Implementing a Backward-In-Time Debugger

Masterarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Christoph Hofer

Juli 2006

Leiter der Arbeit:

Prof. Dr. Stéphane Ducasse
Prof. Dr. Oscar Nierstrasz
Dipl.-Inform. Marcus Denker

Institut für Informatik und angewandte Mathematik

The address of the author:

Christoph Hofer
Eichmatt 38
CH-3326 Krauchthal
hawkie@bluewin.ch
http://www.hawkie.ch

Abstract

In both development and maintenance of software, finding and fixing bugs take a huge percentage of the overall time and resources. Traditional debugging and stepping execution trace are well-accepted techniques to understand deep internals about a program. However in many cases navigating the stack trace is not enough to find bugs, since the cause of a bug is often not in the stack trace anymore and old state is lost, so out of reach from the debugger. Therefore there is a challenge in providing new ways of debugging.

In this work, we present the design and implementation of a backward-in-time debugger for a dynamic language, *i.e.,* a debugger that allows one to navigate back the history of the application. We present the design and implementation of a backward-in-time debugger called Unstuck and show our solution to key implementation challenges.

Acknowledgements

First of all I thank Marcus Denker for all his work and what he did for me. He always supported me and I really enjoyed his collaboration. Thanks for all the discussions we had, for ByteSurgeon, for the unvalued work for Squeak itself. During my work, it was the most important thing that I could count on you!

I thank Prof. Dr. Stéphane Ducasse for his reviews and his feedback. I thank Prof. Dr. Oscar Nierstrasz, the head of the SCG, for giving me the possibility to work in his group.

I always needed the sport as a balancing for my work. Therefore I thank Markus Kobel and the rest of the table tennis club. The same applies for Deborah Hofer for attending me to the university training. You are all my friends, and it is not only the sport, it is also the time after ;-) Thanks go to Christof Lüthi and others for all the hours playing beach volley (or billiard) with me, to Fabian Steiner for playing Squash.

Same important are my other friends: Brige, Linders, WAM, Säm, Möchu and all the others i forgot. Thank you for all plays, all discussions, all the fun and the food.

Then I thank my family. I had not enough time for you and I feel sorry. Especially for you, my dear godchild Joel, because you are to young to understand. I promise we will catch up on everything right after this work!

Last but not least I would like to thank Steffi Gerber, my "secretary", for her (administrative) work for my studies ;-)

Christoph Hofer
July 2006

Contents

Chapter 1

Introduction

Computer programs do not work as they should, which makes software development costly. Already in the early sixties programs to help debugging existed to reduce the time consuming task of finding bugs. A first version of a debugger developed in 1961 called DDT (Dynamic Debugging Technique [Edwa 63]) offered analog features than today's debuggers: insertion and deletion of breakpoints, inspecting and changing registers. Software systems evolved and got more and more complex, but debuggers did not improve much. Computers should help to simplify analyzing complex programs because they have more memory and more power today.

Debuggers offer the ability to stop a program at a chosen place, either due to an error or an explicit request (breakpoint). They provide the current states of the involved objects together with a stack trace. However, while stepping through the code is a powerful technique to get a deep understanding of a certain functionality [Deme 02], in many cases this information is not enough to find bugs. The programmer is often forced to build new hypotheses about the possible cause of the bugs, set new breakpoints and restart the program to find the source of the problem. Often several iterations are necessary and it may be difficult to recreate the exact same context [Lenc 99].

The questions a programmer has are often: *"where was this variable set?"*, *"why is this object nil?"* or *"what was the previous state of that object?"*. A static debugger cannot answer these questions, since it has only access to the current execution stack. There is no possibility to backtrack the state of an object or to find out why especially this object was passed to a method. The *Omniscient Debugger* [Duca 99a, Lewi 03a] is a first attempt to answer these problems, however it is limited to java and instrumentation is done at bytecode load time.

1.1 Key problems

When stopping a program's execution, a lot of information is not available to the programmer. The current stack holds methods that have not yet been finished executing, the other ones are not available anymore. This is a loss of information, which could help in the process of finding bugs. Another problem is that only the current state of objects is accessible but the old state is lost. The state history of objects is important, because faulty values are the source of errors.

1.2 Structure of this Document

This document contains the following chapters:

- In Chapter 2 we describe the current state of debugging tools and other approaches to enhance debugging support.
- In Chapter 3 we present our approach, trace-based debugging. We show an user interface for presenting the execution trace together with additional user interface features.
- Chapter 5 presents Unstuck, our reference implementation. We show what is needed to provide the state of objects at any time and other implementation challenges.
- In Chapter 4 we show examples of finding bugs with Unstuck.
- Chapter 6 contains a discussion about performance and memory usage.
- Chapter 7 concludes and looks at possible future work.
- Appendix A contains installation instructions for Unstuck.

Chapter 2

Problems with Current Debugging Approach

Computer programs often do not work as they should. Finding and fixing bugs is a costly part of software development, therefore debuggers were developped to help with bug fixing. Debuggers help to assert certain behaviour in a program's execution. This chapters describes the problems and the current state of debugging tools.

2.1 Why Stack Trace is Not Enough

After an error occured a normal debugger shows the current stack. The problem is that only methods which have not yet been executed are on the stack, those that have finished execution are no longer available.

2.1.1 Introducing Example

The following example demonstrates the problem: suppose there is a class Foo with two instance variables var1 and var2 and the following methods:

```
Foo≫start
    self beforeBar.
    self bar.
    self moreBar.

Foo≫initialize
    var1 := 0.
    var2 := ''.

Foo≫beforeBar
```

```
    var1 = 0
        ifTrue: [var2 := nil.].

Foo≫bar
    | tmp |
    tmp := 0.
    (var1 to: 10) do: [:each | tmp := tmp + each ].
    self var1: tmp.

Foo≫moreBar
    var2 size > 0
        ifTrue: [ ^var2 at: 1].
    ^''
```

Accessor methods are defined for var1 and var2. Foo new start starts the program execution. The debugger comes up because of an error, var2 is nil in method moreBar (see Figure 2.1).

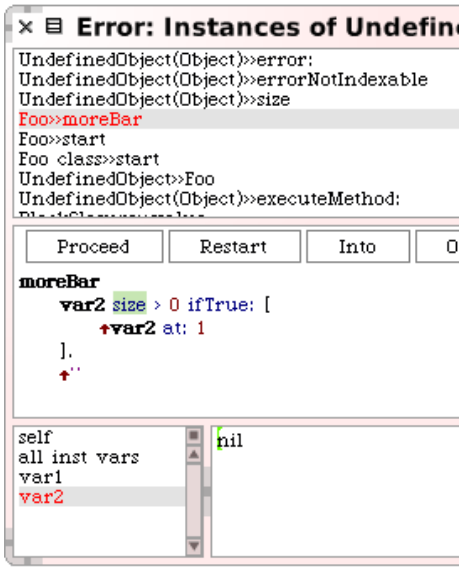

Figure 2.1: Error in the normal Squeak debugger.

In a normal debugger, we see a stack trace: only methods on the stack are shown, those methods which have been completely executed are not available anymore. Figure 2.2 shows a complete execution trace of all methods executed. Only a small part of that (visualized with the dashed box) are part of the stack trace the debugger can show.

When inspecting objects, only the current state is accessible but the old state is lost. Even when selecting a method that is not on top of the stack,

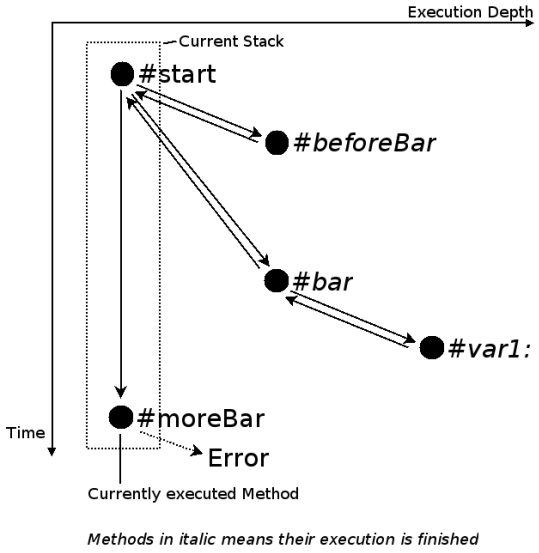

Figure 2.2: Method calls and the resulting stack trace: only the methods in the dashed box are in the stack trace when an error occurs in method moreBar.

the debugger does not revert but presents the same state as before. Assume the situation in Figure 2.2: if we select moreBar: or bar:, the debugger presents the state of the program when the error occurred, even if the state has been different at the execution of these methods.

Recapitulating there are mainly two issues: loss of execution trace and loss of objects old state.

2.2 Recording and Navigating the Complete Trace

Our solution to the problem presented before is to offer a debugger based on traces and a specific interface to navigate backward in time the complete state. During the execution of a program we record runtime data, *i.e.,* method calls and state changes of objects. Therefore we can provide the state an object had in the trace at any time and we know where variables changed their values.

To understand the challenges faced by building a backward in time debugger, *i.e.,* a debugger that allows one to query the state history of a program, we developed a backward in time debugger in Squeak called UNSTUCK. For

its implementation we collect rich information about the program execution in terms of events, which are used to recreate the state of objects at particular points in time.

The contributions of this work are:

- A model for a back-in-time debugger.
- An user interface to present and query the massive amount of data generated by the recording of all the objects states.
- An implementation for Squeak Smalltalk

2.3 A Standard Debugger

Most of the debugging tools today have a common subset of functions and views:

Breakpoints and Watchpoints. A breakpoint is an intentional stop of a program's execution, normally when the execution arrives at a certain point in the source code (or when a specific event occurs in the program). A condition can be added to the breakpoint to only stop when the condition is fulfilled. A watchpoint is basically a breakpoint, which stops the execution when a given expression (usually a variable) changes its value.

Call Stack. This stack is also known as execution stack or function stack. It holds the already called subroutines which have not yet returned (*i.e.,* the execution of the subroutine is not finished). The last called subroutine is on top of the stack and usually holds the breakpoint.

Source Code. A programmer orients himself on the source code, which can be changed or used to modify breakpoints or to use the stepping functions.

Inspect Expressions. This is the main purpose of the debugging functions. Once a breakpoint stops the programs execution, the programmer can evaluate expressions and watch its result, which is used to examine the current state of the program. The expression can be a single variable. With this feature a programmer checks if the program behaves as expected.

Change Values. The programmer can change a variable's value to cause and check certain behaviour.

Stepping functions. Usually there are three of them: step one instruction forward (single step), step to the end of a subroutine and step over (like single stepping, but any subroutine call is treated as one instruction, *i.e.,* the execution remains in the same routine).

