Isolating Relevant Component Interactions with JINSI

Alessandro Orso · Shrinivas Joshi
College of Computing
Georgia Institute of Technology
{orso, sjoshi}@cc.gatech.edu

Martin Burger · Andreas Zeller
Computer Science
Saarland University
{mburger, zeller}@cs.uni-sb.de

ABSTRACT

When a component in a large system fails, developers encounter two problems: (1) reproducing the failure, and (2) investigating the causes of such a failure. Our JINSI tool lets developers capture and replay the interactions between a component and its environment, thus allowing for reproducing the failure at will. In addition, JINSI uses delta debugging to automatically isolate the subset of the interactions that is relevant for the failure. In a first study, JINSI has successfully isolated the relevant interaction of a JAVA component: "Out of the 32 interactions with the `VendingMachine` component, seven interactions suffice to produce the failure."

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Debugging aids*;

General Terms: Algorithms

Keywords: Automated debugging, capture and replay techniques

1. INTRODUCTION

Understanding why a program fails is a complex activity that usually encompasses two tasks. The first task consists of *reproducing* the failure, so that the failing run can be repeated at will and analyzed. The second task consists of discovering *what is relevant* in the failing run—which events caused the failure in question, and which ones were instead irrelevant. Both of these tasks are tricky and make debugging difficult. This is especially true for modern software systems, which are often built through aggregation of components whose interactions with other components and with the environment is increasingly dynamic.

In this paper, we propose an *integrated approach* that addresses these issues and allows for automatically (1) reproducing failing runs at the component level and (2) identifying events that are not related to the failure and can be eliminated, thus simplifying the run to be analyzed and, consequently, debugging. Our approach consists of two main parts:

- We *capture and replay* the interaction of a (failing) component with its environment. The interaction is recorded as a sequence of incoming and outgoing *events*. Such events can be method calls, accesses to field values, and exceptions. This sequence can be replayed at will on the component without requiring a complete environment to be available. In this way, we achieve a twofold advantage: we eliminate the need to reproduce the entire failing run and exercise the component exactly in the same way that led to its failure.
- A recorded interaction is a sequence that can easily consist of thousands, even millions, of events. However, only a subset of this sequence may be required to make the failure occur. We use *delta debugging* to automatically *isolate the failure-inducing events*. The idea is to systematically test subsets of the initial sequence until a set is found where every remaining call is relevant for producing the failure.

We have developed a prototype, called JINSI, that implements the above approach for JAVA programs. Given a system and a component (expressed as a set of classes) within the system, JINSI records interactions between such component and the rest of the system. When an execution fails, JINSI performs delta debugging on the recorded interactions and automatically isolates the subset of events that were relevant for the failure.

To validate and evaluate our approach, we perform a case study in which we use JINSI on a small example subject. Although preliminary, our results are promising: They show that JINSI can indeed isolate short sequences of interactions that make a component fail. Such sequences ease the debugging task by considerably reducing the amount of information that developers must consider.

In this paper, we make the following original contributions:

1. We present our technique for capturing, replaying, and minimizing interactions between a failing component and the rest of the system, so as to obtain a minimal, partial execution that can reproduce the failure.
2. We introduce JINSI, our prototype tool that implements the approach for JAVA software by integrating capture-replay and delta debugging capabilities.
3. We provide a proof-of-concept empirical study that shows how our approach can effectively isolate relevant interactions for a component and so improve the efficiency of debugging.

The rest of the paper is organized as follows. Section 2 introduces an example that we use to motivate and illustrate our approach. Sections 3 and 4 describe JINSI and present a case study that we performed using JINSI. We discuss related work in Section 5, followed by additional applications in Section 6. Finally, Section 7 summarizes the paper and describes future research directions.

2. MOTIVATING EXAMPLE

Before going into details, let us introduce a motivating example. The example consists of a set of users that are running an application on their machines. The application is a simple (imaginary) figure editor, called `easyDraw`, that allows users to perform a variety of graphic operations, such as adding and removing geometric figures, coloring them, and moving them around. Assume that there is a fault in `easyDraw`, and one of the users reveals the fault by

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WODA'06, May 23, 2006, Shanghai, China.
Copyright 2006 ACM 1-59593-085-X/06/0005 ...$5.00.

