http://joda-time.sourceforge.net/.

[7] B. L. Computer, B. Liblit, A. Aiken, M. Naik, and A. X. Zheng. Scalable statistical bug isolation. In *PLDI '05*, pages 15–26. ACM Press, 2005.

[8] C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *ICSE '05*, pages 422–431, New York, NY, USA, 2005. ACM.

[9] C. Csallner and T. Xie. DSD-crasher: A hybrid analysis tool for bug finding. In *ISSTA '06*, pages 245–254. ACM, 2006.

[10] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *ECOOP '05*, number 3586 in Lecture Notes in Computer Science, pages 528–550. Springer, July 2005.

[11] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *ASE '09*, Auckland, New Zealand, November 2009.

[12] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, 2005.

[13] M. Doliner. Cobertura. http://cobertura.sourceforge.net/.

[14] A. Z. Ee, A. X. Zheng, M. I. Jordan, B. Liblit, and A. Aiken. Statistical debugging of sampled programs. In *Advances in Neural Information Processing Systems 16*, pages 9–11. MIT Press, 2003.

[15] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil. Carving differential unit test cases from system test cases. In *SIGSOFT '06/FSE-14*, pages 253–264, New York, NY, USA, 2006. ACM.

[16] S. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *IEEE Trans. Softw. Eng.*, 35(1):29–45, 2009.

[17] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. In *ASE '05*, pages 263–272, New York, NY, USA, 2005. ACM.

[18] C. Hammacher. Java slicer. http://www.st.cs.uni-saarland.de/javaslicer/.

[19] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE '05*, pages 273–282, New York, NY, USA, 2005. ACM.

[20] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02*, pages 467–477, New York, NY, USA, 2002. ACM.

[21] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *ISSRE '05*, pages 267–276, Washington, DC, USA, 2005. IEEE Computer Society.

[22] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *ASE '07*, pages 417–420, New York, NY, USA, 2007. ACM.

[23] B. McWhirter. JAXEN — Universal Java XPath engine. http://jaxen.codehaus.org/.

[24] A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant component interactions with JINSI. In *WODA '06*, pages 3–10, New York, NY, USA, 2006. ACM.

[25] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *WODA '05*, pages 29–35, St. Louis, MO, USA, may 2005.

[26] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *ASE '05*, pages 114–123, New York, NY, USA, 2005. ACM.

[27] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *ISSTA '00*, pages 158–167, New York, NY, USA, 2000. ACM.

[28] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.

[29] D. A. Wheeler. SLOCCount — count source lines of code (SLOC). http://www.dwheeler.com/sloccount/.

[30] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *ESEC/FSE '99*, pages 253–267, London, UK, 1999. Springer-Verlag.

[31] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10*, pages 1–10, New York, NY, USA, November 2002. ACM Press.

[32] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.

[33] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *ESEC/FSE '09*, pages 43–52, New York, NY, USA, 2009. ACM.