Compass

Flow-Centric Back-In-Time Debugging

Masterarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Julien Fierz

2009

Leiter der Arbeit
Prof. Dr. Oscar Nierstrasz
Dr. Adrian Lienhard

Institut für Informatik und angewandte Mathematik

Abstract

Debugging object-oriented programs often is a difficult and time-consuming task. Nearly all of today's debuggers only show the current state of a failing program. The user can see when the state is corrupted, but usually the root cause that leads to that state occurs long before that. Back-in-time debuggers address this problem by recording the execution history of a program run and presenting it to the user for inspection of past states. Those debuggers have proven useful as they help the developer to solve difficult problems better than a standard debugger. However, most of those tools do not provide sophisticated techniques to explore the collected dynamic data, which can make it hard to track down the root cause of an error in large program executions. The approaches are state-centric, which means they provide the past state at different points in time, but they provide no information on how objects were passed around in the system. To address this problem we provide a *flow-centric* approach that focuses on the reference transfers of objects. We present a new back-in-time debugger user interface that provides more efficient exploration of the execution history. The debugger has views and functionality that help the developer understand the failing system and let him explore how objects were passed around. Our initial case studies show that it is possible to find complex bugs more efficiently than with existing approaches.

Acknowledgements

After a year of coding, reading, writing, hacking and dealing with happily colored Squeak objects, it is time to give some credits to the people that helped me with all this.

I would like to start with the people that work at the Software Composition Group (SCG).

Many thanks go to Dr. Adrian Lienhard, the supervisor of this thesis. His research was fundamental for this work and he supported me a lot with good ideas and informative conversations.

Also thanks to Dr. Orla Greevy, who by supervising my *Informatikprojekt* prepared me for the task of writing this thesis.

Thanks to Prof. Dr. Oscar Nierstrasz for giving me the opportunity to write my thesis at the SCG.

Also thanks to all the other members of the SCG for many good pieces of advice and ideas. Everyone in this group does not hesitate to help when asked for advice.

Special thanks to the guys that were working with me in the SCG pool and were always eager to bring some entertainment into it.

Finally I want to thank all my friends and family, at work and in my private life, for always putting up with my grumpy mood when things did not go that well.

Contents

Chapter 1

Introduction

In object-oriented programs, a developer normally spends a significant amount of time in debugging [Zell05]. With common debuggers, this usually works somewhat like this. The debugger opens at the location where an error occurs. The debugger provides the current execution stack and the current values of variables. This information is often not enough to find the error, because the source of it lies in a method execution that is not in the execution stack anymore [Libl05]. Therefore the developer has to set breakpoints where he thinks he might have to look for the bug, and restart the program. For trivial errors or when the source of the error lies near to where the debugger stopped, this works fine, but for more complex bugs, he has to restart several times. In programs with large execution traces and lots of objects, the debugging process can be tedious. The process can be especially difficult if the developer lacks detailed knowledge of the system, which makes it hard to find out where breakpoints have to be set. This is why lately much research has been conducted on alternative debugging techniques like automated debugging [Zell01], query-based debugging [Lenc97] or back-in-time debugging [Lewi03].

In our work we focus on back-in-time debugging because it bears potential to become a very helpful tool to find complex bugs on the fly. In principle it can just be used like any other debugger, only with more information. The developer does not have to learn lots of new techniques or even change the way how he is working. With back-in-time debugging, all the information needed to find the bug is there. However, it can still be a challenge to find the root cause of an error. The problem is that huge amounts of execution data gets collected, which is difficult to explore efficienctly by the developer.

1.1 Problems with Back-in-Time Debugging

In this section we discuss the problems with back-in-time debugging in general and what is missing in current approaches for an efficient search of bugs. A big challenge of back-in-time debugging is how the collected information is presented to the user, because the method traces and object histories can get huge very quickly as programs get more complex. Navigating this data to find bugs is not always convenient in existing approaches like the Omniscient Debugger (ODB) [Lewi03] or Unstuck [Hofe06]. They provide the possibility to find bugs more efficiently than with a standard debugger, but still, the developer can be overwhelmed by the amount of data. Navigating this data can become a complex task where one can get lost quickly.

1.1.1 Missing Flow of Objects

To illustrate limitations of current approaches, we start with an example.

Listing 1.1: A simple bank account example

```
BankAccount class>>example
    | account |
    account := self new.
    account deposit: self getDeposit.
    account addInterest.

BankAccount class>>getDeposit
    ↑ InputReader new readDeposit

BankAccount>>deposit
    ↑ deposit

BankAccount>>deposit: amount
    deposit := amount

BankAccount>>addInterest
    deposit := InterestCalculator new calculateNewDeposit: self deposit interest: 0.05

InterestCalculator>>calculateNewDeposit: deposit interest: interest
    ↑ deposit + (deposit * interest)    "<- crash, because deposit is nil"

InputReader>>readDeposit
    ↑ nil    "<- implementation hidden, for simplicity"
```

The example uses the classes BankAccount, InterestCalculator and InputReader. The BankAccount saves the deposit in a field and has the possibility to add an interest of 5% to the deposit. The InterestCalculator is responsible for computing and returning a new deposit, using the old deposit and the given interest. The InputReader is used to read an amount for the deposit from the user. For simplicity, the actual implementation of the InputReader class is hidden and we just assume that for some reason (probably a bug) the method readDeposit returns nil. The class method example is the main method, it creates an account, sets the deposit, which it gets from the InputReader helper class, and adds the interest. The sequence diagram in Figure 1.1 shows the execution of the example. The flash indicates where the program breaks.

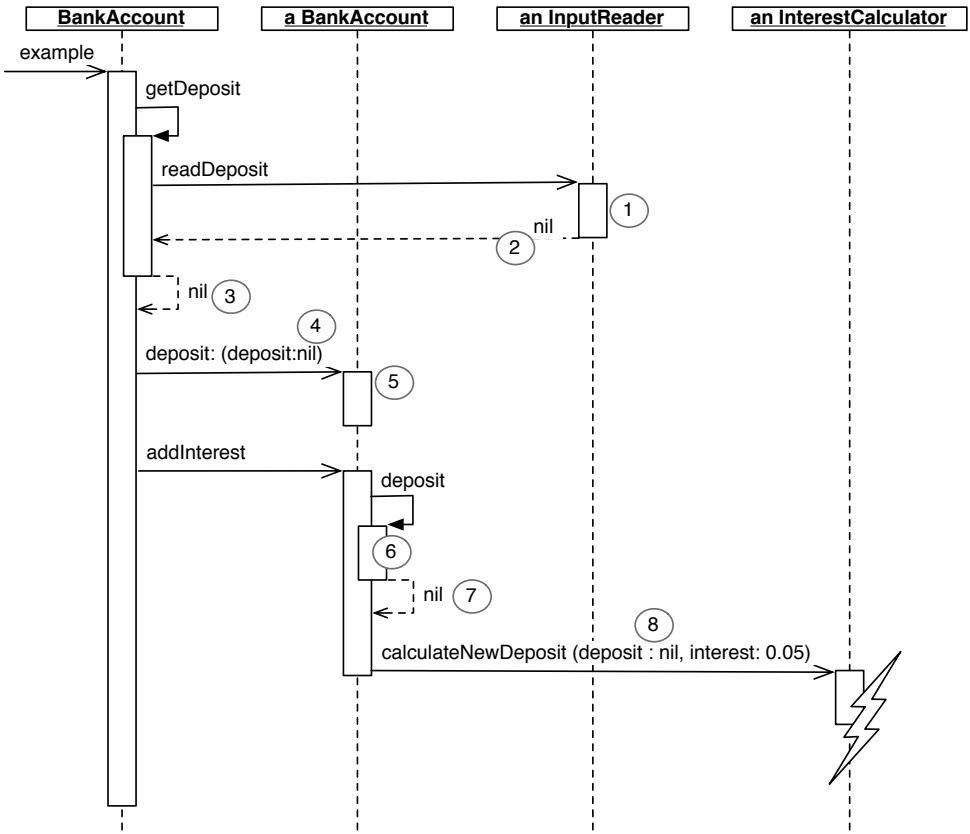

Figure 1.1: Sequence diagram of the crashing example

The error is that the InputReader returns nil, which is then assigned to the deposit field of the account. This is why the program crashes later when the InterestCalculator tries to compute the new deposit. The developer notices the error when a message is sent to nil, then he has to find the source of it, which lies in readDeposit. With a back-in-time debugger it is possible to search the past execution for the error source. The question is what steps have to be taken to achieve that. First the developer notices that the value of deposit is nil, so he could reason about whether this value even is allowed for the deposit, because if it is, someone may have forgotten to check deposit for nil. In this case he will decide that nil is not a valid value (a deposit always should have an amount), so he has to backtrack where nil came from. The following enumeration shows how nil was passed from its source up to the failing code. Each step contains the description how and where nil was referenced. Those steps are also illustrated in the sequence diagram in Figure 1.1.

1. Originated in an InputReader>>readDeposit
2. Returned to BankAccount>>getDeposit
3. Returned to BankAccount>>example
4. Passed as argument to a BankAccount>>deposit:
5. Written into field deposit in a BankAccount>>deposit:
6. Read from field deposit in a BankAccount>>deposit
7. Returned to a BankAccount>>addInterest
8. Passed as argument to an InterestCalculator>>calculateNewDeposit:

This enumeration is the flow of the object nil in the context of our error. With the previously mentioned ODB, it is possible to find the bug. The problem is that in order to backtrack the nil value, it is necessary to investigate the source code. The developer has to identify some of the steps we listed above manually by additionally looking at the source code of executed methods. Of course in this small example this takes not much time, but in more complex applications where there are larger methods it is not always obvious how an object was passed around. The ODB allows the developer to view the state of an object, the history of its fields and where they were set, but it does not provide information about how an object was passed to a specific location in the execution. The possibility of stepping quickly through the flow of an object is missing, and also jumping directly to the location where an object was

instantiated is not possible. This can make navigating the execution history an exhausting task.

The Unstuck debugger already goes a step further by adding an object history, which provides a list of everything that ever happened with a specific object. This means, every access to an object, every time it was passed to a method, every time it was written into a field and so on. This is already an improvement, because it can be seen easily that the particular steps of nil in the above example are part of this object history. In this particular example the history of nil is even the same as the object flow we need to investigate to find the bug. But in general the problem is that there is too much information in the history. The flow of an object in a specific context is a subset of the object history, but identifying it can be hard, because a lot that could have happened with an object is not of interest for that flow. To clarify this point let us have a look again at the flow of the nil object listed before. Between the write in 5 and the read in 6 theoretically nil could be referenced thousands of times, and all of these references would occur in the object history and disturb the search of the origin of nil in the context of our error. So Unstuck does not provide the flow of objects automatically either.

To summarize, the flow of objects is one very important element that is missing in other approaches. We argue that letting the developer see these flows can make the navigation of the execution data more efficient.

1.1.2 Limited Views

Another problem with current approaches is that they present the execution data and in particular the method trace in inconvenient ways, which means if the trace is big it is difficult to get a clear view on it. Figure 1.2 shows an example of the method trace in Unstuck. The method traces are handled similarly in other existing back-in-time debuggers. We need other views of the method trace that make it easier for the developer to understand what happened in the underlying execution. E.g., in Figure 1.2 it is not easy to visualize the flow of an object. In general, existing back-in-time debugging approaches are lacking high-level visualizations of the data that could in some situations support the general understanding of the system, which can significally improve the detection of bugs.

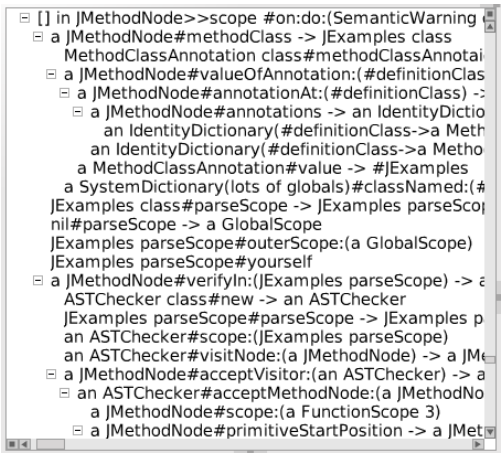

Figure 1.2: Method trace view in Unstuck

1.2 Overview of Our Approach

With back-in-time debugging there is always the problem that masses of data like huge method traces must be handled. As discussed in the last section, we are focusing on improving techniques to navigate this data. The goal is to always let the developer know where he is, and where he could go to get closer to the source of the bug. That's why we have given our implemented tool the name *Compass.* We implemented Compass in Squeak, an object-oriented Smalltalk dialect.

The goal of Compass is to be a back-in-time debugger that provides efficient navigation through the large amount of data that is generated during program execution. To achieve this objective, we define a different approach. Debuggers like the ODB are *state-centric* approaches, which provide the past state at different points in time. We focus additionally on how objects are passed around in the system, so Compass is a *flow-centric* back-in-time debugging approach. Therefore, the main contribution of our work is to provide the first debugger that captures the flow of objects and makes this flow accessible to the developer.

We separate the execution data into two main entities, the execution trace (executed methods and statements) and all the objects created during the execution and their state history. We create a navigation model that defines these two entities as spaces. We research ways of how these spaces can be explored rea-

sonably, and what relationships between the spaces exist. The goal is to find ways of navigating in and between those spaces such that the developer can get an understanding of the execution data effectively, which is crucial for the debugging process.

With this model in mind we can then explain how our debugger is built. It consists of views that present the execution data and are linked in a way that they satisify the navigation techniques proposed in the model. Besides standard features like the source code of the selected method execution or the evaluation panes for objects, the debugger introduces new views and functionality. The method trace is shown as a tree in a fisheye view that improves the navigation of the method trace. The flow of objects can be selected and viewed in different ways. A graph shows how the relationships between objects changed during the execution of a method to let the developer explore side effects of parts of the execution. The control flow dependencies that had influence on an execution of a method are also provided.

1.3 Thesis Structure

In Chapter 2 we present an overview of existing approaches and debugging techniques. In Chapter 3 we research ways to navigate the execution history and build a model for a debugger with them. Then we discuss our implementation in Chapter 4. We validate our approach with several example scenarios for our debugger in Chapter 5 and present conclusions and possibilities for future work in Chapter 6. In Appendix A we provide a quick start guide for installing and using our debugger.

Chapter 2

State of the Art in Debugging

First we have a look at the recent progress in back-in-time debugging, then we describe other debugging approaches that contain interesting ideas for our work.

2.1 Back-in-Time Debugging

Different approaches have already been developed concerning back-in-time debugging. We describe the most interesting approaches here.

Omniscient Debugger (ODB). The ODB is a debugger presented in 2002 which traces the execution of a java application by instrumenting the bytecode at load time [Lewi03]. It allows the user to step through the generated trace and to inspect the state of objects at any time. The user can step through the history of fields of objects to see what values they had at certain points in time and where they were set. So the debugger has the very basic functionality each back-in-time debugger must provide. Figure 2.1 shows a screenshot of the ODB.

