FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Implementing a Reversible Debugger for Python

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Patrick Sabin
Matrikelnummer 0425253

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Univ.-Ass. Dr. M. Anton Ertl

Wien, 8.11.2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Abstract

The programmer usually initiates a debugging process because of a failure and his goal is to find the defect. The defect is always executed before the failure occurs, so it is natural to start at the failure and move backwards in a program to find the defect. However this procedure is usually not supported by actual debuggers.

There are two different methods of implementing a reversible debugger, i.e., a debugger which can run the program forwards and backwards. The first one is the logging-based approach, which records the state of the program after every instruction and allows inspection after the program has finished running. The second one is the replay-based approach, where the debugger runs the debuggee interactively. For this purpose it makes periodic snapshots. The debugger runs the debuggee backwards by restoring a previous snapshot and then running the program forward until it reaches the desired position.

In this thesis, I show that it is possible to implement a reversible debugger by continuous snapshotting of the program state. There are indeed some challenges with using such a feature. For example, there are non-deterministic instructions, which execute differently each instance the interpreter executes them, e.g., a function, which returns the system time. Another example of this is when instructions change some external state like a file on the hard drive, which the debugger does not save when it makes a snapshot. Another problem is that some instructions do something different each time the debugger executes them.

Therefore I present some methods of treating these problems. Accompanying this thesis, I have developed a proof-of-concept implementation of a reversible debugger called epdb for the Python programming language, which solves most of the problems of reversible debugging.

In order to support reversible debugging of programs which have non-deterministic instructions in it, I introduce the new concept of timelines. With timelines, the user can decide which execution path he wants to take. I also introduce stateful resource management to support the management of the external state. This allows the user to investigate the environment corresponding to the actual position inside the program, when he executes the program backwards.

Zusammenfassung

Programmierer beginnen mit der Fehlersuche, weil sie ein falsches Verhalten des Programmes feststellen. Das Ziel der Fehlersuche ist festzustellen wo im Programm der Defekt ist, also der Teil des Programmes, welcher für das falsche Verhalten verantwortlich ist. Der Defekt wird jedoch ausgeführt bevor ein falsches Verhalten sichtbar ist. Daher wäre es sinnvoll in einem Debugger das Programm am Ort, wo der Fehler sichtbar ist, zu beginnen und von dort weg das Programm schrittweise rückwärts auszuführen bis man zum Defekt gelangt. Dieses rückwärts Ausführen wird jedoch von vielen gängigen Debuggern nicht unterstützt.

Es gibt zwei grundsätzliche Strategien um einen rückwärtsausführenden Debugger zu implementieren, das heißt einen Debugger der das Vorwärts- und Rückwärtsausführen unterstützt. Die erste Variante ist die des Log-basierenden Debuggers. Ein Log-basierender Debugger speichert den Programm State nach jeder ausgeführten Instruktion. Nachdem das Programm fertig ausgeführt worden ist kann der Anwender das Programm anhand des Logdatei erneut abspielen und den State zu jedem beliebigen Zeitpunkt im Programm abspielen. Die zweite Variante ist die Snapshot & Replay Strategie. Hierbei erlaubt der Debugger interaktive Steuerung des Programmes. Beim Vorwärtsausführen werden hierbei regelmäßig Snapshots vom State gemacht. Um das Programm rückwärts auszuführen wird ein vorheriger Snapshot aktiviert und das Programm solange erneut ausgeführt bis die gewünschte Position erreicht ist.

In dieser Diplomarbeit möchte ich zeigen, dass es möglich ist einen rückwärtsausführenden Debugger zu schreiben, welcher regelmäßig Snapshots macht und diese nutzt um Rückwärtsausführen zu ermöglichen. Es gibt einige Probleme die beim Rückwärtsausführen auftreten. Zum Beispiel gibt es nichtdeterministische Instruktionen, welche der Interpreter jedes Mal anders ausführt, beispielsweise eine Funktion, die die Systemzeit zurück gibt. Ein weiteres Problem sind Instruktionen mit Seiteneffekten. Diese ändern einen Teil States vom Programm, welcher nicht mittels Snapshots gespeichert wird, wie zum Beispiel eine Funktion die auf die Festplatte schreibt.

Daher möchte ich in dieser Arbeit Methoden vorstellen, die mit diesen Problemen umgehen können. Außerdem habe ich zum Nachweis der Machbarkeit einen Rückwärtsausführenden Debuggers für die Programmiersprache Python entwickelt, welcher die meisten Probleme der Rückwärtsausführung löst.

Um die Rückwärtsausführung von Programmen mit nichtdeterministischen Instruktionen zu ermöglichen, habe ich das neue Konzept der Zeitlinien eingeführt. Mit Zeitlinien kann der Benutzer entscheiden, welchen Ausführungspfad er wählen möchte, wenn er auf nichtdeterministische Instruktionen trifft. Außerdem habe ich das Konzept des zustandsorientierten Ressourcen Managements entwickelt, damit der Debugger auch den externen Zustand verwalten kann. Der Benutzer kann somit auch die der aktuellen Position entsprechende Umgebung des Programmes ansehen, wenn er das Programm rückwärts ausführt.

Contents

Abstract **ii**

Zusammenfassung **iii**

Contents **v**

List of Figures **vii**

List of Tables **vii**

1 Introduction **1**
1.1 Motivation for Better Debugging Tools 1
1.2 Family of Bugs and Related Terms 2
1.3 Methods of Debugging 3
1.4 Epdb 5
1.5 Gepdb 7
1.6 Challenges of Reversible Debugging 8
1.7 Contribution 10

2 Prerequisites **13**
2.1 Python Versions 13
2.2 Types of Debuggers 14
2.3 Python and Debugging 17

3 Reverse Execution **21**
3.1 Execution Modes 21
3.2 Snapshots 24
3.3 Instruction Counting 27
3.4 Timelines 28
3.5 Dealing with Non-Determinism 31
3.6 Dealing with Side Effects 32
3.7 Resources 33

3.8 Instruction Patching 36
3.9 Atomicity 37

4 Epdb Internals **39**
4.1 Snapshot Processes 39
4.2 Bookkeeping in Epdb 44
4.3 Shared Memory 45
4.4 Breakpoints 47
4.5 Implementation of Debugging Commands 48
4.6 Example Patch Modules 50
4.7 Modules of the Standard Python Library 57

5 Performance **59**
5.1 Instruction Counting 60
5.2 Snapshot Performance 61
5.3 Epdb Performance 63

6 Applications **67**
6.1 Web Applications 67
6.2 Smart Phone Development 68
6.3 Visualization of Algorithms 69
6.4 Design and Architecture 69

7 Related Work **71**
7.1 Other Reversible Debuggers 72

8 Further Work **75**
8.1 Smart Snapshot Making 75
8.2 Reversible Debuggable Libraries and Other Tools 76
8.3 Native Instruction Counting and Bookkeeping 76

9 Conclusion **77**

Bibliography **79**

List of Figures

1.1 Timeline example 5
1.2 gepdb 7

3.1 Redo and replay modes 22
3.2 Local snapshot fails activating a snapshot 25
3.3 New timeline 29
3.4 A process and its resources 33

4.1 Star topology 40
4.2 Frame count activation example 43
4.3 Continue activation example 44
4.4 Organization of shared data 47
4.5 Module math 57
4.6 Module random 57
4.7 Module os 57

5.1 Average time used relation between with and without instruction counting . 61
5.2 process creation 64
5.3 fork() overhead 65

7.1 Omniscient Debugger (odb) 72

List of Tables

5.1 Benchmarks for epdb 66

CHAPTER 1

Introduction

Most debuggers out there are traditional debuggers, i.e., debuggers that can only run the program forward. Reversible debuggers, i.e., debuggers that can run the program backwards too, can make debugging easier. Most users of debuggers have experienced the situation, where they overshot the position in the program, where they wanted to examine the value of some variables. With a traditional debugger they would have to restart the program and go through the tedious initialization step. With a reversible debugger, they could accomplish this much faster by running the program backwards. Reversible debuggers have certainly advantages over traditional ones. However, most debuggers are traditional ones because reverse execution requires some kind of simulation, because machine instructions are usually not reversibly executable. Sometimes reverse execution is straight forward, but it can be ambiguous, e.g., when sending a message to another computer.

So alongside this thesis, I have developed the reversible debugger **epdb** for the Python programming language. I have chosen a dynamic programming language, because it is easier to direct[Sos95], i.e., to monitor and control the execution, than in a statically typed programming language. Epdb does not only allow reverse execution, but is also able to deal with ambiguous situations by providing a framework to work in, while allowing the user to control the forward and reverse execution in such situations.

1.1 Motivation for Better Debugging Tools

Debugging can cost a lot of time and money. For example Hailpern states:

> "In a typical commercial development organization, the cost of providing this assurance via appropriate debugging, verification and testing activities can easily range from 50% to 75% of the total development cost." [HS02]

This includes verification and testing activities, but it shows that debugging is an important part of the software development process, that costs lots of time. Many of the problems that occur during debugging are because of inadequate debugging tools. The National Institute of Standards and Technology[Tas03] estimated that based on interviews, that software engineers spent 35 percent of their time on debugging and correcting errors.

The computer science community has largely ignored the debugging problem and so debugging is still more an art than a technique. Therefore this thesis proposes a new debugger called epdb. Epdb extends a traditional debugger with reverse execution capabilities. So it is possible to step through or run a program not only forwards but also backwards. Using this technique I hope to make debugging easier and more straight forward. Eisenstadt[Eis97] shows that many bugs are difficult to trap, because of the cause/effect chasm, i.e., bugs that materialize far away from where they were spawned. Debugging such bugs using a reversible debugger is easier, because the programmer doesn't have to restart the program every he they skipped over the defect.

1.2 Family of Bugs and Related Terms

Book writers and researchers in the area of fault-tolerant systems and dependability often use the terms error, defect, failure, fault, etc. and give each of them a distinctive meaning. However, if you look in different papers you often see different meaning for each of these terms. For example, Laprie[Lap92] defines failure as "deviation of the delivered service from compliance with the specification", but Chillarege[Chi96] notes that in the world of software there aren't well-defined specifications for most products.

For the area of debugging, a smaller subset of these terms is sufficient. In this paper I will stick to the terminology given by Zeller[Zel09]. He distinguishes between defects, infections and failures. A **defect** is an incorrect piece of code. An **infection** is an incorrect programming state and a **failure** is an observable incorrect program behavior. He also uses the term **flaw** to mean defects which cannot be tracked down to a specific location. A **bug** can be an incorrect program code, state or program execution.

To illustrate the meaning of the terms I will give an example. In a typical error-fixing scenario a user reports a bug, because of a failure, i.e. an observable incorrect program behavior. To fix the bug, the programmer tries to locate the defect in the program code to provide a patch. He may use a debugger to locate the defect, or in case it is flawed he may use it to understand what is wrong with the program in order to create a new architecture for the revised version of the program. To locate the bug, he first tries to find the state of the program that is infected. The infection is found after the execution of the defect. With a traditional debugger he will start at some non-infected part of the program and navigate forward in the program until he reaches an infected one. When its state switches from non-infected to infected he has found the defect. However, an infection

is usually not easily observable and it is likely that the programmer will step too far. In this case he has to restart the program. With a reversible debugger the programmer can start at a point where the program shows an observable wrong behavior. This state is obviously infected and the programmer can start from there to run backwards until he finds a non-infected state. If he runs too far he can then run forward again. With a reversible debugger he could use some binary search algorithm to quickly track down the bug. This is especially helpful if there is a big cause/effect-chasm, i.e., the start of the infection is far away from the point of failure, in terms of instructions executed.

1.3 Methods of Debugging

There are quite a few debugging methods programmers use. The simplest is **print & peruse**[Eis97]. Using print & peruse, the programmer typically inserts some *print()*-functions. It doesn't have to be just *print()*-functions, but it can also be some more sophisticated logger. This approach has the advantage that it is almost always available and can be used, even with no additional software installed. This is very useful, if your architecture doesn't support debugging, for instance, in embedded systems. It is also easy to understand and the technique is usable in any other programming environment or programming language. Therefore, programmers don't need to learn a new tool.

Of course there are some disadvantages to debugging by inserting *print()* function calls. For example, the users have to modify the code and after that, they have to undo the changes. By using a more sophisticated logger, which can turn the output off, this avoids the need to undo the changes. However, with a logger the developers have to maintain the logging statements. Too many lines of debugging output would make perusing the information confusing, while with too few lines, the programmer may miss some important hint to a bug. Multiple log levels may improve the situation, but may also raise the maintenance costs. The print or log statements are tightly coupled to the rest of the program and therefore it is not clean software engineering practice. A bigger problem of print & peruse is, that the programmer has to guess where to insert the *print()* functions. If he guesses wrong, he has to restart the whole program. This can be annoying, if the program has to run for quite a long time until it reaches the defect. Guessing good locations for the *print()* functions also requires experience, which cannot be easily taught. For example, someone posted the piece of code shown in Listing 1.1 into the comp.lang.python mailing list, because he couldn't find the bug. This piece of program computes the greatest common divisor. After running it one will see that the *print()* functions give exactly the output one would expect. However, one will also get a ZeroDivisonError at the end. The *print()* functions were not much help here, at least if you arrange them like in the example.

Using a **traditional debugger** would help in the previous situation. I started pdb, the Python debugger that comes with most Python distributions and stepped through the

Listing 1.1: gcd.py

```
a = 12345
b = 54321
m = 1
while m != 0:
    if b > a:
        n = b/a
        m = b % a
        print b, " : ", a, " = ", n, "\t m is ", m
    if a < b:   # use elif here
        n = a/b
        m = a % b
        print a, " : ", b, " = ", m
        a = m
```

program. I discovered in the first run, that the body of both if statements were executed in one iteration, which obviously shouldn't be. A debugger has the advantage here, that the user doesn't need to guess the location of the defect before he starts the program, but he can just start stepping through the program. If he finds something suspicious, he can set a breakpoint there. In contrast to *print()* functions, the debugger is independent of the code, because there is no need to change it. However, there is still some dependence, because using a traditional debugger for longer running programs usually means using breakpoints, which refer to a line of code. Therefore if the program code changes, the breakpoints are lost. Depending on your debugger and debugger configuration, the breakpoints can be even lost after a restart of the program. There is another problem with a traditional debugger. If you have stepped too far and forgot to peruse some part of the program state, you have to restart your application and run forward again. In larger programs, it is often not appropriate to step through the whole program and so it is best to use breakpoints. However, here one needs to carefully guess where to set one's breakpoint. If one guesses wrong, it may be necessary to restart the program again.