2.4 The Squeak Debugger

Squeak is the open-source Smalltalk distribution [Inga 97]. Unstuck is implemented in Squeak, therefore we have a deeper look at the current Squeak debugger.

The Squeak debugger is a typical example for a general debugger. It offers nearly the same functionality described in Section 2.3. There are some differences:

No Real Breakpoints. Squeak does not support real breakpoints. It simulates them by offering the halt method, which is implemented in class Object. This method simply raises a specific resumeable exception with the result that the program is stopped and the debugger pops up. There are helper methods to support conditional halts and more, but these simulated breakpoints are always part of the source code, which means that we have to change the source code for putting a breakpoint. Another disadvantage is that we can not simply remove all breakpoints.

No Watchpoints. The Squeak debugger does not support watchpoints at all. A possible solution is to strictly use accessors for changing instance variables. Therefore a breakpoint in the setter method would be like a watchpoint.

Restart Methods. The debugger allows one to restart the program's execution from any method call. The problem is that the current state of the objects is not changed to the appropriate state they had.

No explicit Debug Mode. Smalltalk is an interpreted language, thus there is no need for a debug mode. The Squeak debugger support the step by step bytecode execution. If an error occurs or a breakpoint is reached, the debugger pops up automatically.

2.5 The Omniscient Debugger

Lewis in [Lewi 03b] describes the idea of Omniscient Debugging and its Java implementation (the "ODB"). The ODB works by collecting events at every

state change and every method call in a program. The debugger provides the state of the program at any desired time. The programmer can select any variable, see where it was set and what values were assigned. The ODB integrates an event analysis engine for searching.

Lewis and Ducassé in [Lewi 03a, Lewi 03b] propose to merge the approach of omniscient debuggers which collect all the run-time information and supports the exploration of the history and event-based tools that monitors program execution and allow queries. It is limited to java and instrumentation is done at bytecode load time.

2.6 Other Approaches

Whyline [Ko 04] implements *Interrogative Debugging* for Alice [1], a 3D world. Here the focus lies on providing an interface to ask questions such as *why* or *why not* things are happening in an Alice world. Thus this debugging facilities are totally tied to the, quite simple, domain model of Alice. Such an approach does not scale when the domain is more complex as in normal development.

Visualising debuggers can work directly via instrumentation on the program being executed, or are based on post-mortem traces [Cons 93, Lang 95]. Visualisation of dynamic information is also related to our work in the sense that it is based on a program trace. DePauw et al. [De P 98] and Walker et al. [Walk 98] use program events traces to visualise program execution patterns and event-based object relationships such as method invocations and object creation.

Query-based debugging [Lenc 97, Lenc 99] use logic programming to express complex queries over a large number of object. Some queries are triggered at run-time while the program is running. The logic queries act as clever program probes. Here the intention is different, in our approach we navigate the history of the program.

Caffeine [Guéh 02] is a Java-based tool that uses the Java debugging API to capture execution events and uses a Prolog variant to express and execute queries on a dynamic trace. Caffeine does not support state history access. TestLog [Duca 06] which uses a logic engine to query the trace of object-oriented applications, is much closer to Unstuck since it offers the possibility to query the previous state of objects. However, *no* user interface

[1] http://www.alice.org

is provided.

OPIUM [Duca 99a] is a tool that allows a user to debug Prolog program using a set of debugging queries on event traces. Prolog is used as a base language and as meta language to reason about events. The main usage scenario of OPIUM is the implementation of a high level debugger for Prolog that allows forward navigation to the next event that satisfies a certain condition. Coca [Duca 99b] supports the debugging of C programs based on events. Opium and Coca are mainly used to show the values of variables. In addition, both Opium and Coca do not support object-oriented programming. In addition, the history of object state is not available.

Auguston [Augu 98, Augu 95] also uses a trace composed of event models and test programs. However it is based on procedural programming languages and does not take into account the specific behavioural aspects of object-oriented languages such as object creation and the state of objects.

PQL (for Program Query Language) allows programmers to express application-specific questions about event patterns at runtime and gives the possibility to act when a match is found [Mart 05]. It's implementation is for Java and uses static and dynamic techniques. The queries can only analyze the current state of objects when a specific query is checked. PTQL [Gold 05] is a Program Trace Query Language to answer declarative queries about program behaviour, and PARTIQLE is their compiler to instrument a Java program by a given query. PTQL is similar to PQL, but more in spirit to SQL.

Chapter 3

Trace-based Debugging

One solution to provide more advanced debugging support is to collect and retain much more information about the execution of a program. For this purpose we collect events representing runtime data. For each method we record the name, the receiver, the arguments and the return value (see Chapter 5). This information is completed by collecting every write access to a variable (instance and local). This means that we record every state change. The collected events are basically a data structure containing the specific runtime information. There is one event for each method executed, its returned value and for every write access to a variable. This data can answer many of the questions a programmer has during debugging, but simple navigation through this mass of data is needed.

We make a trace out of a set of given classes which are interesting for the user to debug. We instrument transparently the methods of these classes to produce the needed data at runtime. Further we will refer to objects from these classes as instrumented objects.

3.1 The Trace and Event Model

A trace is composed of events, depending on the situation the events are holding different information depending of the kind of events they represent and also whether the method execution terminated or not:

- An event representing a message sent describes the selector, the receiver, the arguments, as well as the definition class of the method, *i.e.,* the class which defines the method and holds the source code.
- When the method returns, a return event is generated with the returned value.

- When the value of a variable (instance or temporary variable) is changed, a write access event is generated holding the variable name and the new value.

Additionally every event holds a source range, the mapping between the bytecode and the sourcecode. The depth of an event (which corresponds to the number of unfinished methods) and the timestamp are added when the collector collects the events.

To support object identity checks, each event has a pointer to the original objects (for example to the receiver, the arguments) and in addition for non instrumented objects to their copy.

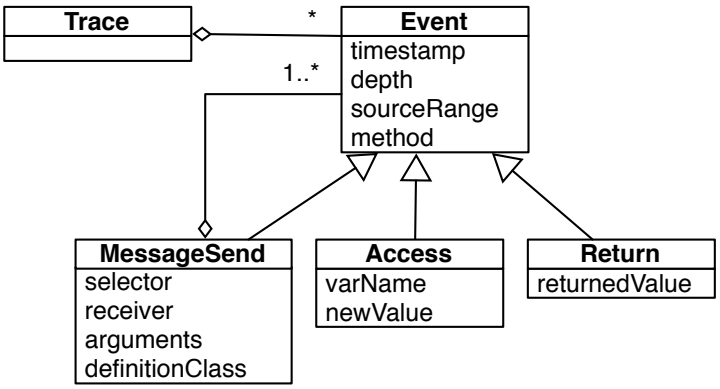

Figure 3.1: The Trace and Event model.

Figure 3.1 shows an UML diagram of the model: a trace consists of several Events. There are specific events, which are containing different information as mentioned before. The Event class shows which information is common to all events. For optimizing the model, a tree is built: an Event belongs to one MessageSend, thus a MessageSend can have multiple Events.

The execution trace holds a huge amount of data, thus we need a methodology for interacting with the debugger, which is described in the following sections.

3.2 User Interface for Navigating the Execution

The user interface of the UNSTUCK Debugger consists of several views on the collected data from the TraceLibrary, enhanced with search and navigation functions. In the following we describe these views. They are identified in Figure 3.2 with numbered black boxes. The corresponding number is specified in square brackets.

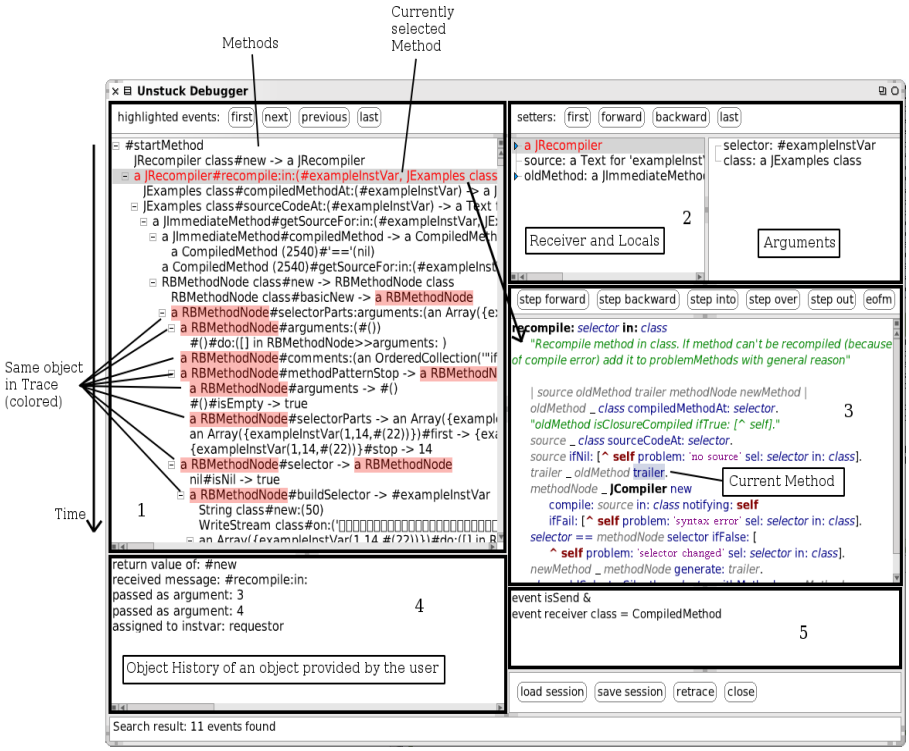

Figure 3.2: The user interface of the UNSTUCK Debugger.

Method trace [1]. Each line represents a method call. The format is of the form

receiver# selector(arg1, arg2, ...) -> return value.

Each line is indented according to the depth of the message sends. Methods can be collapsed if they are of no interest (and of course expanded as well). Methods can be highlighted for remembering them easily. We can step through the highlighted lines. For the receiver and the return value of the selected method the object history can be viewed in the object history [4] using the context menu. This view

selects always the current method in the trace. It can change due to interaction with another view, too.

Object views [2]. There are two views displaying objects according to the currently selected method in the method trace: one displays the receiver of the method and the temporary variables (on the left side: the first line represents the receiver, beneath the temporary variables with the variable name), the other one the passed arguments (on the right side, with the argument's name). If an object is instrumented, it can be expanded to display the instance variables. Thus each line represents an object: these lines can be inspected or used for the object history over the context menu. If an object is an instance variable of an instrumented object, the setter methods of this instance variable and the object it belongs to can be highlighted in the method trace over the context menu. This is really useful to see where this instance variable changed. We can step through this highlighted methods in the method trace, or use the stepping functions provided by the UI: step to the next/previous/first/last value of this instance variable.

