Replaying and Isolating Failing Multi-Object Interactions

Martin Burger
Dept. of Computer Science
Saarland University, Saarbrücken, Germany
mburger@cs.uni-sb.de

Andreas Zeller
Dept. of Computer Science
Saarland University, Saarbrücken, Germany
zeller@cs.uni-sb.de

ABSTRACT

When a program fails, there are typically multiple objects that contribute to the failure. Our JINSI tool automatically captures the failure-causing interaction between objects and isolates a sequence of calls that all are relevant for reproducing the failure. In contrast to existing work, JINSI also isolates relevant interaction *within* the observed component and thus across all layers of a complex application. In a proof of concept, JINSI has successfully isolated the interaction for a failure of the COLUMBA e-mail client, pinpointing the defect: "Out of the 187,532 interactions in the `addressbook` component, two incoming calls suffice to reproduce the failure."

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, developers need to solve two issues: (a) *reproducing* the failure and (b) *fixing* the defect such that the failure no longer occurs. Reproducing is crucial because one can examine the problem and eventually decide whether it has been fixed. Fixing is difficult because the developer needs to search the defect that causes the failure—a search in space and time which becomes the more difficult the larger the program state, and the distance between defect and failure.

As an example of a failure involving both large space and time, consider the following (real-life) problem. The COLUMBA e-mail client[1] offers a feature to import an address book from other systems. As one of us wanted to import his 473 contacts, though, this import failed: An error dialog popped up (Figure 1)—apart from that dialog box, neither an error log file with additional information, nor a stack trace printed on the console is available.

What can we do to find the failure cause? There is no *stack trace* to examine (which otherwise would give us hints of where to look). Stepping through the program in an *interactive debugger* [17] is out

[1] `www.columbamail.org`

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 – Workshop on Dynamic Analysis, July 21, 2008
Copyright 2008 ACM 978-1-60558-054-8/08/07 ...$5.00.

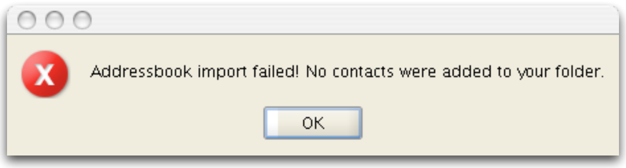
Addressbook import failed! No contacts were added to your folder.

Figure 1: COLUMBA error message. After trying to import an address book, a nondescript error message pops up.

of the question for us non-experts. Backward *slicing* [14] (static or dynamic) falls flat, too, because we would not know where to start. We could *compare code coverage* [5] between a usual run and the failing run, and thus isolate the code that takes care of importing; but then again, where would we look? Finally, we could use *delta debugging* to isolate the failure-inducing input [16]. But delta debugging requires that the program in question be automatically testable, which is not the case here. So we would have to resort to do minimizing the input manually, which not only is a slow and painful process, but also would not point us to the defect.

In earlier work [10], we had presented JINSI, a tool to assist debugging by capturing and minimizing the interaction of a component. As sketched in Figure 2, JINSI wraps around a component and records its interaction with the environment for later replay. By making an educated guess that the `AddressbookImporter` component would be involved, we activated JINSI while importing the address book. JINSI indeed recorded 20 interactions (which includes two incoming method calls) and was then able to replay the failure at will.

The second aim of JINSI, though, is to automatically isolate the failure-inducing interaction. The idea is to apply *delta debugging* on the incoming method calls to isolate a subset of calls in which

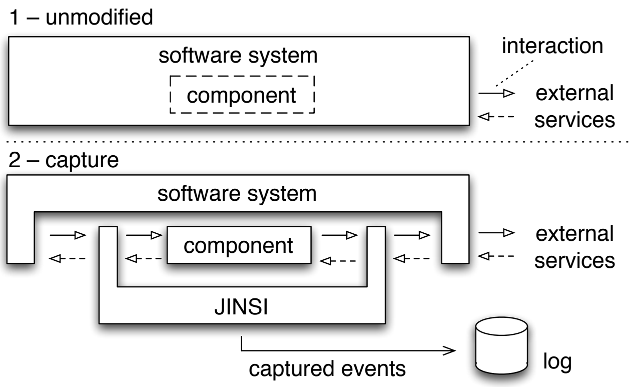

Figure 2: Capturing interaction. JINSI intercepts and records the interaction between a component and its environment.

every call would be relevant to reproduce the failure. Unfortunately, with just two incoming calls, there would not be much to minimize—and no insight to be gained, either, as the last method, called `wizardFinished()`, evidently did all the work and thus was relevant for the failure as well. What we wanted (and needed) was a way to look at what is happening *inside* this component—that is, identifying the objects involved in a complex method and isolating the relevant interaction for these objects just as well. In other words, we wanted to extend JINSI from a single component to multiple objects, isolating relevant interactions at all levels of an application.