A **reversible debugger** solves some of the shortcomings of a traditional debugger. It eliminates the need to restart the program as it is always possible to go backwards. The users don't need to guess breakpoint locations, because it is usually obvious where the program should break: at the position where it gets the first uncaught exception. A reversible debugger achieves more independence by not using breakpoints. Instead the debugger breaks at an exception, which is runtime information, and therefore does not rely on information of the source code. Therefore using a reversible debugger can make things easier.

Listing 1.2: gcd.py

```
print("Name:")
n = input()
print("Hello", n)
```

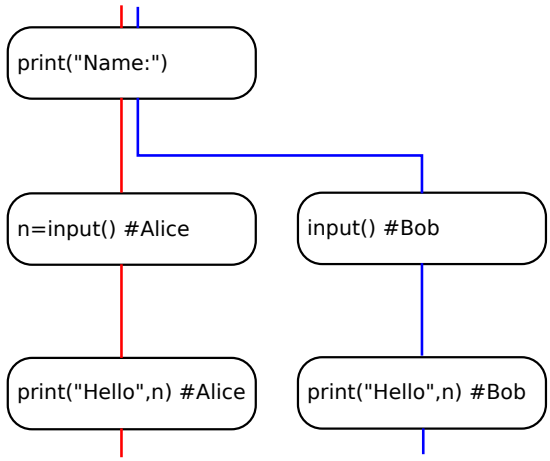

Figure 1.1: Timeline example

1.4 Epdb

The Extended Python Debugger, epdb for short, is a reversible debugger developed alongside this thesis. In this section, I want to give a very brief description of its features. Epdb is based on pdb, but because it is a reversible debugger, it is considerable different. Nevertheless, reusing the code of pdb speeds up the development time of the new debugger. Like pdb, epdb supports the commands *step*, *next* and *continue* to navigate through the program and supports breakpoints by the *break* command. Because epdb is a reversible debugger, it also supports the reverse program flow commands, which are called *rstep*, *rnext* and *rcontinue*. Epdb also allows another kind of navigation, namely activating a snapshot. It is possible to create a snapshot at an arbitrary position in the timeline with the *snapshot* command and later restore to it by using the *activate_snapshot* command.

The Python debugger supports **post-mortem** debugging, which means the programmer can inspect the program state in case the program has finished because of an uncaught exception. Epdb extends this behavior. It does not only allow inspection at program termination, but also allows the programmer to step backwards and inspect previous states of the program. This makes post-mortem debugging more powerful.

The principal architecture to achieve reversible debugging is **snapshot & replay**, i.e., the debugger makes continuous snapshots. To navigate backwards, the debugger

activates a previous snapshot and runs the program forward again. To create a snapshot, epdb uses the system call *fork()*, which makes a copy of the current process.

Timelines

Going back and forth in the program can lead to ambiguous situations. For example, Listing 1.2 shows a program that can have a different output every time the user executes it. Assuming the user has executed the program until the end and has typed the name 'Alice' when the program asked him for a name. If the user now goes backwards to the position where the program asked him for a name and runs forward from there, the debugger has two possibilities to deal with this situation. It could either assume that the user has entered the name 'Alice' and therefore the program could greet 'Alice' again, or it could ask the user for a new name and greet the user with the new name.

To resolve this ambiguous situations, epdb introduces timelines. A timeline is a possible execution path through the program. By default, epdb would execute the *input()*-function the same way as in the first run, i.e., in the previous example the program would greet 'Alice' again. However, if the user wants to enter a new name, he could create a new timeline and then enter a new name in this timeline, e.g. 'Bob'. In this case, these two timelines would coexist and the user can switch between them. Figure 1.1 shows the two timelines for the described scenario.

In order to implement timelines the debugger uses different **execution modes**. It needs them to distinguish, whether the program has already executed the instruction in the current timeline or not. In case it has executed it, the debugger would simulate the previous execution behavior of this instruction. Otherwise, it can execute it as usual.

Another point to consider with timelines is that simulating previous instruction behavior requires the debugger to change the behavior of instructions. Therefore, epdb introduces **instruction patching** in order to change the behavior of instructions. The instruction patching mechanism of epdb is extensible, so that programmers can add new patched instructions. These patched instructions work differently in each mode and can therefore simulate the previous execution behavior of an instruction.

Resource Management

Epdb uses *fork()* to create snapshots and to save the program state. However, *fork()* does not save the whole state of the program, e.g., it does not save files on the disk. Therefore, the debugger needs to manage the state, which is not saved by *fork()*, in some other way. Epdb uses **resource managers** to control the external state, i.e., the state which *fork()* does not copy. For epdb there exists a resource manager class for files, but it would also be possible to implement resource manager for other resources like databases. It allows the users to extend the debugger with new **resource managers** to provide resource management for their own resources. The resource manager work

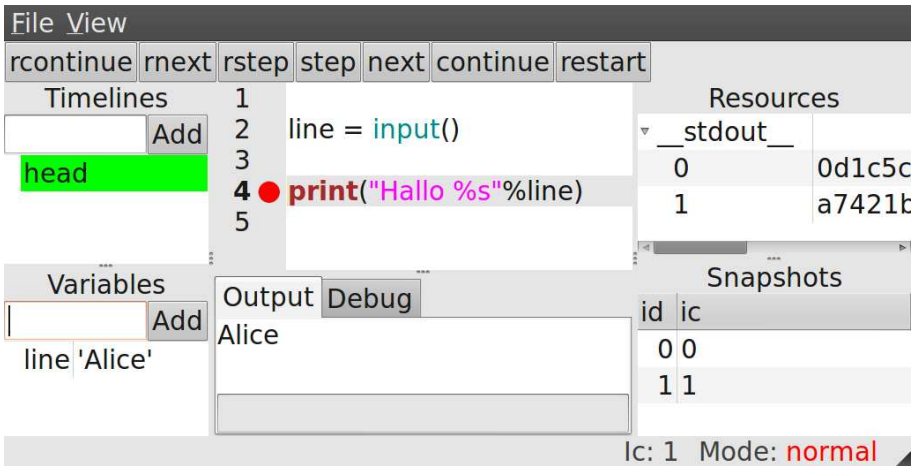

Figure 1.2: gepdb

in conjunction with instruction patching. When an instruction changes some external state, the patched instruction would call the resource manager in order to save the state.

1.5 Gepdb

Epdb is a command line debugger like pdb. However debugging is often easier in a graphical user interface. Therefore I developed gepdb. Gepdb is a graphical front end for epdb. Gepdb has the advantage that it always shows the information a programmer usually needs like the source code, the actual variables, and actual timelines.

Figure 1.2 shows a screenshot of the graphical frontend for the debugger. Gepdb lists all timelines that the program has in this debugging session and allows the user to create new timelines. Each timeline has its own resources and snapshots which the screenshot shows at the right side. In the resource window, gepdb lists each resource along with its history, i.e., it shows at which instruction count a change to the state of the resource occurred. At the bottom right side, the user interface lists the snapshots of the current timeline with their *id* and their *instruction count*. At the left side, the user has the ability to monitor variables, which the user interfaces updates, when the program state changes, e.g., by stepping forward. At the bottom window, the debugger shows the output of the program which typically goes to the console, and it has an entry to do user input, which gets only activated when the program expects some input. The tool bar at the top allows one to navigate forwards and backwards. The source code is highlighted and when the user right clicks right beside the line number, a pop-up menu shows up and allows the user to set a breakpoint. The status bar at the bottom shows the current instruction count and the current mode.

1.6 Challenges of Reversible Debugging

Reversible debugging has not been widely accepted yet. I believe that reverse debugging has some intrinsic problems, which were not challenged by most actual reverse debuggers. It is easy to simulate reverse execution for a program, which actually only does some computation and some control structures such as 'if' and 'while'. However, real-world programs are usually much more complex than that. They interact with the operating system using system calls. Consequently they can execute in a non-predictable way, i.e., one can't determine the state of the program at a position by just looking at the source code, because the whole **environment**, in which the program executes, influences the program execution. So we can define two different kind of program states. The **internal state** is the state of the program without looking at the environment, i.e., the internal state typically consists of the process memory and registers. The **external state** is the state of the part of the environment, which affects the program execution, at a given time in the program.

As Python is an interpreted language, the system calls are not directly visible. Instead, a program consists of functions which may be implemented as a native code module. If functions are implemented in native code, they are able to execute system calls, either directly or by using another library. However, with a Python debugger you can't step into such functions, i.e., they are **atomic** in respect of debugging. In case of functions which are not atomic, I am referring to these functions as **composite** functions.

Determinism

If we look at atomic instructions, we have to distinguish between deterministic and non-deterministic instructions. In this paper, a function is said to be **deterministic** if the outcome of the function is determined by the internal state only. So for example, a function that increments a variable is deterministic, because there is no influence of the environment. Look for example at an atomic function that returns the system time. The program cannot calculate the system time by running an algorithm. Therefore there is a system clock built in the computer and the operating system provides some means to access the clock. The system clock is part of the environment and therefore, the execution of the function is influenced by the external state. Therefore, the function which returns the system time is a **non-deterministic** function.

A deterministic atomic function is easy to handle for a reversible debugger, because it just has to keep track of the internal state, but for a non-deterministic function, tracking the internal state is not sufficient, because the external state also affects the program and may change when the debugger replays a function execution. In fact, it is even more complicated than that. When the user replays a non-deterministic function, he can have either of one two reasonable expectations. He could expect to have the same execution

as in the first run or he could expect to run with the new environment. Both views make sense and an interactive reversible debugger should consider both of them.

Side effects

While the execution of non-deterministic functions is influenced by the environment, functions with side effects change their environment. A function which writes something to the hard-drive is such a function, because it changes the information on the disk, which is not part of the internal state, but of the environment.

If someone debugs a program, and makes heavy usage of the environment, he is usually not only interested in the internal state, but also in the environment. Take for example an external sorting algorithm. If one wants to find a bug, he is not only interested in the variables during program execution, but also in the state of the file, which contains the data the program should sort. Therefore a reversible debugger should provide some way to manage the external state.

Bookkeeping

A reversible debugger must be able to recover previous states. Therefore it has to save the **execution path history** and the **data change history**[CFC01]. Epdb however does not save this history directly. It saves the data change history by making continuous snapshots, but it also keeps a history of the execution path. Thus, epdb needs to calculate the target position in the program when it replays an execution. For a snapshot & replay debugger, which uses multiple processes, it has to exchange this recorded information with the other snapshot processes.

Breakpoints

As a traditional debugger, a reversible debugger should support breakpoints and they should also work when the program runs backwards. The challenge here is especially for a snapshot & replay debugger, because a snapshot & replay debugger has multiple processes. When the user makes a breakpoint in one process, he should see them in other processes too.

Deterministic and Non-deterministic execution

Since the user can navigate through the program backwards and forwards, the debugger can execute instructions for the first time or it can execute instructions which it has executed before. If the debugger executes instructions it has executed before, I am referring to this execution as execution in **redo mode**.

When the debugger is in redo mode, it has two possibilities to execute non-deterministic instructions. It could either execute them like in the first execution, or it could execute them independently from the first execution. In other words, the debugger could choose to use the environment from the first execution[1] or it could use the actual environment. I am referring to the execution of instructions with an old environment as **deterministic execution**, because executing an instruction deterministically would change the internal state the same way as in a previous run. In the case of executing instructions with the actual environment, I am referring this execution as **non-deterministic execution**, because the internal state, which results from the execution of this non-deterministic instruction, is usually not the same as any other previous state.

Non-deterministic and deterministic execution is a challenge for reversible debuggers because they have to decide which type of execution they choose. Most reversible debuggers usually support only one mode of execution, which is usually deterministic execution. However, epdb supports both types of execution by using timelines.

Extensibility

In a real world program, the program execution can be influenced by almost everything which exists in the real world. For example, if the program controls a device, which measures the temperature, the actual temperature becomes an important part of the environment of the program. There are almost infinitely many possibilities to influence the program execution, and thus the programmers of a reversible debugger can't imagine every case which may become important for the users of a reversible debugger. Therefore a sophisticated reversible debugger should allow the user to extend the debugger so that he is able to debug all the devices, which he uses for his application.

1.7 Contribution

Although there already exist a few reversible debuggers, none of them tackle the problem that some instructions can execute differently each time the debugger executes them at the same position. Therefore, the program could execute in multiple different ways. Another problem is that the program may also change the external state, which can be an important source of information for the user, when he debugs the program.

The main contributions of this thesis to solve these problems are timelines and resource management. With timelines users have the choice to replay code either deterministically or non-deterministically. Consequently, users can decide if they want to reproduce the state in which they have been before, or if they want to execute the

[1] In the case an instruction has been executed multiple times, it could also execute the instruction with the environment of the second, third, any later execution

program in a possibly new way. Therefore, users have more control over the program execution.

With resource management, the debugger provides a new way to deal with side effects. A traditional debugger doesn't need resource management, because the program will change external resources when the execution proceeds. A reversible debugger does not change the external state when it runs backwards. Therefore, without resource management the external state of the program would not be in sync with the internal state. Epdb allows to manage the different states of each resource while debugging the program. For this purpose it uses resource managers. A resource manager tracks the different states of some part of the external state, and it is able to restore a previous state of the resource. The debugger interrelates the state of the resource with the corresponding instruction count. This system allows to manage the external state, it is easily extensible, and it also has good performance, because the debugger only has to reset the resources when it actually stops the execution.

CHAPTER 2

Prerequisites

Writing a debugger is quite different from writing other applications. In this section, I want to provide some information on how debuggers generally work. Python already ships with a traditional debugger pdb, which epdb depends on. Pdb makes use of the infrastructure Python provides to implement tools like debuggers, profilers, coverage tools and the like. Therefore I want to explain this infrastructure here too.

2.1 Python Versions

At the time of this writing, there are multiple different versions of Python around, which also work a little differently. For the prototype version of the debugger developed alongside this paper, I had to choose one to work with. First, there are different types of implementation, which target different architectures, e.g., Jython, IronPython, PyPy or CPython. Although many of them are in use, the most important is CPython. The other implementations should work the same as CPython, but may lack one or another feature. A more important distinction is between Python 2.x and Python 3.x. At the time of this writing, Python 2.x is the most used, while Python 3.x is the newer one. Python 3.x has some major changes to the syntax, most visible is the change from the *print* statement to a function. It is difficult to develop a program which works with both versions, and therefore I decided to go for Python 3.x only. Dealing with functions instead of statements, has the advantage that they are easier to patch, a feature which epdb uses to support reversible debugging.