Source code [3]. This view displays the source code of the current selected method in the method trace. Here the source mapping of the events is used to highlight the current event. Normal debugging steps are provided to step through the source code (respectively through the events). The user can select source code and inspect it. He can also change manually the current focus in the execution trace. The object history can display the history for the current selected object. This view is used to program, *i.e.,* the source code can be edited and recompiled.

Object history [4]. This view displays every occurence of a user-selected object in the trace. The events are message reception by the object, object passed as argument, object state change, object's variable assignment or object returned from a method. This is useful for backtracking an object, because if we have an occurence in the trace, we can go backwards through the trace with this object. Back to a previous occurence, see what happend to the object. We see where it was passed as an argument, thus we know from where it came, finally arriving at the first occurrence (normally its creation).

Searching [5]. This pane consists of a simple search field, where the user can query the events. The method trace [1] highlights the found events. Section 3.3.1 presents this functionality in more detail.

Variable	Search domain
event	All events
send	Events representing a method send
return	Events representing a method's return
varAccess	Events representing a variable store (instance or local)
instVarAccess	Events representing only an instance variable store
tempVarAccess	Events representing only a local variable store

Table 3.1: Predefined search variables

3.3 Supporting Trace Navigation

We need supplementary features to locate interesting events and to mark interesting objects that will help finding bugs. In the following sections we describe the searching and coloring functions.

3.3.1 Simple Searching

Searching is important and thus should be simple. This is realized in the following manner: there is only one search field where the programmer can provide a boolean expression to identify specific events. Some predefined variables are available: event for searching in all the trace events (variable access or message send), send for searching only message send events. Table 3.1 presents the predefined variables that the programmer can used. In the current version, it is not possible to define other variables. Appropriate accessor methods are available for the events to access the collected runtime data (as shown in Table 3.2). The expression is used as the selection criteria on the adequate events. The result of the search is a set of events, which are then highlighted in the method trace.

The search expression is expressed in the implementation language, here Smalltalk. With this approach users are familiar with the search language, they can access the needed data using a known language. In addition, they have full access to the domain objects (via *e.g., event sender*). Thus it is easy to add methods to the domain classes to simplify the more complex queries.

Figure 3.3 shows a search example in UNSTUCK. "varAccess varname = 'var2' & varAccess newValue = nil" is the provided search query, *i.e.,* we search the trace for any variable access, where the accessed variable is named var2 and the assigned value is nil.

Query	Result
send selector = #foo	All the executed methods named "foo"
varAccess newValue class = Foo	Every variable assignment, where the assigned object's class is Foo
return returnValue > 4	All returns with a return value greater than 4
events isSend & (event arg1 = 4) & (event arguments size = 1)	Only methods which have exactly one argument, which was 4

Table 3.2: Some search expression examples

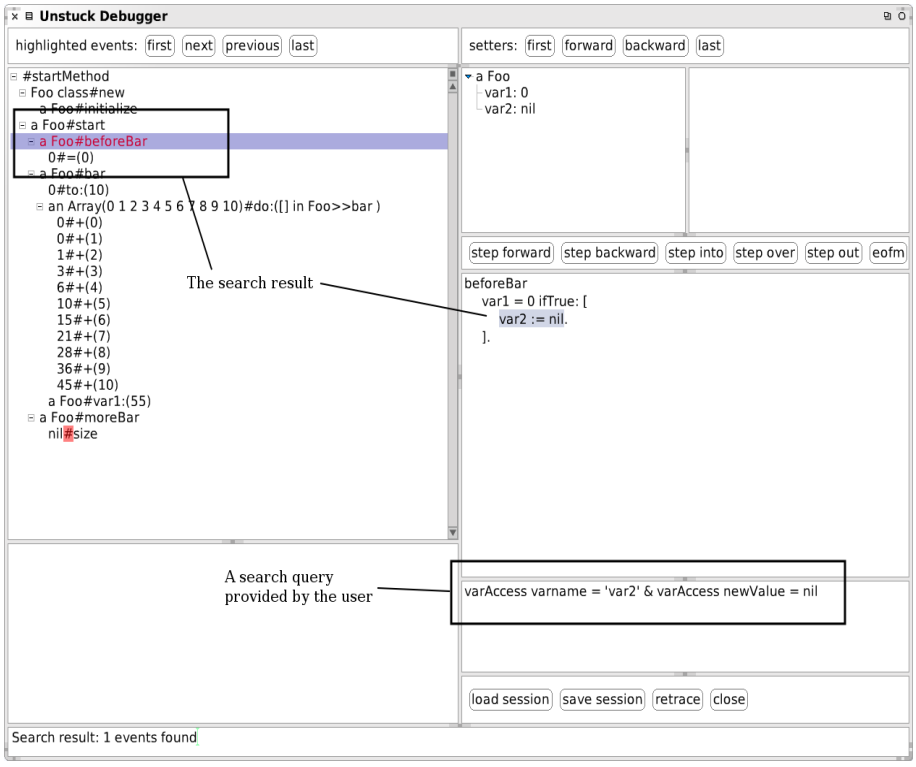

Figure 3.3: Example for a search query

3.3.2 Coloring

Coloring is a useful tool for the developer as it enables tracking objects. The user can assign a color for an object in the trace. Various views (method trace, object views, object history) highlight the object with the assigned color. So it is easy to see if that object was passed as an argument, or if it was the receiver of a message or the instance variable of another object.

Figure 3.4 and Figure 3.5 show how to assign a color for an object using the context menu. Figure 3.6 shows the resulting coloring for an object ("a RBMethodNode"). The user can easily detect the object in the trace and quickly see when it was the receiver of a message, an argument or the returned value.

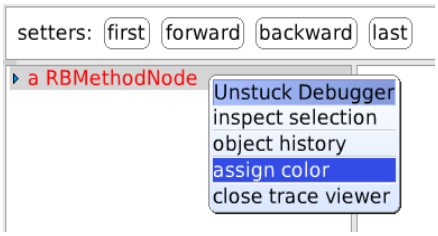

Figure 3.4: Assigning a color over the context menu.

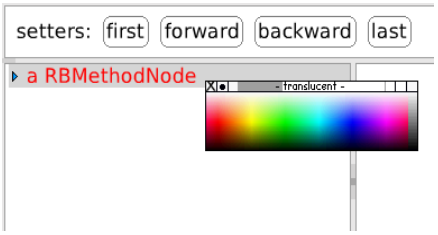

Figure 3.5: A color picker that popped up

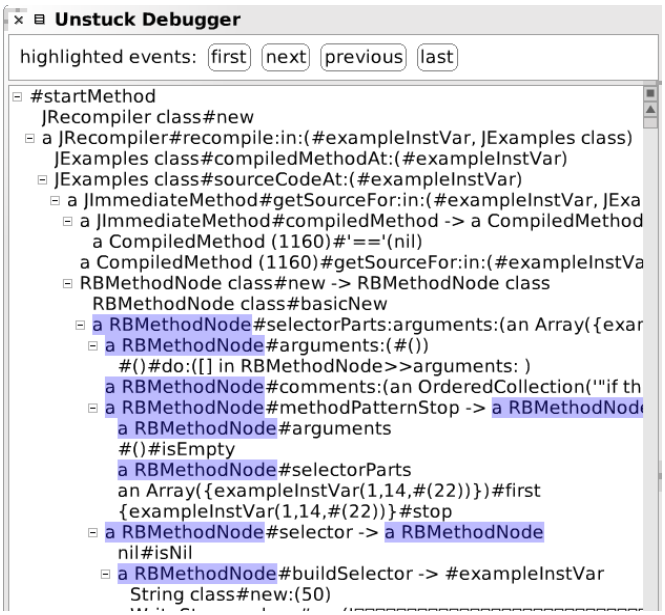

Figure 3.6: A colored object in the method trace.

Chapter 4

Examples

In this chapter we present two examples for finding bugs with UNSTUCK. Section 2.1 already introduced the Introducing Example, which is presented first. Then we show a second example, a bug which cannot easily be fixed with the normal Squeak debugger.

4.1 Introducing Example

Coming back to the problem presented in Section 2.1, here is how we solve it with the UNSTUCK Debugger:

- Start the UNSTUCK Debugger.
- Select the class Foo and provide the code to start the execution (Foo new start).
- The UNSTUCK Debugger instruments the bytecode of the methods of Foo, starts the program and collects the execution trace and presents it in the main user window.
- The error is already visible and it is obvious that nil received the message size.
- We want to see the code with the call of the message size, thus we step one back in the source view. The source code shows now that var2 received the message size
- Select var2 in the source code or in the object view.
- Highlight the modifiers of var2 (see Figure 4.1).

- There are two modifiers: one in the initialize method and one in beforeBar, which is the faulty one.

- Another possibility is to highlight var2 in the object view and use the stepping functions for the modifiers.

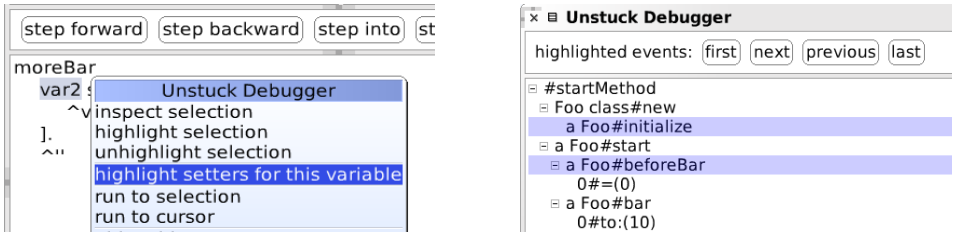

Figure 4.1: Left: highlight the modifiers of a variable over the context menu in the source view. Right: the result of the action made on the left side: highlighted methods in which var2 was modified (*i.e.,* changed the value)

The Unstuck Debugger offers us the information we need: the modifiers of var2. They are only available because the old state and the execution trace are not lost. We do not have to think about breakpoints but instead can directly navigate to the source of the bug.

4.2 AST Bug

The following example is a real bug that was not fixed with the normal Squeak debugger. Section 5.2.2 presented Swapping Compiled Methods, *i.e.,* if the collector is collecting events, we execute the instrumented method, otherwise the not instrumented one. This is necessary for Unstuck to build the trace, because it works on the abstract syntax tree of Smalltalk. ByteSurgeon works also on the abstract syntax tree, leading to an endless loop without Swapping Compiled Methods. This example proofs that Unstuck is able to debug such programs.