Basically all information of the execution history is there, but as we already have shown in Section 1.1, this information is not presented to the developer in a way that allows efficient navigation. We called the ODB a state-centric

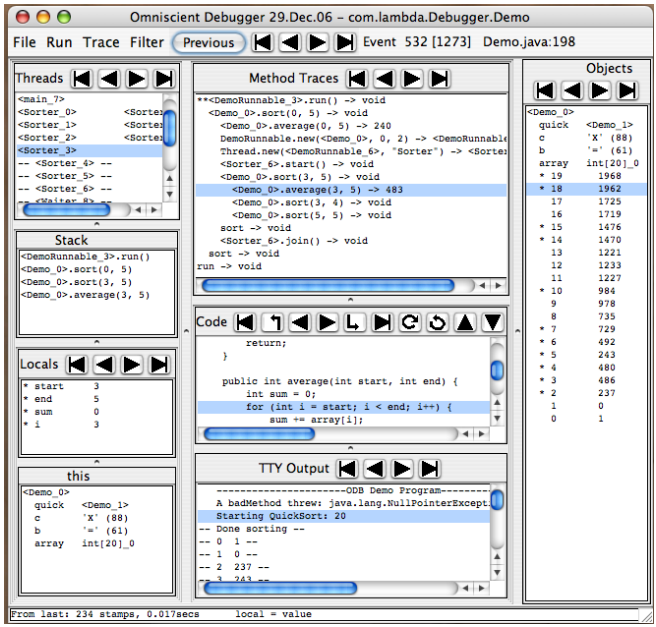

Figure 2.1: ODB

approach, that allows the developer to examine the state of objects and the history of the state. But it is not possible to see automatically where an object came from or to jump to its allocation. We also mentioned the problem of missing high-level views that would ease the navigation of the execution data. All in all the ODB has proven the concept of back-in-time debugging very well, but does not provide very much additional help for navigation.

Unstuck. Unstuck is a back-in-time debugger developed in Squeak [Hofe06]. It is very similar to the ODB but adds features like a simple query language for filtering execution events, the history of an object and the possibility to color appearances of certain objects for better recognition. These features already improve navigating the execution history, but we also have shown in Section 1.1 that Unstuck neither provides the flow of objects nor high level views.

Trace Oriented Debugger (TOD). This debugger was inspired by the ODB but mainly addresses the problems of performance and high memory requirements, because these are the main reasons why back-in-time debug-

ging is not established in commercial products yet [Poth07]. Pothier et al. show that the scalability of such debuggers can be improved by using a high-speed distributed database backend that allows effective storage and querying of events.

Concerning the user frontend, the TOD provides some standard views, like an object inspector or a control flow view. These views allow one to step through the execution and to see where certain values of fields have been set. Additionally, the developers already add a high-level view of the data. In particular they created so called event *murals*, which are graphs that show a number of events per unit of time. These events can be calls to methods of a particular object or calls to a particular method. The goal of these *murals* is to create a view out of a very large number of events that allows the developer to identify abnormal behavior patterns.

The problems with the TOD are similar to the ones with the two previous approaches, but at least it provides some high level views.

Reversible debuggers. The approaches presented up to now all work by collecting events to present the user information from earlier states of the execution [Koju05, Lieb98], but they do not allow the developer to pick a point in time where he wants to restart the execution, or only in a very limited way. The objective of reversible debuggers is to go a step further and let the developer not only explore the execution history, but rather let him go back to earlier points in time to restart the execution from there. This is achieved by reverting step by step every executed statement.

The problem with these approaches is that reverting the memory into a state where the execution was before is rather tricky. Also, they do not really contribute to efficiently navigating around an execution history.

Commercial back-in-time debuggers. There are already some commercial products involving back-in-time debugging. CodeGuide is a commercial Java developing environment which has integrated its own back-in-time debugger. But the debugger is rather limited and does not provide many ways to navigate the execution data. RetroVue is another commercial back-in-time debugger that offers some more possibilities to view the data. It implements the most common functions similar to those of the ODB. But neither of those debuggers provides any more than these basic functionalities.

2.2 Other Debugging Approaches

Until now we have presented some existing approaches in back-in-time debugging. Now we have a look at some other debugging techniques, some of which might be very interesting to combine with back-in-time debugging and therefore have inspired features of our approach.

Program Slicing. When an error occurs in a program execution, we always have to search for its source. That means we have to go backwards from the statement where the error occurred and search there for the possible sources. But mostly a big piece of the source code does not even influence the infected statement at all, so actually you wouldn't have to search for the error in those parts of the application. The problem is that for developers it is hard to see which code is or is not affecting a statement.

This issue is addressed by a technique called *program slicing* [Weis81, Zell05, Ko04], which for a particular statement (called the slicing criterion) finds a set of other statements that might influence it, and therefore excludes the statements that absolutely can't affect it. This set of influencing statements is called a slice. There are two possible ways of getting it. Static slicing uses no other information than the underlying source code. This makes determining the slice very hard, and usually the resulting slice of a slicing criterion tends to be very big, because lots of statements might possibly influence it but do not necessarily in an execution. Dynamic slicing does its analysis on an actual program run [Kore88], which means it uses the execution trace to determine the slice, and therefore we can get exactly the statements that influenced the slicing criterion in the underlying program run. As in back-in-time debugging we record the execution trace, we can use it to determine dynamic slices.

Mark Weiser has shown in a study that people break down code into slices when they are debugging [Weis82], so it seems to make sense to design our debuggers in ways that help people identify those slices more efficiently.

Automated Debugging. An other idea is to automate the debugging process [Clev00]. This means the programmer should only have to do very few but necessary steps to prepare the debugger, which then should use some algorithms to automatically isolate the defective code statements. Andreas Zeller has in particular developed an approach called *delta debugging*. The goal of this technique is to isolate defective code by comparing the program run that is failing

with one that has once worked. The debugger is automatically testing what changes have led to the error until only a minimal set of changes remains. The only thing the developer has to do is to provide a test function that indicates if the error has occurred or not.

There are different applications of *delta debugging* on different domains of program execution, e.g., input, code or program state. In the first one, an application crashes with some input that worked in an older version (e.g., the *gcc* compiler crashes when it tries to compile some source code). Now the debugger isolates the part of the input that is responsible for the error by changing the input bit by bit and always running the program and applying the test. The result is a minimal subset of the input that makes the error happen [Zell02b]. Applying *delta debugging* to code is similar: there are two different versions of a program, one that fails and one that used to work. Now the debugger incrementally changes the code from one version to another and always executes it and applies the test. The result here is a minimal subset of the code that makes the test fail [Zell01]. Finally the same can be done with program state, comparing the state of a passing and of a failing run and narrowing down the state to a small set of variables that are responsible for the failure [Zell02a].

There are several problems with this approach: first you always need a version of a program that once worked in addition to the current version. Second, it may not always be that easy to write the test that checks if the bug occurs or not. Another problem is that the automatic testing may take a long time, depending on the complexity of the program and the number of differences of the two versions. All in all automated debugging surely makes sense in some situations and for some types of bugs, but there are also several drawbacks.

Query-Based Debugging. In large applications, there is a huge number of objects, and their relationships can be very complex and therefore hard to understand for a developer. The approach of query-based debugging addresses this problem by letting the developer explore the object space with queries [Lenc97, Lenc99]. While running a program, the developer can define a search domain and conditions that make up a query, which then returns all the elements of the domain for which the conditions hold true. For example, he can define the domain as two classes and set a condition that returns true if the two instances of those classes reference each other. The query then returns all pairs of instances of the classes that satisfy the condition. This can help the user find defect object relationships, e.g., when he knows that certain objects should not reference others.

The problem with this approach is that you have to know what relationships could be defective. When an error occurs the developer has to find the source of the bug, but sometimes he does not know at all which object relationships could be defective, so he has no clue what query he should write. Another problem is that such queries can become slow if the search space is big; as there are sometimes thousands of instances of some classes, testing the conditions for all tuples of these classes can be very expensive.

Nonetheless query-based debugging can be a helpful tool for finding bugs, but still needs to be combined with other tools to be effective. A very practical application of it can be program understanding, where the user can use queries to find out how the objects generally are related to each other. It can help the developer to analyze frameworks by doing queries on applications of it.

Graph based debugging. There are other approaches that deal with object relationships. Tools like the Data Display Debugger (DDD [Zell96]) or Fujaba [Geig02] support debugging by allowing the developer to explore the program state. They provide graphical front ends to present data to simplify debugging, e.g., by visualizing the state with graphs. Sometimes with such graphs the developer can see much better how objects are related to each other than with common object explorers. Problems are of course the big amount of objects normally existing in a system, so it is necessary to delimit what is actually viewed.

Bug patterns. Another interesting approach allows the developer to search for bugs even on the static level. The idea is that a lot of bugs follow a common pattern [Hove04, Wasy07]. An example of a common bug pattern is that Java developers often forget to override the hashCode() function when they override the equals() function, because that way it is possible that two objects are equal but their hashes are not, which can lead to problems, e.g., in maps. There are a lot more of those patterns. With static analysis of the code (in some newer approaches also dynamic analysis is used) and taking into account possible bug patterns, some potential error sources that may be hard to find by hand can be detected in large programs. An advantage of this approach is that it can be used for finding potential errors in systems with large amounts of code, even if the error has not yet led to a crash of the application. The disadvantage is that it misses bugs that are complex or that do not satisfy a common pattern.

2.3 Summary

We have presented several back-in-time debugging approaches in this chapter. The problem of the ODB, Unstuck, the TOD and other approaches is that although they provide all the necessary data to find bugs, navigating this data is hard in large execution traces. This makes finding complex bugs a difficult task. Even though Unstuck provides the history of every access to objects, it has no information about how objects were passed to a specific location in the execution. Furthermore, views like the method execution trace are not very practical in all of these approaches.

We have looked at other debugging concepts. Dynamic slicing can be used to reduce the trace to the statements that are of interest to the developer, while query-based and graph based debugging reveal relationships of objects. These and other concepts contain interesting ideas that can be integrated in back-in-time debuggers.

To summarize, our key observation is that the existing back-in-time debugging approaches are state-centric. This is the main problem we address in this work.

Chapter 3

Approach

In Section 1.1 we discovered problems of state-centric back-in-time debugging approaches. Exploring the data of the execution of a large object-oriented program can be hard. In this chapter, we therefore develop new ways of exploring this data.

To achieve our goal we investigate what kind of data is available when debugging. In a first step we design a model which allows us to see how the different data entites are related. This reveals what kind of navigation can be used to explore the data in order to find bugs. The next step is then to define a debugger using this model, to create views from it, and to link the views intuitively.

3.1 Exploring the Execution History

First we have to think of what kind of information we get when debugging. On one hand the executions of methods or even statements are recorded, which define the control flow. On the other hand we have the objects, and how they evolve during the execution. These are the two main entities that appear in a programs execution. We look at those two entities as two different spaces, the *control flow space* and the *object space*. First we search ways to navigate within each of the spaces. Then we need also to find relations between them, to make navigation between those two spaces possible. This is clarified in Figure 3.1. The plane at the top represents the control flow space, showing the method execution events. The bottom plane is the object space, showing the objects

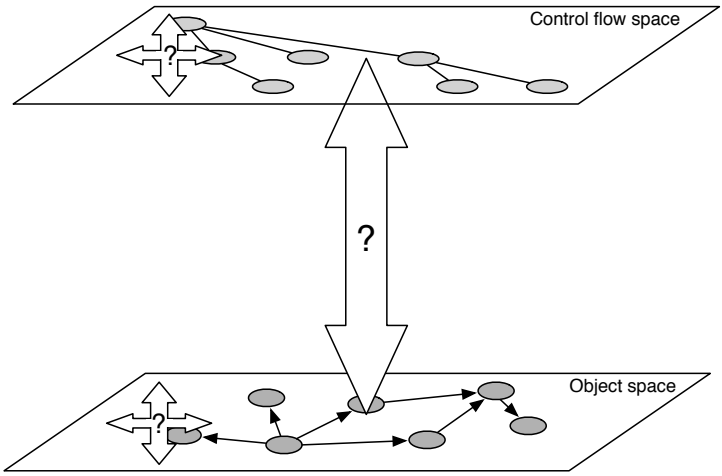

Figure 3.1: Two spaces: Control flow and objects

and how they are related. The figure points out again what our main objective is, namely to find effective ways to explore the execution data.

In the following sections we have a closer look at each of the spaces.

3.1.1 The Control Flow Space

The control flow of a program run is defined as the order and nesting in which the statements were executed. In our approach we mostly focus on the executions of methods and the statements that influence the execution of other statements. Figure 3.2 shows the control flow of the bank account example presented in Section 1.1. On the left the method execution trace is shown and on the right there are the bytecode instructions of one of the method executions. We show these two levels of detail separately because sometimes we are only interested in the method execution trace rather than in the whole statement execution trace.

So in the control flow space there are different entities like the execution of methods and statements. Figure 3.3 shows more formally what those entities in the control flow space are and how they are related. Method executions are the dynamic part of the metamodel. They activate methods that consist of statements. The methods and statements are the static part of the metamodel.

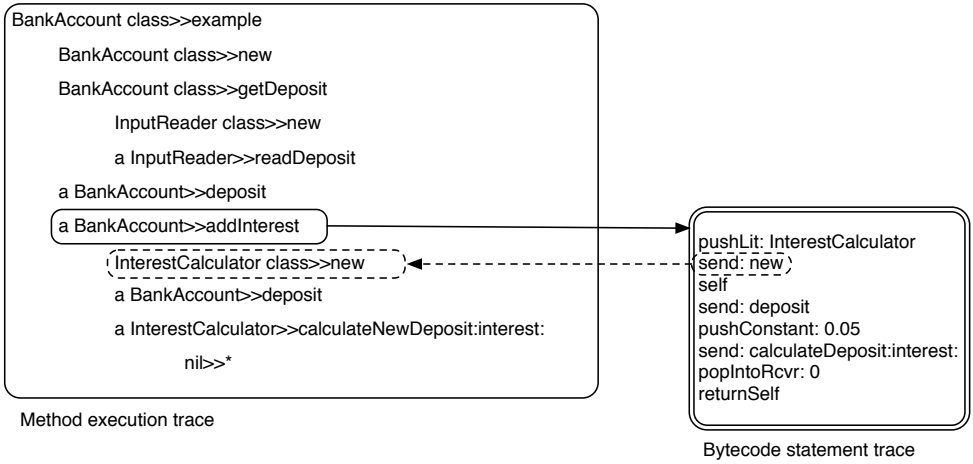

Figure 3.2: Trace of a program execution

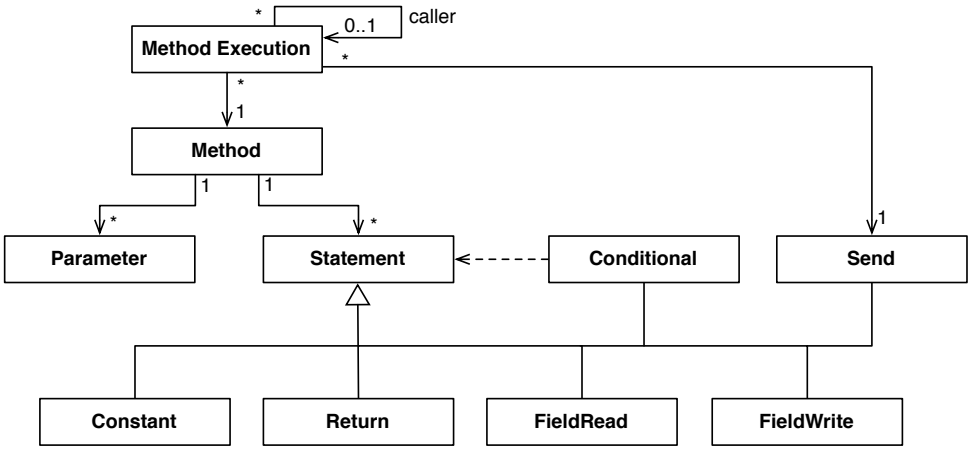

Figure 3.3: Metamodel of the The control flow space

For simplicity reasons they are not modeled as an abstract syntax tree here, but simply as a list of statements per method.