2.2 Types of Debuggers

There exists a whole bunch of different kinds of debuggers, each with their own different purpose. The kind of problem, programming language, programming environment or system affects the requirements to a debugger. I want to give an overview of some types of debuggers in order to position epdb in the world of debuggers.

Machine Level Debuggers

Machine or low-level debuggers operate on machine code. People use them for programs written in assembler, or to debug the code a compiler has generated, or to reverse engineer a program where the source is not available.

One of the main properties of a debugger is the ability to halt the process. The debugger does this halting when it reaches a breakpoint. To achieve halting there are two principle options for the developers. Either they could execute it on a virtual machine which allows directing the program, or they make use of the operating system or hardware support. So a machine level debugger uses the option of hardware or operating system support. For the x86-architecture[Sei09], there is support for two kinds of breakpoints — **soft breakpoints** and **hardware breakpoints**.

Soft breakpoints are machine code which the debugger injects into the program code at run time. For the x86 architecture, this is the machine code for the interrupt number 3, or *INT 3* for short. In fact, the debugger does not inject in the sense of adding new code, but by changing a byte of the opcode. This works, because the opcode of *INT 3* is very short, i.e., only one byte long. When the CPU hits the *INT 3* instruction, it stops the execution and triggers an interrupt, which the debugger handles. Before continuing the execution, the debugger has to restore the old instruction, that it overwrote before with the *INT 3* opcode.

As you can see, soft breakpoints change the program code, which the program loader has loaded into the memory. This has some implications when the user tries to use a debugger to look for malware, because malware often checks the CRC sum of the code and will kill itself, if it changes. Consequently, malware developers can hinder the use of soft breakpoints to debug their malicious code.

Hardware breakpoints solve this problem by offering debug registers. These registers hold the address where the program should be halted. Additionally, there are some flags, which allow creating breakpoints for three conditions: break when the particular address is executed, break when the particular address is written, break when the particular address is read or written. Before an instruction is executed, the CPU first checks if a hardware breakpoint is set. Consequently, it is possible to debug a native program without modifying it, but it is difficult to get around the limited number of registers for general purpose debuggers.

Source Level Debugger

A source level or symbolic debugger maps the source code directly to the application's machine code. A compiler transforms source code into machine code that executes on the hardware platform. The source level debugger's job is to map the machine instructions back to the original source code. This is usually not trivial, because the transformation done by the debugger doesn't need to be bijective, injective or surjective, and therefore the reverse transformation may not be unambiguous. However debuggers typically solve this by compiling some additional information about the mapping into the compiled version. This is the reason, why users should use the "-g" switch for gcc if they intend to debug their program afterwards.

Usually, the users of a high-level programming language prefer a source level debugger, because they usually "think" in their programming language instead of thinking in machine code. However, sometimes the source to code mapping is not available, e.g., when the software vendor only ships a binary version, and in this case, the programmer has to fall back on a machine level debugger.

Interactive vs. Logging Debugger

Program execution is usually much faster than humans can perceive and understand the changes to the state the program has performed. Therefore a debugger should have some means to make them understandable to the user. An **interactive debugger** does this by allowing the user to stop the program and continue the execution from there on. Stopping here means, that the debugger shows the user some kind of prompt or waits for some other action from the user. It usually supports breakpoints and typically the running commands *step*, *next* and *continue*. The user can inspect variables or the stack at any point of the program and they may even modify the program execution by inserting some instructions or modifying variables.

A **logging debugger** on the other hand executes the program without halting. It does however log important information from the execution, but there is no user interaction. After the execution has finished, the user can inspect the log file. On top of the log file, the debugger could present a user interface, which would allow the user to navigate through the different states of the program. Therefore, it could almost feel like the program is running, while it is in fact, only replayed from the log. As the debugger has to do some logging, the program is still somewhat slower than in normal execution.

There are three ways to achieve logging of the programming state while running. The program could run on a virtual machine (see 2.2) which does the logging, or it could have some support from the interpreter of the programming language (see 2.2), or it could use program instrumentation. With program instrumentation the debugger would actually change the code when the program loads. For native code, the debugger would have to change the machine code, but for object-oriented code which executes on

virtual machine like the Java Virtual Machine, the debugger could overwrite the class loading mechanism[LB98].

Debugger for Interpreted Languages

Interpreted languages don't compile to machine code and therefore a machine level debugger would only debug the interpreter itself and not the program written in the interpreted language. Most interpreted languages provide however some means to implement a debugger. For example, the interpreter could call some debugging routine before executing some line of code. The developers of a debugger for an interpreted language have the advantage, that they only need to use one programming language and don't need to understand machine code at all.

Virtual Machine Based Debuggers

One problem of using program directing using the mechanism built-in the hardware or operating system is that they have to change the code. Because they usually operate in user mode, they may also have problems to access code, which they don't have access to, for example code, in the kernel of the operating system[KNM06].

One way to implement a debugger instead of using soft or hardware breakpoints for directing is to execute the program on a virtual machine. The virtual machine can emit events to the debugging process, which gathers the information to present it to the user. One can also implement a virtual machine[DF04] [KTD05], so that it stops executing further instructions on special events. This would represent a break in a program similar to a software or hardware breakpoint, but without modifying any registers or code in the memory.

Reversible Debugger

A reversible debugger is a debugger that allows the user to navigate through the program in reverse. There are multiple ways to achieve this. The debugger could be a logging debugger, which logs the program execution and allows to examine every state of the program after it ends. Using this way, it is however not possible to inject some piece of code into the program flow. Another strategy would be to record the changes of every instruction and when running backwards, undo the changes using the recorded information. A further one is to make continuous snapshots in the program and when going backwards the debugger could recover a previous snapshot and replay the program from there until it reaches the desired position in the program.

Epdb is an interactive reversible debugger. It actually executes instructions and halts the process to give the user live interaction. To simulate reverse execution it uses the snapshot & replay mechanism.

2.3 Python and Debugging

Python comes with a debugger called *pdb*. Pdb makes use of another module called *bdb*. The purpose of *bdb* is to provide a basis for other debuggers. It uses the *sys.settrace()* function to intercept the program execution. Although it is sufficient to have a hooking function like *sys.settrace()* to implement a full featured debugger, lots of organizing work has to be done, e.g., keeping track of breakpoints, resetting the trace function and so on. *Bdb* builds some framework around *sys.settrace* and provides a higher level interface to develop a debugger. The *pdb* module, which uses *bdb* as a basis, adds functionality such as a user interface and higher level debugging features. The whole debugger is written in pure Python, while only the *sys.settrace()* function is implemented in C. It doesn't seem useful to reimplement all those features the debugger already supports. The prototype accompanying this thesis is based on pdb and extends it with all reversible debugging capabilities. This is feasible because all the code is open source.

Python Low-Level Dispatching

The *sys* module, which is part of the Python standard library, provides the function *sys.settrace()*. This function takes one argument — the trace function. When the program sets the trace function, the interpreter calls it whenever one of the following events happen:

call The interpreter calls a function.

line The interpreter executes a line of code.

return A function is about to return.

exception A function throws an exception.

c_call Similar to call, but used when a C function is called.

c_return Similar to return except that it works for C function.

c_exception A C function throws an exception.

The *trace* function takes three arguments: The stack frame, event and arg. The stack frame is the top stack frame of the code that raises the trace event. The arg depends on the event type. At the 'return' event it is the return value; at the 'exception' event it is the tuple $(exception, value, traceback)$; at the 'c_call' event it is the function to be called. The other events have the arg *None*.

When the trace function finishes, the interpreter uses the return value to reset the trace function. This means that the function should return itself to keep the trace function active or otherwise when it returns *None*, the trace function gets deactivated.

Bdb

The main part of the *bdb* module is the *Bdb* class. This class provides four functions — *user_line(), user_call(), user_return(),* and *user_exception()*. These are template methods[GHJV94] which pdb implements. Bdb calls them when the trace function *trace_dispatch()* receives an event of the type 'line', 'call' or 'return'. Bdb calls these methods not directly in the *trace_dispatch()* function. Instead, the trace function calls one of the methods *dispatch_line(), dispatch_call(), dispatch_return()* or *dispatch_exception()* first. These methods do some breakpoint checking before actually calling the *user_*()* methods. The *trace_dispatch()* function and *dispatch_*()* take care of resetting the trace function. Bdb only resets the trace function if there is a breakpoint in the code of the current stack frame or some other reason to stop inside the function. Using these optimizations Bdb reduces the number of calls to the trace function.

user_line()

If the trace dispatch function is set, *user_line()* gets called every time before the interpreter executes a line of code. Bdb usually calls this template method when there is a trace-dispatch event of type 'line', but does some additional work. It checks for breakpoints or other stop information and only calls *user_line()* if a reason to stop exists. So Bdb doesn't guarantee to call *user_line()*, but epdb needs to stop at every atomic instruction to implement instruction counting. In order to achieve this, epdb always sets some stop information to make bdb call *user_line()* on every atomic instruction.

user_call()

Every time before the interpreter executes a function call, it sends a 'call'-event to the trace function, if it is set. Bdb does some preprocessing before it calls the template method *user_call()*. It checks if there is some possibility to stop in this function either by a breakpoint or another reason, as if the user has used the step command. For epdb, the *user_call()* method gets always called for composite functions, because each of the navigation commands work internally like a repeated *step* command to allow instruction counting. The epdb implementation of *user_call()* increments the actual frame count, as every function call also increases the number of stack frames by one.

user_return()

The method *dispatch_return()* calls the template method *user_return()* when there is either a step over the return of a function or the user used the *return* command. Epdb doesn't support the *return* command yet, but as it simulates every command as repeated *step, user_return()* always gets called when the function returns. Epdb also decrements the actual frame count.

user_exception()

Similar to the previous methods, *user_exception()* gets called if there is the possibility to break at the exception, either because of a breakpoint or because of a step command. However epdb uses repeated *step*, and so *user_exception()* gets called at every thrown exception in a composite function.

Pdb

The *Pdb* class inherits from Bdb and Cmd. Bdb provides the basis for debugging the program while the Cmd class handles the user interaction like showing a prompt or parsing user input. *Pdb* also overwrites the *user_*-methods of *Bdb*. Epdb reimplements these methods. However, pdb implements some other methods which are important in conjunction with epdb.

interaction()

The interaction method shows a prompt and shows the user input. This is a method which epdb calls when it stops the program flow, e.g., by reaching breakpoint.

do_*()

The do_*-methods are part of the command dispatch pattern[Boo03] of the base class *Cmd*. These methods are called when the user enters some input, e.g., when the user enters *step*, the method *do_step* gets called. Epdb overwrites some of the commands, especially *step*, *next* and *continue*, and it also adds some new ones, notably *rnext*, *rstep* and *rcontinue*.

CHAPTER 3

Reverse Execution

Epdb does reverse execution by restoring a previous snapshot and executing as many instructions as needed to reach the current instruction minus one. If all executed instructions are deterministic, the process will exactly stop one instruction before the current instruction. The debugger has then to count the instructions, which it does by using the trace function, but one has to be careful that pdb has some optimization, which turns the *trace* function off, if there are no breakpoints. So this optimization has to be switched off.

3.1 Execution Modes

Epdb distinguishes three different execution modes. It uses normal mode when the programmer asks to run or step forward over instructions the first time. To simulate backward running, it activates a previous snapshot and runs the program in replay mode. As it is possible to navigate back and forth, it is possible to run an instruction in the same environment more than once. To allow implementation of deterministic execution behavior, the debugger uses redo mode to execute non-deterministic instructions deterministically. The debugger uses different execution modes to allow controlling instructions differently, depending on its current state.

Normal Mode

Normal execution doesn't differ much from a traditional debugger. One difference is that epdb needs to count the instructions, which pdb doesn't do. For non-deterministic instructions, it also has to record the external state which affects the execution behavior. For instructions with side effects it has to record the external state before the instruction changes it, in order to restore it when the user runs the program backwards, later on.

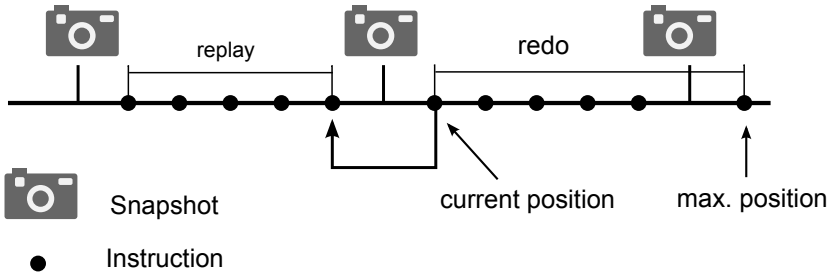

Figure 3.1: Redo and replay modes

Replay Mode

After restoring to an earlier snapshot, the debugger tries to recover a future state from that point on. It does this recovery in replay mode. In the simplest case, it executes the statements as in forward execution. This works for deterministic statements without side effects, e.g., the line:

```
i = i + 1
```

is such a statement. It neither changes the external state, nor does it depend on it. However, there are statements which are dependent on some external state, i.e., they have side effects. Let's look at the example of writing something to disk, using *write()*. When the debugger executes the *write()* instruction in replay mode, the instruction would write something to the disk again. This behavior is usually not wanted. Therefore the debugger has to use a patched version of the *write()* instruction, which actually doesn't write anything to disk in replay mode.

In the *write()*-example, there is another problem to consider, because the *write()* function call returns the number of bytes written. Therefore it has not only side effects, but it is also non-deterministic. So replaying the *write()* call should return the number of bytes written in the original run. So the debugger has to record the return value in normal mode and reuse it in replay mode.

Redo Mode

Redo mode is similar to replay mode. Like replay mode, epdb uses redo mode when the instruction counter is at a position in the program, which was already executed before in the actual timeline[1]. Epdb uses replay mode when it simulates backward running. The part of the code executed in replay mode is always code which the user would not expect to get executed at all. In redo mode, the debugger actually executes code which the user would expect to get executed, but which also has been executed

[1] You may want to read about timelines in section 3.4 before continuing

before. Replay mode is usually not visible to the user because when the debugger has finished replaying instructions, it switches into redo or normal mode before it interacts with the user. So when the user sends the debugger a command to run the program backwards, the debugger executes code in replay mode, but when he sends a command to run forward, the debugger executes the code in redo or normal mode.