4.2.1 Starting Position

A programmer sent us the (unsolved) bug with the following information: "The program works on Smalltalk's abstract syntax trees and the goal of that part of the program is to store a method's node. The program makes a copy of the original tree to do some transformation without touching the original. During the copy process, some back references are not set correctly leading to an error while working with the tree."

The programmer guessed that the bug is somewhere in the postcopy implementation of the nodes. Therefore, we recompile the corresponding method, because the abstract syntax tree is copied right after recompiling.

4.2.2 The Error in the Squeak Debugger

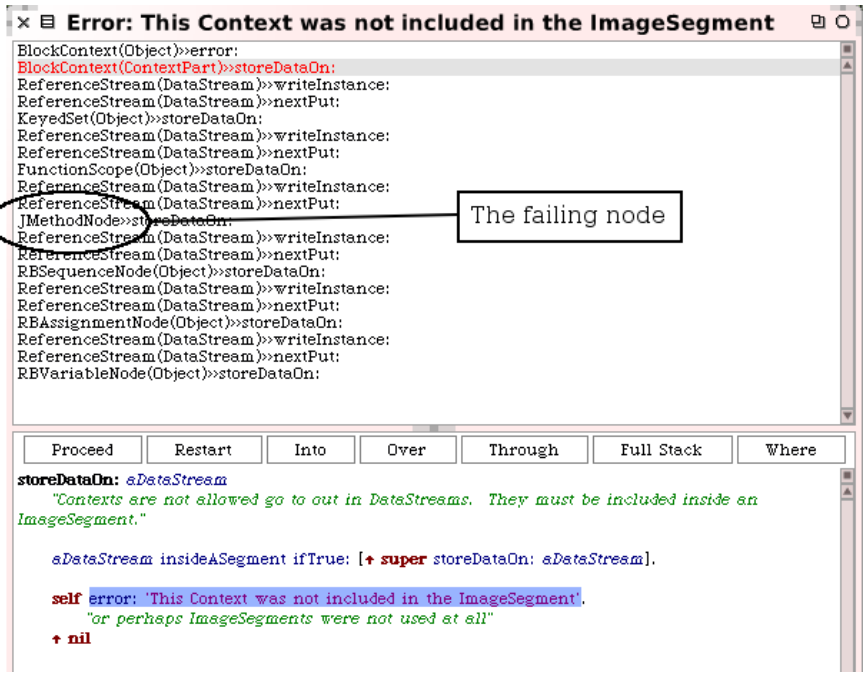

Figure 4.2: The error in the Squeak debugger

Figure 4.2 shows the debugger after the error occurred. The program tried to store a JMethodNode. The preparation process of the node sets the function scope (which is the symbol table for declared local variables of methods or blocks) to nil, which is done on the original node. The copied node still holds the function scope, which can not be stored. As there is a reference to the copied node somewhere in the original tree, the error occurs.

It is hard to find the bug with the normal debugger because of the following problems:

- **Stack Trace.** Figure 4.2 shows the stack trace of the error. We have the ability to look at the full stack, but this does not help, because the full stack is huge and the bug is in a method that finished executing, *i.e.,* is not on the stack anymore.

- **Copies.** A node and its copy are equal, but not identical. The only difference is the identity hash, thus it is painful to distinguish them and to know if a node belongs to the copied tree or the original.

4.2.3 Fixing the Bug with Unstuck Debugger

We have to provide a set of classes and the code to start the program for Unstuck. Figure 4.3 presents the start window of Unstuck Debugger and what we choose: all classes from packages that could contain interesting classes and the following code: JRecompiler new recompile: #exampleInstVar in: JExamples. JAstCompressor writeAll. We recompile the method exampleInstVar from JExamples. The method node of this method produced the error.

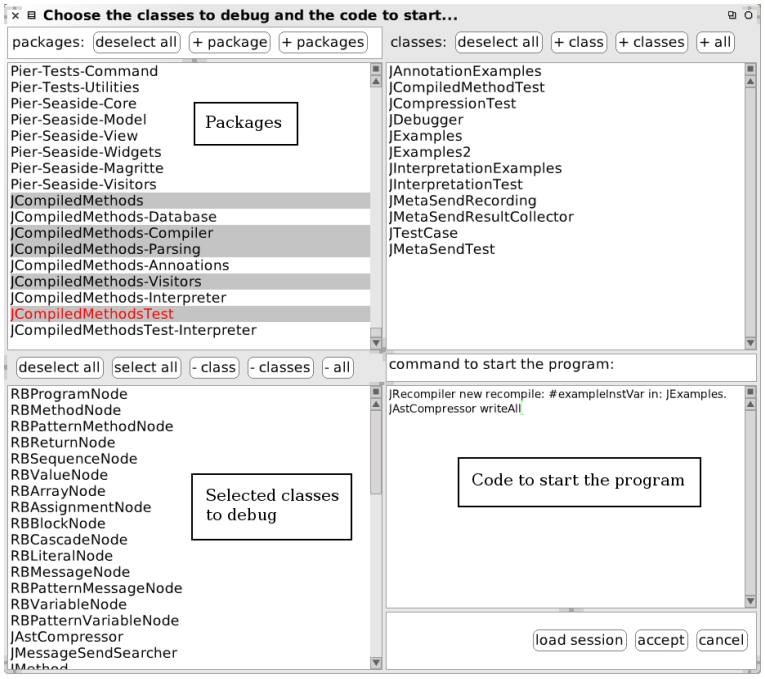

Figure 4.3: The start window of Unstuck Debugger with the selected classes and the provided code

Once Unstuck Debugger presents us the resulting trace, we begin to track down the bug. The first idea is to color the two abstract syntax trees with different colors to distinguish them. Then we should come across a suspicious node. We color the original tree with blue (the one prepared for storage) and the copy with red (which occurs at the end of the trace).

We know that the method #storeDataOn: is of interest, we pull it down to JMethodNode to have it instrumented.

Color the function scope with green (for better remembering).

Color the JMethodNodes. We color the node containing the function scope red. A bit earlier in the trace we found a JMethodNode that received the message #prepareForStorage. We color it blue.

We go down the tree and mark the next nodes in parallel: a JMethodNode has a RBSequenceNode as the body with the corresponding color.

The RBSequenceNode contains two statements holding two RBAssignmentNodes, which are colored too.

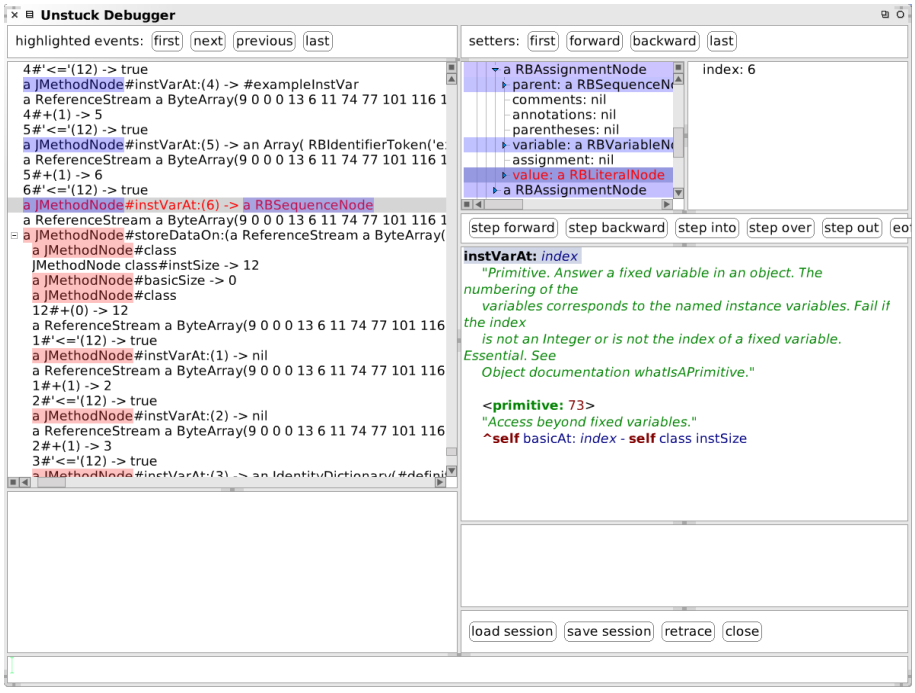

Figure 4.4: The beginning of the coloring.

Each RBAssignmentNode holds a variable and a value. In our example the variable is represented by a RBVariableNode and the value by a RBLiteralNode. We color these nodes blue (as they belong to the original). Then we like to do the same with the copied tree and color them red, but they already

appear in blue (see Figure 4.5). We found two suspicious nodes. It seems that there could be the bug. Taking the first node, the RBVariableNode, we are interested where it was assigned as the variable of the RBAssignmentNode, because there the two trees were mixed.

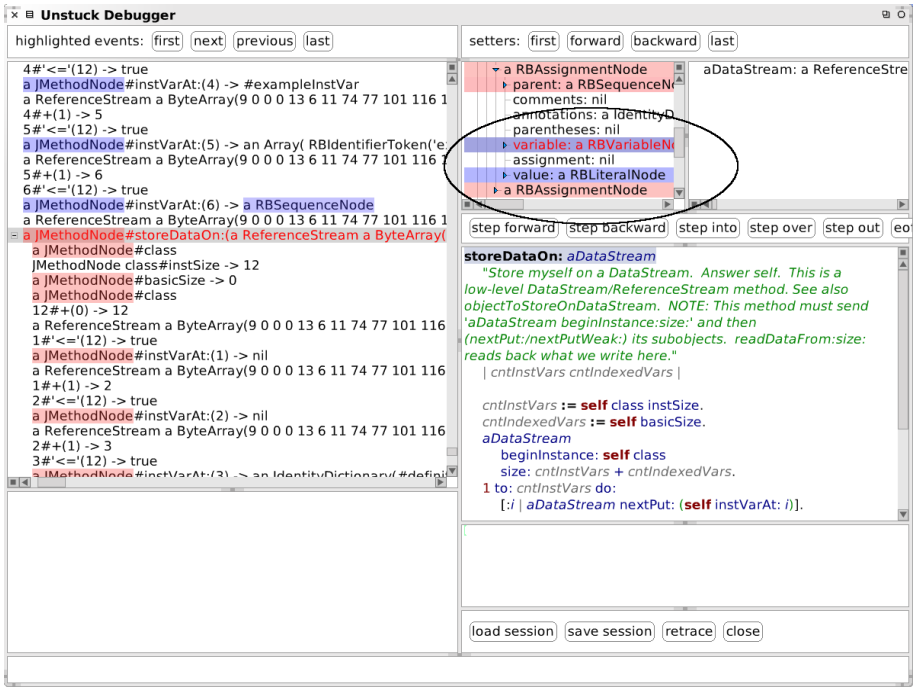

Figure 4.5: Found suspicious nodes: the two nodes colored in blue could be colored in red too.