In this paper, we report an early success with this approach. These are our contributions:

1. A technique to automatically capture and replay object interaction *within a component* rather than just the component and its environment; and

2. A capture/replay approach based on *observed objects* rather than observed code, allowing us to identify precisely the interaction within a component.

Our early experiences, as presented in this paper, are very promising. In the case of COLUMBA, the approach successfully isolates the two objects involved in the failure. Since it suffices to record interaction at the source (e.g., user input), interactive programs still show an acceptable performance while recording. Last but not least, the approach is fully automatic, and, in the case of COLUMBA, pinpoints the very method that is faulty.

The remainder of this paper is organized as follows. After summarizing our earlier work on JINSI and call minimization (Section 2), we describe how to capture intra-component interaction (Section 3) and how to isolate the relevant inter-object interaction (Section 4). Section 5 demonstrates the results of applying the approach to COLUMBA, effectively solving the initial problem. After discussing related work (Section 6), we close with conclusion and consequences (Section 7).

2. THE ORIGINAL JINSI APPROACH

To make this paper self-contained, let us give a brief overview of the *original* JINSI technique at first, as described in [10] and [11]. JINSI works in two phases:

Capturing phase. The capturing phase uses instrumentation[2] to capture the interactions between a component and the rest of the application. In the context of JINSI, a *component is a set of classes*, defined by a list of class and/or package names.

Isolation phase. The isolation phase replays the recorded interactions and isolates the set of failure-inducing interactions via delta debugging.

Let us give some detail on these phases and finally discuss some open issues of the original JINSI.

2.1 Capturing Interaction

To capture the *interaction between a suspicious component and the remainder of the application* (the *external code*), JINSI implements a capture/replay technique that (a) identifies all interactions between observed and external code, (b) correspondingly instruments the application, and (c) captures interactions at runtime.

[2]We use JAVA bytecode instrumentation to modify existing classes directly in binary form. In this way we can add new methods and other features to the application's code as soon as the JVM loads it. In our case the tool of choice is JAVASSIST.

The instrumentation allows JINSI to capture all interactions in the form of different events: method and constructor calls, return values, field accesses, and exception flow. For each of these, JINSI distinguishes *incoming* and *outgoing* events: Incoming events originate from the external part; outgoing events originate from the component.

The concrete instrumentation depends on the event to be captured. To capture incoming method calls and their return values (including exception flow), for instance, JINSI

- renames the observed method M itself to M';

- adds a logging proxy method P with the same signature as M, which in turn calls M'; and

- replaces all calls from observed code to M by calls to M'.

While capturing, JINSI records two events: one describing the incoming method call itself, including attributes needed for replaying like the given parameters (their types and unique IDs, see below); and one representing the returned value (or the exception thrown).

When capturing data, the type of information ranges from simple scalar values to complex objects. While scalar values can be captured inexpensively in terms of time and space, gathering information about composite objects is much more expensive. In our approach, we (1) only have to capture those objects that directly affect the concrete computation, and (2) incrementally capture the information at runtime as soon as it is required. Besides the events, all we require to replay a program run are unique object IDs, their types, and scalar values. This way, JINSI can dramatically reduce the space and time costs of the capture phase.

2.2 Isolating Relevant Interactions

JINSI uses delta debugging to automatically *isolate the failure-inducing interactions* by systematically testing subsets of the initial sequence—until a set is found where every remaining call is relevant for producing the failure. Applying delta debugging on interactions has several advantages for debugging, in particular, over earlier applications of delta debugging:

- It subsumes delta debugging on input [16] as a special case, since all input-processing methods are part of the interaction to be minimized; at the same time, it needs no knowledge about input structure and does not need any specific scaffolding to manipulate the input.

- Compared to delta debugging on program states [2], it is much more robust, scales to applications with non-accessible state (e.g., native code or distributed applications), and naturally exploits the abstraction layers available in the program.