Figure 3.1 shows the difference between redo and replay mode in the view of the actual timeline. Each timeline has a current and maximum instruction count. Epdb executes everything after the maximum instruction count in normal mode, but also sets the new maximum instruction count after executing something in normal mode. Everything after the current instruction count until the maximum instruction gets executed in redo mode. To step some instructions back, the debugger activates a previous snapshot and runs forward until it reaches the desired position. While it runs forward to the desired position the debugger sets itself to replay mode. After it reaches the target position, it switches either to redo or normal mode, depending on whether the current instruction is on the maximum position of the timeline or not. Instruction execution in redo mode and replay mode is usually very similar and most of the time patched instructions execute the same way in redo and replay mode.

Undo Mode

Undo mode was the first model to implement side effect management, however it is now replaced by the resources concept. I will describe the undo mode here, because I also want to document the the reasons for the decisions I made, while developing epdb for the sake of completeness. If you want to know how epdb does side effect management now, read Section 3.7.

Undo mode was not really a mode, but in the case of running instructions with side effects backwards, the debugger should undo those side effects. The design was that the debugger keeps a list of all instructions that were executed and which have side effects, together with their instruction count and some additional information for undoing them, which it gathered in normal mode. When the debugger runs backwards, it would look up the list to find the instructions that it has to undo and runs them in a special undo mode.

As it turned out the undo mode approach wasn't very practical, especially because epdb supports timeline switching. With timeline switching, it would be very complex to find which instructions to run in undo mode and after that, which to run in redo mode. It would also not be very efficient if the resource would support restoring previous states as some databases do.

3.2 Snapshots

Snapshotting can be a very convenient feature, even without being able to run a program backwards. For example, take a program, which has a long initialization phase, that requires the user to enter a lot of input from the keyboard. If the defect in the program is at the end, every time the user restarts the application, he has to go through the tedious initialization step. With snapshots, he could have set one snapshot after the initialization and instead of restarting, he could have restored the snapshot then.

Snapshotting was the first feature implemented to achieve the goal of reversible debugging. The technique I have chosen for implementing snapshots was using the system call *fork()*. *Fork()* creates a new process by duplicating the calling process. The decision to use *fork()* has some implications. One of the most important is that the operating system must support this system call, which is the case for Unix-like operating systems, but for example, not for Microsoft Windows. The advantage of this approach is that it is simple and independent of the programming language used and so, the implementation approach could be easily applied to a debugger for another programming language as well. Creating a process with *fork()* is very efficient, because it uses copy-on-write. With copy-on-write the operating system doesn't copy the memory of the process, but instead sets a bit for each memory page that the newly created shares with the old process[Bac86]. If one of the processes writes to a shared page the operating system then copies the page. Therefore, creating a snapshot using *fork()* is extremely fast.

There are two different strategies to create snapshots; I call them local and global snapshot creation. **Local** means that the current executing process knows only about the inherited snapshots, and therefore cannot activate later snapshots, i.e., snapshots which are taken at a higher instruction count. Let's take the example in Figure 3.2, which illustrates an example where local snapshots fail. The type of diagram is an adapted UML sequence diagram. The adaption concerns especially the time of lifelines, which in this diagram, is not time measured in seconds, but instead, measured as the number of instructions executed, i.e., the time in the lifeline is measured in the current instruction count (ic). Therefore the processes can send messages backwards, because the processes can have different instruction counts at the same real time. The diagram indicates this backward sending by giving the *activate()* messages an additional upward or downward direction. It also models the history of a snapshot, which result from the *fork()*, by branching the lifelines. This should mean that the snapshot has copied the creating process along with all its variables. In this diagram, a process makes two snapshots, one after the first and another after the second instruction. After the third instruction, it activates the first snapshot. This snapshot wants to activate the second snapshot, but this fails because there was never an assignment to the variable *s2* in the diagram. Keep in mind that the diagram is very much simplified, in order to keep it neat. In fact, a snapshot wouldn't start running, but instead would create another process using *fork()*, which then runs as the new activated snapshot.

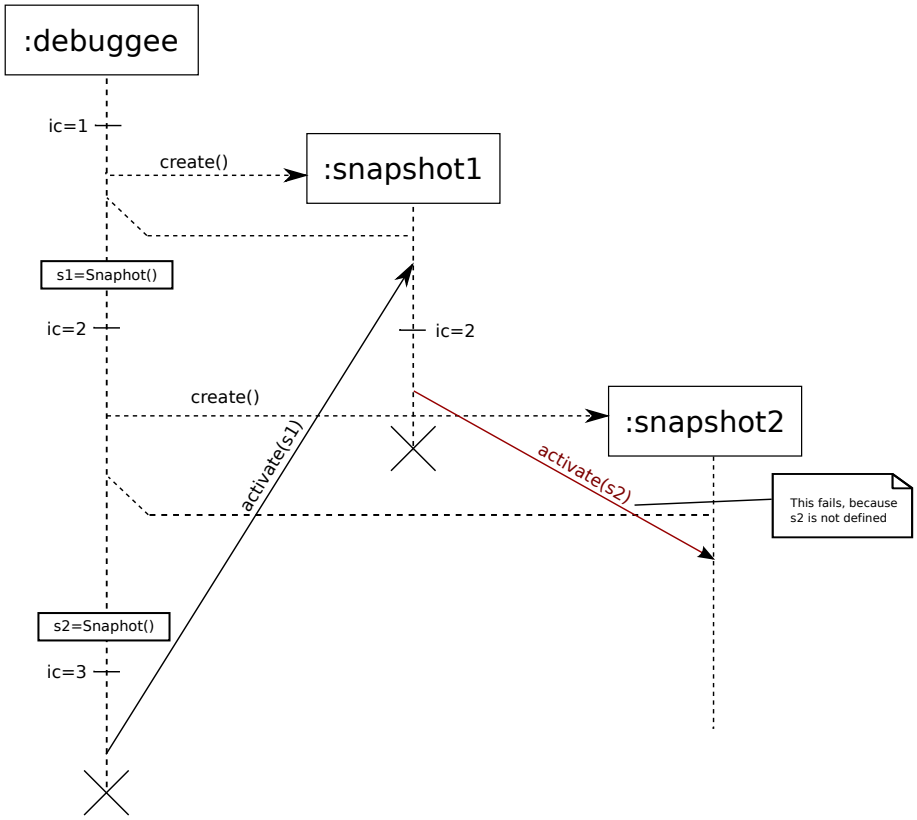

Figure 3.2: Local snapshot fails activating a snapshot

With the **global** strategy, this kind of forward snapshot activation would work, because the debugger would save the state independent of the actual snapshot process. This behavior seems to be more desired. However, it is more difficult to implement, because there needs to be an independent process, which administrates the snapshots, by registering and activating them. As these drawbacks don't seem to be much of a problem compared to the benefits of global snapshots, I chose to implement the global snapshot approach.

Making Snapshots

There are two reasons why the debugger makes a snapshot. Either the debugger makes it to save the internal state after a non-deterministic function, or it makes it to make replaying instructions faster.

Non-deterministic functions depend on the environment. Therefore, the changes to the internal state are not predictable for the debugger without the knowledge of the environment at the time of the execution. One way to get around this problem is to

Listing 3.1: sleep() example.py

```
import time

time.sleep(3)

print("Sleeping done")
```

make a snapshot after the non-deterministic instruction. When the debugger encounters the instruction again in redo mode, it then can activate the next snapshot, instead of executing the instruction. As a result of this, the debugger restores the internal state, not by executing instructions, but by restoring a snapshot. This makes it easy to handle non-deterministic instructions correctly, even without knowing the details of how they work.

If the user debugs a longer running program, the replaying of code could take a lot of time, because the debugger has to recover from a snapshot from long ago. The debugger can reduce the time it needs to replay instructions, by making continuous snapshots when debugging forward. Then it can use a snapshot in the more recent past and start replaying from there. Epdb has a timer variable, which the debugger increases with the time it needs to execute each instruction. When this variable exceeds the time of one second, the debugger makes a snapshot of this position and resets the timer variable. Consequently, replaying will always take less than one second, assuming the execution speed of the program is the same as in the first run. This is because the debugger would never execute the last instruction, because it uses forward activation of snapshots (see Section 3.4).

In the example in Listing 3.1, the *time.sleep()* function waits for 3 seconds. When the debugger replays this instruction, it would actually wait for 3 seconds again. However, epdb makes a snapshot after the *time.sleep(3)*, because the timer variable would exceed the one second limit (in fact it would reach a value of slightly above 3 seconds). Therefore the debugger wouldn't replay this instruction, but would use the snapshot after the *time.sleep()* instead. When the user however steps in redo mode over the *time.sleep()*, the debugger would use forward activation and restore the next snapshot, which is immediately after the *time.sleep()*. Therefore, in redo mode the simulation of the execution of the *time.sleep()* function would be much faster than in normal mode.

Reasons for Using fork()-Snapshots

Using *fork()* for making snapshots has some drawbacks, most notably the operating system has to manage a lot more processes, which may affect the overall performance. Therefore, one can think of implementing snapshots without using *fork()*. The Python programming language stores all its global variables in a publicly accessible dictionary

called *globals*, and the local variables in a publicly accessible dictionary called *locals*. One might try to make a deep copy of these dictionaries, save them and use them to restore them later on. However, this doesn't always work, because some information isn't easily accessible within the Python interpreter, especially if a function is implemented in C instead of Python, which is quite common. Let's look at the standard implementation for file access. The standard library makes use of the standard C implementation of file access. This makes the interpreter portable among all platforms that support C. This implementation allows one to write to a buffer, which is usually not written immediately, but rather when the program calls *flush()* or *close()* on the file descriptor. The Python interpreter however, has no access to the buffer and therefore can't change it, which is necessary in case of going backwards. Using *fork()* allows the debugger to create a complete copy of the process, including the buffer for I/O.

Another advantage of using *fork()* in conjunction with files is that the operating system makes a copy of the file descriptor. Therefore, the file descriptor in the process is open regardless if another process closes it later on. If one tries to implement a process copy on the user side, he would need to take care of that. He would also have to take care of the correct initialization of the registers of the CPU.

An additional advantage of *fork()* is that the operating system uses copy-on-write, which makes creating a copy of a process extremely fast as discussed in Section 5.2.

3.3 Instruction Counting

In order to step an arbitrary number of instructions backwards, it is necessary to count the instructions. In order for the user to step one instruction backwards, the debugger would restore the last snapshot, and then would run the number of instructions to its original location, minus one. Therefore the debugger has to count the number of executed atomic instructions. Epdb doesn't work on bytecode, but uses the trace function to implement a debugger, which Python provides. With this approach, the length of one instruction is usually one line of code. One exception to this are function calls. In case of function calls, the debugger also executes the lines of code inside this function before finishing the line of code where the function call occurred.

The easiest way to implement instruction counting is to set the trace function for every instruction and then to increment an instruction counting variable every time an instruction is executed. Compared to a traditional debugger, this may results in some performance loss. The reason is that in traditional debugging the debugger can set the trace function only for execution frames that contain at least one breakpoint or in one of its succeeding frames. This approach allows optimization in many cases, because usually only very few breakpoints are used compared to the size of the code.

Listing 3.2: rnd.py

```
import random

random.seed()
r = random.randint(0,10)
print(r)
```

3.4 Timelines

If the user runs a program backwards and later decides to go forward again, the debugger has two reasonable ways to execute the instructions. First, it could execute them as they are (non-deterministic execution), or it could make them work the same way as they worked in the first run (deterministic execution). The difference of the two ways emerges when we look at non-deterministic instructions which may change the internal state of the program in a different way each time they get executed. Take for example a program which generates a true random number as you see in Listing 3.2. Let's say the user has debugged towards the end of the program and the it prints the number 4. Then the user decides to step backwards to the *seed()* instruction and after that he goes forward using the step command twice. Now the debugger could either generate a completely new random number or it could show the old one, which is 4.

Both ways to go forward again make sense and can be useful sometimes. Take for a example a program which makes Monte Carlo experiments, and for some reason the program provides a wrong result. If the programmer wants to examine the program in order to understand whats happening, he would most likely use the deterministic version to execute the program. If however he wants to modify the configuration of some variables and sees if the algorithm still returns a wrong result, he would prefer the non-deterministic version. Thus, a reversible debugger should support deterministic and non-deterministic running. Note that logging debuggers like odb[1] only support the deterministic version. However, simply adding two commands to step forward, called for example dstep (deterministic step) and nstep (non-deterministic step), introduces another problem. If the user debugs in reverse, then steps forward using nstep, and then debugs in reverse again and then uses dstep, there would be two ways to go forward again. Either the program can show the output of the first run or of the second run.

To solve these issues, epdb introduces timelines. A timeline is a deterministic way through the program, which can also go only through a part of the program. For this purpose, each timeline has a current and a maximum instruction count. The **current instruction count** is the position inside the timeline where the next instruction, which the debugger should execute, is located. The **maximum instruction count** is the latest

[1] see Section 7.1

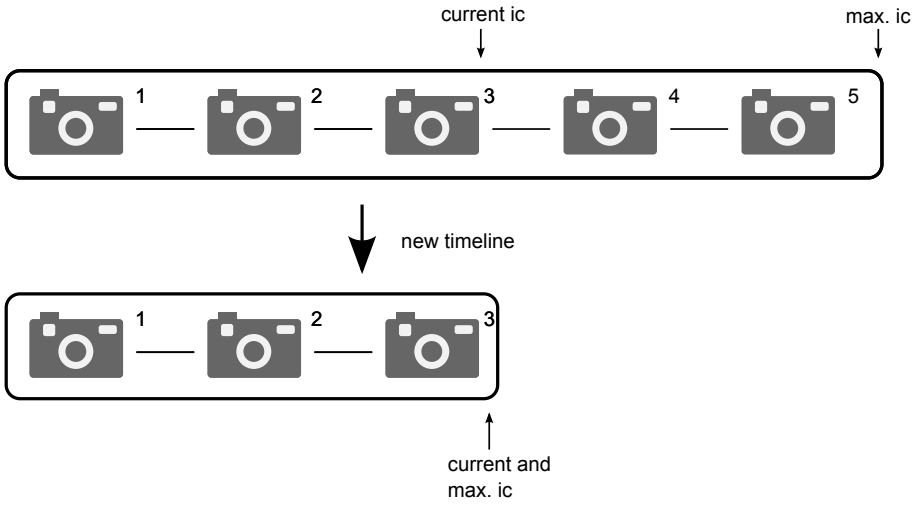

Figure 3.3: New timeline

position inside the timeline the user has already executed.