One type of statement that has a special relationship to other statements is a conditional statement, also called control flow statement. It influences the continuation of a program by evaluating a condition and is therefore important for the control flow. As methods are executed by the send statement, they are also dependent on these control flow statements.

We now want to find out what possibilites there are to explore this execution trace.

Stepping the trace. An obvious way to navigate the execution trace is to simply step through it. That means step over statements (forward or backward) or step into or out of method executions. In Figure 3.3 the possibility of stepping into a method execution is given by the relationship of the send statement and the method execution. Stepping out is defined by the method execution pointing to its caller.

Stepping the trace gives the developer the possibility to reproduce what happened during the program execution. It is a feature that is provided by almost every back-in-time debugger.

In our model we do not have the executed statements (we modeled the statements statically). As we only have the method executions and the aliases, we do not really step over statements, but rather over the aliases. That means when we are stepping in our model, not really all statements that were executed are taken into account, but only the ones that led to the creation of an alias or that triggered a message send.

Control flow dependencies. We now propose to additionally use control flow dependencies as a possible mean for navigation. This means we take into account special statements that influence the control flow, the control flow statements. In the metamodel in Figure 3.3 this is shown by a dependency from the conditionals to the other statements. If the boolean value that was taken by such a statement wouldn't have had that value, a certain subtrace would not have been executed. In some cases it may be of interest to show the developer this dependency, so he can directly navigate from a method in the subtrace to the method where the conditional statement was executed. Some bugs can be found quicker using that kind of navigation, because if the value that is used

in a conditional statement is wrong because of an error, the developer doesn't have to check all the statements that were executed after the conditional and the method he was looking at when he saw the dependency.

To view an example of this, remember the bank account example from Section 1.1. The method InputReader>>readDeposit just returned nil (Which is the source of the error), now let's see a little modified version:

```
InputReader>>readDeposit
    ↑ self isReadyForInput
        ifTrue: [ self readInput ]
        ifFalse: [ nil ]
```

Let us assume that in the example isReadyForInput returns false because we forgot to initialize the InputReader properly. We later try to send a message to the nil object that is returned from readDeposit, which leads to the error. This message send is dependent on the condition in readDeposit, because it controls to which object we send the message later. By inspecting the dependency we can get faster to the source of the error. This topic is related to dynamic slicing introduced in Section 2.2. Often when a developer looks at a statement, he wants to find the parts of the program that have influenced its execution, namely he is interested in the particular dynamic slice, and one thing that influences this dynamic slice are the conditionals mentioned before.

3.1.2 The Object Space

Next we introduce the object space. It is defined as the full object history, which means it includes the current state of the objects and all states before that. The space uses the alias model proposed by Lienhard [Lien06, Lien08a]. Additionally to the object history this model provides the flow of objects that allows a developer to backtrack where an object came from at a certain location in the program execution. The flow of objects is captured by the creation of *aliases*. Aliases represent reference transfers of objects, e.g., when an object is passed as an argument to a method, an alias is created representing the argument reference. The aliases are chained, i.e., each alias points to its *origin*, the alias that represented the previous reference. These chains make up the flows of objects. Figure 3.4 shows the entities of the object space, namely aliases and objects and how they are related. There are different kinds of aliases for different kinds of references. The write alias is created when an object gets stored in the field of another object. It points to its *predecessor*,

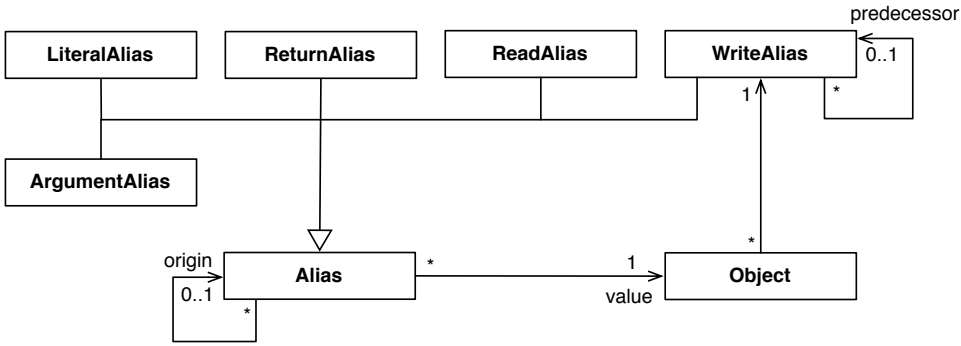

Figure 3.4: Metamodel of the object space

the write alias of the object that was stored before in that field, which makes it possible to retrieve the historical state of an object.

As we have defined the model of the object space we now find different ways to explore it.

Object relationships. One way of navigating the object space is to investigate the relationships of objects, by exploring the field pointers of an object to other objects. In the metamodel in Figure 3.4 this is defined by the relationship between objects and write aliases. By exploring object relationships the developer can step by step reveal how they are connected to each other. Often the user has some specific objects in his focus where he can concentrate on. To facilitate this process there are techniques for exploring relationships automatically, e.g., the developer may want to know if there exists a path from one object to another. Another useful information is which objects have been referenced in a certain time period. Figure 3.5 shows the relationships between a bank, an account of that bank and some other objects.

Object state history. Another way of investigating the object space is to track the changes of objects. In Figure 3.4 this is defined by the write aliases that point to their predecessor. By using these predecessors the history of a specific field can be explored. Using the history of all fields of an object, it can always be reverted to its state at a certain point in time. The whole history of the state of objects can be explored that way. Again, the developer will mostly

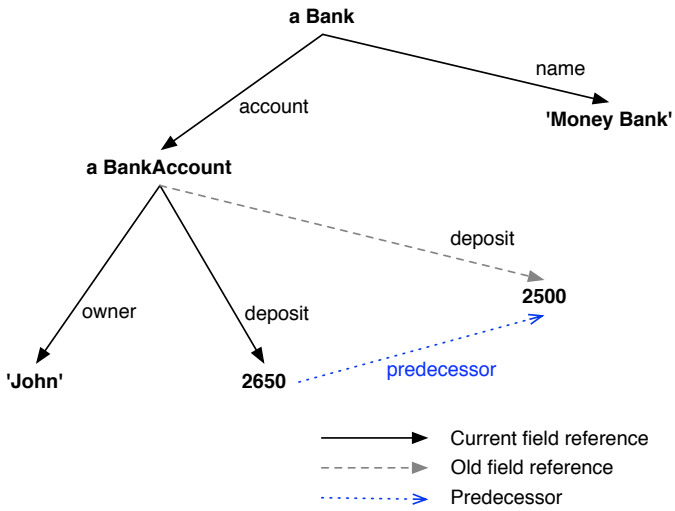

Figure 3.5: Object relationships and history

concentrate on one or more specific objects. For example he wants to know how an object has changed between two points in time, which can be provided automatically. Figure 3.5 shows an example of the object state history, as the amount of the deposit changed there. The object 2650 was written into the deposit field of the bank account and replaced the object 2500.

Object flow. The flow of objects defines how objects were passed around in the system. In Figure 3.4 this is modeled with the alias pointing to its origin alias, that is, its previous reference transfer. We can use this feature for navigating in the object space. E.g., when the developer is looking at a field of an object, he often is interested how this object got there; he wants to know how it was passed around in the system before it was stored in the field. By starting at the alias that is saved in the field and following the chain of origins he gets exactly that information. What the developer does in this example is to go backwards, by navigating in the opposite direction from where an object came and following it to its root. We refer to the path he goes as the *backward flow*. But going in the other direction is interesting too. At a certain point in time the developer might be interested in what happened afterwards with a specific object. The question is where that object did go after a certain point in time. We call this the *forward flow* of an object. Of course the handling of the

forward flow is somewhat more complicated than the handling of the *backward flow*, because there is not just one single path of aliases but rather a tree. E.g., an object in a method execution can be passed as an argument to two other method executions, which results in two paths that need to be examined if one wants to know what happened to the object.

As an example concerning the flow of objects, we refer to the flow of nil in the bank account example presented in Section 1.1.

3.1.3 Mappings Between Spaces

In the previous sections we presented possibilities to investigate the control flow and the object space. The next important step is to find out how to navigate *between* those two spaces. The goal is to get from one space into the other by using reasonable mappings between the entities of the spaces. Figure 3.6 combines the two metamodels of the spaces and illustrates how the entities of the two spaces are related. In the following we will explain the mappings. Figure 3.7 illustrates the mappings with an object diagram of an executed method and the aliases that were created during the execution.

Alias to method execution. As Figure 3.6 shows, one mapping from the object space to the control flow space that comes naturally is a direct relation of the aliases and the method executions, because every alias has a reference to the method execution it was created in.

Figure 3.7 illustrates this. Two aliases were created during the execution: one that represents the constant 2500 and one that represents the assignment. They both are connected to the method execution.

Alias to statement. Aliases can even be mapped more precisely by directly associating them with to statements that led to their creation. This means that an alias can always be mapped to a statement, which is also shown in Figure 3.6.

As each object flow is actually represented by a set of aliases, the flow can always be mapped to a set of method executions, respectively a set of statements. By examining the flow of objects one follows the method executions and statements where the object was passed through. There may be other sets of aliases that could be of interest to know their corresponding statements, like all aliases of

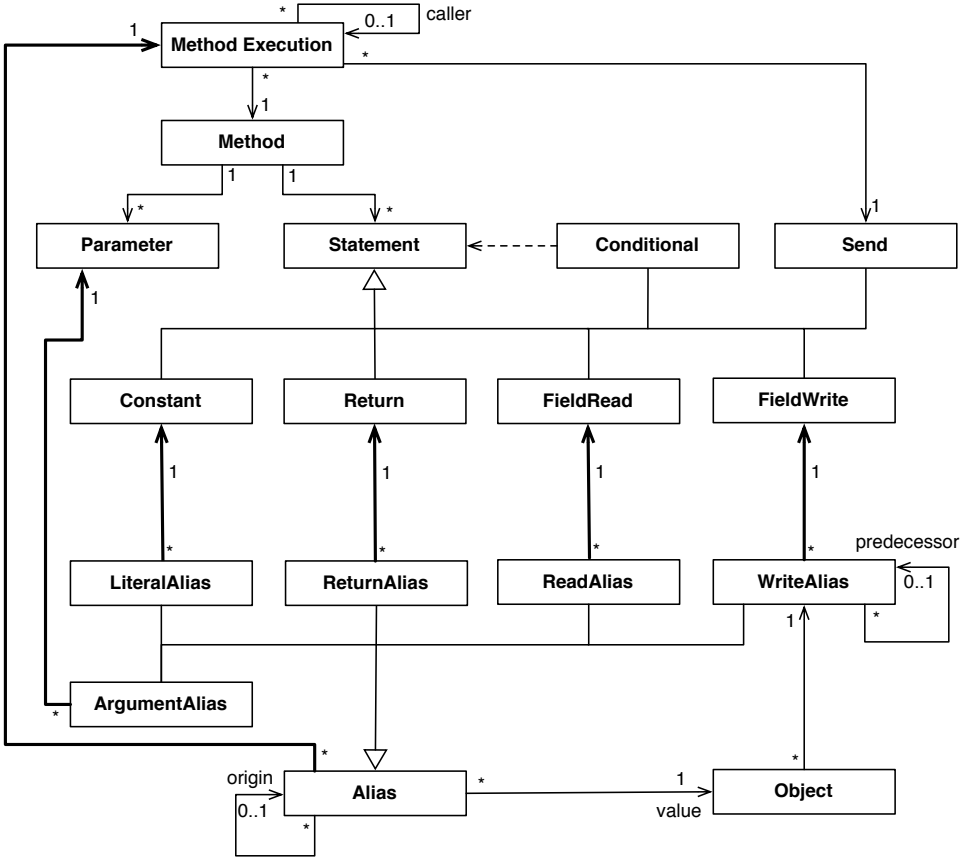

Figure 3.6: Full metamodel

one object, which results in all the method executions and statements where this object occurred.

In Figure 3.7, the aliases point to their corresponding statement, which are the assignment statement for the write alias and the constant statement for the literal alias.

Created aliases in method execution There are also meaningful mappings from the control flow space to the object space. Each method execution can be mapped to a set of aliases, namely the aliases which were created in it. In the metamodel in Figure 3.6 this is defined by the relationship between the alias

and the method execution. With this information one can see which objects were referenced during the execution of a method.

In Figure 3.7 the method is connected to all aliases that were created during its execution.

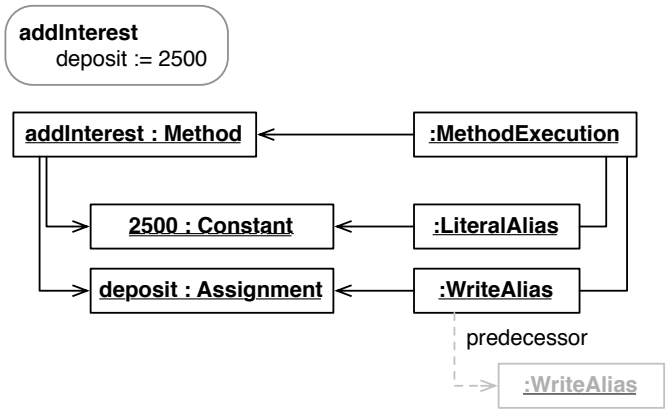

Figure 3.7: Relationships between control flow and object space entities

Side effects. What a developer often is interested in are the side effects of a specific part of the trace. That means he wants to know how the relationships (defined by field references) between objects changed during a certain time period. Commonly it is good practice to select this period as the time interval of a method execution. Then we can map the subtrace which consists of the set of methods that were executed during that period to a set of objects and aliases that represent the side effects during that time.