2.3 Open Issues

The original JINSI technique, as described in [10] and [11], is not without issues, though. In our introductory example, we applied JINSI on the component that runs the import process—the (quite obvious) `AddressbookImporter` class. The original JINSI is indeed able to capture and replay the interactions between this component and the rest of the application. There were 20 of them, including one constructor and two method calls. Unfortunately, the method that triggers the failure, the `wizardFinished()` method, does not contain the actual bug, but in turn uses another component that possibly contains the defect. So, although JINSI was able to minimize the `AddressbookImporter` interaction, we were not able to pinpoint the bug. Obviously, choosing the proper component remains a burden for the developer.

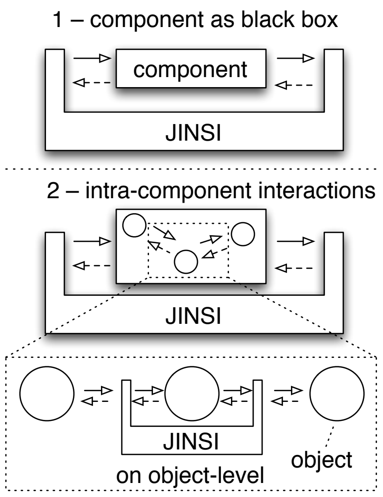

Figure 3: JINSI extended by *intra-component interactions*. We extended JINSI by the ability to capture and replay *interactions between objects within the component* itself. (For the sake of clarity we omitted the event log.)

In order to debug the actual defect, it may be required to repeat the process of defining an estimated boundary and running JINSI over and over again until the proper one is found. One could proceed in an incremental top-down approach, starting with the GUI layer and proceed towards the system, but still, this would a tedious task. What is needed is an approach that not only captures the *inter-component* interactions (between the component and the external code), but the *intra-component* interactions as well: all the interactions *within* this component. And, if we could replay these intra-component interactions at will, we could again use delta debugging to isolate the failure-inducing ones inside of the component of interest.

This alternate approach is shown in Figure 3. By capturing all *interactions between instances of classes declared in the component of interest,* the component no longer is a black box, but what happens inside the component is subject to capture/replay and minimization as well—thus providing a much finer-grained diagnosis.

In the following two sections, we describe our extensions to the JINSI approach—extensions that preserve the advantages of the original approach, while being able to minimize *intra-component* interactions in form of *inter-object* ones.

3. CAPTURING INTRA-COMPONENT INTERACTIONS

To observe *intra-component* behavior in form of interactions between all *observed objects*—instances of classes declared in the component—we had to extend JINSI to enable it to also capture these *intra-component interactions*.

In our previous approach, JINSI only observed *code* and thus captured and isolated interactions caused to happen by an executed piece of JAVA code. As an example, consider the classes shown in Figure 4. The class `AdditionOperator` is observed in the component of interest, whereas its superclass `BinaryOperator` is located in some external part of the application (unobserved). If we run an application that uses these classes, we would only observe all interactions in `AdditionOperator` (i.e., the incoming calls to `operate()`), but interactions with code inherited from `BinaryOperator` would not be captured. In our extended approach, JINSI captures and replays the behavior of *objects*. Thus,

```
package world;
public abstract class BinaryOperator {

    protected long a;
    protected long b;

    public void setA(long a) {
        this.a = a;
    }
    public void setB(long b) {
        this.b = b;
    }
    public abstract long operate();

}

package component.of.interest;
public class AdditionOperator extends BinaryOperator {

    public long operate() {
        return this.a + this.b;
    }

}
```

Figure 4: Gathering information for replaying objects. When observing instances of `AdditionOperator`, we have to instrument methods in the super class `BinaryOperator` as well, otherwise we could not replay the object's original behavior because the captured interaction would not include calls to setter methods.

all interactions of instances of class `AdditionOperator` would be observed, even if its code is inherited.

Furthermore, while observing code, interactions will not be captured if they occur between code within the component. For example, if a method call directly occurs between two fragments of observed code, that call will not be captured, even if two different objects are involved. When observing objects, this type of interactions has to be captured and replayed as well. That results in the following extensions to the original JINSI.

3.1 Capturing Calls

Our extension of the JINSI approach is straightforward: JINSI has to set up a capturing boundary around every object defined in the observed component as well as around the component itself. So, in contrast to observed code, JINSI has to check *at runtime* whether an interaction crosses an object boundary.