Every *step*/*next*/*continue* command in redo mode runs the program deterministically like dstep. For non-deterministic execution, epdb can create a new timeline by copying the current timeline up to the current execution point. In the new timeline, the programmer is in normal mode and therefore the debugger executes every instruction non-deterministically from there on. By creating a new timeline, the user can debug every instruction non-deterministically, because when creating a new timeline, the debugger sets the maximum instruction count to the current instruction count. therefore the debugger is in normal mode, which always executes non-deterministically. Figure 3.3 shows the operation of creating a new timeline. The new timeline is a copy of the old timeline up to the current execution point. Epdb avoids copying snapshots later than the current execution point and sets the maximum instruction count of the new timeline to the current instruction count. Consequently the execution mode is normal.

Every timeline in epdb has a name and it is possible to switch between them. This allows the user to make different runs of the program and to compare those two, without needing to restart the whole program. Inside a timeline, everything is in redo mode and therefore deterministic, but if the user wants non-deterministic execution he could create a new timeline, which switches to normal mode and executes non-deterministically from there on until the user gives a command to run the program backwards.

Reference Counting

A timeline consists of a number snapshots and each snapshot can belong to multiple timelines. A snapshot can have two different uses. Either it is part of the timeline and

therefore epdb uses it for snapshot & replay, or the user can use it to set the program to an earlier state by manually restoring a previous snapshot. Having many snapshots may have an impact on the performance, because each snapshot has its own process. Therefore, the debugger wants to keep the number of snapshots to a minimum, but some snapshots are needed because they are part of the timeline. Making a timeline with few or even no snapshots may also have an impact on the performance, because every reverse navigation command would need to reset a very old snapshot or even restart the program. However, it would be advantageous to delete unneeded snapshots. A timeline does not need a snapshot, if the snapshot is not part of this timeline. However, it does need it, if it is part of the timeline. Epdb uses reference counting on snapshots. For each snapshot, there is a count to how many timelines it belongs. The users can only remove snapshots, which don't belong to any timeline. If they want to remove a snapshot that still belongs to a timeline, they have to delete the timelines the snapshot belongs to first. Every time epdb creates a new timeline, it increments the reference count on every snapshot that belongs to the new timeline by one, and every time epdb deletes a timeline, the reference count on every snapshot of this timeline gets decremented. Epdb doesn't support removing snapshots from timelines (except by deletion), because snapshots can be used to replay non-deterministic behavior.

Forward Activation

Let's consider an example in redo mode where the actual position is one instruction before a snapshot and the user steps forward. The debugger has two ways to react. Either it can activate the new snapshot or it can execute the next instruction. Both ways should usually result in the same internal state in case of a deterministic instruction, and because of the resource management, external resources should be in the same desired state in both cases. Consequently the debugger should provide some means to simulate deterministic execution for non-deterministic instructions. One way to achieve a correct replay of a non-deterministic function is to make a snapshot immediately after the instruction. Then, when the debugger runs the program forwards in redo mode, it activates the new snapshot instead of executing the instruction. I call this activation of snapshots, while running forward, **forward activation**. The advantage of forward activation is that it makes it easier to patch non-deterministic instructions. The patched instruction can request the debugger to make a snapshot after its execution in normal mode. When the debugger executes the instruction later on, it doesn't execute it, but instead recovers the snapshot and thus recovers the correct internal state. Therefore, the only code a patched version of a non-deterministic function without contains is an additional request in normal mode for the debugger to make a snapshot.

Forward activation also has the advantage that it restores the correct internal state, even if an earlier patched instruction was defective and led to the wrong program state. Another advantage is that the deterministic execution can be a lot faster, because the

debugger doesn't need to execute every instruction, but instead activates a later snapshot and runs the program just from there. Epdb therefore always uses forward activation and guarantees this to the user, so that he can rely on this property to write his own patched versions for his instructions. There is, however, the small disadvantage that there is no guarantee to the user that all instructions will get executed in redo mode when the user runs the program forward, which can make a difference when the user wants to do something special in the patched instruction, such as doing some visualization. However, I think this shouldn't be much of a problem most of the time, because the state of the program is usually more important than the execution of instructions, and epdb emphasizes the importance of the program state even more by introducing stateful resource management[1].

3.5 Dealing with Non-Determinism

As stated in Section 3.4, there are two ways to execute a non-deterministic instruction. Executing an instruction non-deterministically is usually simple, because the debugger just has to execute it without further directing. However, for deterministic execution though, it has to do additional work. It has to use the information which it recorded from the previous run to simulate the running behavior of the previous run. As epdb does non-deterministic execution with the use of timelines, I want to concentrate on deterministic execution here.

To execute instructions deterministically, epdb has to log the effects of atomic non-deterministic instructions. Epdb considers all functions of the standard library as atomic functions. If one treats the effects of all atomic non-deterministic functions correctly, one implicitly treats all composite non-deterministic functions, because the non-determinism always results from some external state, which can only be accessed by functions that escape the interpreter in some way. These functions have to be atomic for a source level debugger. Composite functions may have non-deterministic behavior because they depend on some non-deterministic functions either directly or indirectly (i.e., by calling other functions in between), but there are no other sources of non-determinism. So if one fixes all atomic non-deterministic functions by simulating a deterministic behavior, all composite non-deterministic functions will be simulated correctly too.

To simulate deterministic behavior, the debugger therefore needs to record the non-deterministic behavior when the function is executed in the first debugging run, i.e., when it executes in normal mode. There are two different approaches for recording non-deterministic behavior. The debugger could make a snapshot after the instruction in normal mode and then rely on the forward activation property as described in Section 3.4. The other approach is to save the data from the environment which changes the

[1] see Section 3.7

internal state. When the function is replayed in replay or redo mode, it is not executed, but the recorded behavior is looked up and simulated. Typically, a non-deterministic function manifests by having a non-deterministic return value. In this case, the debugger would store the return value with the corresponding instruction count where the function was called. Epdb stores this key/value pair in a shared dictionary called *nde* (non-deterministic effects). When the debugger replays the function, it would look up the behavior and simulate it.

I want to illustrate this with an example. Assuming the debugger executes the function *time()*, which returns the actual time. Then the patched function would save the actual time in the *nde* dictionary at the current instruction count, before it returns. The debugger later executes it in redo mode, e.g., because the user stepped back and then stepped forth. Then the the patched function would return the value of the dictionary *nde* at the position of the current instruction count. Since the instruction count is the same as it was when the debugger has executed the instruction in normal mode, it would therefore return the same value as in the first run.

There is a caveat with multiple processes and their synchronization. The process which runs the function first is different one than the one which replays it, and the *fork()* of the processes is done before they are executed. Therefore the record of the behavior should be done in some data structure, which is independent of the process, e.g., a shared memory as described in Section 4.3.

3.6 Dealing with Side Effects

Atomic functions with side effects may change the external state of the program. When redoing or replaying some piece of code, the debugger should restore the external state of the program. A function which has side effects can be either deterministic or non-deterministic. A deterministic function with side effects does change the external state, but the internal state is not affected from the environment. Therefore the debugger can ignore the effects of the environment to the execution in this case. If the function is however non-deterministic, the debugger can use the same approach as mentioned in Section 3.5.

Side effects don't affect the internal state, and if the programmer is only interested in the internal state, the debugger can even ignore them. However, ignoring them may completely mess up the environment, so that when the debugger switches to normal mode, either by creating a new timeline or by running over the maximum instruction count of the timeline, the state of the environment may be inconsistent. Therefore, it makes sense that the debugger also manages the external state, i.e., it should log changes to the environment and restore the state when it executes instructions in replay or redo mode, or switches to another timeline. The debugger, however, can reduce the amount of external state recoveries by using the fact that side effects don't change the internal

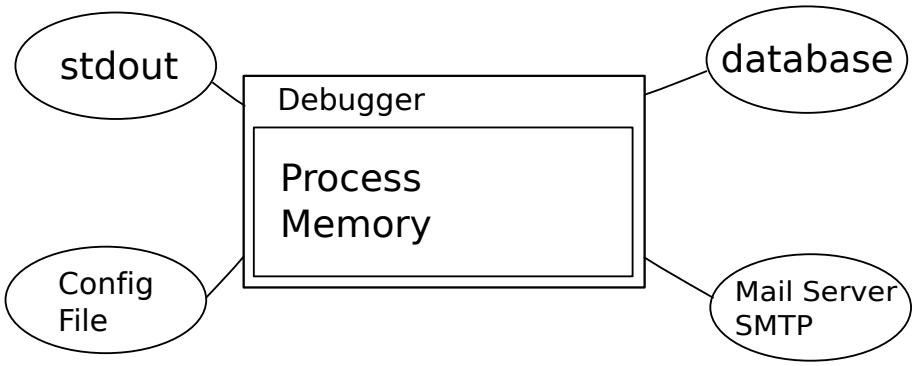

Figure 3.4: A process and its resources

state. Since the user only examines the state when the program breaks, i.e., the debugger shows the user a command prompt, the debugger doesn't have to restore the state when it is replaying multiple instructions, but only before it interacts with the user, or when it switches from replay or redo to normal mode. The debugger does only one restore of the external state after the user sends it one navigation command. This is fortunate, because restoring the external state could take quite a lot of time. However, as it has to do the restoring only once, the user may not even notice the time the recovery needs if it takes less than about 100ms, which is about the time an eye needs to blink.

The debugger has to manage the external state. The external state is only the subset of the program that affects the program execution or is affected by it. However not every instruction changes the whole external state, but rather only parts of it, e.g., an instruction that writes something to disk only changes the state of the file, but does not change the text on the terminal. Consequently, it makes sense to further divide the external state into **resources**. For example, a file which the program uses is such a resource.

3.7 Resources

Epdb allows controlling external resources like databases, files or other processes. A resource is some part of the external state. Figure 3.4 illustrates a debugger with four different resources. When using epdb, the debugger does not have its own process but is part of the program which the debugger executes[1]. The memory, which the process and the debugger use, represent the internal state which epdb saves when it makes a snapshot. The state of the database, the mail server, standard output and config file represent the external state of the program, and each of these components is a separate resource.

[1]In debuggers which would work on native code, the debugger would typically have its own process

Some operations on resources do not allow reversible debugging. For example, one usually can't undo the deletion of a file. However, if the debugger saves the content of a file before it deletes it, the debugger can undo the deletion. To restore the content, epdb doesn't try to undo the deletion, but instead it saves the state of the file before it gets deleted. Then, when running to the instruction count before the file deletion, it restores the file.

Epdb supports an abstraction to implement such additional actions like saving and restoring state of resources to make operations reversible. For every kind of resource the program uses, epdb needs a **resource manager**. There exists, for example, a resource manager to deal with files and another for databases. Every resource has a state, which the resource manager labels with an identifier. The identifier should be unique and therefore a UUID[LMS05] offers some reasonable implementation. When an instruction changes a resource, the patched version of this instruction calls the *save()* method of the corresponding resource manager after it has changed the resource. The resource manager then returns the identifier of its state, which epdb manages. The resource manager also has a *restore()* method, which epdb uses to restore to some previously saved state of the resource. As the resource managers should persist switching snapshots and therefore switching processes, it is necessary to make them distributed, and therefore pickleable[Bea09], in order to serialize the object. A resource manager should also implement the magic *__reduce__()* method. Listing 3.3 shows the skeleton of a resource manager.

This design of resources isn't arbitrary, but inspired by how some modern databases deal with their data. For example, the Oracle database supports a technology called flashback[MAA+10]. With flashback, the user is able to restore the database to any previous state. Every state of the database gets an ID, which the user can use to identify a state. Some NoSQL databases provide similar version control of data. For example, CouchDB[ALS10] keeps the history of each piece of data and provides a unique identifier for each version. Making a resource manager for such a database should be straightforward. For files, it should be possible to implement a file system which allows restoring old versions of the files by never deleting them. Using such a file system would allow the debugger to transparently implement reversible debuggable files. NILFS[Lay09], for example, has many such properties required for a file system for reversible debugging.

Managing Resources

The resource managers offer a simple interface to save and restore the external state. However, they just assign the different states some identifier and thus there is no relation between the identifier for the state and the instruction count to which the state belongs. This relation is necessary for the debugger, because it has to restore the external state for a given instruction count. For example, if the debugger stops in redo

Listing 3.3: Skeleton Manager

```
class SkeletonManager:
    def __init__(self):
        "Initialize the manager"

    def save(self):
        "Save the actual resource and return an ID"
        return id

    def restore(self, id):
        "Restore the state from an ID"

    def __reduce__(self):
        "Make the manager serializable"
        return (SkeletonManager, ())
```

mode at instruction count 10 after an instruction, which has printed something to the screen, then it has to restore the state of the screen for instruction count 10. However, the resource manager doesn't know anything about instruction counts. Therefore, the debugger has to manage the identifiers it receives from saving resources. Epdb establishes the relation between state identifiers and instruction counts by using a distributed dictionaries for each resource the program uses. This dictionary has at each instruction count, where the resource changes, the corresponding state identifier. The debugger updates this dictionary when the resource manager saves the state of its resource. Then, it uses the current instruction count as key for the new entry in the dictionary and the state identifier, which the resource manager returns, as value. When the debugger needs to restore a resource, it can look up state identifier for the highest instruction count in the dictionary, which is lower then the current instruction. This state identifier it can then send to the resource manager to restore the desired state of the resource.

Each timeline may have different external states at the same instruction count number. Therefore, each timeline needs to keep its own dictionaries for the resources. When the user creates a new timeline, the debugger has to copy the resource dictionaries, so each timeline has its own. If the current position in the timeline is not at the end of the timeline, the debugger generates only reduced copies of the dictionaries. These reduced copies only contain instruction numbers up to the current instruction count. Newly created timelines only describe the execution up to the current instruction count, and therefore, its resources only need the state for instruction counts up to the timelines maximum instruction count, which is equal to the current instruction count.

The resource dictionaries have to be distributed between processes. Each timeline can consist of multiple snapshots. The debugger may activate any of these snapshots and in this case, the process associated with the activated snapshot needs access to the

resource dictionaries.

3.8 Instruction Patching

Epdb has three modes of execution: normal, replay and redo. While replay and redo almost always work the same, there is very often a difference between normal and replay/redo mode. Take for example the *write()* function, which writes some bytes to disk. In normal mode, it should actually write something to disk, but in replay mode, it wouldn't be advisable, because the bytes would already have been written to disk by the first execution in normal mode, and writing to disk again would change the file to some wrong state. In replay and redo mode, the file shouldn't be changed at all, because managing the state of the file is the job of the resource manager. In fact, *write()* should do nothing, except returning the number of bytes written in the first run.