We select the RBVariableNode and highlight the modifiers (*i.e.*, where the instance variable variable of the RBAssignmentNode changed its value), see Figure 4.6. There is only one place and we go to it (by jumping to the first highlighted line in the method trace). Since this is the only modifier, it is probable that the bug is there.

We are in the #postCopy method of RBAssignmentNode and the source code modifying the instance variable is highlighted (see Figure 4.7). There we found the bug: variable := variable postCopy should be variable := variable copy. Two lines beneath (value := value postCopy) there is the cause for the second suspicious node (the same error).

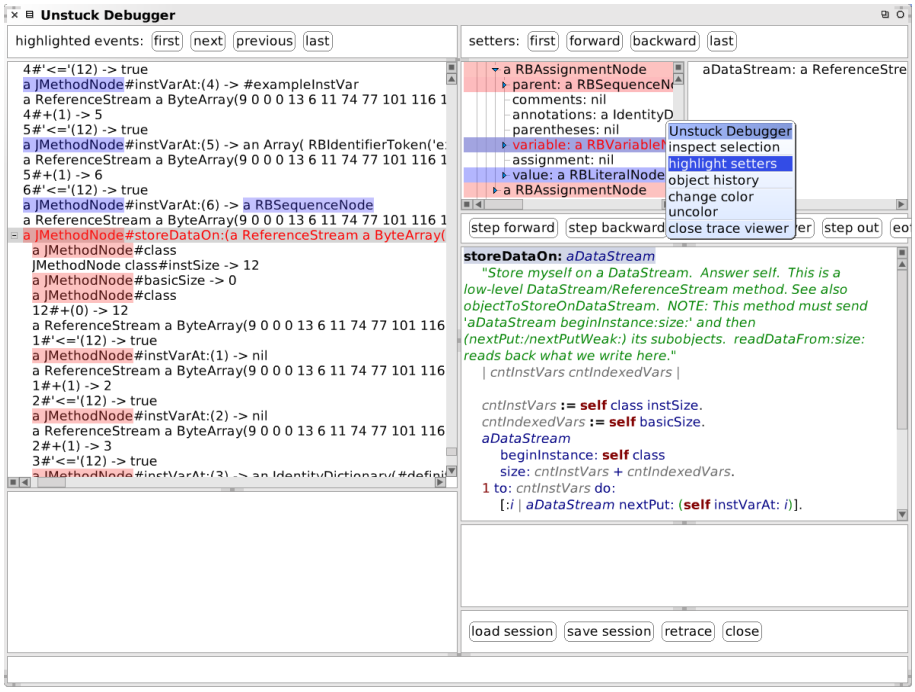

Figure 4.6: Highlight the modifiers of the interesting variable.

4.2.4 Explanation

The faulty code produces these mixed trees: since the subnodes of an RBAssignmentNode are not copied, the two trees shares them.

The abstract syntax tree is bidirectional, *i.e.,* each node contains a reference to its parent. The first instance variable of a node is the parent. The #storeDataOn: method puts all instance variables into a stream. This explains how the copy of the JMethodNode is called to store itself on the stream (see Figure 4.9): the original JMethodNode put its RBSequenceNode into the stream. The RBSequenceNode puts its statements, *i.e.,* the RBAssignmentNodes. An RBAssignmentNode put its RBVariableNode into the stream. There the walkover to the copy happens: the RBVariableNode puts its parent into the stream, which is an RBAssignmentNode of the copied tree. Then each node puts its parent, *i.e.,* it is going up the tree to the root, the JMethodNode. Figure 4.9 shows this way graphically.

4.2.5 Conclusion

The bug's source is simple, but the consequences are immense. Finding the bug with the normal Squeak debugger can take hours. It is not even said

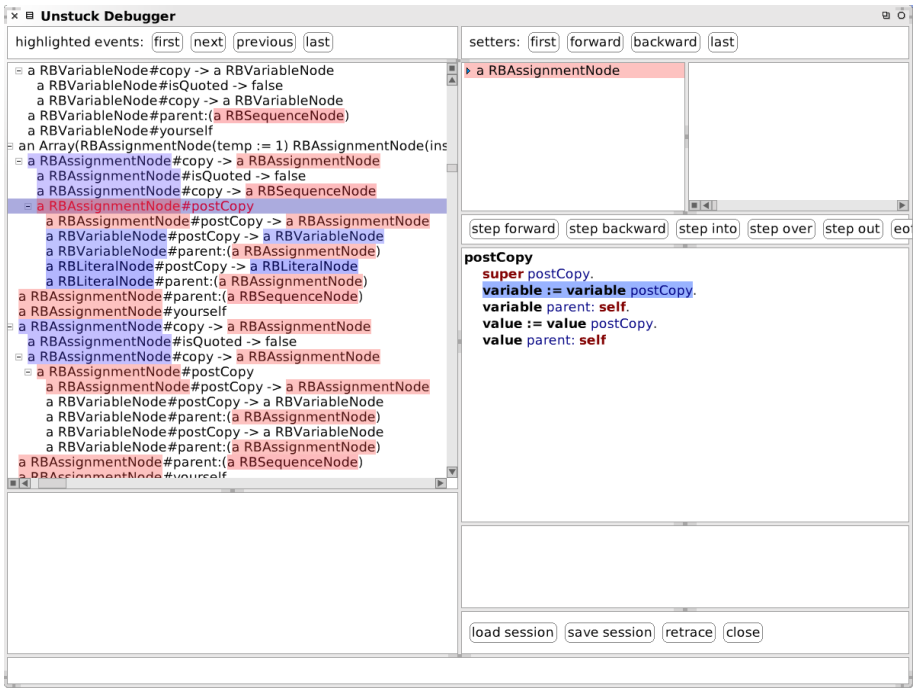

Figure 4.7: The place where the only change of the interesting variable happend.

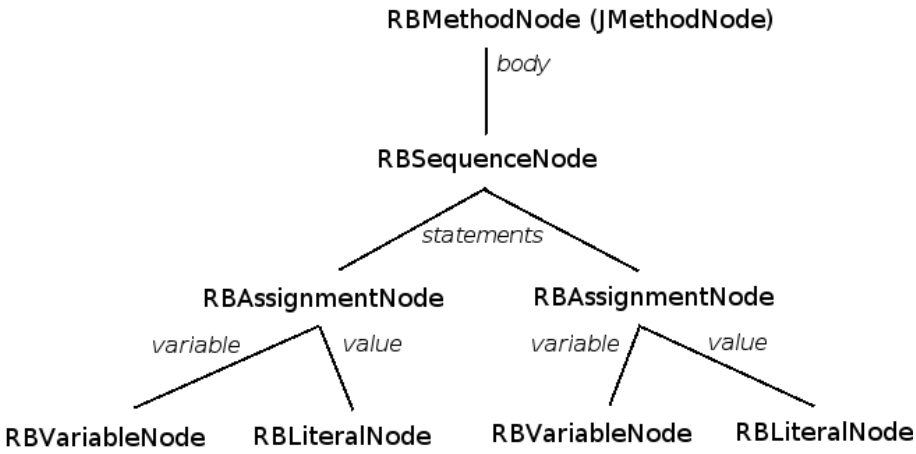

Figure 4.8: The abstract syntax tree for the method.

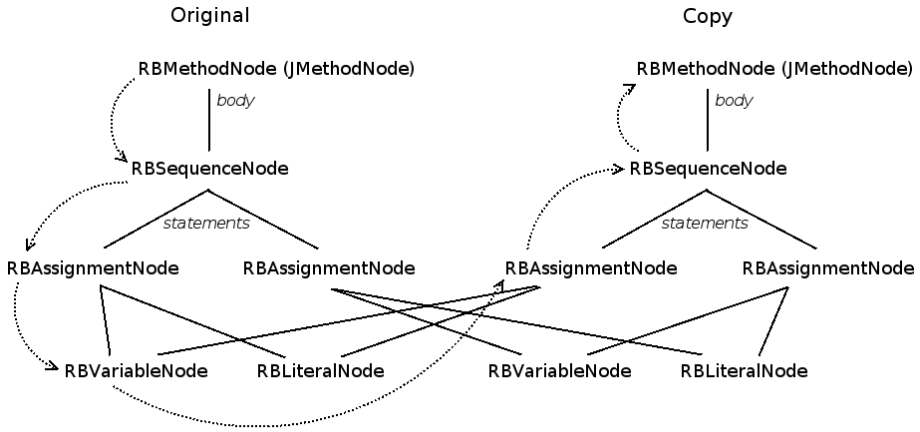

Figure 4.9: The original abstract syntax tree and its copy, sharing the same nodes as the result of the bug.

that it would be solved, because the bug was not fixed over three months. With Unstuck Debugger we fixed the bug within minutes. This examples shows how easy it is to track objects in the execution of a program, even structures of objects. Time was not wasted by searching each place where a variable was modified, these places were just given. All this additional information is very useful for finding and fixing bugs and should be available for programmers.

Chapter 5

Implementation

Unstuck Debugger is implemented in Squeak, an open-source implementation of Smalltalk [Inga 97]. Unstuck Debugger is based on the TraceLibrary which offers execution trace infrastructure. Basically the debugger collects the events, orders them and prepares the state reconstruction.

There are a few challenges when implementing our approach. To provide the state of any object at any time in the execution trace, we need a way to generate the needed data. Further we need to collect this mass of data and prepare it for presentation.

To generate events (method invocation, variable access and method return), the methods are instrumented using ByteSurgeon which is a high-level library to manipulate method bytecodes [Denk 06]. Figure 5.1 shows the different layers. The following sections describe each layer and how they work together.

5.1 Trace Library

The TraceLibrary supports the generation of execution traces from a set of classes and the code to start the program. ByteSurgeon instruments the methods of the given classes to generate the events at runtime. During the execution, a collector gathers these events and forms the program trace. Section 3.1 describes the events and the resulting trace model. The following sections shows how events are processed and how we obtain the state of objects for any time in the trace.

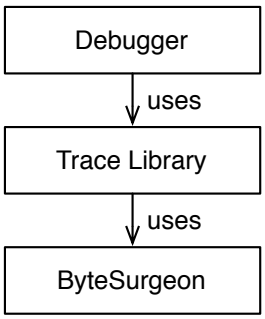

Figure 5.1: The TraceLibrary is built on top of ByteSurgeon and the Unstuck Debugger is build on top of the TraceLibrary.