To capture these object interactions, we add *runtime checks* for all interactions that *possibly cross these boundaries*; the actual decision is made during the program run. Basically, we instrument all classes declared in the component for possible outgoing and incoming interactions; we instrument external classes for possible incoming interactions; and in the case of super classes of observed ones but defined in external, we instrument for possible outgoing interactions as well, see below. For instance, an outgoing method call is eventually captured if it satisfies these conditions:

1. The interaction must come from an *observed object.*

2. The source object must be different from the target: If it is the same object, the call does not leave the object at all.

Note that at the object level, a single interaction can be both incoming and outgoing if the source and target objects are both observed, because it is an interaction between two different objects within the component of interest.

For the other types of possible interactions, an analogous runtime check is used in order to decide at runtime. JINSI extends the captured and stored events by attributes that identify source and target objects (unique object ID and type); apart from this extension, the set of captured attributes remains unmodified.

3.2 Observing Objects

Besides this additional runtime check extension, JINSI now instruments *super classes* of observed class instances as well, because calls to an object could be targeted at a super class —in contrast to observing code as discussed above. When we observe code, we only instrument the code directly defined in the component of interest; if a super class (of a class declared in the component) is external, we do not instrument that super class for capturing interactions. In the case of observed code that approach is absolutely fine[3]. However, in order to be able to replay object behavior, we have to capture the behavior that is defined in super classes as well, even if declared outside of the component of interest.

Hence, JINSI also instruments super classes for potentially incoming and outgoing interactions. In these super classes, the runtime check accordingly decides whether an interaction is incoming, outgoing, or both: (1) If the concrete object is an instance of a class declared in the component of interest, the interaction can be incoming as well as outgoing; (2) if it is an instance of a class declared outside of the component, the interaction can be incoming only.

3.3 Performance

All these additional run-time checks are expensive. Therefore, we implemented a *staged* approach: Instead of capturing all intra-component interactions directly, we add an intermediate phase. As we can replace the external code with JINSI replaying the component with the recorded executable events, we can use these events to replay the run and to capture interactions of all objects within the component of interest at the same time in a separate phase (see 1 and 2 in Figure 5). This way, we can apply the leaner capture of observed code, which has a small overhead, in the field; the finer-grained but slower capture of object behavior can be done offline within JINSI.

4. ISOLATING INTRA-COMPONENT INTERACTIONS

After capturing the intra-component interaction, we can now replay it at will—and we can replay *subsets* to determine which subsets are relevant. This is the main idea of delta debugging. However, due to the large number of calls involved, we found it useful to first reduce the set by applying a slicing technique similar to dynamic slicing.

4.1 Object Slice

Before applying delta debugging, we compute an *object slice* which eliminates all objects and their corresponding events that are not related to the failure [7]: We start with the objects that are most likely related to the failure—if an exception is thrown from a specific method call, for instance, we would choose its target as a starting point. Now, we include all events where these initial objects are involved: either as source or target of an interaction (e.g. other incoming calls). All these events are added to the list of possible relevant events, in addition to the initial ones.

This way, we obtain a list of possible relevant interactions that describes the constructions and usages of all objects that are involved in the interactions that reproduce the failure. Because we iterate only once over the initial event log, the complexity of the slicing algorithm is linear.

[3]If we are interested in interactions of that code as well, we include the super class in the observed code.