In Figure 3.7 the write alias that was created points to its predecessor, the value that was saved in the deposit field before. So the side effect of the method in the example is that the deposit field points to a new value.

3.2 Designing the Debugger User Interface

Now that we have elaborated techniques for exploring the execution data we can come up with a design for our flow-centric debugger. We are going to define

several views that display the execution data and combine them to a functional debugger. First we give an overview of the debugger by shortly introducing each of its views shown in Figure 3.8.

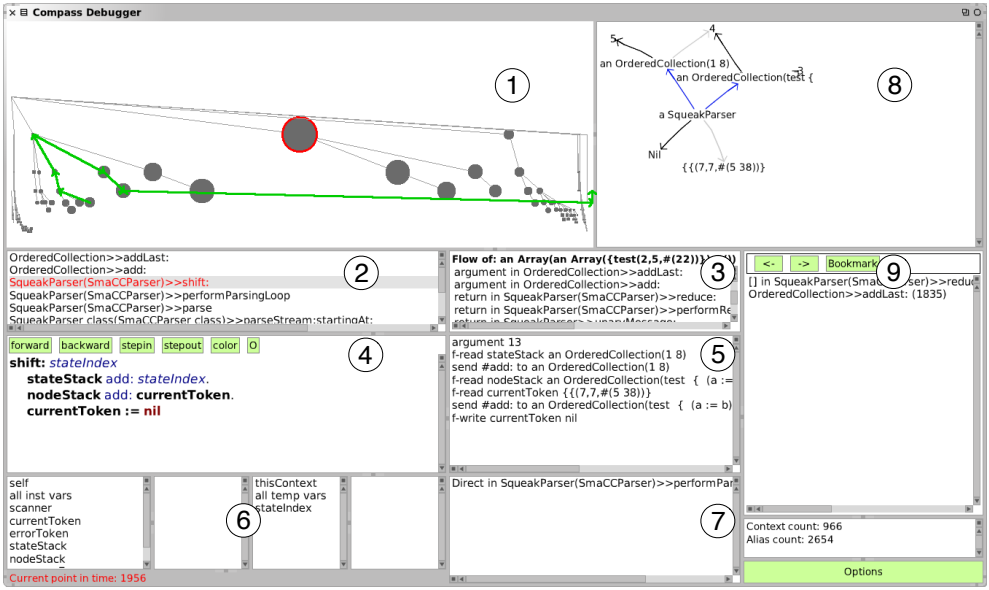

Figure 3.8: Compass debugger frontend.

1. The top view is the visualization of the execution trace, representing the method executions as circles.

2. This list shows the execution stack of the currently selected method execution.

3. This list contains each reference transfer of the flow of an object selected by the developer. One way to select an object flow is by double clicking on an alias in the sub-method statements list (see 5).

4. This pane shows the source code of the currently selected method execution and provides possibilites to step through the execution trace.

5. In the sub-method statements list the aliases and message sends are shown that occurred during the method execution.

6. The four panes at the bottom let the developer view the values of variables.

7. The dependency pane shows if the selected method execution is depending on any control flow statements.

8. In the side effects graph pane, the changes of relationships of objects in the subtrace of the selected method execution are shown.

9. The navigation history allows the developer to step back to method executions he has been before, or to step forward again. It is also possible to bookmark method executions in order to stick to important locations.

A very important characteristic of the debugger is the synchronisation of the views. In most of the views the user can interact with the data, but the focus is always exactly at a specific point in time, and all the views are adapted to it. E.g., when the developer selects an alias in an object flow, the debugger jumps to the method execution it was created in and selects the statement in the source code and sub-method pane where the alias was created.

A feature that was motivated by the Unstuck debugger was the coloring of objects. In several views there is the possibility to assign a color to an object. Every time the object appears in a view, it is shown in the color that it was assigned. This is a good visual aid for the developer that makes it sometimes easier to understand the execution data.

All the views presented before and their correlations must be understood and handled by the developer when debugging. We now have a closer look at each of the views, how they are connected to each other and how they integrate the navigation techniques we discovered in the previous section.

3.2.1 Method Trace

In a back-in-time debugger one must be able to navigate the method trace quickly. In the problem section (Section 1.1) we mentioned that in other back-in-time debugging approaches good visualizations of the method trace are missing. Generally they visualize the trace simply as a tree list, which becomes huge when a large application is traced.

In Compass we decided to have two views to visualize the trace, one graphical view that contains the full method trace (1 in Figure 3.8), and a list that shows only a slice of the trace (2), namely the execution stack of the currently selected method execution (this is actually the execution stack that is provided by standard debuggers). Figure 3.9 shows an example of the two trace views, applied to a run of our bank account example (we slightly extended the example to get

a bigger trace here). The circles in the full method trace visualization represent method executions. The edges connect them with their parent execution. The elements are ordered from left to right by their start timestamp and from top to bottom by their depth in the stack.

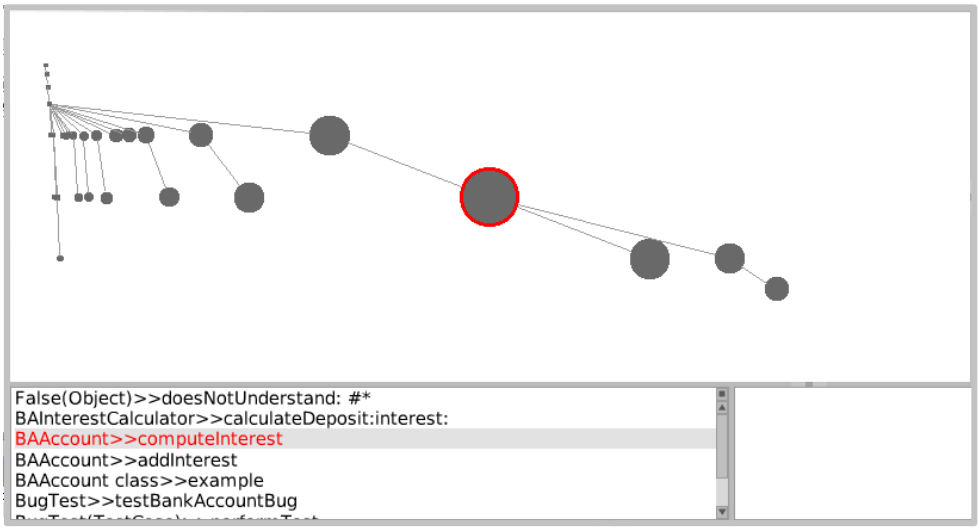

Figure 3.9: The method trace views

The idea of this visualization is to generally have every single method execution on the screen, but to accentuate the part of the trace that is in focus at the current point in time. That is why it is shown as a *fisheye view*. The focused execution is in the center of the screen and is marked by a red border. Now the distance on the time axis between two executions that lie nearer to the focus is much bigger than distance between two executions that are far away from the focus. The ones that are farther away also have a smaller size. The same holds for the depth axis. That means that method executions with a small time and depth difference to the focused execution are well visible on screen, while the executions with a big time and depth difference are rather near the bounds of the screen (but still visible). We will see how this visual representation can help the developer when we cover the flow objects later in this chapter.

A disadvantage of this view of the method trace is that the labels of the method executions are not shown, because there is not much space left for them. Hence it cannot be seen directly which circle represents which method execution. We

address this problem by showing the name of the method execution when the mouse hovers above its circle. Another aid is the previously mentioned coloring of objects. When an object has a color assigned and is the receiver of a method execution, its circle is drawn in that color.

Another problem is that the parent of a method execution can be far away from the current focus, but often it is important to see what the parent is. This is why additionally to the full trace, the current execution stack is also shown in another view. Like in other debuggers it is visualized in a list. That way the developer can easily navigate the chain of parent executions.

By clicking on a circle in the full method trace or in the execution stack, the currently focused method execution can be changed.

3.2.2 Method Execution

The next two views we are discussing are the source code pane (4) and the sub-method pane (5), which represent the currently selected method execution. The first pane shows the source code of the method, the second contains a list of the aliases that were created during the execution (In our model this is the *created aliases in method execution* mapping). When an alias is selected, the corresponding statement in the source code is selected (this is the *alias to statement* mapping). Additionally to the aliases, also the message sends are contained in the sub-method list and are also highlighted in the code pane when selected.

In the source code pane there are buttons to step through the execution trace. We defined stepping the trace in our model in Section 3.1.1 (*Stepping the trace*). The buttons let the developer navigate in and out of methods, and over statements.

3.2.3 Control Flow Dependencies

We have now seen the basic possibilities to navigate the execution trace. In our model we also identified the *control flow dependencies*. These are statements that have influenced the control flow. In the dependency panel (7) there is a list of control flow statements that have led to the execution of the currently selected method. When clicking on a dependency, the debugger jumps to the method execution where it was executed and selects the control flow statement in the source code. There are two different kind of dependencies: direct and indirect

dependencies. In a direct dependency, the control flow statement is executed in the execution stack of the method execution. In an indirect dependency, there is a control flow statement that has influenced the object flow of the receiver of the focused method execution. Indirect dependencies are included because a control flow statement that has influenced the flow of the receiver is responsible that exactly that object has been sent the message. Figure 3.10 illustrates this. The method on the bottom right is the focused method execution we need to get the dependencies. The arrow shows the flow of the receiver of the method execution. In the stack where the receiver was coming from there is a control flow dependency so we have an indirect dependency. Also, in the execution stack of the focused method execution itself there is a control flow dependency, which is a direct dependency. Let us review the example shown in Section 3.1.1.

```
InputReader>>readDeposit
    ↑ self isReadyForInput
        ifTrue: [ self readInput ]
        ifFalse: [ nil ]
```

Here we have an indirect dependency, because the if-statement is responsible that nil is returned and later used as receiver for a message, instead of another object.

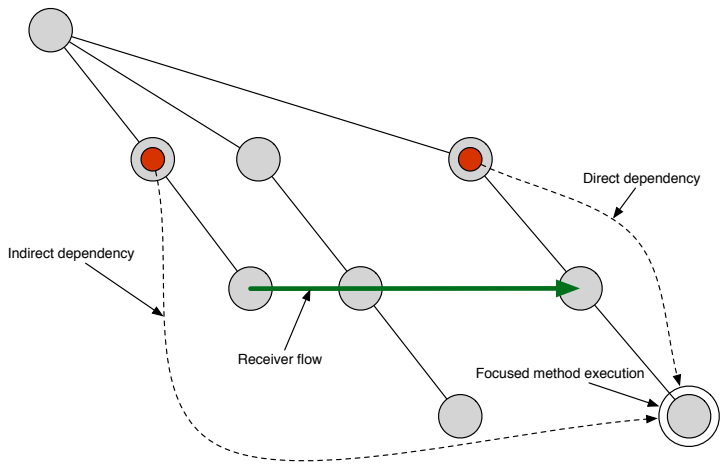

Figure 3.10: Direct and indirect control flow dependencies

3.2.4 Objects

In the model section we defined different possibilities to navigate the object space. One was the exploration of the *object relationships*. The 4 panes at the bottom of the debugger (6) show the state of the receiver object of the currently selected method execution and the local variables. They allow the developer to inspect the values and therefore to explore the object relationships. The states of the objects are always reverted to the currently focused point in time, so the *object state history* is taken into account. The history of a specific field in a specific object can also be viewed.

Our approach is flow-centric, so one of the most important features of the debugger is its ability to show the flow of objects (See *object flow* in the model section). In different views that involve objects the developer can tell the debugger to show how the object was passed to the current location in the execution trace. There is a view that lists the single steps (aliases) of the object flow (3). When selecting one of the aliases, the debugger jumps to the location where it was created. Additionally the flow is visualized in the method trace view. Figure 3.11 illustrates this by showing the flow of nil in the bank account example. The green arrows show through which method executions the object

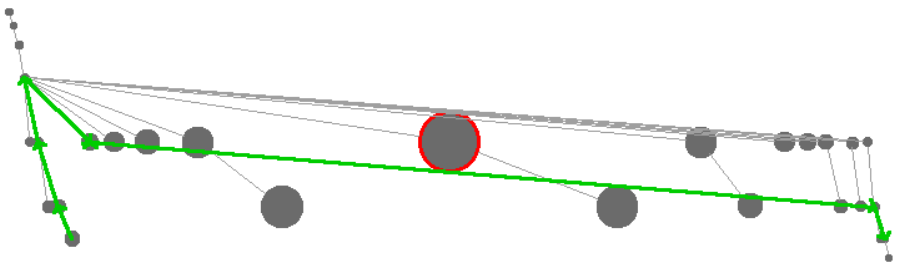

Figure 3.11: Object flow visualized in method trace

passed. One of the benefits of our representation of the method trace is that this flow can be interpreted very easily. When an arrow connects two method executions that are not in a parent-child relationship, this means that the object was written into a field and then later read from it. When an arrow points from a method execution to its parent, this generally means that the object

was returned. Often an object gets returned over several method executions, like in the bank account example at the beginning. Finally, when an arrow goes in the opposite direction, i.e., from a method execution to a child, this generally means it was passed as an argument (In some rare cases, it could also be an object written to a field and read from it in the child execution). In our example this was nil that was written into the deposit field, and later read from it. To summarize, we can say that with this visualisation, the developer can get a good feel of how an object was passed through the system. This leads to a better understanding in comparision to looking at the object flow as a list of aliases.

The object flow covered before was the backward flow of an object. We also defined the forward flow, which is used to determine what happened with an object after a specific location in the execution trace. When the developer chooses to see a forward flow of an object, a window pops up that contains the aliases of the forward flow in a tree list view. At the beginning only the root element of the tree is shown, which is the alias that represented the object at the point in time the forward flow was shown. The developer then can expand the tree to see what different paths the object has taken after that point in time. When an alias in the tree is selected, the backward flow starting at that alias is shown in the debugger. Figure 3.12 shows an expanded forward flow of an object. After the object is instantiated and returned, it takes three different paths. It is twice given as an argument and returned once.

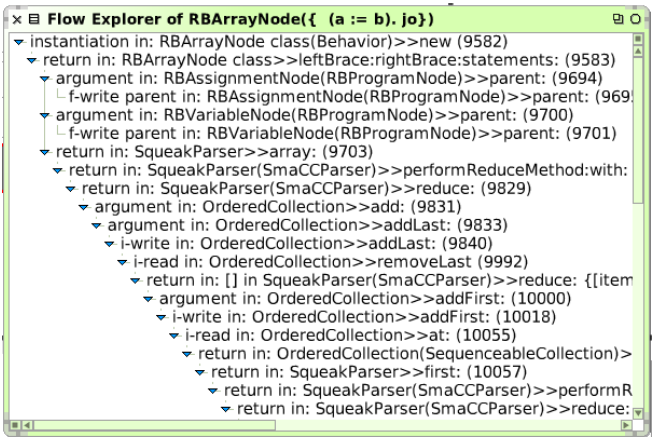

Figure 3.12: Forward flow example

3.2.5 Side Effects Graph

In the model section we have shown a mapping from a subtrace of the execution to the object space, the *side effects*. These are the effects that a specific subtrace has on objects, which means how the relationships of objects changed during the execution of the subtrace.