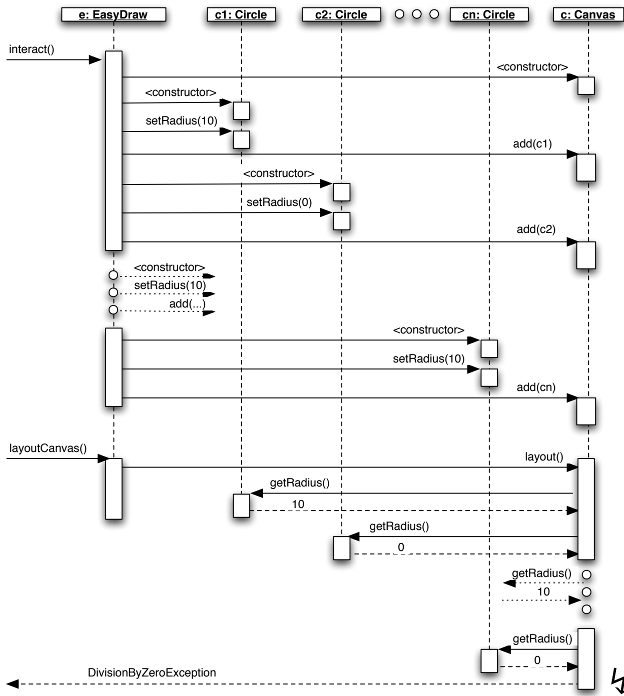

Figure 1: A faulty canvas. After adding n circles, layouting the canvas throws an exception.

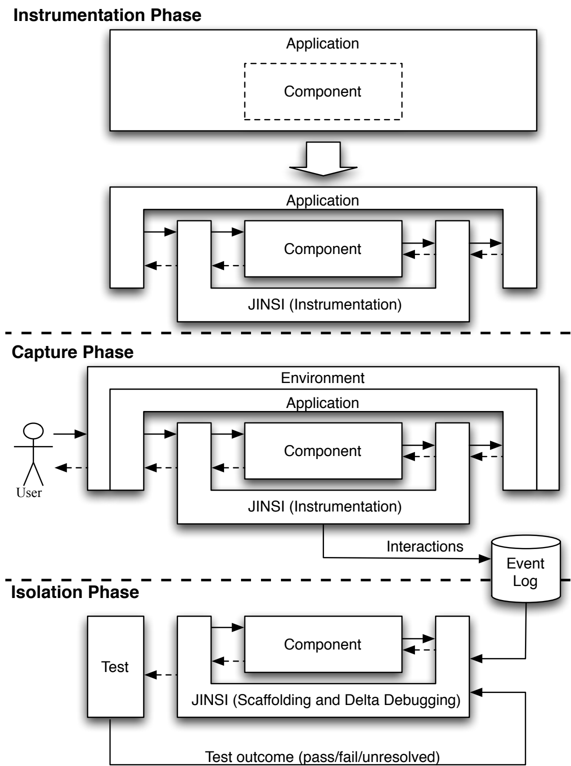

Figure 2: JINSI phases. After instrumentation, JINSI captures events, to be minimized by delta debugging.

performing the following actions (see Figure 1): create a canvas `c`, add a number n of circles, `c1`, `c2`, ..., `cn`, to the canvas, and layout the figure. This last operation results in an invocation of `c.layout()`, throwing a `DivisionByZeroException`.

Typically, the user would be given the possibility to send back some crash-related information to the application developer. In this case, such information would be the exception that was thrown and the stack trace at the time of the failure. The stack trace would simply include the three methods that were last active: `main()`, `EasyDraw.layoutCanvas()`, and `Canvas.layout()`.

Assuming that `easyDraw` is a non-trivial application, debugging the problem with such a stack trace would be fairly difficult. The programmer would need to (1) recreate a set of interactions that cause an exception at that point in the program, and (2) trace back the cause of the error, possibly restarting the failing application over and over. What the developer would need is a trace of all the interactions between the failing component and the rest of the application. However, collecting and sending to the developer a complete trace is typically impractical—there could be millions of interactions to be recorded and transmitted. Besides creating problems in terms of storage, bandwidth, and even privacy, such traces would likely contain too much noise and would not necessarily help to improve the efficiency of the debugging activity.