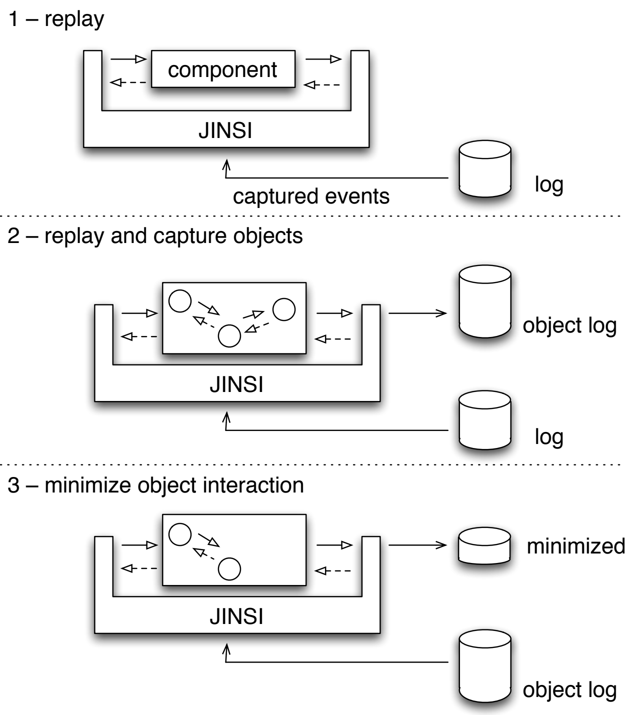

Figure 5: Isolating intra-component interactions. The captured interaction can be (1) replayed to reproduce the failure, (2) replayed to capture detailed interaction on object level, and (3) isolated to create a minimal test case.

One drawback of this approach is that the developer has to manually determine the initial objects of interest. However, for a given event log with recorded object interactions, JINSI can suggest an initial set of objects. Currently, this automated suggestion is based on exceptions only, so that it does not work for non-crashing bugs where no exceptions are thrown. In that case, the developer has to choose the objects on his own, or she could apply delta debugging with the help of JINSI on the whole set of interactions directly.

In our introductory example, COLUMBA shows no thrown exception (for example, printed on the console). Actually, an exception is thrown within the faulty component as we will see in Section 5. Indeed, the import wizard catches all exceptions that are thrown while importing and instead displays a generic error message (see Figure 1). Thus, the GUI absorbes all exceptions and its stack traces, so the actual hint for the location of the defect. Because JINSI captures the exception flow while observing objects, it is able to suggest a suspicious object as described above.

4.2 Delta Debugging

After reducing the number of events using object slicing, JINSI uses delta debugging to isolate the failure-inducing interactions within the component of interest.

Because the worst-time complexity of the algorithm is quadratic, the algorithm may take a long time in order to minimize a large set of interactions. In a complex object oriented application like COLUMBA or in a long-running server program, you would expect many of these interactions; however, as in [7], the earlier slicing phase takes care of reducing the sequence length (and thus the execution time for delta debugging.)

5. PROOF OF CONCEPT: COLUMBA

Let us now demonstrate JINSI on a real bug in a real program—

the failing address book import in COLUMBA described in the introduction. COLUMBA is a free software e-mail client written in JAVA, developed since June 2001 (first release) until August 2007. Since COLUMBA is a large (about 1,700 JAVA classes) and complex software system (it includes support for different e-mail protocols, cryptographic protocols, spam filter, address book, and more), it is an appropriate subject for an initial proof of concept.

In order to trigger the failure shown above, we imported an address book file with 100 records in an existing COLUMBA address book that already contains 300 entries. All addresses were generated (random name, etc.); however, we regard the type and the amount of data as realistic.

We defined `org.columba.addressbook.model` as component of interest because it implements the address book's data model. That package contains 19 JAVA classes. Given that boundary and the original application to JINSI, we triggered the failure manually while JINSI captured the whole interaction between the component and the external code. After this run, the event log file contained 166,130 *inter-component* interactions, thereof 49,855 incoming constructor and method calls.

In order to obtain the *intra-component* interactions, we had JINSI replay the component interactions while recording the object interactions within the component. In this second step, JINSI captured 187,532 interactions, containing 62,558 incoming calls. Compared to the first step, the amount of calls increased by about 25%. That is not as much as expected, however the actual amount of interactions highly depends on the concrete component and its structure. In our example, the observed component directly interacts with the rest of COLUMBA, so the number of intra-component calls that do not leave the component is about 13,000—compared to about 50,000 calls captured in the first step. Nevertheless, we successfully captured the interactions on object level within the component.

Within the previously captured object interactions, JINSI exactly classified one object as suspicious: An incoming method call to that object caused an exception being thrown. After slicing for that object, the event log only contained 32 interactions, including one constructor call and seven method calls. Thus, JINSI is able to reproduce the original failure by replaying 32 out of 187,532 interactions; as in [7], we found that slicing is a very effective alternative to delta debugging when it comes to minimizing call sequences.

To minimize the remaining 32 interactions, we used delta debugging. Eventually, the simplified interactions contained only 20 events, including one incoming constructor and one incoming method call. This means that constructing the object and calling that method causes the original failure. Figure 6 depicts the failure-inducing interactions. First, the empty constructor is called in order to construct an instance of class `ContactModel`; within that constructor four empty vectors are instantiated and an universally unique identifier (UUID) is obtained. Second, the method `getPreferredEmail()` is called.