Those changes are visualized by a graph, similar to the visualization of side effects in [Lien08b]. The nodes of the graph are the objects that lead to the side effects. There are differently colored edges, representing either a newly existing field reference from one object to another, a field reference that is no longer existing, or an object that was accessed from an existing field reference. In the debugger, the side effects graph gets always updated for the subtrace of the currently selected method execution.

Figure 3.13 shows an example of the side effects graph applied to the subtrace of a method execution that added an interest to the deposit of a bank account. The black edge points from the bank account to the newly computed deposit

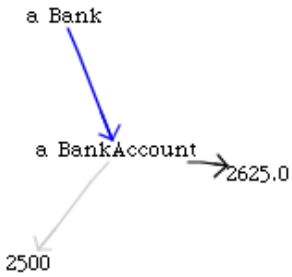

Figure 3.13: Side effects graph example

(2650), the light gray edge points to the deposit (2500) that was saved before and is now not referenced anymore by the account. Additionally, a blue edge points from a bank object to the account. This means that during the execution of the subtrace, the account object was read from a field of the bank object. We added these read accesses to the graph because they can be helpful to reveal relationships between objects. In this example we not only see that the amount of the deposit of a account changed, but also that it is connected to the bank to which it belongs.

The names of the fields themselves are not contained in the graph, but by right-clicking on an object, all its newly set fields show up in a list, together with the

objects that were stored in them. By clicking on a field in this list, the debugger jumps to the location where the field was assigned the new value.

Finally, viewing the side effects can be useful for a better comprehension of the whole system, which leads to a more effective error search.

3.2.6 Navigation History

Lots of bugs can be found more easily with a back-in-time debugger compared to a standard debugger. But sometimes the bugs can still be very complex, and require the developer to do much exploration in the execution trace. Parts of the execution that involve the error (as source or consequence) may be distributed arbitrarily. To find the real source of the error, the developer must sometimes check these parts to get understanding of the error. Hence he must move around a lot in the execution trace, but has to remember important locations, so it is easy to get lost. That is why Compass integrates a navigation history. It lets the developer do simple things like going several steps backward (and forward again), similar to a navigation in a web browser. By stepping back or forward, the debugger does not only jump to the method execution that was selected before, but also selects the statement that was selected when leaving the method execution. As mentioned the developer has to remember some locations that are important, so another feature is to bookmark method executions to quickly get back there if necessary.

Chapter 4

Implementation

In this chapter we discuss the implementation of our debugger. This involves the altered Squeak VM (the Object Flow VM [Lien08c]), which is responsible for gathering the execution data. This data is then analyzed and presented to the developer by our debugger front-end written in Squeak. In the following sections we cover details of the implementation of the virtual machine and the debugger and some of its components like the fisheye trace view.

4.1 The Object Flow VM

The Compass Debugger is implemented in Squeak, an open source Smalltalk. Common Squeak applications run on the Squeak Virtual Machine [Inga97]. Our approach, however, needs an altered version of this VM, namely the Object Flow VM [Lien08c]. This approach addresses the problem of the memory usage by discarding information that is not needed anymore. It is implemented at the VM level to make use of the garbage collector to efficiently delete unneeded historical data. The VM captures and manages execution data in a very efficient way and provides important data like the flow of objects, but lacks a user interface. For this work we created such a user interface, but we also had to extend the VM to add more detailed information required for our user interface. In the following we have a closer look at this virtual machine.

4.1.1 Motivations for extending the VM

There are several reasons in favor of gathering execution data at the level of the VM. An important point of working at this low level is that no source or bytecode of the traced application needs to be modified in order to fetch execution data, which means that no recompiling or bytecode manipulation is needed. Moreover, the operations can be implemented directly and very efficiently, which is favorable for performance reasons. Another benefit that comes with the VM is that one can make use of the garbage collector to throw away execution data that is no longer needed.

As Squeak is open-source, everyone has full access to the implementation of the virtual machine and can change whatever is needed. A great feature is that it is written in a subset of the Squeak language itself, which is then translated to C. That means that also the implementation of the VM is mostly platform independent, so the changes to the VM only have to be done once. The so called Blue Book [Gold83] provides a detailed description of the design of a Smalltalk-80 VM, which is mostly equivalent to the Squeak VM.

However, there are some difficulties one has to deal with when altering the VM. The main disadvantages are that one has to know very well how the virtual machine works, which is not always trivial, and that altering it can be very tricky and produce bugs that are hard to track down. Every change to the VM has to be evaluated carefully to check for possible unwanted side-effects.

4.1.2 Changes to the VM

In this section we summarize how the Object Flow VM works. For a more detailed explanation see [Lien08c].

The Object Flow VM captures the execution history by implementing the *alias* model. In the Squeak VM references to objects are implemented as direct pointers to these objects. Now the main change to the VM is that such pointers do not point anymore to the objects themselves, but rather to an *alias* object which represents a reference to the object and has itself a pointer to the object. Figure 4.1 illustrates how fields point to objects in a standard and in the altered VM. In the Object Flow VM, the field points to a *write alias*, which is created when an object is written to a field. The VM always creates aliases when an object is referenced in some way, e.g., when an object is returned by a method execution, not the object itself but rather a *return alias* that represents the

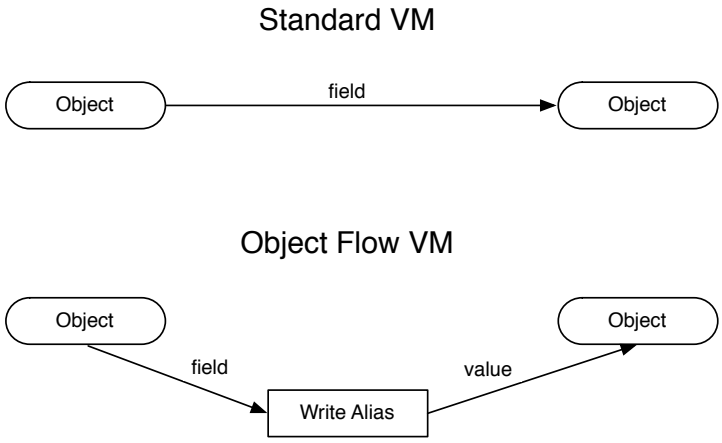

Figure 4.1: Object pointers in VM

reference to the object is returned. In Figure 4.2 there is an overview of how aliases fit in the common object relationship model and Figure 4.3 shows all kinds of aliases that can be created.

There are some properties that must hold in the VM. Firstly, the aliases should be transparent at the application layer, which means e.g., they have to forward the messages they receive to their corresponding object. The aliases exist only in the VM to track execution history; in the application it seems as if they are not there. So the aliases are not allowed to have any influence on the program flow, but there are ways to access them later to reconstruct execution history. A second property that must hold is that the old model of the VM still has to work, so collecting the execution data and therefore the creation of aliases can be turned off without causing any problems.

The aliases not only have a pointer to their corresponding object, they also must contain information to model the history of object state and the object flow. Basically each alias contains the timestamp of its creation, the method or block execution it was created in and the program counter of the statement that was responsible for the alias creation. As the aliases point to their method execution, the method trace is given by all the method executions that are kept in the system.

The state history is modeled with a special field used by write aliases that stores

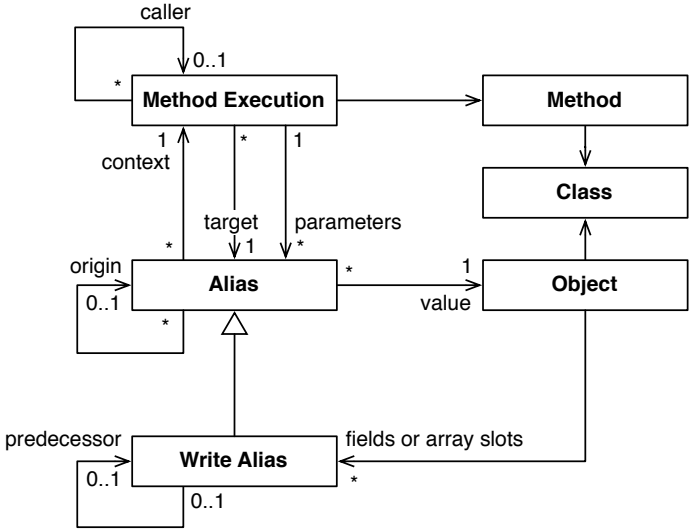

Figure 4.2: The object flow metamodel [Lien08c]

their predecessor, i.e., the object that was stored in the field before the new object was written into it. That way, the state of an object can be restored at any time by going through the predecessors of the aliases stored in the fields of the objects and picking the predecessors that were stored at the desired time.

Each alias additionally contains a field that stores its origin, the alias that was there when the reference was transferred. Every alias has an origin except for the *allocation alias*, which is created when an object is instantiated, and the *literal alias*. With help of the origins the flow of an object can be reconstructed. If the developer needs to know how an object was passed to a certain location in the execution trace, he has to start at the given alias and backtrack the origins up to the allocation alias, which eventually results in the full backward flow of the object.

This implementation of the alias model allows the virtual machine to garbage collect also data of the execution history. Because aliases are just normal objects, they get garbage collected as soon as there are no other objects referencing them anymore. That means, as soon as an object in an application is not referenced anymore, all its aliases are not referenced either and they get garbage collected and all their referenced predecessors, origins and method executions

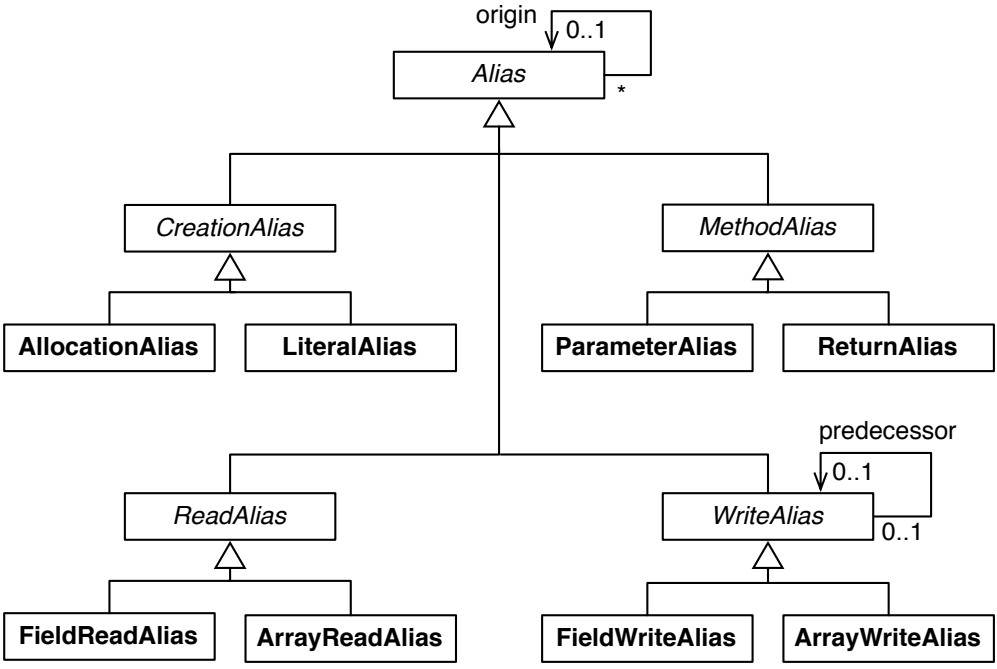

Figure 4.3: The alias class hierarchy [Lien08c]

too, if they are not otherwise referenced. This decreases the amount of data that must be kept in memory, which is also an advantage for our debugger, as the amount of data we have to navigate is diminished.

Besides the changes to the VM itself, some additional code is needed in the image that is used with the new VM. This code contains the definition of the aliases and provides possibilites to access them.

4.2 The Compass Debugger

In this and the following sections we present some details of the implementation of the Compass debugger. We describe the debugger itself and in detail the implementation of some of its components that need more explanation.

4.2.1 Debugger Details

The Compass debugger inherits from the standard Squeak debugger, for several reasons. As mentioned in Section 4.1.2, the method executions themselves are used to keep the method trace in memory. The Squeak debugger can already handle those method executions, which makes things easier for us. Moreover, it already adds a source code pane and has possibilities to map byte code instructions to the source code text. Additionally it has panes for evaluating variables of the current method execution and its receiver. To summarize, there are two main advantages of extending the standard debugger: firstly it spares us some work, because it already provides views our debugger needs anyway (with some adjustments), and secondly it makes a part of our debugger look like the standard debugger, which is good for new users, as they see something familiar.

4.2.2 Source Code and Sub-method Pane

In this section we discuss the implementation of the panes introduced in Section 3.2.2. As mentioned before the source code pane is taken from the original Squeak debugger. This pane shows the code of the currently selected method execution. Compass adds another view that contains the sub-method data for this method execution. This view is a list of the aliases and message sends that have occurred during the execution of the method. To create this list, all aliases

that point to the selected method execution are combined with all method executions whose senders are the selected method execution and ordered by there timestamps. When one of those aliases or sends is selected, the corresponding statement in the source code pane is selected. This is done with the help of a map provided by the Squeak debugger, which maps single byte code indexes of a method to its corresponding source code text. As the aliases and method executions save the program counter where they were created respectively called, this map can be used to get the selected source text.

4.2.3 Fisheye Method Trace

The method trace presented in Section 3.2.1 is a special view that displays all method executions. The idea is to accentuate the currently selected method execution and its neighbours but to not exclude completely the ones that are farther away. This goal is achieved by introducing a fisheye view [Furn86]. The method execution that currently is in the focus of the debugger is always in the center of the viewing rectangle, and according to its position in time and depth the positions and sizes of the other method executions are computed. In Figure 4.4 there is an example of a fisheye projection.

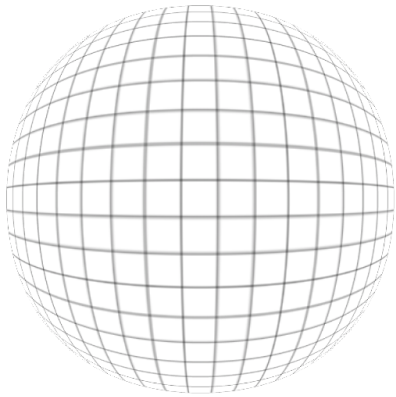

Figure 4.4: Circular fisheye projection

Our view differs a bit from this classical fisheye projection, because the x values are computed independently from the y values, as we will show in the following.