In the next sections, we describe our approach that addresses these issues by collecting and reporting a minimized execution trace that contains only interactions that are relevant for the failure. Such a trace would allow for a more efficient and effective debugging than either a generic crash report or a complete execution trace.

3. THE JINSI APPROACH

We call our approach, and the tool that implements it, JINSI (for "JINSI Isolates Noteworthy Software Interactions").[1] JINSI has three phases: instrumentation, capture, and isolation. Figure 2 provides an overview of these phases. In the *instrumentation phase*, JINSI is invoked with an application and a component (specified as a list of classes); it then produces a new version of the application instrumented for recording the component's interaction. In the *capture phase*, JINSI's instrumentation intercepts, captures, and records all of the interactions between the component and the rest of the application while users are running the software. The interactions captured include method calls, field accesses, and exception flows. In the *isolation phase*, JINSI applies delta debugging to systematically isolate those interactions that are relevant for the failure.

In a possible scenario of use, the developer identifies a component `c` in an application that is failing for some input and wants to identify a minimal set of stimuli for the component that would reproduce the failure. To this end, the developer uses JINSI on the application, specifying `c` as the component of interest, while rerunning the application against the input that caused the failure. JINSI instruments the application, records the incoming and outgoing events for `c`, and computes a minimized set of interactions that is then reported to the developer. The developer can use this minimized interaction set to efficiently debug the component problem.

To show how JINSI works, we use our example from Section 2. Assume that the component instrumented by JINSI is canvas `c`. The failing execution results in a trace containing $3n + 3$ interactions:

- one incoming call to `c`'s constructor
- n incoming calls to `add(c1)`, ..., `add(cn)`
- one incoming call to `layout()`
- n pairs of interactions (one of which returns 0):
  - outgoing call to `getRadius()`
  - incoming return from `getRadius()`
- one outgoing exception `DivisionByZeroException`

[1]"Jinsi" is also the Swahili word for "method," which is the most common interaction recorded by our approach.

When JINSI performs delta debugging to isolate those interactions that are relevant for the failure, it finds that a sequence of five interactions is sufficient to cause the failure:

1. an incoming call to `c`'s constructor,
2. an incoming call `add(c2)`, and
3. an incoming call `layout()` as well as
4. one *outgoing call* to `getRadius()`, and
5. the corresponding return with value 0.

All the other interactions are completely irrelevant to the failure.

With such a short sequence of interactions, debugging is far easier than with a complete execution trace. In this example, the developer would easily infer that the `layout()` method does not handle circles with a radius of zero, thus causing the failure.

In the rest of this section, we discuss the three phases of our approach and the details of the technique. In the discussion of the technique, we use the following terminology. We refer to the component whose interactions must be recorded as the *observed component* (or simply component) and to the classes in the observed component as the *observed classes* (or code). *Observed methods* and *observed fields* are methods and fields of observed classes. In an analogous way, we define the terms *external code*, *external classes*, *external methods*, and *external fields*. Basically, we indicate as *external* anything that is either in the part of the application that does not include the component or in the library code.

3.1 Instrumenting Code

To capture the interactions between the component of interest and the rest of the application, JINSI leverages a capture/replay technique that (1) identifies all of the interactions between observed and external code, (2) suitably instruments the application code, and (3) efficiently captures interactions at runtime [6]. JINSI's instrumentation is designed to capture all types of interactions between two parts of the code: incoming and outgoing method calls, incoming and outgoing return values, accesses to field, and exception flows.

The type of instrumentation inserted varies based on the type of event to be captured. For example, to capture incoming calls and corresponding returns, JINSI performs two steps:

1. In a first step, JINSI replaces each public method `m()` in the component with a proxy method and an actual method. The *actual method* has the same body as `m()` (modulo some instrumentation), but has a different signature that takes an additional parameter of a special type. The *proxy method*, conversely, has exactly the same signature as `m`, but a different implementation. The proxy method (1) creates and logs the appropriate call event, (2) calls the actual method by specifying the same parameters it received plus the parameter of the special type, (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).
2. In the second step, JINSI modifies all calls from observed methods to other observed methods by adding the additional parameter of the special type mentioned above. In this way, we are guaranteed that calls that do not cross the boundaries of the observed code 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).

Due to lack of space, we do not describe how our instrumentation technique operates for all types of events we capture. Complete details about the instrumentation are provided in [6].