5.1.1 Event Processing

A collector, a TraceCollector, collects these events at runtime and processes them to remember the order, to optimize the data structure and to prepare state reconstruction of the objects (receiver, variables and arguments) participating to the trace.

Order Remembrance. The collector tags the events with a timestamp to remember the order of the events in which they arisve. The depth of the event is also calculated by the collector.

Data structure optimization. We create an event tree from a sequence of events, using the return events as marker of the end of a method execution. Back pointers to navigate from the subevents to the parent events are also managed as part of this process.

State reconstruction preparation. The collector handles every occurrence of objects in the trace for later state reconstruction. By state reconstruction we mean the ability to reconstruct the exact state of an object at any point in time as we will explain in Section 5.1.2. We distinguish three cases for treating all the objects participating into an event (*i.e.,* receiver, arguments, variables, return values): instrumented objects, not instrumented objects and collections.

- Instrumented objects: they do not need any special handling. The collector gets the state changes of such objects from instance variable write access events. With these changes we are able to provide the

state of these objects at any time by applying the latest change before that point in time.

- Non instrumented objects: to remember their state the collector copies these objects.
- Collections: because a collector does not get any events for changes in a collection, a copy of a collection is saved with the current timestamp. Basically the collector creates a new collection of the same kind and processes every object of the collection as if it would be in the trace, *i.e.,* recursively apply the same process: check if the object is instrumented or not, if it is a collection and process them as described above. We need to copy the collection, because the original collection could be modified in the future execution of the program.

When a collector gathers an event from an instance variable write access, then the new value represents a state change of an instrumented object. This change, the current timestamp and the variable name is remembered for the corresponding object. This is useful for later state reconstruction, because we do not have to go through the whole trace and collect the needed changes. Note that the previous behavior is not necessary since we could walk over the trace and collect all changes belonging to an object. Here they are just ordered at runtime and act as a cache.

5.1.2 State Reconstruction

State reconstruction is the process of reverting an object's state to any desired point in time in the trace. As explained above a collector prepared the state reconstruction. Depending on the type of objects, the reconstruction is different:

- Instrumented objects: for every instance variable we take the latest change before the desired time and apply this change. The applied value is reverted to the desired time, too (see Figure 5.2).
- Non instrumented objects: no reconstruction needed, we just take the copy the collector has made and associated with the event.
- Collections: we take the last occurence of the collection in the trace and every object inside is reverted to the desired time.

The following examples show the special handling of collections: the first example adds a collection to the receiver which is aCollection too. Let's assume that the method `addAll:` is not instrumented but the expression is inside an instrumented method.

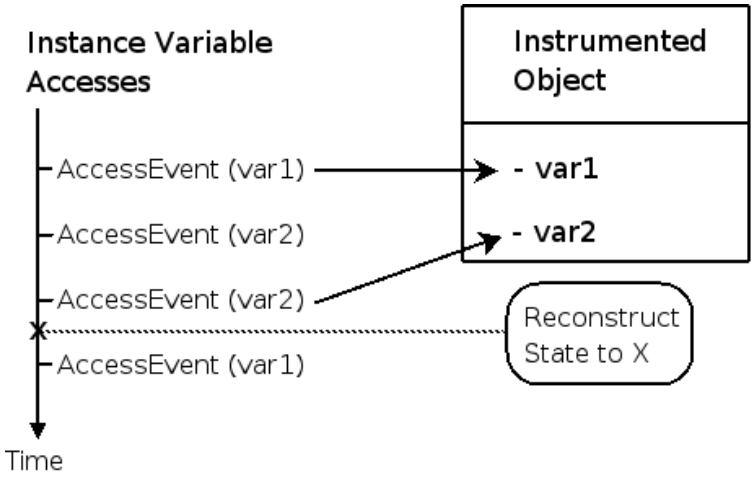

Figure 5.2: The state reconstruction process for instrumented objects.

```
...
aCollection addAll: anotherCollection
...
^ someExpression
```

When the collector treats the MessageSend for the method addAll:, it processes the two collections (because they were involved in the method's execution, as the receiver and as an argument). The collector creates a new collection with the current objects inside to remember which ones were in the collection at this time and handles each object inside as it would be an object in the trace (as described in Section 5.1.1). The same happens when the collector receives a Return event. To get the state of a collection right before this method was executed, we take the new collection created by the collector and reconstruct the state of the objects inside. To get the state of the collection after this method was executed, we take the second collection the collector created when it was treating the return event. This collection includes the newly added objects, thus we get the right state back.

```
...
aCollection foo: otherCollection
...
^ someExpression
```

with foo defined as:

```
foo: collection
  collection  removeFirst
```

Similarly to the previous example, let's assume that the method foo: is not instrumented. In addition, let's assume that we have two objects inside otherCollection, one instrumented and other not. After the execution of foo:, the collector creates a new collection: the instrumented object is put in it, and a copy of the not instrumented one. Thus we remark that the otherCollection has changed. To get the state at the end of the method's execution, we take the newly created collection and put the two following objects inside: the reverted instrumented one (by applying the latest changes) and the copy of the not instrumented object.

5.2 Event Gathering Using ByteSurgeon

ByteSurgeon [Denk 06] is a tool for transforming Smalltalk bytecode at runtime. It provides high-level abstractions, thus developers do not need to program at bytecode level. Bytecodes are low-level instructions for the Virtual Machine stack machine. ByteSurgeon can insert code before, after or better: instead of an instruction. This code is passed in form of a string of Smalltalk code. If we insert code after an instruction, it will be executed right after the execution of the instruction. Additionaly ByteSurgeon provides the same functionality for methods instead of instructions.

For accessing runtime information (such as the receiver of a message, the passed arguments), ByteSurgeon provides meta variables. They have a special syntax (<meta: #var>) and can be added to the string that represents the code to insert. ByteSurgeon provides the receiver, the arguments and the returned value of a message send, the variable name and the new value of a write access to a variable. The IRInstruction (an intermediate representation, which represents a bytecode instruction) delivers the static information, *i.e.,* the selector and the definition class of a message send and the source range of all events.

As an example we describe how to generate the method send events of a method:
ByteSurgeon iterates over the IRInstructions, thus we work with an IRSend, which represents a message send at bytecode level. It provides the static information (selector, source range, definitionclass). ByteSurgeon provides the runtime information over meta variables (the receiver is accessible with <meta: #receiver>, the arguments with <meta: #arguments>). ByteSurgeon takes a string to insert code, thus we generate the string as follows:

```
'TraceCollector default take:
    (MessageSend withSelector: ', instr selector printString   // include the selector
    ,' withArguments: <meta: #arguments>                       // include the arguments
```

```
withReceiver: <meta: #receiver>                    // include the receiver
withSourceRange: ', instr sourceRange printString  // include the source range
, ' class: ', instr superOf printString,')'        // include the definition class
```

This generates our needed event and the collector stores it. Then we instrument every send in a method with the following code:

```
aCompiledMethod instrument: [:instr |
   instr isSend ifTrue: [instr insertBefore: theString]].
```

5.2.1 Just-In-Time ByteSurgeon

Instrumenting a method is costly, particularly for longer methods. To save time in the trace generation process we use the idea of Just-In-Time ByteSurgeon: the TraceLibrary instruments only methods that are executed, which is done at runtime. For realizing this behaviour we use ByteSurgeon itself: we insert a preamble for each method which has the following functions:

- Instrument the current method.
- Re-execute the current method.

To realize this we need access to the following information:

- The current method.
- Receiver, arguments and the definition class of the current method.

We use the meta variables of ByteSurgeon to make the information available, if it is not directly provided (like receiver, selector and arguments). When instrumenting with ByteSurgeon we can pass a dictionary to the instrumentation process. The dictionary has name value pairs, thus we can assign a name to an object and then access it with a meta variable in the code string (with <: #name>).

Therefore the preamble has the following form:

```
TLUtil instrumentMethod: <: #compiledMethod>.
^<meta: #receiver> perform: <: #selector> withArguments: <meta: #arguments> inSuperclass: <: #class>.
```

We install this preamble before every method with a prepared dictionary containing the method (which is a CompiledMethod accessible over <: #compiledMethod>), the selector of the message and the definition class (accessible over <: #class>).

In Squeak we can send a message (execute a method) programmatically to an object by using the message #perform:withArguments: or in our case #perform:withArguments:inSuperclass:. When providing the superclass the lookup process starts at the provided superclass and not at the receiver's class.

5.2.2 Swapping Compiled Methods

Just-In-Time ByteSurgeon raises the problem of circulary dependencies. Figure 5.3 shows the problem: assume that we have an (already) instrumented method Y and an original method X, which is executed. Just-In-Time ByteSurgeon initiates the instrumentation of method X and ByteSurgeon does the instrumentation. If ByteSurgeon uses an already instrumented method (like method Y) for the instrumentation process, the runtime events are generated and the TraceCollector collects them. These events do not belong to the trace the user wants.

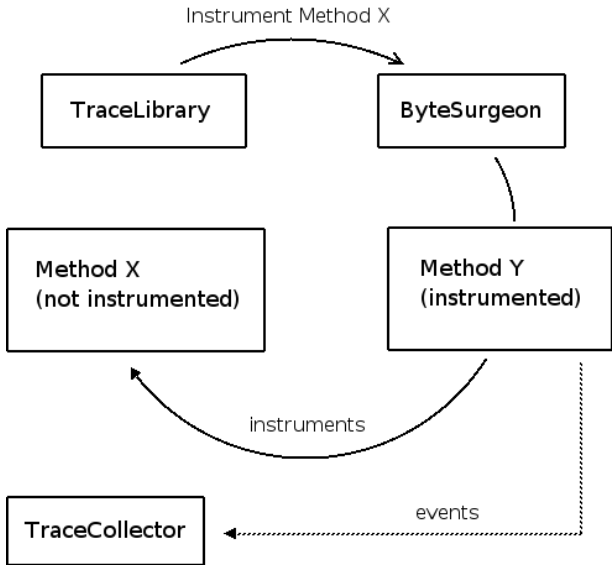

Figure 5.3: The problem of the Just-In-Time ByteSurgeon

The first approach to solve the problem stated before would be to appoint the TraceCollector to not handle these events and to throw them away. Be-

fore starting the instrumentation process we could tell the TraceCollector that from now on every event should be rejected.

We introduced a better solution for the problem: swapping compiled methods. Basically there are two methods, the original one which is not instrumented and the instrumented one. Based on the situation the demanded method is executed. Therefore if the normal trace building process is running, we execute the instrumented method. If ByteSurgeon is instrumenting a method, we execute the not instrumented method. The advantage of this solution is that no unnecessary events are generated.