The debugger knows how many bytes were written in the first run by using a modified version of *write()*, which actually stores the number of bytes written in some debugger internal variable. Epdb supports the shared dictionary *nde*, which the *write()*-function of a patch module can use to store information the function needs to implement deterministic execution behavior.

As we see, a reversible debugger requires a different implementation of a function in replay/redo than in normal mode. Consequently, it needs a way to patch the function, but this patching shouldn't change the execution while the program runs without a debugger. The patching has to be done at runtime or when the program loads. For dynamic languages, patching the function at runtime seems to be the cleaner way. Patching a function at runtime is often called **monkey patching**[Zia08]. Monkey patching is often used in conjunction with software testing. There, it is used to inject a fake or mock object into the the runtime environment, instead of the object itself. It may also be used to fix a bug in a running server, which shouldn't reboot to fix the bug. As I believe that patching of instructions has nothing to do with primates, I want to use the more generic term **instruction patching** in this paper. It is more generic, because it does not necessarily mean that the patching is done at runtime. It may also be done using program instrumentation at class loading for a statically typed language like Java. If I emphasize that the patching takes place at runtime, I will use the term **dynamic instruction patching**. The name instruction patching also has the advantage that it emphasizes that the debugger replaces atomic instructions which also have some time context.

With dynamic instruction patching, the function or object is simply replaced at runtime with another one. As functions, objects, and classes are first-class objects in Python, the patching can be done using simple assignments. Using instruction patching, the debugger can use a different version of a function, which distinguishes between normal, redo and replay mode.

Epdb uses instruction patching to implement patched versions of functions and ob-

jects, but it is still unclear when the patching takes place and how the code is organized to provide patched versions. Epdb distinguishes between the *builtins* module and normal modules. The *builtins* module is loaded even without an *import* statement, so the debugger patches *builtins* at the start of the program. For the rest of the modules epdb overwrites the import mechanism, to patch them at import time. Python allows this by overwriting the *__import__()* function.

The code for the patched versions of the functions is located in their own module which has the same name as the original module, but has two additional underscores at the beginning. To patch the `random.py` module, the programmer would write a `__random.py` module with the patched code. When the program imports a module under debugger control, epdb merges the module with the patch module. It looks for a symbol in the original module and if it finds it, it looks in the patch module to find it there and in that case overwrites the original one. In this case, the module provides the original symbol with prefix *__orig__*, in order to allow the patched implementation to access the original implementation. If the patched module doesn't provide the symbol of the original one, the original implementation is used. If a patch module provides a symbol, which does not exist in the original module, it will be ignored and not accessible by the program. Such symbols can be used by the patch module to use patch module locale variables, which are not accessible from the program itself.

Listing 3.4: ex.py

```
def foo():
    return 1

def bar():
    return
```

Listing 3.5: __ex.py

```
def foo():
    return 2

def baz():
    return
```

Listing 3.6: view

```
def __orig__foo():
    return 1

def foo():
    return 2

def bar():
    return
```

Listing 3.4, 3.5 and 3.6 show an example of merging two modules. The module in Listing 3.5 overwrites the function *foo()*, but keeps a copy *__orig__foo()*, which works the same as in figure 3.4. Since the `__ex.py` doesn't overwrite the function *bar()*, it looks the same after merging as in `ex.py`. The *baz()* function has no corresponding function in Listing 3.4, and therefore it is only local to the `__ex.py` module.

3.9 Atomicity

How far can a programmer step into an instruction? The answer to this question very much depends on the programming language and the debugger. Python is an interpreted

language and it doesn't seem to make sense to debug the interpreter itself using a Python debugger. The CPython interpreter isn't even written in Python itself, but in C. If a user wants to debug the interpreter, he better chooses a debugger for C code such as gdb. Even if we ignore the interpreter itself, we may still encounter C code, because a module for Python can be written in C. The Python debugger pdb stops here and ignores stepping into functions in modules written in C, however, it allows stepping into modules of the standard library. This is helpful for people who develop the standard library, but is usually annoying for programmers who don't. Many of the standard library modules do some wrapping, before actually calling some module written in C. Therefore, most of the time a user doesn't get a lot of insight just by stepping into a standard library module. If they jump into such a module by accident, they often try to come out as fast as possible. If the debugger allows stepping into a standard library function, jumping into the standard library code is sometimes unavoidable by the person doing the debugging. For example, if a function call, which the user is interested to debug, is on the same line as a call to a standard library function, but the debugger would execute the standard library function first, then a step at this line would access the standard library function before the user defined function.

Epdb avoids stepping into standard library functions. One reason is that it is often more annoying than useful. Another is that it makes it easier to count instructions correctly, because the library function might have non-deterministic parts in it. If epdb would step into the standard library function, the implementer of reverse debugging capabilities would need to make those function calls inside the standard library work in reverse. Those are often very poorly documented, if some documentation even exists. The standard library itself however has very good documentation, and therefore it is usually easier to implement reverse debugging for a standard library call.

One could argue that implementing reversible debugging for primitive functions is preferable to implementing reversible debugging for more complex ones, because the complex ones consist of primitive functions, and programmers would only have to implement a few primitive ones. However, it is often not possible to implement reverse debugging for primitive operations, while it is possible to undo more complex ones. Let's take for example a program which sends a message to another process which shows the message on the screen. A primitive operation would be, in this case, the sending of the message. There is no way a program can undo that. When the message is sent, it gets on the wire, the other process receives it, and does its actions and then the debugger has no information what the other process has done. However, if we look at the more complex function "send a message to the other process to write something on the screen", the debugger has suddenly much more knowledge of what is happening. For example, to undo this action, it could send a message "undo the effects of the last message I sent you" and the other process could restore the previous state. Of course, designers of such a system need to pay close attention to such requirements.

CHAPTER 4

Epdb Internals

Up to now, I have presented the overall architecture for a reversible debugger. Epdb is a fully working prototype and therefore I had to do many more specific decisions and detailed design. I want to present them in this section.

4.1 Snapshot Processes

Epdb has three kinds of snapshot processes. There is one **controller** process, one **debuggee** process, but multiple **snapshot** processes, i.e., as many as there are snapshots.

Epdb uses the controller process to manage the communication between the snapshots. If there wouldn't be a controller process, every process would need to know about every other process, because every process may activate any other snapshot and this snapshot may activate any other snapshot again. The decision would be between a full mesh or a star topology, but a full mesh topology would require a lot of work to keep the connections alive and a lot of connections between the processes, i.e., $n(n-1)/2$, where n is the number of snapshots. A star topology requires only n connections. However, it requires one additional process, the controller process. Epdb uses a star topology like in Figure 4.1. This communication topology always routes messages between a snapshot and the debuggee process through the controller process.

Beside the snapshot and the controller process, there is also the debuggee process. The debuggee process is the one that does the debugging at the moment, by interacting with the user. It is the process which actually does work, while the others are just waiting most of the time. There is only one debuggee process, because otherwise the two or more processes would confuse each other. When the debuggee activates a snapshot, this snapshot initiates a new debuggee and the previous debuggee process terminates, so that there is only one debuggee at any time.

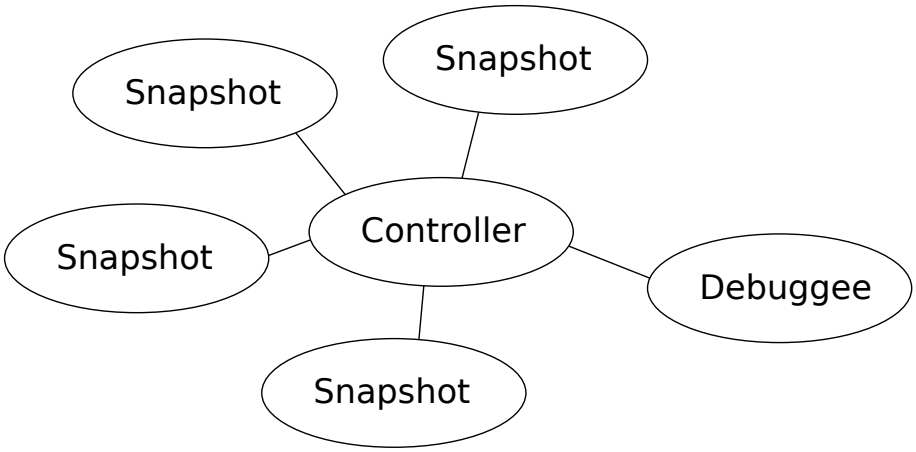

Figure 4.1: Star topology

Epdb implements each snapshot as a process. When the debuggee makes a snapshot, it calls the system call *fork()*, which creates an exact copy of the process, except for some details like the process id. The newly created snapshot process registers itself with the controller and then waits until it gets the **activation** message from the controller. When it gets an activation message, it spawns a new process using *fork()* again, because otherwise the snapshot would get lost, if it would start running without making a copy of itself. The newly created process then becomes the debuggee. The **end** message is another message the controller sends to the snapshot processes. This message instructs the processes to terminate. The controller uses this message to delete a snapshot, either when the user requests to delete it, or when the debugger ends, every snapshot process gets an end message in order to make them terminate. The controller does this form of closing in order to be able to use the system call *wait()* on them, to prevent them from becoming zombie processes.

The debuggee uses the controller to activate snapshots. It does so by sending the controller an **activation** message. In this message, it also sends the *id* of the snapshot to activate. The controller then looks the *id* up and sends an activation message to the addressed snapshot, which spawns a new debuggee. There are two other types of messages the debuggee may send to the controller. One is the **list** message, which makes the controller process list all snapshots, and the other is the **quit** message. This message makes the controller send the end message to every snapshot, waiting for their termination, and then terminating itself.

Snapshot Communication

As there are multiple processes which depend on each other, there must be some sort of communication between them. Python supports some of them out of the box, namely

signals, queue, shared memory and sockets. There are also third party modules for interprocess communication, for example, POSIX or System V queues. The requirements for the communication system don't seem very high. There are not so many messages going around, but it is necessary that two process, which are not in a parent/child relationship, can easily establish a connection. I tried quite a few forms of interprocess communication solutions, before I decided to go for sockets.

Signals

Signals are a very simple form of communication. The Python standard library has support for them. Useful for implementing them is the *signal.pause()* function. This makes the process stop until it receives a signal. Signals seem to be an adequate solution, if you have only local snapshots. For global snapshots, there is much more communication between the processes and signals, so there don't seem to be a sufficient solution to it. Signals also suffer from portability issues.

Queue/Pipes

Python comes with a *multiprocessing* module. This module contains the classes *Queue* and *Pipe*. Although they are used differently, they share almost the same advantages and disadvantages. One big advantage is that they are not only bundled with Python, but also implemented portably and work on almost every platform. However, they have a problem: it is not easy to create a Pipe or Queue between two process, which do not know about each other (i.e. for example in a parent-child relationship). This makes it very difficult to establish communication, especially for global snapshots.

Shared Memory

There is support for shared memory in Python. Python supports a *Manager* object. This object is a separate subprocess. Using shared memory this way wouldn't save any processes. It would be possible to save the data related to snapshots in the shared objects. Consequently, every process would have access to all snapshot data, but shared memory doesn't solve the synchronization problems. These could be resolved using locks, but this would be more complicated than necessary.

POSIX/System V Queues/Pipes

As the Python Queue/Pipes had some shortcomings with respect to establishing communication, I looked at POSIX and System V queues and pipes. These are not part of the standard Python distribution, but there are third party modules supporting them. As it turned out, the problem of establishing a connection between two foreign processes isn't well supported.

Sockets

Python supports programming using sockets, so one can start with them without preparation. They also solve the problem of establishing a connection between two foreign processes since there is the function *bind()*, which gave the socket a name. As sockets are usually used for network programming, there are of course some security concerns, such as ensuring, that no one on the Internet is able to control another user's machine. Luckily, there are Unix Domain Sockets, which do not work over TCP or UDP, but over a local file. There are still portability problems here, because they are not well supported under Microsoft Windows, but as I broke compatibility before, this shouldn't matter.

Snapshot Activation Modes

The snapshot implementation of epdb supports three different modes of snapshot activation. The type of snapshot activation is transparent to the users, i.e., they will not see which activation mode the debugger uses. The reason for different activation modes is that the debugger usually doesn't stop after it activates a snapshot, but goes some steps ahead. The most often used snapshot activation mode is **counting activation**, but epdb uses **frame count activation** and **continue activation** in some special cases.

Counting Activation

With counting activation, the debugger activates the snapshot and instructs it to run forward until it reaches a given instruction count. Then the debugger interrupts the execution and calls the *interaction()* method. Epdb uses counting activation, if it knows in advance at which instruction count it has to break. This is always the case with *rstep*, *rnext*, *rcontinue* and *step* and often, but not always with *next* and *continue*.

Frame Count Activation

Epdb uses frame count activation in a special case, where counting activation doesn't work, due to the fact that the debugger doesn't know at which instruction count to stop when it activates the snapshot.

In Figure 4.2, the program is inside the function *foo()* and calls the function *bar()*. The user also has executed the *bar()* function partially in normal mode and after that, has run the program backwards up to the invocation of the function *bar()*. The current instruction count of the timeline is at the position immediately before the invocation of *bar()*, and the maximum instruction count is inside the function *bar()*. Inside the function *bar()*, the debugger has also made a snapshot, either because the timer variable exceeded its limit, or a patched function requested to make a snapshot for deterministic execution. When the user initiates *next*, the debugger has no information at which instruction count it should stop, because the target location at the position where *bar()*

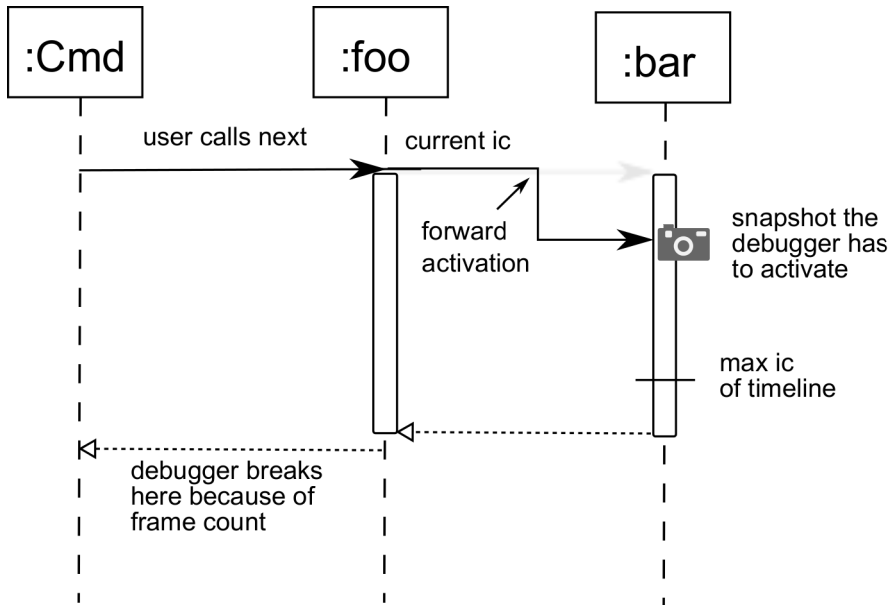

Figure 4.2: Frame count activation example

returns was never executed before, and therefore the debugger hasn't saved any execution path information at that location yet.