In the method trace view, a circle represents a method execution and an edge

always connects it with its sender. The position on the x-axis represents the point in time the execution of the method started, and the position on the y-axis is the depth in the stack trace.

The position of an element is computed in the following way. Definition (4.1) shows how a distance factor f_t is computed that represents how far away a method execution is from the focused method execution on the x-axis. The variable t_{focus} is the time of the focused method execution and t is the time of the method execution whose position we want to compute. C_t is a constant with which we can control how quickly the method executions go towards the border of the viewing rectangle.

$$f_t = 1 - \frac{1}{|t_{focus} - t| \cdot C_t + 1} \tag{4.1}$$

Using this formula, f_t is always in the interval $[0, 1)$. The more f_t tends to 0 the nearer the method execution lies to the focus. If the factor was 1 it would mean that the method execution would be infinitely far away, so it can actually never take this value. In (4.2) the factor is now used to compute the exact position on the x-axis of the viewing rectangle, whereas w is the width of this rectangle.

$$x = \begin{cases} \frac{w}{2} + f_t \cdot \frac{w}{2} & (t \geq t_{focus}) \\ \\ \frac{w}{2} - f_t \cdot \frac{w}{2} & (t < t_{focus}) \end{cases} \tag{4.2}$$

To see an example of this, let us assume the viewing rectangle has a width of $w = 100$ (which means that the x value of the focus is $x_{focus} = 50$), the time of the focused method execution is $t_{focus} = 10$ and we have three other method executions with times $t_1 = 8$, $t_2 = 6$ and $t_3 = 4$. Let's additionally assume that $C_t = 0.5$. Applying the computations we get $x_1 = 25$, $x_2 = 16.5$ and $x_3 = 12.5$, so the results show clearly that although the time differences between the method executions are always the same, the distances between them become much shorter as they are farther away from the center, and they go rather fast towards the border of the viewing rectangle.

The computation of the y value of the position is computed analogously so we do not show the formulas here. Instead of the time values simply the depth values of the method executions are used to compute a distance factor and the computations are done on the y-axis of the viewing rectangle. What still needs to be computed is the size of the circles that represent a method execution. We

simply take the distance factor of the time axis to compute the size, shown in (4.3), where s_{focus} is the size of the focused method execution.

$$s = (1 - f_t) \cdot s_{focus} \tag{4.3}$$

This equation simply means that the farther away a method execution is from the focus, the smaller its size.

In the above equations we actually have two values that can be varied, the constant C_t used in (4.1) and s_{focus} used in (4.3), which is actually also a constant. Compass offers options to change those two constants, which allows the developer to adjust the distance between two method executions and the size of the method executions.

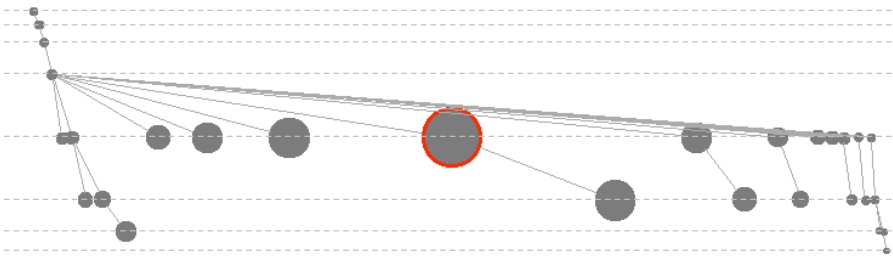

Figure 4.5: Fisheye trace view

We call our view a fisheye trace, but as mentionend, it is rather a pseudo fisheye view because we do not actually use classical methods to compute it, and so some other properties hold on our view. For example, it shows the whole space of method executions, but additionally all method executions with the same nesting level remain aligned on the y-axis. This is because the x and the y values of the method executions are independent from each other, and so if the depth of two method executions is the same, the y value of their center is the same. In Figure 4.5 there are lines which show that method executions with the same depth are always on the same line, other than in Figure 4.4, where lines are projected as a curve.

4.2.4 Side Effects Graph with Mondrian

As shown in Section 3.2.5, the side effects graph is used to view side effects in the subtrace defined by the currently selected method execution, i.e., it shows how the relationship between objects changed during the execution of the method and all its child method executions.

The graph is implemented using Mondrian [Meye06], a tool for scripting visualizations in Squeak. We extended Mondrian to do the layout of the graph by calling GraphViz[1] externally. GraphViz is a tool that uses powerful algorithms to do graph layouts with good performance. Mondrian then uses that layout to draw the graph.

As the determination of the nodes and the edges and the computation of the graph layout require quite some time for large subtraces, the generation of the graph is done in a background process. The graph gets updated in the debugger as soon as all computations are done.

The nodes of the side effects graph represent objects that are modified or pointed to, e.g., objects that have new field references and objects that are now referenced by fields are both contained in the graph. Edges represent either new field references from one object to another, field references that do not exist anymore or objects that were accessed through a field reference of another object. The nodes and edges of the side effects graph are computed in the following way.

- First the subtrace of the current method execution has to be determined. We also need the start and the end timestamp of the subtrace. The start timestamp is simply the timestamp of the selected method execution. Concerning the end timestamp, the last method execution of the subtrace must be determined and the timestamp of the last alias in this execution is the desired end timestamp.
- To find out what new field references were created, all write aliases that were created between the start and end timestamp are taken into account. For each of these write aliases, two nodes are inserted in the object graph, if they are not already in there. One node is the target object of the alias which is the object that was written into the field. The other node that is inserted is the owner of the field, which is the receiver of the method execution of the write alias.

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

- Now one edge is definitely inserted, leading from one object to the other object now contained in its field. Additionally, there's a check if the write alias has a predecessor. If yes, an edge is also created to this object with a light gray color, to indicate that the object is not referenced anymore.
- As mentioned also the field read accesses of objects are included in the graph. This is done by running through all read aliases between the start and the end timestamp. Similarly to the write aliases, the read object and the owner of the field are inserted into the object graph, if not already there, and a blue edge is added pointing to the read object.

4.2.5 Control Flow Dependencies

In this section we show how values in conditionals and loops are detected that have influenced if a method was executed (We introduced the control flow dependency pane in Section 3.2.3). We only detect dependencies for complete method executions, not for single statements, because finding those dependencies inside of one method can be done rather easily.

In the following we describe the steps to find the dependencies of the method execution that is in focus. The analysis starts at the parent method execution of the focused method execution. This parent is searched for control flow statements that have influenced the execution of the focused method. All found dependencies are kept in a list. Then the same is repeated with the next parent method execution and recursively for the whole chain of parent method executions. That way the direct control flow dependencies in the execution stack of the focused method execution are collected.

We also defined indirect dependencies, which are the control flow statements that influenced the receiver of the focused method execution. These dependencies are found by traversing the object flow of the receiver. For each alias of the flow its method execution is taken and the same procedures as described above are applied to that execution. That way all indirect dependencies are collected.

The execution data that is available to us does not suffice to find the conditional dependencies. It only contains method executions and aliases, but no trace of the executed statements is recorded. Therefore in order to find conditionals, a static code analysis needs to be done. This analysis is done at the bytecode level, by using the intermediate representation of the method provided by the NewCompiler package. The intermediate representation contains all instructions

of the method, which we now search for *JumpIf* instructions, because those jumps have influence on the control flow. E.g., if-conditions in the source code are implemented with those jumps in the byte code, but also loops with while conditions. Thus by analyzing directly on the intermediate level, we get all control flow dependendencies by simply taking into account those jumps. When such a conditional jump is found, it is checked if the child method execution in question is affected by the jump, if yes, the conditional is added to the list of dependencies. E.g., if a child method was executed in the block of an if-condition, a dependency is added.

If a method execution is dependent on a control flow statement, then the method executions in its whole subtrace are also dependent on that statement. In our debugger it is possible to jump to that statement from every of those depending method executions. Unfortunately, rather than only to jump to that statement, we would rather like to directly select the aliases that were used to create the condition. That way it would also be possible to automatically create a dynamic slice, which could be used to mark code that has affected a certain method execution. The way it is implemented now, the developer can jump to the conditional, and then has to explore by himself what value was used by the conditional statement.

Chapter 5

Evaluation

In this chapter we show two examples that demonstrate the capabilites of our debugger for a efficient search of bugs. We solve the problems with the Compass debugger and discuss what advantages we had in comparision to other approaches.

5.1 Object Flow Example

This first example demonstrates how the object flow can be used effectively to track down a bug where a defect object was passed around. We added a little modification of a method in the NewCompiler package to infiltrate an error that is hard to track down.

Problem. The IRBuilder of the NewCompiler package is used to generate bytecode. We created a small example method buildTruncaterMethod that uses the builder to generate a method that simply sends the message truncated to the receiver. The code of the method is shown in Listing 5.1. After building the method (1), the compiled method gets returned (2) and is used in the last line to execute an example (3). We now modified the initialize method of the IRBuilder so the created IRMethod is in a faulty state. This modification does not have any consequences until the method is compiled (2), but then an error occurs because a message was sent to nil. Examining the execution stack in a common debugger does not reveal why the problem occurs.

Listing 5.1: Compiling a method with the IRBuilder

```
buildTruncaterMethod
    | irMethod compiledMethod |

    "1. Create intermediate representation of method"
    irMethod := IRBuilder new
                    numRargs: 1;
                    addTemps: #(self);
                    pushTemp: #self;
                    send: #truncated;
                    returnTop;
                    ir.

    "2. Compile method"
    compiledMethod := irMethod compiledMethod.

    "3. Use method"
    ↑ compiledMethod valueWithReceiver: 3.5 arguments: #()
```

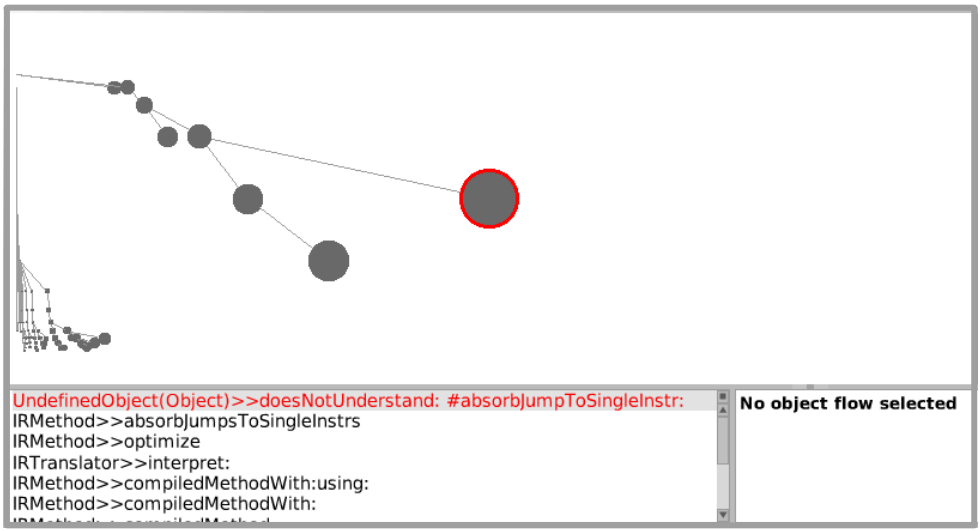

Figure 5.1: Opening the debugger after error occured

Solution. When we run buildTruncaterMethod, an error occurs and we open our debugger. 237 method executions and 625 aliases were recorded during

the execution. Figure 5.1 shows the initial state in the method trace. We see that doesNotUnderstand was sent to UndefinedObject (nil). When we step out of doesNotUnderstand, we get to the parent method execution and see in the code and submethod pane (Figure 5.2) that absorbJumpsToSingleInstrs: was sent to nil, which obviously is an error. We now want to know what the reason was that nil was there to take the message instead of another object. Therefore we

Figure 5.2: Method execution where error occured

tell the debugger to show how the object was passed to this statement. We do this by right clicking on the message send statement and selecting the flow of the receiver from the menu that pops up. The debugger visualizes the flow of nil in the method trace and in a list (Figure 5.3). In the flow list we can see

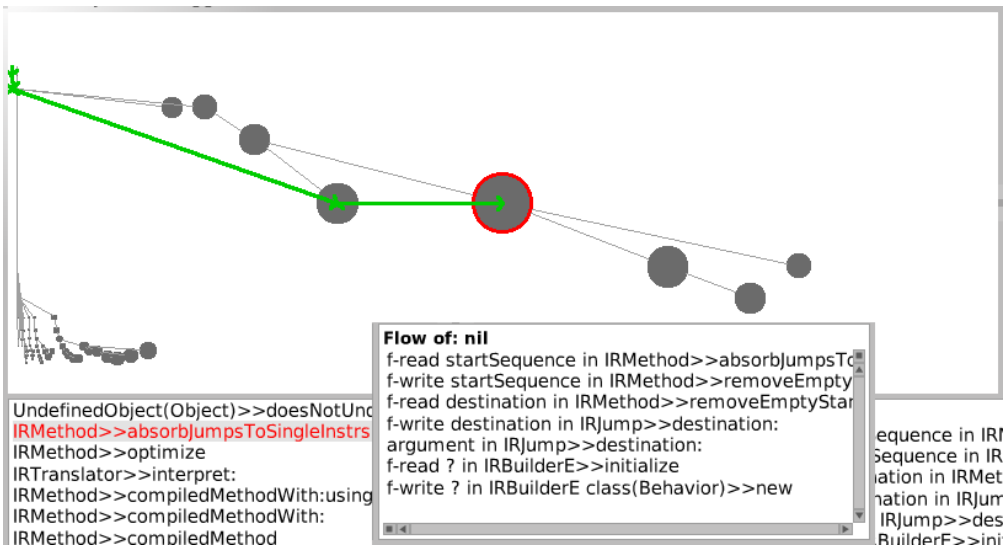

Figure 5.3: The flow of nil

the complexity such a flow can have. The object was stored in more than one

field and passed several classes. We can now step back through the flow. When we look at the flow, what attracts our attention is that the root of the flow was a field write in a new method. This means that nil was written into an instance variable of an object at the time the object was created. This is the standard initialization of an instance variable. So we can conclude that the nil object we are dealing with results from an instance variable that was not properly initialized. The next thing that happened to nil in the flow was that it was read from that uninitialized instance variable in the initialize method execution of the object that contains the variable. As this is the first time the instance variable is used, we jump to that statement and try to see if we get some clues in the method execution of the statement. In Figure 5.4 a part of the method is shown. As the statement of the flow is selected, we see that currentSequence is the variable in question. We also see that this variable gets assigned a value one line below, and this is actually the reason for the error. The variable gets assigned its value too late, for the program to work correctly, the two lines must be exchanged.