There are two preambles to realize this approach: one for instrumented methods and one for not instrumented methods. These preambles swap the method if needed. We can tart and stop the TraceCollector to collect events or not. To decide which of the two methods is executed we take if the TraceCollector is collecting or not as the selection criteria.

The following code represents the preamble for not instrumented methods:

```
TLTraceCollector isCollecting ifTrue: [
   (<: #class> compiledMethodAt: <: #selector>) replaceMyselfWith:
      ((<: #class> compiledMethodAt: <: #selector>)
         properties at: #instrumentedMethod).

   ^<meta: #receiver>
      perform: <: #selector>
      withArguments: <meta: #arguments>
      inSuperclass: <: #class>.
].
```

If the TraceCollector is not collecting events, the current method continues, because it is not instrumented. If the TraceCollector is collecting, the preamble replaces the current method with its instrumented counterpart, which is stored as a property of the not instrumented method.

The preamble for instrumented methods swaps the method if the TraceCollector is not collecting events.

We use meta variables to provide the needed information, which is stored in a dictionary and passed to the installation procedure of the preambles (like installing the Just-In-Time ByteSurgeon preamble in Section 5.2.1).

5.2.3 Combine Just-In-Time ByteSurgeon and Swapping Compiled Methods

To get Just-In-Time ByteSurgeon and Swapping Compiled Methods working together we need to modify the preamble used in Just-In-Time ByteSurgeon (Section 5.2.1) and we need to prepare the methods of the given classes for tracing. This section describes the modified preamble for Just-In-Time ByteSurgeon, the preparation process and when methods are instrumented at runtime.

Preparation. For each method we install the preamble for not instrumented methods with the corresponding information (selector, definition class, the method itself). Further we install the modified version of the Just-In-Time ByteSurgeon preamble.

Modified preamble for Just-In-Time ByteSurgeon. The modified preamble looks as follows:

```
TLTraceCollector isCollecting ifTrue: [
    TLTraceCollector stop.
    TLUtil instrumentMethod: <: #compiledMethod>.
    TLUtil insertInstrumentedPreamble:
        <: #class> selector: <: #selector> original: <: #originalWithPreamble>.
    TLTraceCollector start.
    ^<meta: #receiver>
        perform: <: #selector>
        withArguments: <meta: #arguments>
        inSuperclass: <: #class>.
].
```

The preamble ensures the combination of Just-In-Time ByteSurgeon and Swapping Compiled Methods:

- If the TraceCollector is not collecting events, the method execution continues as normal. In the other case we should change to the instrumented method. Because of the Just-In-Time ByteSurgeon idea, there is no instrumented method yet, thus it will be created.
- We stop the TraceCollector for the upcoming instrumentation process. From this point on only original (not instrumented) methods are executed to prevent the creation of unnecessary events.
- The next step is to instrument the original method to produce the desired events. After this we install the preamble for instrumented methods.

- After the end of the instrumentation process we start the TraceCollector, *i.e.,* it will collects events again the instrumented method will be executed from this point on.
- Finally we re-execute the current method (which will be the instrumented one).

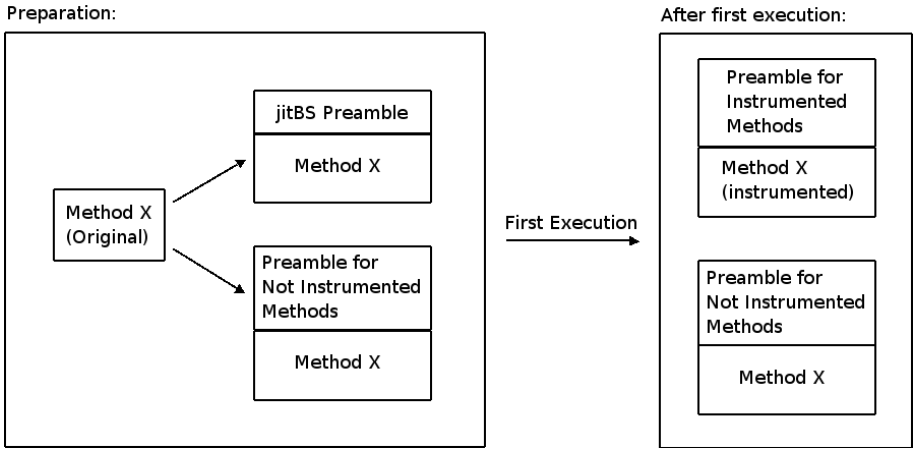

Figure 5.4: The resulting methods of Just-In-Time ByteSurgeon and Swapping Compiled Methods combined

Figure 5.4 shows the resulting methods: in the preparation process we create two methods from the original one. One with the modified preamble of Just-In-Time ByteSurgeon, which is the actual method after the preparation, and one with the preamble for not instrumented methods. When the first execution runs (and the TraceCollector is collecting events), the instrumentation process begins (with the original method). We install the preamble for instrumented methods before the resulting method of the instrumentation. After the first execution there is still the method with the preamble for not instrumented methods and the instrumented method with the corresponding preamble.

This solution ensures the desired behaviour: we instrument only methods that are executed (while the TraceCollector is collecting events). The instrumented method is only executed if needed.

In Section 4.2 we describe an example of bug fixing with Unstuck (the AST bug), where we need Swapping Compiled Methods. Without this behaviour it is impossible to generate the trace while using Just-In-Time ByteSurgeon. Therefore Section 4.2 proves that our approach is working.

Chapter 6

Discussion

Collecting that much runtime data raises questions about performance and memory usage. In this chapter we discuss both.

6.1 Performance

In this section we present performance benchmarks [1] for the preparation time, the instrumentation time and the runtime. We used three examples:

- the Introducing Example, presented in Section 4.1
- the AST bug, presented in Section 4.2
- a Pier trace: we took all test suites from a test package of Pier. We selected every model class from Pier and every model class from Magritte to debug and ran these nine test suites.

	Prepared Methods	Time (s)	Average (s)
Introducing Example	10	0.19	0.019
AST Bug	943	17.6	0.019
Pier Trace	1714	30.4	0.018

Table 6.1: Preparation time

First of all, the TraceLibrary prepares all methods from the given classes for later tracing. This process includes installing the modified preamble for

[1] Machine used: AMD Athlon 64 3200+ (2GHz), 1GB RAM, Squeak 3.9beta

Just-In-Time ByteSurgeon and the preamble for not instrumented methods (as described in Section 5.2.3). Table 6.1 presents the results. Preparing a method takes less than 20 milliseconds. The needed time is constant, a fact that does not surprise, because we insert the two preambles before a method and this is independent from the method itself. In Squeak a class has averaged 16 methods per class, thus we prepare about three classes per second. The instrumentation seems to take quite long, especially considering that it is only for preparing and instrumentation and tracing is not done yet. Later when discussing the instrumentation time, we will show that we save time with preparing methods. For the preparation process we rely on ByteSurgeon and the user itself. If the user can select only important classes, the preparation time is acceptable.

	Instrumented Methods	Time (s)	Average (s)
Introducing Example	6	0.19	0.03
AST Bug	127	10.2	0.08
Pier Trace	338	28.6	0.085

Table 6.2: Instrumentation time

We measured the time used to instrument methods with ByteSurgeon. An instrumented method generates all the runtime data we collect for later debugging. Instrumenting a method takes approximately 80-90 milliseconds. The 30 milliseconds from the Introducing Exampleare not significant, because the methods are short and the instrumentation process depends on the length of a method (longer methods have more instructions, thus we have to insert more code). The instrumentation time was from the beginning a problem, thus we introduced Just-In-Time ByteSurgeon. Other improvements outside from Unstuck were made to bring down the instrumentation time. The problem is the amount of data we produce and the question is if we can really bring down the instrumentation time significantly. We will investigate this problem in the future (see Section 7.2.2).

The preparation and instrumentation time measurements proof that the use of Just-In-Time ByteSurgeon is worthy. In the AST bug example, we have a preparation and instrumentation time of 27.8 seconds. We have 943 methods and an average of 0.08 seconds to instrument a method. Therefore instrumenting every method would take approximately 75 seconds (943 * 0.08 seconds), which is more than the 2.5 fold time. The same calculations shows that in the Pier trace instrumenting every method would take approximately the 2.5 fold time.

	Overall runtime (s)	Effective runtime (s)	Simulated runtime (s)
Introducing Example	0.65	0.27	0.045
AST Bug	33.5	5.7	1.5
Pier Trace	1350	1291	5.2

Table 6.3: Various runtimes

As the last benchmark we measured the overall runtime used to generate the trace (including preparation and instrumentation time). Deduced from this time we have the effective runtime, which is the overall runtime subtracting preparation and instrumentation time. We measured the simulated runtime too, to have a time to compare. The simulated runtime is the time used to step through the program in the Squeak debugger, which is done programmatically (without any user interface). For the AST bug it takes about the four-fold time to generate the trace compared to the simulated run. The simulated run does only the stepping, while we record a whole trace and process the events. The pier trace is an example for a really large trace (see Section 6.2).

The AST bug is a real scenario. The performance there is good. It takes round half a minute for Unstuck to get the trace and the preparations. The bug was solved with Unstuck within minutes, but remained unsolved with the normal debugger.

With the Pier trace we reached the limit. Normally a user debugs one test case, but here we ran nine test suites. We plan to test and optimize the performance and memory usage in more detail (see Section 7.2.2). We assume that the garbage collector could be the problem: if a certain amount of space is used, the garbage collector does a full garbage collection instead of an incremental one. As the trace grows, the garbage collector needs much more resources, which may be the cause for the slowdown.

6.2 Memory Usage

To get an idea how much memory a trace uses we measured it. We included the space used by the state preparation process and data structure optimization. We used the same three examples presented in Section 6.1. Table 6.4 shows the results.

	Number of events	Memory usage (Kb)	Average per event (Kb)
Introducing Example	74	16	0.21
AST Bug	2725	800	0.29
Pier Trace	389689	88800	0.22

Table 6.4: Memory usage

The AST bug takes only 800 Kb of memory, which is rather few. The big Pier trace uses much more memory, but if we take the number of events as a measurement for the size of the trace, the memory usage is more or less proportional to the trace size.

	Memory usage with changes (Kb)	Memory usage with copies (Kb)	Factor
Introducing Example	16	16	1
AST Bug	800	2300	2.9
Pier Trace	88800	?	-

Table 6.5: Various memory usages

Table 6.5 shows interesting results: we measured the memory usage if we copy every object in the trace to remember the state at any time. In the Introducing Examplethe memory usage remains the same, there are not much objects participating in the trace, thus the copies do not carry weight. The AST bug shows a more significant result: the trace working with copies instead of changes uses nearly the three-fold amount of memory. The overall runtime for generating the trace remains nearly the same. We use more time to copy objects, but the state preparation process disappears.