With the simplified failure-inducing interactions at hand, JINSI could create an executable minimal unit test case as shown in Figure 7. Running these two JAVA-statements, the developer would be able to reproduce the original failure easily—and immediately focus on `getPreferredEmail()` as obvious failure cause.

Figure 8 shows the original code of class `ContactModel`, including the suspicious `getPreferredEmail()` method. The error is easy to spot. If a `ContactModel` is created, the attribute `emailAddressVector` contains an empty vector of e-mail addresses. Accessing the e-mail via `getPreferredEmail()` returns the first element of the contact. If there is none, the code is supposed to return `null`, as indicated by the "backwards compatibility" comments. Unfortunately, this code is never reached. If the contact has no e-mail address, the iterator `it.next()` will not return `null`, as obviously assumed in the code. Instead, it will throw a `NoSuchElementException` exception, thus causing the import to fail and resulting in the error message in Figure 1. The "backwards compatibility" code is in fact unreachable.

```
 1. INCALL ContactModel()
 2.   OUTCALL Vector()
 3.   OUTCALLRET Vector #13828
 4.   OUTCALL Vector()
 5.   OUTCALLRET Vector #13829
 6.   OUTCALL Vector()
 7.   OUTCALLRET Vector #13830
 8.   OUTCALL Vector()
 9.   OUTCALLRET Vector #13831
10.   OUTCALL UUIDGenerator()
11.   OUTCALLRET UUIDGenerator #13832
12.   OUTCALL #13832.newUUID()
13.   OUTCALLRET String #13833
14. INCALLRET ContactModel #13827
15. INCALL #13827.getPreferredEmail()
16.   OUTCALL #13828.iterator()
17.   OUTCALLRET Iterator #13837
18.   OUTCALL #13837.next()
19.   OUTCALLTHROW NoSuchElementException
20. INCALLTHROW NoSuchElementException
```

Figure 6: Simplified interaction. Out of 187.532 recorded interactions within the `org.columba.addressbook.model` component, JINSI has isolated 20 which suffice to reproduce the failure. `#n` means object with id `n`

```
public void testContactModel() throws Exception {
    ContactModel model = new ContactModel();
    model.getPreferredEmail();
}
```

Figure 7: Minimal test case. The `getPreferredEmail()` method has a defect, causing an exception being thrown.

To trigger this failure, one needs to create a contact without e-mail address. This is what happens during the import: If any contact has no e-mail address, the import fails. We could thus filter out contacts without e-mail addresses before importing them—or fix the COLUMBA code such that `getPreferredEmail()` returns `null` if there is no e-mail address. And this is how we eventually were able to use COLUMBA after all.

To measure performance, we did some early time measurements: We stopped the time from starting the import process by clicking on "Import" until the error message (Figure 1) pops up. We measured the JINSI overhead in four scenarios:

Without capturing. This means running the original COLUMBA.

Code, but no written events. This means to observe code (JINSI instruments COLUMBA and intercepts interactions during its run); however, the recorded events are not written to the event log on disk.

Code. Here, JINSI observes and captures inter-component interactions, and writes the event log file on-the-fly. Until the event is written to disk, JINSI blocks the application under observation. Currently, JINSI uses a small buffer, but implementing the producer-consumer pattern (events are given to a long

```
public class ContactModel implements IContactModel {

    private Vector emailAddressVector = new Vector();
    private Vector phoneVector = new Vector();
    private Vector instantMessagingVector = new Vector();
    private Vector addressVector = new Vector();

    public ContactModel() {
        this.id = new UUIDGenerator().newUUID();
    }
    public Iterator getEmailIterator() {
        return emailAddressVector.iterator();
    }
    public String getPreferredEmail() {
        Iterator it = getEmailIterator();
        // get first item
        IEmailModel model = (IEmailModel) it.next();
        // backwards compatiblity -> its not possible
        // anymore to create a model without email
        if (model == null) return null;
        return model.getAddress();
    } ...
}
```

Figure 8: The actual defect. `Iterator.next()` never returns `null`. Instead, a `NoSuchElementException` will be thrown if the iteration has no more elements.