3.2 Capturing Events

The capture phase takes place while the application is running (e.g., in the field or during testing). At runtime, the probes added to the code in the instrumentation phase suitably capture the interaction between the component and the external code and record them in the form of events with attributes. The events, together with their attributes, are recorded in an *event log*.

3.2.1 Capturing Execution Events

JINSI captures three main kinds of event: method calls, accesses to field, and exceptions.

Method calls. The most common way for two parts of an application to interact is through method calls. In our case, we must account for both calls from the external code to the component (*incall*s) and calls from the component to the external code (*outcall*s). Note that the technique does not need to record calls among observed methods because such calls occur naturally during replay.
Our technique records four kinds of events related to method calls: (1) OUTCALL events, for calls from observed to unobserved code; (2) INCALL events, for calls from unobserved to observed code; (3) OUTCALLRET events, for returns from outcalls; and (4) INCALLRET events, for returns from incalls. These events contain enough information (i.e., attributes) to be able to reproduce the event during replay. For example, INCALL events have three attributes: the receiver object, the signature of the method being called, and the list of parameters.

Field accesses. Interactions between different parts of an application also occur through accesses to fields. To account for these interactions, our technique records accesses to observed fields from external code and accesses from the component to external fields. In the case of accesses from external code to observed fields, we only record write accesses—read accesses do not affect the behavior of the component and, thus, do not provide any useful information for replay.
Our technique records three kinds of events for accesses to fields: (1) OUTREAD events, for read accesses from observed code to external or library fields; (2) OUTWRITE events, for write accesses from observed code to external or library fields; and (3) INWRITE events, for modifications to an observed field performed by external code. Also in this case, the events contain enough information to be replayed (e.g., the object containing the field, the field name, and the value being written for INWRITE events).

Exceptions. Exceptions too can cause interactions between different parts of an application. Moreover, interactions due to exceptions occur through implicit changes in the applications' control flow and are typically harder to identify than other types of interactions.
To capture interactions that occur due to exceptions, our technique records two types of events: (1) EXCIN, for exceptions that propagate from external code to the component; and (2) EXCOUT, for exceptions that propagate from the component to external code. EXCIN and EXCOUT events have only one attribute that consists of the type and object ID of the corresponding exception.

3.2.2 Capturing Partial Information

When capturing data flowing through the boundary of a component (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 compos-

```
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;
}
```

Figure 3: The `hasPositiveElement()` method. Replaying interaction requires only a minimum of recorded information.

ite objects. Whereas capturing scalar values can be done inexpensively, collecting object values is computationally and space expensive. A straightforward approach that captures all values through the system (e.g., by serializing objects passed as parameters) would incur in a tremendous overhead and would render the approach impractical. (In preliminary work, we measured time overhead of over 500% for a technique based on object serialization.) Our key intuition to address this problem is that (1) we only need to capture the subsets of those objects that affect the computation, and (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 in Figure 3. The method `hasPositiveElement()` 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` contains millions of elements, we 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 ID [2] and type. (We need to record the type to be able to recreate the object during replay.) With this approach, object IDs, types, and scalar values are the only information required to replay executions, which can dramatically reduce the space and time costs of the capture phase.

3.3 Isolating Relevant Events

A common task in debugging is to *simplify the input*—that is, to find out which part of the input is relevant for the failure in question.

[2] Our technique associates a unique ID to every object by suitably instrumenting constructors.

```
 1 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✘
 2 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
 3 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
 4 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
 5 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✘
 6 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✘
 7 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
 8 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
   ⋮
19 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
20 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
21 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
22 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
23 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
24 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
25 <SELECT NAME="priority"␣MULTIPLE␣SIZE=7> ✔
26 <SELECT NAME="priority"␣MULTIPLE␣SIZE=7> ✘
```

Figure 4: Simplifying failure-inducing HTML input (from [10]). Delta debugging simplifies input by running systematic experiments.

The benefits of a simplified input include cutting away irrelevant information; it also normally results in shorter program runs which cover less of the code.