We tried to generate the trace with only copying objects for Pier too, but we did not manage to generate the trace and evaluate it. We did it for one single test suite which holds a tenth of the events of the full trace. The resulting trace uses nearly the three-fold amount of memory which the normal full trace uses. This shows that the memory used has to be very large. Together with the evaluation of other single test suites (which were possible to evaluate), we estimate a memory usage of over two gigabytes for the whole trace of all nine test suites.

If we copy every object, the trace uses more memory. For the AST bug it is the three-fold amount, for the Pier trace we can only estimate, but we need a lot of memory. The system crashes if we try to generate the full trace with only copying objects (not to mention to evaluate the memory usage). These are the reasons to work with changes.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In many cases the Unstuck Debugger provides an improvement over conventional debugging: if we have a faulty value it is easy to find the place where it was set uncorrectly. The bug can be chased in the program history. Such a kind of behavior is not possible in a normal debugger where this work has do be done manually using multiple restart and breakpoints. When we select a message send in the Unstuck Debugger it is like we stopped the program there with a breakpoint. We have the same information, but there is no need to put a new breakpoint for stopping the program in another situation. In the debugger we can just go there. Completed with backtracking and searching functions the debugger helps us finding bugs much faster.

7.2 Future Work

To have all this information available there is a price to pay: instrumenting the bytecode and collecting the trace information which slows down applications. But it needs to be compared to the time won when finding bugs. Up until now the Unstuck Debugger has been working on a limited still challenging case studies such as debugging abstract syntax tree and compiler internals. We plan to analyze the performance and memory usage in more detail (Section 7.2.2). In addition as Smalltalk offers a dynamic programming style with on the fly recompilation, we plan to investigate if it is realistic to instrument the complete environment and be able to debug any application without having to provide a program seed.

Other possible enhancements are support for threads, which is discussed in Section 7.2.1 and the ability to restart execution in the past: the Unstuck Debugger should be able restart the execution trace from any point in the

past which requires to recreate stack execution on the fly. UNSTUCK provides this feature, but it is experimental and needs more work to get stable.

7.2.1 Threads

UNSTUCK is not designed for multi-threaded programs. This sections describes what is needed to debug multi-threaded programs with UNSTUCK and how it can be realized.

UNSTUCK needs certain adjustments to be able to debug multi-threaded programs:

- **Event assignments to threads.** Each event belongs to one thread resulting in multiple traces, one trace for each thread.
- **Global Identifier for events.** Events are ordered within a trace, but we still need a global ordering for events to know if something in one thread happened before or after another thing in another thread. Therefore we can catch bugs from bad or not synchronized threads.
- **New view for threads.** This view displays the threads of a program. According to the selected thread, the method trace shows the corresponding trace.
- **Extend navigation functions.** Additionally to the new view we need to extend the navigation functions. Stepping can be scoped to a thread, but also global steps are desired.

Gathering additional information, like the thread an event belongs to, raises another problem: UNSTUCK must not force synchronization between the multiple running threads. Therefore if we use a synchronized area for event collecting or processing, we would force an unwanted synchronization between the threads. Bugs resulting from bad or not synchronized threads might disappear because of the synchronization effect of UNSTUCK.

We could use multiple TraceCollectors to avoid the described synchronization problem. Each thread has exactly one collector to assure independency among threads and their trace collecting process.

For detecting the current active thread we need an atomic operation to get this information. With it we can assign an event to a thread by sending it to the intended collector. Before that we need to assign a collector to each thread. We could use our own, small modified threads. These threads know a collector. Instead of sending the generated events directly to the collector,

we let the (active) thread do it. Thus each thread sends the events to the right collector and we get one trace per thread.

A rather simple solution to get a global identifier for each event is to take the current timestamp. The problem here is that in Squeak the clock's precision is in milliseconds. It is not sure that each event has a different timestamp. A solution for this problem would be a counter implemented in the virtual machine as a primitive. A primitive is a call to the virtual machine and during the call everything else is blocked. Therefore the operation is atomic and we get the correct global ordering, without any synchronization effect.

7.2.2 Performance and Memory

Generating and collecting the execution trace of a program is costly. We already analysed the performance and memory usage of Unstuck (Section 6.1 and Section 6.2). We need to do a more detailed performance analysis to detect exactly which parts of the system take most of the resources. With this analysis we are able to introduce and validate optimizations to get better performance in Unstuck.

The space usage analysis shows that Unstuck requires much memory if the execution trace grows. We need to test if we can bring the space usage down to enable the handling of big traces with Unstuck. We could use a sort of garbage collection for the trace, *i.e.,* we exclude unneeded information. Another idea would be to compress the trace.

7.2.3 Scoping and use of reflection framework that provides it

Currently Unstuck uses ByteSurgeon to generate the execution trace. We could change the backend to a reflection framework that provides scoping like Geppetto [Röth 06]. If we could scope the event generation process, numerous problems would disappear. Swapping Compiled Methods (Section 5.2.2) is not the nicest solution: with the reflection framework we scope the event generation to our tracer and get the requested effect, without swapping the method to execute. To support threads in Unstuck we could scope the event generation to each thread. Based on the scope a different collector processes the events.

A problem with a reflection framework is the acquirement of specific information, like the mapping of the source code to the bytecode. Unstuck needs this information which is normally not accessible over the reflection

framework. In a future evaluation we will check if we can extend the framework to solve this problem and benefit from the advantages (*i.e.,* scoped reflection) that a reflection framework provide.

Appendix A

Installation Instructions

In this appendix we give a short overview on how to install Unstuck.

1. Install ByteSurgeon:

   - Use at least a 3.9beta image.
   - Load package AST from SqueakSource.
   - Load package NewCompiler from SqueakSource.
   - Enable the "use new compiler" preference.
   - Recompile the image using the "Recompiler" class.
   - Install ByteSurgeon from SqueakSource.

   Alternatively download a prepared image from the website.

2. Use the Unstuck repository from SqueakSource to the TraceLibrary and then Unstuck (in this order).

3. Use "Unstuck start" to start the debugger.

List of Figures

List of Tables

Bibliography

[Augu 95] M. Auguston. *Program Behavior Model Based on Event Grammar and its Application for Debugging Automation.* In 2nd International Workshop on Automated and Algorithmic Debugging, Saint-Malo, France, Mai 1995. (p 9)

[Augu 98] M. Auguston. *Building program Behavior Models.* In European Conference on Artificial Intelligence ECAI-98, Workshop on Spatial and Temporal Reasoning, Brighton, England, August 1998. (p 9)

[Cons 93] M. P. Consens and A. O. Mendelzon. *Hy+: A Hygraph-based Query and Visualisation System.* In Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2, pages 511–516, 1993. (p 8)

[De P 98] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. *Execution Patterns in Object-Oriented Visualization.* In Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98), pages 219–234. USENIX, 1998. (p 8)

[Deme 02] S. Demeyer, S. Ducasse, and O. Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2002. (p 1)

[Denk 06] M. Denker, S. Ducasse, and É. Tanter. *Runtime Bytecode Transformation for Smalltalk.* Journal of Computer Languages, Systems and Structures, vol. 32, no. 2-3, pages 125–139, July 2006. (pp 29, 33)

[Duca 99a] M. Ducassé. *Opium: An extendable trace analyser for Prolog.* The Journal of Logic programming, 1999. (pp 1, 9)

[Duca 99b] M. Ducassé. *Coca: An Automated Debugger for C.* In International Conference on Software Engineering, pages 154–168, 1999. (p 9)

[Duca 06] S. Ducasse, T. Gîrba, and R. Wuyts. *Object-Oriented Legacy System Trace-based Logic Testing.* In Proceedings 10th European Conference on Software Maintenance and Reengineering (CSMR 2006), pages 35–44. IEEE Computer Society Press, 2006. (p 8)

[Edwa 63] D. J. Edwards and M. L. Minsky. *Recent Improvements in DDT.* Research Report AIM-60, MIT Artificial Intelligence Laboratory, 1963. (p 1)

[Gold 05] S. Goldsmith, R. O'Callahan, and A. Aiken. *Relational Queries over Program Traces.* In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), pages 385–402, New York, NY, USA, 2005. ACM Press. (p 9)

[Guéh 02] Y.-G. Guéhéneuc, R. Douence, and N. Jussien. *No Java without Caffeine: A Tool for Dynamic Analysis of Java Programs.* In ASE, page 117. IEEE Computer Society, 2002. (p 8)

[Inga 97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. *Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself.* In Proceedings OOPSLA '97, ACM SIGPLAN Notices, pages 318–326. ACM Press, November 1997. (pp 7, 29)

[Ko 04] A. J. Ko and B. A. Myers. *Designing the whyline: a debugging interface for asking questions about program behavior.* In Proceedings of ACM CHI 2004 Conference on Human Factors in Computing Systems, volume 1, pages 151–158, 2004. (p 8)

[Lang 95] D. Lange and Y. Nakamura. *Interactive Visualization of Design Patterns can help in Framework Understanding.* In Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1995), pages 342–357, New York NY, 1995. ACM Press. (p 8)

[Lenc 97] R. Lencevicius, U. Hölzle, and A. K. Singh. *Query-Based Debugging of Object-Oriented Programs.* In Proceedings OOPSLA '97, ACM SIGPLAN, pages 304–317, October 1997. (p 8)

[Lenc 99] R. Lencevicius, U. Hölzle, and A. K. Singh. *Dynamic Query-Based Debugging.* In R. Guerraoui, editor, Proceedings ECOOP '99, volume 1628 of *LNCS*, pages 135–160, Lisbon, Portugal, June 1999. Springer-Verlag. (pp 1, 8)

[Lewi 03a] B. Lewis and M. Ducassé. *Using events to debug Java programs backwards in time.* In OOPSLA Companion 2003, pages 96–97, 2003. (pp 1, 8)

[Lewi 03b] B. Lewis. *Debugging Backwards in Time*. In Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), October 2003. (pp 7, 8)

[Mart 05] M. Martin, B. Livshits, and M. S. Lam. *Finding application errors and security flaws using PQL: a program query language*. In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), pages 363–385, New York, NY, USA, 2005. ACM Press. (p 9)

[Röth 06] D. Röthlisberger. Geppetto: Enhancing Smalltalk's Reflective Capabilities with Unanticipated Reflection. Master's thesis, University of Bern, January 2006. (p 47)

[Walk 98] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. *Visualizing Dynamic Software System Information through High-Level Models*. In Proceedings OOPSLA '98, pages 271–283. ACM, October 1998. (p 8)