Importing the faulty address book	Runtime [sec]
Without capturing (no instrumentation)	$\ll 1.0$
Capturing code but no written events	$\simeq 1.0$
Capturing code and writing to hard disk	$\simeq 5.0$
Capturing objects and writing to hard disk	$\simeq 7.0$

Table 1: Performance of capturing. At the moment, writing captured events to disk is still too expensive. When capturing is enabled but no events are written to disk, the overhead is acceptable.

queue and picked up in a separate Thread) should be very helpful.

Objects. This is the same as "code", but JINSI observes objects within the component. So, all runtime checks are executed.

The plain capture takes about 1 second, compared to the original COLUMBA where the addresses are imported instantly (stopped time is much smaller than 1 second). Hence, the overhead caused by JINSI to intercept all interactions is acceptable. However, our current implementation causes some overhead when writing the captured events to disk.

6. RELATED WORK

This paper contains two core ideas: recording and replaying on both component- and object-level, and isolating relevant method calls by applying object slicing and delta debugging. Earlier work has explored each of these ideas in isolation. However, the present paper is the first to combine all of them.

Efficient Test Case Minimization. The work closest to ours is the minimization approach of Leitner et al. [7]. They apply a static slicing technique and delta debugging to efficiently isolate the set of method calls that is relevant for a failure. The two main difference to our work are: (1) As their approach is based on random unit testing, they minimize a set of *random method calls* rather than a set of previously recorded interaction, as in JINSI; (2) they directly minimize the code of a random test case by applying *static slicing minimization*, a method based on static program slicing, and delta debugging, whereas JINSI applies both slicing and delta debugging on recorded object interactions. Based on these minimal interactions JINSI could create a minimal test case as illustrated in Figure 7. Leitner et al. again improve the minimization approach of Lei and Andrews [6].

ReCrashJ. Artzi et al. presented RECRASHJ [1] that reproduces crashes from JAVA programs. In contrast to our work, it does not use a log of events, but an in-memory record of stack elements created by checkpoints at each method entry. Finally, RECRASHJ generates a test for each stack frame. Both approaches can generate test cases that reproduce a failure, whereas JINSI can additionally reproduce the relevant steps in the execution trace that lead to that failure.

Cause-Effect Chains in Computer Systems. In [9], Neuhaus et al. presented MALFOR, a system that isolates the *processes that cause an intrusion*. MALFOR records interactions at system level and after the intrusion has been detected, the recorded interactions are replayed to isolate the relevant processes. In contrast to JINSI, MALFOR isolates the interplay of multiple programs.

Mock Objects. In [13], 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; JINSI uses mock objects to pass proper objects when replaying incoming calls, however, the actual outgoing calls are redirected to JINSI and not to the passed mock.

Omniscient Debugging. Bil Lewis' ODB debugger records an entire JAVA program run. It then allows the programmer to interactively explore the recorded states, thus accessing all aspects of a run [8]. As JINSI reduces the recorded run to the relevant calls, the remaining run could easily be observed using ODB-like techniques.

Delta Debugging. Besides minimizing method calls, as in [6] and this paper, delta debugging has been used to isolate various failure-inducing circumstances [15]. 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 [11] 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, especially when capturing and isolating inter-component interactions.

7. CONCLUSION AND FUTURE WORK

Capturing and replaying interaction is not only a helpful tool for reproducing failures; it can also be helpful for isolating those events that were relevant for the failure. By combining capture/replay with isolation of relevant events, JINSI kills two birds with one stone. It is easy to deploy even in production code, since initially, only incoming events for the top-level components need to be recorded. At the same time, JINSI can provide relevant diagnoses, because computationally expensive techniques such as delta debugging can be offset to the offline replay of recorded interactions. Finally, focusing on method calls allows us to exploit abstractions as provided by the programmer—both in the search for relevant events as in the presentation of the final diagnosis.

All of this has to be taken with a grain of salt, as we still need to substantiate these claims. Yet, we are confident to gain more

```
 1. INCALL ContactModel()
 2.   OUTCALL Vector()
 3.   OUTCALLRET Vector #13828
 4. INCALLRET ContactModel #13827
 5. INCALL #13827.getPreferredEmail()
 6.   OUTCALL #13828.iterator()
 7.   OUTCALLRET Iterator #13837
 8.   OUTCALL #13837.next()
 9.   OUTCALLTHROW NoSuchElementException
10. INCALLTHROW NoSuchElementException
```

Figure 9: Minimal interaction. After eliminating irrelevant objects, only ten interactions and two incoming calls remain.

experience in the upcoming months. This is what our future work will focus upon:

Scalability and stability. Right now, JINSI can be applied to medium-sized programs like COLUMBA. To make it work on large JAVA programs like ECLIPSE, we need to fix some issues with the instrumentation that fails on some complex classes. One of these issues is caused by JAVASSIST, the class library used for the actual byte code manipulation. However, we are in contact with its developer in order to fix that bug[4].