In earlier work [10], Zeller and Hildebrandt had devised a method called *delta debugging,* an automatic means to simplify failure-inducing input. Delta debugging would repeat the failing test while systematically suppressing parts of the input, eventually coming up with a simplified input where every remaining part is relevant for producing the failure. An excerpt from such a simplification process is shown in Figure 4, where delta debugging minimized an original 896-line HTML input which caused the Mozilla browser to crash. Each input was checked whether the failure still occurred (✘) or whether the test passed (✔). Eventually, the input was reduced to a single failure-inducing HTML tag; printing out this `<SELECT>` tag would suffice to make Mozilla crash.

Unfortunately, what works nicely in a lab environment can still have major problems in practice, especially when it comes to applying one method to a variety of scenarios. In fact, applying delta debugging to input involves a number of nontrivial requirements:

We need a means to *reproduce the run automatically.* Correctly reproducing a failure can be among the most time-consuming tasks in debugging—especially if the application in question is not prepared for automation. In the Mozilla example above, reproducing the failure included setting up an environment that simulated user interaction to load the HTML file and attempt to print it.

We need a means to *control the input.* Inputs are frequently produced by other components—in the case of Mozilla, an HTML server—that may be hard to control. Again, one must hope for the application to provide appropriate interfaces; Mozilla, for example, can also read HTML input from a file.

We need knowledge about *the input structure.* In Figure 4, the input is simplified on a character-by-character basis. Yet, HTML has its own syntactic structure, which should be exploited when simplifying. Non-textual inputs can hardly be simplified effectively without knowledge of their structure.

JINSI overcomes all of these problems. Programmers can use JINSI to record component interactions and replay them at will. Applied to the `easyDraw` application discussed earlier, for instance, JINSI would capture interactions during an interactive (and possibly remote) session, and then replay the recorded interactions—which

trivially implies complete control over the invoked method calls. Thus, JINSI users have a means to reproduce the run, and a means to control the input.

When it comes to simplifying the interactions, JINSI uses delta debugging to suppress method calls rather than input. Applied to the interactions shown in Figure 1, for instance, JINSI systematically suppresses subsets of events and assesses the result: if the `Canvas` raises an exception, 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 interaction remains—the five interactions discussed in Section 2.

The benefit of minimizing interaction rather than input is that the original input structure is reflected in the structure of method calls and their arguments. For instance, the original user interactions, adding circles to the canvas, are all reflected in individual method calls. As these are systematically suppressed during JINSI's isolation phase, the process can often be equivalent to appropriate transformations/reductions of the original input.

One possible downside of JINSI is that the programmer must choose a *boundary* at which interaction is to be recorded and replayed. However, if this boundary is well-chosen, JINSI can totally replace simplification of input. Moreover, because the size of the component that we capture can ideally range from one class to the whole program, JINSI can also be used to record and replay entire applications. For example, consider a program that uses a standard parser to process its input. Assume that, whenever some syntactic entity is read, the parser invokes a method to process the entity. If the programmer chooses this point as the boundary between observed and external code, JINSI will effectively record and replay the application input. Even better: As JINSI suppresses the method calls associated with the processing of some entity, the application will act as if the entity was missing in the input—that is, it will act as if the input had been simplified according to its syntactic structure. This makes JINSI not only more general, but also far more versatile than simplifying input. (In Section 6, we provide more examples of possible usage scenarios for JINSI.)

4. CASE STUDY

To assess the feasibility of our approach, we performed a case study on an example subject: an implementation of a *vending machine.* We picked this example because it is simple enough to be explained in a paper, yet not entirely trivial. Moreover, it is a pre-existent example that was not created by any of the authors and has been used in a number of papers.

The vending machine example consists of three classes. The `Coin` class represents coins of various values, such as nickels, dimes, and quarters. Class `UI` is the user interface for the overall application. Through the interface, users can insert coins (`Coin` instances), ask for having their coins back, and buy an item. The `VendingMachine` class implements the actual functionality of the vending machine and is operated through the user interface (i.e., an instance of `UI` drives the `VendingMachine` based on the inputs received from the user).