The debugger also can not run *next* as in normal mode, because there is a snapshot which it has to activate to guarantee the forward activation property. In this case, epdb uses frame count activation. Before it calls the function, it saves the number of stack frames it actually has. Then it activates the latest snapshot inside the function and passes it its actual frame count. Then the activated snapshot runs the program until it reaches this frame count or less. Consequently, the debugger precisely stops at the position at which the function returns. The "or less" condition is necessary to support exceptions which may leave the function at a lower frame count.

Continue Activation

Continue activation is similar to frame count activation, but used in conjunction with the *continue* command. Consider an example where no future breakpoint in redo mode exists, but the debugger has made a snapshot, which it should activate. Figure 4.3 illustrates this. In this case, the debugger has to activate the latest snapshot and execute it over the maximum instruction count of the timeline. At this point, it has to switch to normal mode and run it until it finds a breakpoint in normal mode. As the debugger has no information at which instruction count the breakpoints occur[1], it has to switch

[1]breakpoints reside on lines in source code, not instruction counts

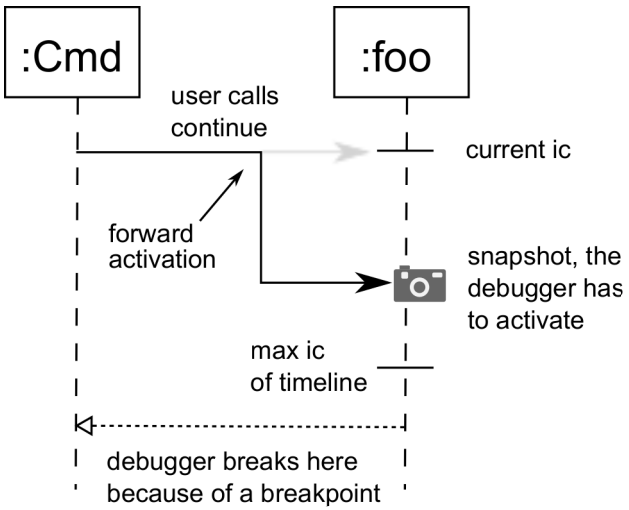

Figure 4.3: Continue activation example

in order to run the program until it reaches a breakpoint. At the snapshot activation, it has to instruct the new process to run until it reaches a new breakpoint. I call this type of snapshot activation — **continue activation**, because it makes the new process run in continue mode after it gets activated.

4.2 Bookkeeping in Epdb

Epdb records the data change history with the use of snapshots, but it also has to record the execution path history. It does this by counting instructions. This works for the *rstep* command, because *rstep* only requires to activate a previous snapshot and then to run from there until it reaches the number of instructions minus one. However, for *rnext* and *rcontinue*, this isn't sufficient. With just the instruction count, there is no way to predict at which instruction number a breakpoint resides or where the debugger called the corresponding function. In order to make *rnext* and *rcontinue* work, the debugger has to track additional information.

First, the debugger needs to stop at a breakpoint when the user initiates a reverse execution command. The user is able to set a breakpoint after it has executed all the code and then can run the program backwards to this breakpoint. Therefore, the debugger can't simply save the instruction counts with the breakpoints when it encounters one, while running the program forward, because the breakpoints may not exist yet. Consequently, the debugger has to save the instruction counts for every line of code, which it executes. Epdb does this in a dictionary, the **continue_dict**. It uses a tuple containing the filename and the line number as key. This information specifies exactly one line

of code. As a value it uses a list of instruction counts, which are executed on this line of code. Using this dictionary, it is possible to calculate the position of the breakpoint when running multiple instructions backwards or when running forwards in redo mode.

For *rnext* to work, the information saved in the *continue_dict* is not enough, because the debugger needs to know on which instruction counts the functions have returned. It keeps another dictionary, the **rnext_dict**. Every time the debugger encounters a *user_return()*, it adds an entry to the dictionary with the current instruction count as the key and the instruction count of the corresponding *user_return()* as the value. Consequently, it is possible to calculate the position in the program where a *rnext* should stop the program execution.

There is another situation where the debugger needs additional information and that is in case of a *next* command, when the debugger is in redo mode. There are maybe snapshots inside the function and therefore epdb must activate the latest snapshot inside the function, because of the forward activation property. Epdb keeps another dictionary, the **next_dict**, which is the reverse form of the *rnext_dict*, i.e., it has the instruction count from the corresponding *user_call()* as the key and the instruction count at the *user_return()* as the value. Using this dictionary, the debugger can calculate the nearest snapshot to the target position and therefore is able to guarantee forward activation.

With *next_dict* and *continue_dict*, there is the problem that the information the debugger saves may be used by a snapshot made earlier. As this earlier snapshot hasn't recorded the information of *next_dict* and *continue_dict* for later instruction counts, it wouldn't have this information. Therefore epdb must use a shared dictionary for *next_dict* and *continue_dict* to make this information available to all snapshot processes. This shared dictionary should belong to a timeline, i.e., each timeline should have its own version of *continue_dict* and *next_dict*.

With *rnext_dict*, the situation is a little different. It is only needed for information in the past. Therefore, the debugger doesn't need to share the dictionary with other processes, because each process can track the information it needs.

4.3 Shared Memory

As described in 3.5, the debugger has to record the behavior of non-deterministic functions and exchange the recorded data with other processes. Section 3.7 showed that epdb makes use of resource managers, which must be synchronized among all processes. Section 4.4 shows that breakpoints need to be distributed. Epdb also has to keep track of the execution path history which should be available to all snapshots, and last, but not least, it has to keep information of timelines synchronized over all processes.

To synchronize all the data, epdb uses its own server process, which it starts at its own startup. Most of the data can be handled in a shared dictionary, but the breakpoint

implementation[1] makes some usage of lists. Therefore, the implementation should support lists as well. Epdb does not need synchronization, i.e., locking before accessing data, which most shared data implementations provide, because only one process is active at the same time.

In order to implement a shared dictionary, I needed some communication technology. I first considered using the *multiprocessing*-module. There exists a *Manager* or alternatively *BaseManager* classes, which would allow implementing a shared dictionary easily. As it turned out, the module wasn't well designed for using it with *fork()*, but instead, with a *Process*-data structure which doesn't guarantee the use of *fork()*. Using *fork()* in conjunction with the *Manager*, I ran into some nasty race conditions which I wasn't able to fix due to the complex implementation that does much more than I actually needed.

Therefore, I decided to implement a simple shared dictionary on my own. This isn't very complex, as Python provides the well implemented *pickle* module, which allows serialization of objects. For inter-process communication, I decided to go for Unix domain sockets for the reasons discussed in Section 4.1. At the start of the debugger, epdb launches a process which manages the dictionary and listens for incoming connections. All other processes connect to the managing process at their start. To access the dictionary, the client sends a tuple of three elements containing the method name, the positional arguments and the named arguments. The server executes the method with the arguments and returns a tuple of two elements. The first element indicates whether the method was successfully executed or if an exception was thrown. The second contains the return value or the exception. I also added a proxy[GHJV94] at the client side for more transparent usage of the dictionary. This implementation also has the advantage that it works with complex objects like resource managers, as long as they are pickleable. However, this can be easily achieved by implementing the *__reduce__* method of the class.

This implementation works well for what it is intended for, but also has some limitations. For example, it is not possible to receive an iterator of the dictionary, because iterators are not pickleable. However these limitations are no problem in the described case.

Figure 4.4 shows how the shared objects are arranged. First, there exists a timelines object, which is a dictionary that holds all timelines. A timeline is a more complex object, which consists of dictionaries for non-deterministic effects, resources, managers and, the execution path history information stored in the dictionaries *next_dict* and *continue_dict*. Epdb stores the information of breakpoints independently from the timeline.

[1] the breakpoint implementation is almost copied from pdb, except that epdb stores breakpoint data on the distributing server

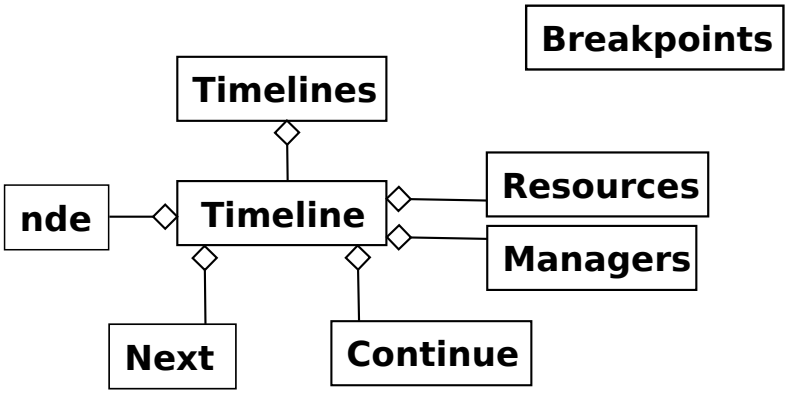

Figure 4.4: Organization of shared data

4.4 Breakpoints

There are different processes communicating with the user. Therefore breakpoints should appear to the user in the same way regardless which process is currently the debuggee. Without changing the bdb implementation, every process would handle breakpoints separately. This means that after restoring a snapshot, the user would probably see fewer or different breakpoints. Therefore, I decided to implement a distributed version using the shared memory approach. As it turned out, I could reuse the implementation of breakpoints from bdb and pdb, but I had to replace the data structures, which store the breakpoint information, with shared data structures, i.e., remote proxies, which forward the function calls to the shared memory server. For the breakpoint implementation, I needed a shared dictionary and a shared list, because bdb uses those data structures to index the breakpoints in different ways.

For the implementation of shared breakpoints, there are two reasonable ways to implement them. They could either belong to a timeline, or they could be global. If they belong to a timeline, the users would see different breakpoints for each timeline. In the global approach, the users would see the same breakpoints when they switch to another timeline. I decided to go for the global approach, because breakpoints belong to some line in the source code. As the source code doesn't change when the timeline changes, this seemed more reasonable to me. However, the timeline approach has advantages too. For example, the user may want to search for a different bug in two different timelines and therefore would need different breakpoints in each timeline. Therefore, a full-featured implementation of a reversible debugger might implement both.

4.5 Implementation of Debugging Commands

Although pdb provides an implementation for the standard debugging commands *step*, *next* and *continue*, their implementation doesn't combine well with instruction counting. The reason is that pdb doesn't call a trace function on every instruction executed, but only on those, which are located in frames, that have a breakpoint in it. This optimization makes debugging faster, but doesn't allow applying instruction counting. Therefore, it is necessary to rewrite those functions.

Step

The *step* command in epdb works similarly to the one in pdb, as it only steps one instruction forward. Therefore, it doesn't interfere with instruction counting, because instruction counting requires the debugger to stop after each atomic instruction. Step goes exactly one atomic instruction ahead, and stops exactly where the instruction count mechanism increments its instruction count. Therefore the stop mechanism is the same as in pdb in normal mode, but epdb also supports the redo mode. In redo mode, epdb first checks if there is a snapshot at the next instruction for the current timeline. If it is the case, it activates the snapshot instead of stepping one instruction forward, to guarantee the forward activation property of the debugger. After a successful step in redo mode, epdb has to restore all resources to their state for the actual instruction count.

Next

Next works similar to *step*, except when the current line contains at least one function call. Then the *user_call()* method is called before *user_line()*. In the pdb implementation, the *_stop_frame* is set to the current frame. This means that the debugger only stops when it returns from the function call to the current frame. It also means that the instruction count would only be increased by one, because the trace function would only be called once. This is undesirable because epdb would confuse the instruction counts in replay mode, because there, it would step into the function.

To fix this issue, epdb implements *next* similar to *step* in that it calls the trace function on every call, but doesn't always calls the *interaction()* method. If the debugger runs over a function call, it calls the *user_call()* method beforehand and the *user_return()* method afterwards. Before calling *next* the debugger stores the number of stack frames which the debuggee currently has. When it steps into the function, the function call sets up a new stack frame and increases the number of frames. When it leaves the function, it deletes a stack frame and decrements the number of frames. The debuggee may create additional frames by calling other functions, but when it equals the number of frames it stored previously, it has returned from the function. If it is less than this number, then it has left the function via an exception, and therefore has reached the position where

the next command should present the user a command prompt. Furthermore, the next command should also consider breakpoints. A breakpoint check in *user_line()* solves this.

Like *step*, *next* has to treat normal and redo modes differently. In redo mode, epdb tries to find an appropriate snapshot, and then activates it. In contrast to *step*, *next* requires two different activation modes. First, a snapshot activation, combined with going a number of steps forward (see counting activation in Section 4.1), and second, a snapshot activation until the stack depth reaches a given number (see frame count activation in Section 4.1). As *step*, *next* has to restore the resources before the debugger shows the command prompt.

Continue

Continue suffers from the same problem as *next*, in conjunction with instruction counting. The epdb version of *continue* calls the trace function on every line of code. In the *user_line()* method, it checks if the actual line is a breakpoint, and in this case, provides a command prompt. In redo mode, *continue* calculates the halting position using the *continue_dict* and tries to find the best positioned snapshot to reach it. However, if the next breakpoint is after the maximum instruction count of the timeline or doesn't exist (in this case it is the end of the program), epdb uses the snapshot with the highest instruction count from the timeline and activates it until it finds a breakpoint, or the program ends. This type of activation is called continue activation as described in Section 4.1

Rstep

There is is no implementation for all the reverse debugging commands in pdb. Therefore, I had to implement them in epdb from scratch. For reverse execution, epdb ignores the external state at first and does replaying by activating a previous snapshot. Then, it does counting activation until it reaches the current instruction count minus one. It replays some instructions in replay mode, and before it shows the command prompt to signal that reverse stepping has finished, it switches to redo mode, and thus sets the resources to their state at this instruction count in the actual timeline. Consequently, the debugger doesn't need to reset the state after every replayed instruction, but only once per initiated user command.