Improving isolation performance. In addition to object slicing, we want to use additional analyses in order to reduce the search space. In particular, we want to include a side-effect analysis [12] to eliminate all calls to pure methods; thus, methods that do not change the object's state (except for method calls that cause the failure, e.g. by throwing an exception). A capable tool implementing a *dynamic side-effect analysis* may be JDynpur[5].

Minimizing outgoing interactions. Currently, we simply let *outgoing* events occur since *incoming* return events require corresponding outgoing interactions. However, the simplified interaction as shown in Figure 6 could be minimized further by removing objects that are not connected to the actual failure. In that way, JINSI could classify more interactions as irrelevant; the possible outcome is illustrated in Figure 9.

Tracking dependencies. When minimizing interactions, JINSI does not leverage any hierarchy or other structure in the involved objects. We are experimenting with various *strategies* on where to start minimization. One promising approach is inspired by dynamic backward slicing: Starting with the failing method, we apply JINSI to minimize the interaction with its arguments (and their arguments recursively). Such a strategy would effectively mimic extracting the dynamic backward slice, except that many more irrelevant calls and objects could by eliminated.

Synthesizing fixes. One important issue in all automated debugging approaches is the *missing statement problem:* One can only isolate and focus on what is there, but not suggest what *should* be there. We are currently experimenting with *generated method calls* in the style of AutoTest [7] to learn where and how missing statements can be generated.

[4]`http://jira.jboss.com/jira/browse/JASSIST-43`

[5]`http://www.st.cs.uni-sb.de/models/jdynpur/`

Evaluation. Once we can apply JINSI to a large body of programs, we want to evaluate its efficiency and effectiveness as applied to real-life bugs. In particular, we want to apply JINSI to the *iBugs* repository [3] and compare its effectiveness and efficiency to other automated debugging approaches.

Information about JINSI is available at

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

Acknowledgments

This work was supported by grant Ze509/2-1 from Deutsche Forschungsgemeinschaft to Saarland University. Our work was greatly facilitated by recent open-source JAVA programming frameworks, such as JAVASSIST [4]. Valentin Dallmeier and David Schuler provided valuable comments on earlier revisions of this paper.

8. REFERENCES

[1] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP*, 2008. To appear.

[2] H. Cleve and A. Zeller. Locating causes of program failures. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 342–351, New York, NY, USA, 2005. ACM.

[3] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proc. of the 22nd IEEE/ACM Intl. Conference on Automated Software Engineering*, November 2007.

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

[5] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proc. of the 24th Intl. Conference on Software Engineering*, pages 467–477, New York, NY, USA, 2002. ACM.

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

[7] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 417–420, New York, NY, USA, 2007. ACM.

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

[9] S. Neuhaus and A. Zeller. Isolating cause-effect chains in computer systems. In *Software Engineering*, volume 105 of *LNI*, pages 169–180. GI, 2007.

[10] A. Orso, S. Joshi, M. Burger, and A. Zeller. Isolating relevant component interactions with JINSI. In *WODA '06: Proc. of the 2006 Intl. Workshop on Dynamic Analysis*, pages 3–10, New York, NY, USA, 2006. ACM.

[11] A. Orso and B. Kennedy. Selective Capture and Replay of Program Executions. In *Proc. of the Third Intl. ICSE Workshop on Dynamic Analysis (WODA 2005)*, St. Louis, MO, USA, May 2005.

[12] A. Rountev. Precise identification of side-effect-free methods in Java. In *IEEE International Conference on Software Maintenance*, pages 82–91, 2004.

[13] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic Test Factoring for Java. In *Proc. of the 20th IEEE/ACM Intl. Conference on Automated Software Engineering (ASE 2005)*, Nov 2005.

[14] F. Tip. *A Survey of Program Slicing Techniques*. Centrum voor Wiskunde en Informatica, 1994.

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

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

[17] A. Zeller and D. Lütkehaus. DDD—a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, 1996.