This application, and more precisely the `VendingMachine` class, contains a defect. In Figure 5, we see a part of the source code of the `VendingMachine` class, illustrating the error. Every time a coin is inserted, through a call to method `insert()`, its value is added to the current credit (variable `currVal`). When the credit reaches the cost of an item (75 cents), the `enabled` flag is set to true. However, after a sale, the `enabled` flag is reset only if the credit is 0. In other words, if there is any credit residue after the sale, the machine erroneously stays in the enabled state, and a following request to vend is successful even if the credit is insufficient, which is a defect.

```
class VendingMachine {
   private int currValue;
   private boolean enabled;

   public VendingMachine() {...}

   public void insert( Coin coin ) {
      int value = coin.getValue();
      currValue += value;
      if (currValue >= COST)
         enabled = true;
   }

   public void returnCoins() {...}

   public void vend() throws IllegalCreditException {
      if (enabled) {
         System.out.println( "Get your drink");
         currValue -= COST;
         if (currValue == 0)
            enabled = false;
      }
      else {...}
      // invariant
      if (currValue < 0)
         throw new IllegalCreditException();
   }
}
```

Figure 5: A `VendingMachine` Component. The `vend()` method has a defect, causing an exception being thrown.

For example, if a user adds 80 cents, requests an item and then requests a second item, he or she would receive both of them. At this point, the execution would terminate with an exception because the class invariant that the credit must be ≥ 0 has been violated (the credit would be -70 after the second sale).

Figure 6 shows parts of a possible sequence of interactions between the `VendingMachine` and the rest of the application that may lead to a failure. For our case study, we generated random user inputs and kept providing such inputs to the application, through the `UI`, until we generated a failure.

We repeated the operation ten times and kept the longest sequence, which consisted in 28 user operations (*insert*s, *coin returns*, and *vend*s). We then invoked JINSI on the application, specifying `VendingMachine` as the component of interest, and then re-run the instrumented application.

In its capture phase, JINSI produced an event log that contained 32 interactions. In its isolation phase, JINSI reduced these 32 interactions to seven, listed in Figure 7.

These seven events are indeed the minimal set of interactions necessary to reproduce the fault in the `VendingMachine` component. They create an instance of the component, add 85 cents by inserting appropriate coins, and invoke `vend()` twice. After the second call to `vend()`, the `VendingMachine` object would fail with an `IllegalCreditException`. Removing any of these events would result in a sequence that does not cause the failure to occur—that is, each single event is relevant for producing the failure.[3]

Although JINSI's performance is not an issue at this stage of the research, we measured how long it took to isolate the relevant inter-

[3]To simplify the presentation, the list above does not show the outgoing events, which correspond to OUTCALL events from `VendingMachine` to `Coin.getValue()` and corresponding OUTCALLRET events. Such events are not stimuli for the component, but rather events from the component that JINSI suitably consumes by providing the appropriate scalar values for the OUTCALLRET events.

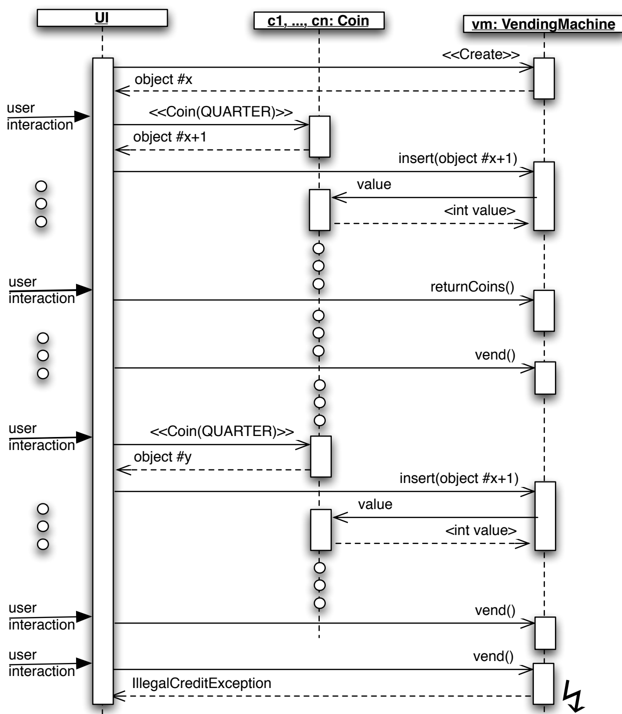

Figure 6: Event flow within the vending machine. After some interactions, the vending machine throws an exception.

actions and how many iterations of the delta debugging algorithm were performed. JINSI minimized the sequence in 11 seconds, performing 388 iterations, on a dedicated Pentium III machine with 1GB of memory running the GNU/Linux Operating System (Kernel version 2.4.26).