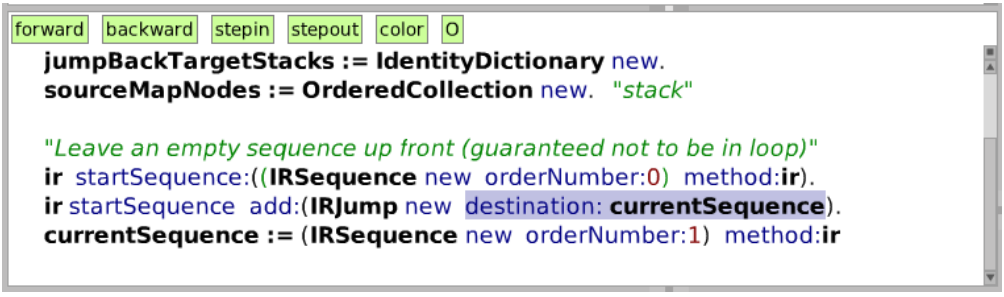

Figure 5.4: Defective method

Discussion. This first example shows how the object flow can ease the task of debugging when provided automatically. Between the source of the error and where the syptom occurs, more than 200 methods are executed. With a normal stack-based debugger, the developer must read code without any further information or restart the debugger with breakpoints. With our debugger, when one wants to know where an object originates from, like in this case nil, he can quickly browse the flow. In the example nil was written and read from three instance variables and was passed through three different classes before it arrived at the error's location. This is what makes the example harder to debug with other approaches, because to get to the origin of nil, one has to

do a lot more manual investigation. In other debuggers (e.g., the ODB), when the error occurs, the developer sees that the value was read from an instance variable and then can ask where it was assigned, so the first step back he can do very easily too. But then he has to find out where the value came from, so he has to read the code. Then he will find out that it was read from another instance variable and goes to the assignment of it, and so on. Step by step he can reconstruct the object flow, but firstly he needs time for that, and secondly he can not go to the root of the flow directly, which was essentially the key for solving this problem.

As seen in this example, it can be effective to go directly to the root of an object flow. In this case this revealed the problem rather quickly, and there are likely lots of other problems where this is the case too. So it might be recommended to first have a look at the root of the flow, but of course the problem can lie somewhere else in the flow, so when nothing is found at the root, we have to traverse the object flow to go on with our search. The question is then, if it is better to start at the end of the flow or at the root. Probably the developer has to evaluate by himself what way he wants to go. E.g., when the flow is very long because the object in question is used very frequently without causing problems, it might be better to start at the end of the flow because the error is probably rather local. But in the end it is always the developer who has to measure where he wants to start with the error search.

To summarize, Compass and especially the object flow gives the possibility to find the error quickly in this example. Nevertheless we need to add some remarks. Even with the flow, if the developer does not focus on the right parts of it, he may need the same amount of time as with another approach. And finding the error source can be a lot harder than in this example. Let's say we didn't exchange the two lines of code but rather just delete the assignment. Without further knowledge of the NewCompiler the developer would have found that the variable was unassigned, but he would not be able to fix it without further analysis.

5.2 Multithreading Example

When more than one process was traced, the method trace simply has more than one root method execution. The flow of objects still can be visualized in the method pane and therefore reveals how objects are passed around between processes. The following example demonstrates an error that occured in a real

multithreaded application. We created a simpler example of it that contains the same error.

Problem. The application is used to download software updates from the internet and install them concurrently. This task is handled by two classes, the UpdateDownloader, which downloads the updates from the internet, and the UpdateLoader, which installs the downloaded updates in the Smalltalk image. The UpdateDownloader is started in its own process. When it has downloaded an update, it puts the code that is needed to install the update into a shared queue. The UpdateLoader, which is running in another process, reads from the queue and waits until an element is added. When this happens, it reads the element from the queue and uses the provided code to install the update. This is done until the downloader puts a special delimiter symbol into the queue, meaning that everything has been downloaded. The two following listings show the code of the two processes.

Listing 5.2: The update loader process

```
[ this := docQueue next.
  nextDoc := docQueue next.
  nextDoc ~= #finished ] whileTrue: [
    Compiler evaluate: nextDoc.
    docQueueSema signal ].
```

Listing 5.3: The downloader process

```
self getUpdates withIndexDo: [ :code :i |
    aSemaphore wait.
    aQueue nextPut: i.
    aQueue nextPut: code ].
aQueue nextPut: ''.
aQueue nextPut: #finished
```

The symptom of the error in this example is the update loader process indefinitely waiting on the empty queue. The application is frozen as long as the update loader is waiting for a new element in the queue. By interrupting the current process a debugger can be opened. Examining the current execution stack shows simply where the update process is stuck. Somehow the download process seems to have stopped putting anything into the queue, but when completing the last download, it should have put the delimiter symbol into the queue. The question is now why the download process did not conform to the

expected protocol. Examining the stack and reading the source code does not help reveal the problem.

Solution. We start Compass, about 1000 method executions and about 3000 aliases were recorded. In the beginning, the debugger shows the last execution of next, where the process is waiting for an element in the queue. We step out and get to the method execution where the updates are loaded. The code and submethod panes of this execution are shown in Figure 5.5. Looking at the

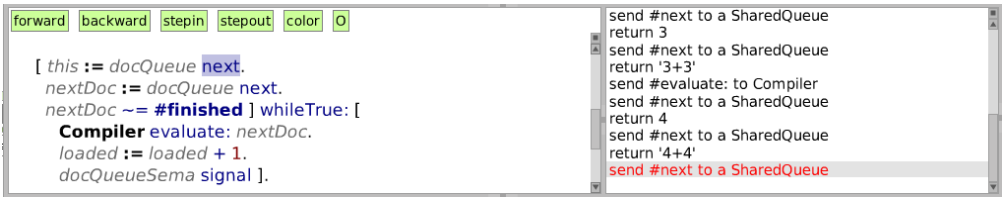

Figure 5.5: Execution of the update loader

executed statements (only a part of them are shown in the figure), we discover that the queue is read four times. After that, the method waits indefinitely on the queue. To know why nothing is put into the queue anymore, we take a look at the other process, so we have to get into a method that was executed in the other process. Before we do that we bookmark the current method execution, because it is one of the important locations in this execution trace and we might want to come back here. To get into the other process, we can show the object flow of an element that was read from the queue, because we know it was put in there by the other process. The flow of such an element is shown in Figure 5.6. Following back this flow, we get into the other process, until we

Figure 5.6: Object flow of a element in the queue

are in the execution of withIndexDo:. This is the loop of the retrieved updates which should put them into the queue (by evaluating the provided block). In Figure 5.7 the source and submethod pane are shown. Here we see that the

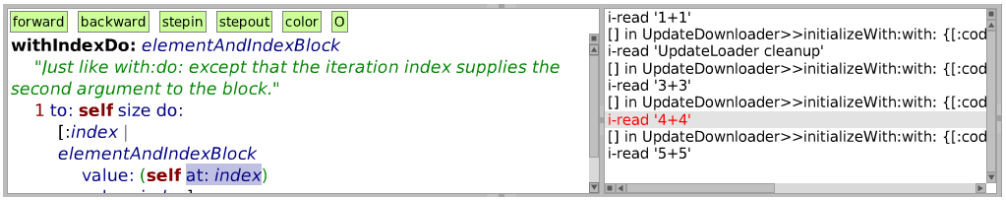

Figure 5.7: Loop on downloaded updates

fifth update is retrieved, as it is in the array (i−read '5+5'), but then the block is not evaluated anymore, and therefore nothing is put into the queue. Now we must conclude that something definitely must be wrong with the downloading process, because the method withIndexDo: should absolutely not stop abruptly. So we assume that the process was terminated somehow. By investigating what was done in this process, we do not find out anything. We see that the updates simply were put into the queue; nothing more happened. An assumption we can make is that another process somehow must have terminated this process, so we go back to the method execution that executes the updating code by clicking on the bookmark we made before.

Looking back at Figure 5.5 we see that every piece of code that is read from the queue is evaluated by the compiler. So we can step into those evaluations and see what happened (actually we have to step through several methods until we get to the DoIt method that executes the code). When we arrive in the DoIt method execution of the update code UpdateLoader cleanup, we remark something in the side effects graph, shown in Figure 5.8. In the graph we see that after the

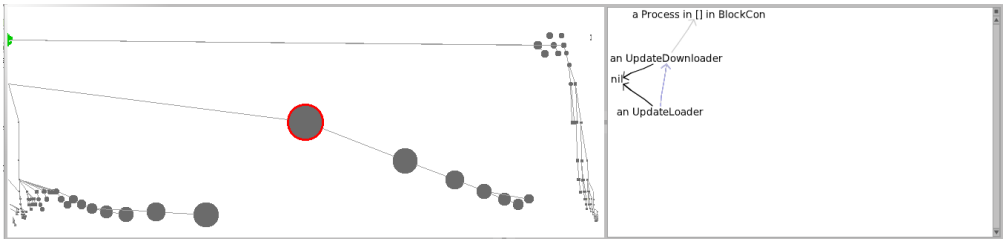

Figure 5.8: Side effects of the executed update code

execution of the code, the UpdateLoader does not have an instance variable anymore pointing to the UpdateDownloader and the UpdateDownloader itself has no instance variable anymore pointing to its process. They both have new

references to nil, so we immediately see that their instance variables were nilled out. We know now the problem lies in this part of the code. Stepping further into it we discover that the called method cleanup does terminate the process and additionally nils out the downloader (Figure 5.9). It does that because this update cleans up the image and by doing so inadvertently terminates the process. That is why nothing is put into the queue anymore and so we have found out that this update accidentally stops the download process.

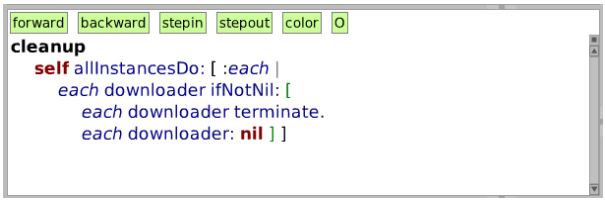

Figure 5.9: Cleanup method that stops process

Alternative solution. In this particular example, there are other ways of finding the error. Some of the above steps, like looking at the execution trace of both processes to find out that the process suddenly stopped, are probably necessary for understanding the problem. But to get to the code that stops the process, there are faster ways than looking at the evaluation of the compiler. In the UpdateLoader, the downloading first gets initialized and started with the following code:

Listing 5.4: Initializing of download process

```
retrieveUpdatesOntoQueue: aQueue withWaitSema: aSemaphore
    self downloader ifNotNil: [
        self downloader terminate ].
    self downloader: (UpdateDownloader new
        initializeWith: aQueue with: aSemaphore).
    self downloader start
```

In the start method of the UpdateDownloader actually starts the process. When we suspect something is wrong with the downloader process, we might want to investigate what happened with the downloader object instantiated here. So we use another feature of Compass, the *forward flow* explorer, which is shown for the downloader object in Figure 5.10. In this forward flow we see that after storing the object in the instance variable, much later it gets read from it

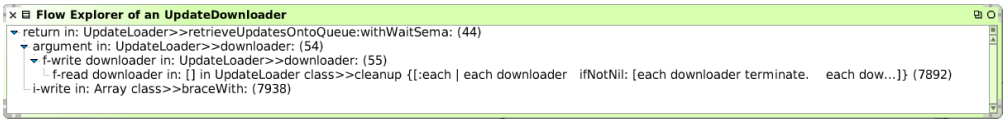

Figure 5.10: The forward flow of the downloader object

again. When we go the that location where the downloader object was read, we are immediately in the method execution where the downloading process gets terminated. The downloader gets read there from the instance variable and is told to terminate its process. Now inspecting the stack we find out that this code is executed as part of a previous update, so we go somehow the opposite way as in the previously presented solution.

Discussion. As the error in the example was not trivial and we had to handle a large trace, we needed several features of the debugger to find the source of the bug. In this case the debugger was very useful to understand the system, which was necessary to identify the cause. We have shown the use of the object backward flow (Where did an object come from?), the forward flow (Where did an object go to?), the object graph and the bookmarking of method executions.

The example also makes clear that with our debugger more than one way exists to solve the problem. In this example we have presented two possibilities to detect the source of the error. The second solution was more effective, but there is no obvious reason why one should explore the forward flow of the downloader object. But if someone gets stuck somewhere he just has to try out some things with objects that seem important to the problem, like in this case the downloader object. What the developer does next also depends on his knowledge of the domain, the better this knowledge is, the better he chooses his next step. The important thing is that Compass provides enough functionality to help the developer with each of those steps.

In the beginning of this section we mentioned our debugger is not designed for handling more than one process, but in this example we have seen it works not too bad already. Nonetheless, the support for multithreaded programs could be heavily improved. Right now, the method trace is a bit confusing as processes switch from time to time. Showing the trace of every process separately could help improve that, but then also ways must be found to handle the visualization of object flows that involve more than one process, i.e., objects

being passed between processes. To summarize, by separating the different processes more clearly, navigation of multithreaded applications would become more convenient.

Chapter 6

Conclusion

In this work we presented Compass, a flow-centric back-in-time debugger with the objective to make navigation of the execution data more efficient. We combined classical features of back-in-time debuggers with new ideas. In this chapter we summarize what our concepts contribute to debugging and what future work can be done.

6.1 Compass revisited

We developed a model that revealed how the different entities of the execution history of a program are related and how they can be navigated efficiently. Compass is based on that model. The next list contains the navigation concepts we implemented in our debugger.

- Stepping the trace
- Control flow dependencies
- Object relationships
- Object history
- Object flow
- Alias to method execution mapping
- Alias to statement mapping
- Created aliases in method execution

- Side effects

The most important contribution that distinguishes our debugger from the other approaches is the accessibility of the flow of objects. Other interesting contributions are the side effects graph and the control flow dependencies. The other navigation concepts listed before are in some way also contained in other approaches. In the following we discuss in more detail our contributions.

Other approaches do not provide the flow of objects. To find out where an object came from the developer has to read and understand code to see how an object was passed around in the system. With the object flow the steps an object has taken can easily be followed and the developer can understand what happened with it in the system and focus on finding out what went wrong. We have shown in our examples that this makes finding bugs much easier if a defective object has been passed a long road to where the error occured.

We introduced a new visualization for the method trace, the fisheye method trace. Together with the other views in our debugger this trace view has proven very useful, as it helps the developer to get a good understanding of the underlying trace. There are possibilites to visualize different components in this trace, like we did with the object flow, that can easily be interpreted when seen in the trace.

The side effects graph helps the developer to understand how objects are related. As it shows the side effects of the currently selected method execution, it provides additional visual information to understand the part of the code the developer is investigating. When an error in the relationships of objects is suspected, this can be especially useful.

The control flow dependencies allow the developer to see quickly if the execution of a method depended on a conditional statement executed elsewhere in the trace. This can help the developer a lot to get a good understanding of the system. For example, by examining direct dependencies the user could quickly see if a method execution is part of a code that was executed in a loop.