These results are promising and motivate further research. In terms of the effectiveness of the technique, the results support an "existence argument": cases exist in which our technique can produce benefits in generating minimal failing executions for debugging. In terms of efficiency, the performance of JINSI is acceptable, if not good, especially considering that it is still an initial prototype completely unoptimized.

5. RELATED WORK

This paper contains two core ideas: recording and replaying component interactions, and isolating relevant method calls. Earlier work has explored each of these ideas in isolation. However, the present paper is the first to combine both.

Test Case Minimization. The work closest to ours in spirit is the minimization approach of Lei and Andrews [4]. Just like us, they apply delta debugging to isolate the set of method calls that is relevant for a failure. As their approach is based on random unit testing, though, they minimize a set of *random method calls* rather than a set of previously recorded interaction, as in JINSI.

Omniscient Debugging. Bil Lewis' ODB debugger records all states of a JAVA program run. It then allows the programmer to interactively explore the recorded states, thus accessing all aspects of a run [5]. As JINSI reduces the recorded run to the relevant calls, the remaining run could easily be observed using ODB-like techniques.

1. INCALL `⟨init⟩()`
2. INCALL `insert(Coin(25))`
3. INCALL `insert(Coin(10))`
4. INCALL `insert(Coin(25))`
5. INCALL `insert(Coin(25))`
6. INCALL `vend()`
7. INCALL `vend()`

Figure 7: Relevant calls. Out of 32 recorded interactions with the `VendingMachine` component, JINSI has isolated seven which suffice to reproduce the failure.

Mock Objects. In [7], Saff and colleagues introduce the idea of using *mock objects*, which automatically reproduce part of the environment, to improve the efficiency of re-testing—JINSI also relies on mock objects to allow for replaying without a complete environment, but for different goals.

Delta Debugging. Besides minimizing a set of method calls, as in [4] and this paper, delta debugging has been used to isolate various failure-inducing circumstances [9]. As discussed above, we find JINSI more general and more versatile than simplifying input; we also find it more light-weight and more robust than isolating state differences.

Selective Capture-Replay. SCARPE [6] is a technique and a tool for selective capture/replay. Given an application, the technique allows for capturing and replaying a part of the application specified by the user. JINSI leverages some of the technology behind SCARPE.

6. POTENTIAL APPLICATIONS

We believe that the combination of capture/replay with delta debugging, all applied on component interactions, has the potential to greatly improve debugging. We describe why the proposed idea might succeed and also why it might fail.

6.1 Why JINSI Could Succeed

In recent years, research in dynamic analysis and debugging has made considerable advances, promising to make programmers more effective when debugging programs. However, being able to *reproduce* a run is still the basic requirement for any debugging method—if we cannot reproduce a run, there is no way to tell whether we have actually fixed the defect.

This is where JINSI becomes valuable. The obvious scenario of use for our approach consists of applying it in-house, when developers are testing their code. In this case, developers could ran their system tests while capturing the interactions of one or more components (e.g., modified or newly created components). When one of the monitored components fails, the approach saves the recorded interactions and starts performing the delta debugging step until the failure-inducing events have been identified. At this point, this set of events would be provided to the developers in the form of a test case. The test case can be used as a minimal test cases that reproduces the component failure and lets developers investigate its causes to eliminate them. For best integration with current test practices, JINSI could even produce a JINSI test, reproducing the failure at will.

However, JINSI can also be applied to *deployed software.* There are different possible instances of this scenario. For example, developers could configure the deployed system so that the interactions of different components are captured at different deployment

sites. In this case, when a component that is being monitored fails, the minimal test case generated by JINSI could be packaged and sent back to the developers for further investigation. This scenario (although it introduces privacy issues that will have to be addressed) is very compelling because it would allow developers for effectively investigating field failures whose resolution could greatly benefit users.

Finally, we want to point out that JINSI is a very general, versatile, and lightweight approach. In particular, it requires no domain knowledge about the structure of the input (as in other delta-debugging approaches) and does not rely on any special operating system or library support for capture/replay (as in other capture/replay approaches). Overall, we believe that JINSI has the potential to become a valuable tool for everyday's debugging tasks.

6.2 Why JINSI Could Fail

There are several ways in which JINSI may not work as expected.

First, although capture/replay approaches using mock objects have been shown to scale to large and complex applications [7] and to suitably capture and replay events [6], it is possible that specific aspects of a component's interactions may not be adequately captured, especially if real-time aspects or thread schedules are relevant for the failure.

Second, our approach can easily minimize all *incoming* events that are controlled by the environment, such as method calls and direct modifications to the observed component's state (i.e., field accesses); JINSI's replay infrastructure can control which of these events to replay and which ones to skip. However, *incoming* return events depend on the occurrence of corresponding *outgoing* calls, which cannot be controlled because they originate in the observed component. Currently, we simply let such *outgoing* events occur and produce the corresponding *incoming* events. We are working on better ways to control and manipulate the values returned by outgoing calls, in order to minimize this information as well.

Finally, choosing an appropriate boundary between observed components and environment may not always be easy; unfortunately, JINSI provides no specific support for this task. To address this issue, we are considering the possibility of applying JINSI in a recursive fashion, by capturing the interactions of lower-level ("inner") components while higher-level ("outer") components are being replayed. In this way, it may be possible to eventually isolate *interaction chains* from the external input down to low-level objects.

7. CONCLUSION AND FUTURE WORK

JINSI provides a combined solution for two nontrivial debugging tasks: (1) reproducing the failure, and (2) understanding what is relevant for the failure to occur. The approach is versatile and has few special requirements; it has been demonstrated to work on a simple, yet non-trivial JAVA application. We expect the approach to scale to larger programs, and to be applicable in a large number of debugging scenarios, including capturing data from deployed programs.

Although we have a working prototype, our work is still in its early stages. Besides addressing the risks as discussed in Section 6.2, our future work will focus on the following topics:

More events. Right now, JINSI is best at minimizing incoming calls and field accesses. We would like to improve JINSI's control on *outgoing* calls and return values, as discussed in Section 6.2. We would also like to apply JINSI to failures involving inter-thread and inter-process communications.

Better diagnoses. We are currently working on extending JINSI to isolate not only the relevant interactions between the observed components and their environment, but also the interactions within the observed components. This extension may lead to the identification of *chains of relevant interactions* that reveal which parts of the program execution were actually relevant for a failure.

Isolating vs. simplifying. Besides simplifying relevant interactions, as described in this paper, delta debugging can also be used to isolate relevant *differences* between a passing and a failing execution. In such a setting, JINSI would tell the relevant differences between two recorded sets of interactions: one where the failure occurred and one where it did not.

More case studies. Finally, we are investing a considerable effort in making JINSI usable for more and larger programs, as well as in generally improving its usability and performance. These improvements shall allow us to conduct large-scale case studies and better evaluate JINSI's usefulness. Eventually, we plan to turn JINSI into a publicly available plug-in for the ECLIPSE programming environment.

Acknowledgments

This work was supported by grant Ze509/2-1 from Deutsche Forschungsgemeinschaft to Saarland University as well as by NSF awards CCR-0205422 and CCR-0306372 to Georgia Tech. Our work was greatly facilitated by recent open-source JAVA programming frameworks, such as BCEL [1], EASYMOCK [2], and JAVASSIST [3]. Valentin Dallmeier, Christian Lindig, and Andrzej Wasylkowski provided valuable comments on earlier revisions of this paper.

8. REFERENCES

[1] Byte-Code Engineering Library (BCEL). http://jakarta.apache.org/bcel/.

[2] EasyMock home page. http://www.easymock.org/.

[3] Javassist home page. http://www.csg.is.titech.ac.jp/~chiba/javassist/.

[4] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *Proc. 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 267–276, Chicago, Illinois, Nov. 2005.

[5] B. Lewis. Debugging backwards in time. In M. Ronsse, editor, *Proc. Fifth Int. Workshop on Automated and Algorithmic Debugging (AADEBUG)*, Sept. 2003.

[6] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proceedings of the Third International ICSE Workshop on Dynamic Analysis (WODA 2005)*, pages 29–35, St. Louis, MO, USA, May 2005.

[7] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic Test Factoring for Java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated Software Engineering (ASE 2005)*, pages 114–123, November 2005.

[8] D. Saff and M. D. Ernst. Automatic mock object creation for test factoring. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, pages 49–51, June 2004.

[9] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 1st edition, 2005.

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