6.2 Future Work

Firstly, there are possibilities to improve the method trace view. Lots of other visual aids could be integrated, e.g., using colors for the circles to indicate attributes of it. Also, there are disadvantages that should be addressed too, e.g., it would be nice to have a good way of labelling the circles, so the developer can

recognize better which circle represents which method execution. Also, other high-level views could be added. The idea is to use views of dynamic data and integrate them into the debugger. There exist lots of ways to visualize the dynamic data on a high level (e.g., for reverse engineering). The question is if using those views while debugging back-in-time can improve the effciency of bug detection.

The problem with the current implementation of the side effects graph is that it can get very big for large subtraces, and if it is too big it is not useful anymore at all. So for small subtraces this graph works very well and can be useful, but for large subtraces improvements are necessary. One idea is to do something similar like we did with the method trace, which means, show the full graph but accentuate parts of it the developer is interested in. It is a bit more complex to realize than the fisheye method trace, but could make the side effects much more useful

The control flow statements as they are implemented in our debugger are useful for understanding the system. But the initial goal was to let the developer skip the examination of big parts of the execution trace if values taken by the control flow statements were infected. We have shown in the bank account example that this can help sometimes, but it seems that this is only useful for a very specific type of error. It could be that often the developer is misguided when examining those dependencies. However, the idea behind it was to involve dynamic slicing in a back-in-time debugger, but with the control flow statements we only provide a part of the dynamic slice. We could improve this by letting the developer choose a statement, computing the full dynamic slice and then highlighting the code that affected the statement. This would help the developer to focus on the code that really influenced the code he is interested in.

The above paragraph points out something else that might be considered, namely hiding parts of the execution data that seems not to be important to the issues the developer focuses on. Our approach deals with the full execution trace and how it can be navigated, but it could help to additionally reduce the data that must be navigated. The idea is to have in principle all the execution data, but to not display all of it depending on what the developer is interested in, so the knowledge of the developer about the problem can be taken into account. This is what is done with the dynamic slice, where the developer focuses on a specific statement and wants to accentuate the code that influenced the statement. There are other possibilites to filter out parts of the execution data. E.g., the developer can choose to exclude the method executions that come from specific packages or classes.

Appendix A

User Guide

This appendix gives instructions on how to install the necessary components to run the Compass Debugger. What is needed is an Object Flow VM, an image that has some packages installed for the special VM to work properly, and of course the packages of Compass and its dependencies. Additionally, Compass requires Graphviz to be installed on the machine, because it needs it to create some graph layouts. After all the installation instructions there is a guide which explains the usage of the debugger.

A.1 Installation

A.1.1 Downloading Compiled VM and Prepared Image

The easiest way to obtain a ready-to-run version is to download the virtual machine and demo image from the following website:

`http://scg.iam.unibe.ch/Research/ObjectFlow/`

The download is a zip archive that contains the compiled VM for the appropriate platform (currently available are VMs for Mac OS X Intel and Ubuntu Linux) and a prepared image. At the time of this writing the provided pre-compiled VMs have been tested on Mac OS X 10.5.5 for the Intel processor, and Ubuntu 8.04 (Hardy Heron).

A.1.2 Preparing Own Image

Loading FlyingObjects

The Object Flow Debugger requires some support code in the image (for example, the definition of the class Alias and extension of the class Process). To make your own image ready to be run on the Object Flow VM do the following:

1. After backing up your image, start it up using a standard Squeak VM.
2. In the Monticello browser add the following SqueakSource repository.

   ```
   MCHttpRepository
       location: 'http://www.squeaksource.com/FlyingObjects'
       user: ''
       password: ''
   ```

   Load the package FlyingObjects and then save and quit.
3. Start the image using the Object Flow VM.
4. Load the package FlyingObjectsUI from the same repository as in step 2 and save your image.
5. To test your installation, run the tests in the package FlyingObjects−Tests. All tests are expected to pass.

Loading Compass

If you have correctly installed the Object Flow VM along with its support packages, you should now be able to install Compass. As Compass has many dependencies, we provide an installation script for speeding things up. Here is what you have to do:

1. In order to make the object graph work, you need to install GraphViz on your computer. You can get it here: http://www.graphviz.org/
2. In an image working with the Object Flow VM, add the following SqueakSource repository to the Monticello browser.

   ```
   MCHttpRepository
       location: 'http://www.squeaksource.com/OmniCompass'
       user: ''
       password: ''
   ```

3. Load the package CompassInstaller.
4. To load all the required packages, execute the code

```
CompassInstaller bootstrap
```

A.2 Debugging with Compass

This section provides a quick guide to get started with Compass.

A.2.1 Starting the Debugger

There exist two ways of recording a program execution and starting the debugger.

The first way of recording data is to use the flyDuring: method implemented in the class Object. For instance, to trace the bank account example, execute the following code:

```
self flyDuring: [ BAAccount example ]
```

The execution of the block is recorded. Now the Compass debugging interface can be started by executing

```
CompassDebugger start
```

The debugger will then show the recorded data. When closing the debugger, the traced data gets deleted. If an error occurs while tracing the code, as usual a small debugger window appears. In addition to the default buttons the new button labelled 'Compass' opens the Compass debugger at the location where the error occurred.

The second way to debug is to use unit tests. We extended SUnit to re-run a failed test and record its execution before the failure is shown in the debugger. Whe re-running a test, as usual the small debugger window pops up, and as discussed above the Compass debugger can be started by clicking the 'Compass' button.

A.2.2 Using the Debugger

Figure A.1 illustrates the Compass debugger user interface. This section gives an overview of the different views and actions provided by Compass.

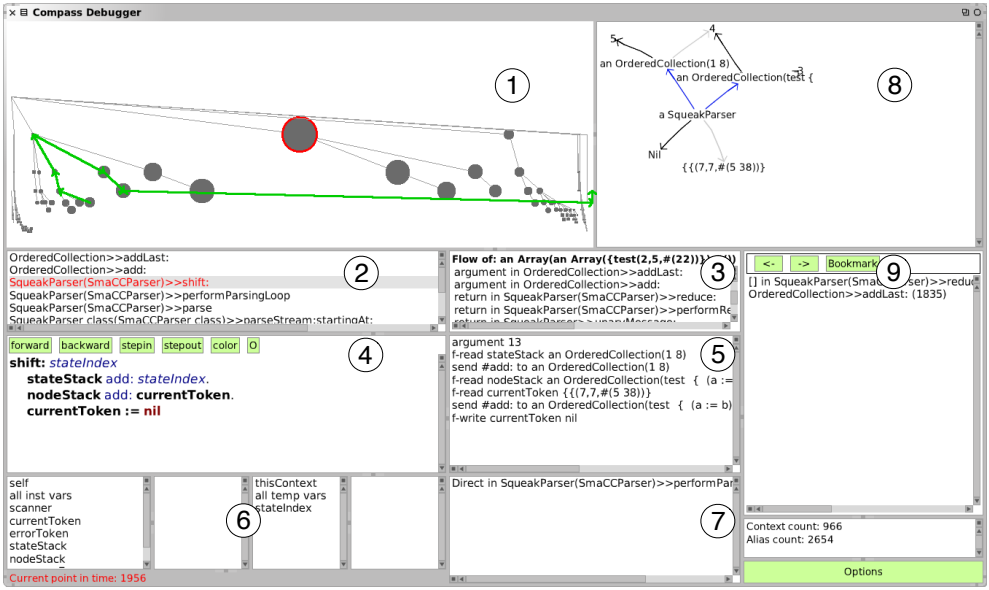

Figure A.1: Compass debugger frontend.

1. Execution trace. This view shows the execution trace as a tree in which nodes represent executed methods and block closures. Lines represent the caller relationship from top to bottom right. Nodes are ordered from left to right by the start timestamp of their execution and from top to bottom by their depth on the call stack. The trace can be navigated by clicking on the circles. The thick green arrows represent the flow of the object that was selected in one of the other views (see below).

2. Execution stack. This view shows the execution stack as it existed at the time when the selected method execution was started.

3. Object flow. This panel shows the flow of a specific object. The list contains the transfers of a reference of this object (*e.g.*, argument, return, field

write, field read, etc.). This allows one to backtrack the flow of the object to find out how the object was passed into this method. The flow given by this list is the same as the one shown graphically in the execution trace (1). By selecting a reference transfer from the list, the focus of the debugger changes to the method execution in which this transfer took place. One way to select an object flow is by double clicking on an alias in the sub-method statements list (5).

4. Source code. This is the source code of the method of the selected method execution.

5. Executed program statements. This list shows the reference transfers (aliases) and method sends that occurred during the execution of the selected method or block execution. When an item is selected the corresponding source code statement in the source code pane (4) is highlighted. Additionally, important actions can be executed from the context menu (right-click on an item). The available actions depend on whether an alias or a message send is selected. The most important action is to show the flow of an object. When choosing this action, the flow is shown in the previously described object flow pane (3) and it is drawn in the execution trace (1). You can also choose to explore the forward flow, which brings up a window that shows a tree of how the object was transferred starting at the current selection.

6. Variables. These four panes are the same as in the original debugger. They allow one to inspect the fields of the receiver and of local variables of the selected execution context with respect to the point in time of the current focus. By right-clicking on a variable, similar actions can be triggered as in pane (5), *e.g.*, selecting an object to highlight its object flow.

7. Dependencies. This list shows the control flow dependencies of the currently selected method execution (that is, the list of control flow statements that were responsible that the selected method was executed). By clicking on a dependency, the debugger jumps directly to the method execution and selects the control flow statement in the source code pane (4).

8. Side effects graph. The side effects graph summarizes the side effects that the execution of the currently selected method and all transitively called

methods produced. A black arrow between two objects indicates a field or array slot update. The black arrow points from the updated object to the newly assigned object. To support the understanding of this graph, the following additional information is provided to show the connection between the different objects in the graph. A light gray arrow indicates the previous value of a modified field and a blue arrow indicates a field or array read event (dereference). By right-clicking on an object, a menu with a list of the new field values comes up. By selecting a value, the debugger jumps to the location where the object was written into the field.

9. Navigation history. Like in a web browser, the navigation history can be used to go step by step back- and forward. In our case, the steps are the context switches (changes of focus in the Compass user interface). If the context is changed, by clicking on the back button you get to the previously selected context. Also bookmarking is supported to be able to quickly jump to bookmarked locations in the execution history.

List of Figures

Bibliography

[Clev00] Holger Cleve and Andreas Zeller. "Finding Failure Causes through Automated Testing". In: *Proceedings of the Fourth International Workshop on Automated Debugging*, Aug. 2000.

[Furn86] George W. Furnas. "Generalized Fisheye View". In: *Proceedings of CHI '86 (Conference on Human Factors in Computing Systems)*, pp. 16–23, ACM Press, 1986.

[Geig02] Leif Geiger, Ag Softwaretechnik, Technische Universität Braunschweig, Albert Zndorf, Ag Softwaretechnik, and Universität Paderborn. "Abstract Graph Based Debugging with Fujaba". 2002.

[Gold83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[Hofe06] Christoph Hofer. *Implementing a Backward-In-Time Debugger*. Master's thesis, University of Bern, Sep. 2006.

[Hove04] David Hovemeyer and William Pugh. "Finding bugs is easy". *SIGPLAN Not.*, Vol. 39, No. 12, pp. 92–106, 2004.

[Inga97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself". In: *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pp. 318–326, ACM Press, Nov. 1997.

[Ko04] Andrew J. Ko and Brad 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*, pp. 151–158, 2004.

[Koju05] Toshihiko Koju, Shingo Takada, and Norihisa Doi. "An efficient and generic reversible debugger using the virtual machine based approach". In: *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pp. 79–88, ACM, New York, NY, USA, 2005.

[Kore88] B. Korel and J. Laski. "Dynamic program slicing". *Information Processing Letters*, Vol. 29, No. 3, pp. 155–163, 1988.

[Lenc97] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. "Query-Based Debugging of Object-Oriented Programs". In: *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming (OOPSLA'97)*, pp. 304–317, ACM, New York, NY, USA, 1997.

[Lenc99] Raimondas Lencevicius, Urs Hölzle, and Ambuj Kumar Singh. "Dynamic Query-Based Debugging". In: R. Guerraoui, Ed., *Proceedings of European Conference on Object-Oriented Programming (ECOOP'99)*, pp. 135–160, Springer-Verlag, Lisbon, Portugal, June 1999.

[Lewi03] Bill Lewis. "Debugging Backwards in Time". In: *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG'03)*, Oct. 2003.

[Libl05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. "Scalable statistical bug isolation". In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pp. 15–26, ACM, New York, NY, USA, 2005.

[Lieb98] Henry Lieberman and Christoper Fry. "ZStep 95: A reversible, animated source code stepper". In: John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, Eds., *Software Visualization — Programming as a Multimedia Experience*, pp. 277–292, The MIT Press, Cambridge, MA-London, 1998.

[Lien06] Adrian Lienhard, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. "Capturing How Objects Flow At Runtime". In: *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA'06)*, pp. 39–43, 2006.

[Lien08a] Adrian Lienhard. *Dynamic Object Flow Analysis*. PhD thesis, University of Bern, Dec. 2008.

[Lien08b] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. "Test Blueprints – Exposing Side Effects in Execution Traces to Support Writing Unit Tests". In: *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*, pp. 83–92, IEEE Computer Society Press, 2008.

[Lien08c] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. "Practical Object-Oriented Back-in-Time Debugging". In: *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, pp. 592–615, Springer, 2008. ECOOP distinguished paper award.

[Meye06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. "Mondrian: An Agile Visualization Framework". In: *ACM Symposium on Software Visualization (SoftVis'06)*, pp. 135–144, ACM Press, New York, NY, USA, 2006.

[Poth07] Guillaume Pothier, Éric Tanter, and José Piquer. "Scalable Omniscient Debugging". *Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, Vol. 42, No. 10, pp. 535–552, 2007.

[Wasy07] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. "Detecting object usage anomalies". In: *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 35–44, ACM, New York, NY, USA, 2007.

[Weis81] Mark Weiser. "Program slicing". In: *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pp. 439–449, IEEE Press, Piscataway, NJ, USA, 1981.

[Weis82] Mark Weiser. "Programmers use slices when debugging". *Commun. ACM*, Vol. 25, No. 7, pp. 446–452, 1982.

[Zell01] Andreas Zeller. "Automated Debugging: Are We Close". *Computer*, Vol. 34, No. 11, pp. 26–31, 2001.

[Zell02a] Andreas Zeller. "Isolating cause-effect chains from computer programs". In: *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 1–10, ACM Press, New York, NY, USA, 2002.

[Zell02b] Andreas Zeller and Ralf Hildebrandt. "Simplifying and Isolating Failure-Inducing Input". *IEEE Transactions on Software Engineering*, Vol. SE-28, No. 2, pp. 183–200, Feb. 2002.

[Zell05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oct. 2005.

[Zell96] Andreas Zeller and Dorothea Lütkehaus. "DDD — a free graphical front-end for Unix debuggers". *SIGPLAN Not.*, Vol. 31, No. 1, pp. 22–27, 1996.