Rnext

A *rnext* means to avoid stepping into a function that returned to the actual position in the program. In epdb, *rnext* does the opposite of a preceding *next*. If the user has sent a *next*, and afterwards a *rnext* command, then he is at the same position in the source

code as before. The only thing that may change is the mode, i.e., if the debugger was previously in normal mode, it is after a *rnext* always in redo mode.

However, without additional tracked information, the debugger doesn't know which function to step in and which to skip while it is replaying. A *rnext* can mean to step only one instruction backwards, or it can mean to step multiple instructions backwards depending on if the last instruction was a function call or not. To solve this problem, I decided to track every instruction count at a function call with the correspondent instruction count at the return of the function. Epdb stores this information in a dictionary *rnext_dict* [1], with the return instruction count as the key. Then, the debugger can look up the corresponding call instruction count. If it does not exist, it can simply step one step backwards. If it exists, it can step back until it reaches the call instruction count. Both of these types of stepping backwards only require counting activation.

Like *rstep*, *rnext* sets the resources to their appropriate state only before it shows a command prompt. *Rstep* always ends in redo mode.

Rcontinue

Similar to *rnext*, for *rcontinue* to work, the debugger has to track additional information. Without this information, the debugger wouldn't know at which breakpoint to stop. It could be that it has to stop at the first line with a breakpoint it encounters, or at the second or any following.

Epdb keeps a dictionary *continue_dict* with a tuple, that contains the filename and line number as the key, and a list of instruction counts as the value. If the programmer navigates backwards, epdb can iterate through every breakpoint and check if there is an instruction count in the dictionary. Then, it takes the highest instruction count and uses it to calculate the steps to run forward. When it reaches its goal, it sets the resources and then switches to redo mode as *rnext* and *rstep* always do.

In principle, it is possible to implement this dictionary locally for each process, because the program can reproduce it while running forward and it is usually not used to jump forward. However, since epdb allows timeline switching, the corresponding continue dictionary is not available if the user is in a different timeline, and therefore can't switch to another timeline. Thus I made the *continue_dict* shared, because I feel it is the cleanest solution.

4.6 Example Patch Modules

I implemented a simple patch module set, which illustrates the principle of dealing with non-determinism and side effects. For non-determinism, I chose the *time()*-function of

[1] see Section 4.2

the *time* module. For functions which also have side effects, I chose the built-in file editing functions *open(), read(), write(), close()*.

Time

The *time.time()* function is a typical example of a non-deterministic function. It returns the actual time as a floating point number. This number represents the numbers of seconds passed since January 1st 1970, at 0:00. The *time.time()* function does not have side effects, because it doesn't actively change the time.

There are two ways to implement *time.time()*. The patched function could either store the return value in a stored dictionary or it could instruct the debugger to make a snapshot after the instruction.

Implementation with a Shared Dictionary

In order to create an patched version of the *time* module, the debugger needs a *__time* module. If this module exists, the debugger then merges these two modules. Listing 4.1 shows an implementation for the *__time* module.

One tricky aspect in the implementation of the *time* module is that the module name is the same as the function name. Since the function needs to access the original module, it needs to import the original *time* module. However, this module has the same name as the function and therefore every reference to *time* would reference the function instead of the module. Thus, the module should import the *time* module as a different name. In the example, the *time* module is imported as *timemod*.

The implementation of the *time* function distinguishes between normal and redo/replay mode. In normal mode it first uses the original implementation to get the system time. It then saves the time in the shared dictionary at the actual instruction count and after that it returns the time value. In redo and replay mode it looks up the value in the dictionary and returns it. By this means the function always returns the same time value in redo mode as it returned it in normal mode, when the debugger executed it the first time.

Implementation with Snapshots

The alternative implementation of *__time* instructs the debugger to make a snapshot after the instruction. Listing 4.2 shows an implementation of time using the snapshot approach. The dbg module provides the object *snapshottingcontrol* with a method, which allows to do exactly this. In normal mode, the implementation calls this method and then returns the system time using the original function. There is no implementation for redo or replay mode, because the debugger never executes the function in redo or replay mode, but instead activates the snapshot after the instruction.

Listing 4.1: __time.py

```
import time as timemod
import dbg

def time():
    if dbg.mode == 'normal':
        value = timemod.__orig__time()
        dbg.nde[dbg.ic] = value
        return value
    elif dbg.mode == 'redo' or dbg.mode == 'replay':
        return dbg.nde[dbg.ic]
```

Listing 4.2: Alternative implementation of __time.py

```
import time as timemod
import dbg

def time():
    if dbg.mode == 'normal':
        dbg.snapshottingcontrol.set_make_snapshot()
        return timemod.__orig__time(a)
```

The advantage of this approach is, that it is very straight forward. For every non-deterministic function without side-effects, the implementation with snapshots looks almost the same, even for more complicated situations, e.g., a function which changes a referenced object. It also seems feasible to automate the process of patching such instructions. However, this approach also has a disadvantage, because the debugger always makes a snapshot, when it reaches such an instruction, which could reduce the system performance.

File Handling

Implementing reversible debuggable file handling is much more complex than a *time()*-function. First, the file handling functions are not only non-deterministic, but also have side effects. Moreover, they interact with each other, so they cannot be considered independently. To simplify the matter, I only considered the core functions of file handling which are *open()*, *read()*, *write()* and *close()*, and ignored others like *seek()* or *trunk()*.

It is important to note, that simulating file access very much depends on the file and therefore it is clearly impossible to give one "best solution". For instance, the file could be a special file like `/dev/null` or `/dev/random`, which works completely different than a regular file. Therefore, a simulation for accessing a `/dev/random` file

Listing 4.3: __builtins.py: open

```
def open(file, mode="r", buffering=-1, encoding=None,
         errors=None, newline=None, closefd=True):
    if dbg.is_dbg_callee():
        return builtins.__orig__open(file, mode,
            buffering, encoding, errors, newline, closefd)

    fd = builtins.__orig__open(file, mode, buffering,
            encoding, errors, newline, closefd)
    args = (file, mode, buffering, encoding,
            errors, newline, closefd)
    fp = FileProxy(fd, args)
    return fp
```

would be different than one for a regular file. The file can also be accessed by another process and therefore it may be necessary to take this into account.

One can use the code of the example with epdb to explore the typical flushing behavior of the function calls (i.e., the buffer is only written to disk on *close()* or *flush()* calls). To make the implementation quite simple, but nevertheless of practical value, I made the following assumptions to how the file is accessed:

- Only regular files are considered
- It is assumed that no other process deletes the file
- The type of the file object returned by open isn't used
- The implementation of the file does not expose it's buffer
- The implementation of the file only flushes in case of a *flush()* or *close()* call

Open

The *open()* function is a factory function which returns a file object of some type. The type depends on the mode in which the file is opened. For instance, it can be *BufferedRandom* or *TextIOWrapper* depending on the opening mode. In the simulating example, the patched open call returns a *FileProxy* object. This object supports the *read()*, *write()* and *close()* method. The proxy passes the calls to the *builtins* file object.

I used a proxy in this example, because it is not easily possible to access the buffer of the file object. This means that it is not possible to change the buffer of the file, which is written when the process ends, for example: the programmer steps over a *write()* call and then decides to step back. This means that the debugger restores a previous

Listing 4.4: __builtins.py: FileProxy

```
class FileProxy:
    def __init__(self, file, args):
        self.__file__ = file
        self._args = args
        self.fn = fn = args[0]
        resource = dbg.current_timeline.\
            new_resource('file', fn)
        rm = resources.FileResourceManager(self.fn)
        self._fileresourcemanager = dbg.current_timeline.\
            create_manager(('file',fn),rm)
        id = self._fileresourcemanager.save()
        if not dbg.ic in resource:
            resource[dbg.ic] = id

        def write(self, b): "..."
        def read(self, n=-1): "..."
        def close(self): "..."
```

snapshot and terminates the previous debuggee. This means that all changes to the buffer are flushed out, which should be considered by the activated process.

Another problem with open file descriptors is their behavior in conjunction with *fork()*. Let's consider a program which opens a file and writes to it. After that, it forks another process and then both processes close the file. This would actually mean that the stream would be written twice, which is probably not intended. Epdb solves this problem by using resources, which have a state. If the debugger closes a process, e.g., because the user deletes a snapshot, it can recover the external state by resetting the file resource to its actual state.

The implementation of the *open()* function can be seen in Listing 4.3. The first thing this implementation checks is if the callee is some debugger related module, because the debugger uses *open()* as well. In this case, the debugger does not use the patched version of *open()*, but the original one. The implementation of *open()* works the same in redo, replay or normal mode, because *open()* is a deterministic[1] function with side effects. Instead of returning a *File* object, the patched version of *open()* returns a proxy to it, i.e., the *FileProxy*. This proxy provides its own implementation of *read()*, *write()* and *close()* methods.

Listing 4.5: __builtins.py: read()

```
    def read(self, n=-1):
        if dbg.mode == 'normal':
            dbg.snapshottingcontrol.set_make_snapshot()
            value = self.__file__.read(n)
            return value
```

FileProxy

As shown in Listing 4.4, the *FileProxy* does not only provide patched versions of *read()*, *write()* and *close()*, but also manages the file resource manager. It generates a new resource and a resource manager. Then it saves the actual state and stores the *id* in the shared resources dictionary. Keep in mind, that the current timeline can change between the *FileProxy* initialization and its method calls (e.g., by creating a new timeline) and therefore, the resource dictionary should not be stored as a member variable for using it in other method calls.

Read

The *read()* method is a non-deterministic function without side effects. It is non-deterministic, because the debugger doesn't know what the method is going to return. The *read()*-method relies on an external state, the file. However, it doesn't change the external state and so it has no side effects. These considerations allow one to implement the *read()* method using snapshots. The implementation shown in Listing 4.5 achieves this by calling the *set_make_snapshot()* method of the *snapshotcontrol* in normal mode. There is no implementation of *read()* in replay or redo mode. This is because epdb would never call *read()* in replay or redo mode, but would instead activate the next snapshot, because of forward activation. This snapshot has the correct internal state after the *read()* call in normal mode.

Write

The *write()* method is a non-deterministic function with side effects. The side effects are obvious, because it changes an external state, the file. It is also non-deterministic, because the *write()* method also returns the number of bytes written and this number does not have to be the same as the number of characters in the argument. A possible implementation of *write()* may look similar to Listing 4.6. As *write()* is non-deterministic, it is easiest to just let the debugger do all the work and then to tell it to make a new

[1] at least if we consider the internals of *open()* such as the integer descriptor, as being of no particular interest

Listing 4.6: __builtins.py: write()

```
    def write(self, b):
        value = self.__file__.write(b)
        id = self._fileresourcemanager.save()
        dbg.current_timeline.\
            get_resource('file', self.fn)[dbg.ic+1] = id
        dbg.snapshottingcontrol.set_make_snapshot()
        return value
```

snapshot after the write call, as was done in the *read()* example. However, *write()* also changes the external state and therefore it should also tell the resource manager to save the state and to put the state identifier into the resource dictionary, so that the debugger can restore the state of the file later on. Similar to *read()* the implementation of *write()* doesn't need an implementation for replay or redo mode because epdb uses forward activation.

Close

The *close()* method is an example of a deterministic function with side effects. An implementation may look similar to the code in Listing 4.7. As it is deterministic, it is not necessary to make a snapshot after this function call. However, it is necessary to save the state of the file resource after *close()*, because *close()* flushes the buffer. As the patched *close()* method does not instruct the debugger to make a snapshot after its return, it is possible that the debugger calls the *close()* function in redo or replay mode. The implementation also proxies the *close()* method in these modes as opposed to the non-deterministic *read()* and *write()* methods.

I do not consider this implementation of file handling a pretty one, but more of a hack. However, I think this example shows the power of the framework, that it can be used to simulate even such complex operations like file handling quite easily. A better solution to this problem, in my opinion, would be to use a special file system for the files the application uses. This file system should not delete or change files, but instead save a new version of the old file and it should allow the restoration of old versions of the files. This is similar to the way some databases with Multi Version Concurrency Control (MVCC) save their data. The file system in user space (FUSE) technology allows to implement such a file system entirely in user space. Using a special file system would also allow a way to deal with some special files like `/dev/random`. The file system could store old versions of the number generator. However, implementing such a file system is out of the scope of this thesis.

Listing 4.7: __builtins.py: close()

```
    def close(self):
        if dbg.mode == 'normal':
            self.__file__.close()
            id = self._fileresourcemanager.save()
            self._resource[dbg.ic+1] = id
        elif dbg.mode == 'replay' or dbg.mode == 'redo':
            self.__file__.close()
```

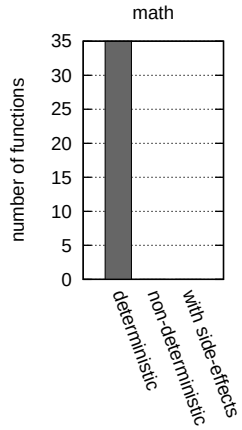

Figure 4.5: Module math

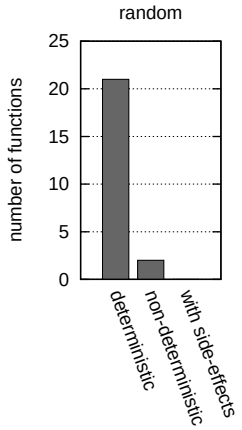

Figure 4.6: Module random

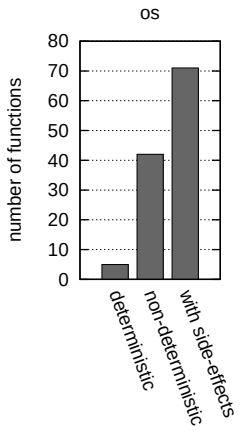

Figure 4.7: Module os

4.7 Modules of the Standard Python Library

In this section, I want to give an idea, how much work it would be to enable reversible debugging for a program. The amount of implementation modules a program needs, very much depend on the program. If it is a program, which only does some calculations, the program may not need any implementation modules at all. However, for a program, which makes heavy usage of multiple resources and system calls, it can become a very complex task. Some modules don't need any patching at all. For example, the module *math* has 35 functions, which are all deterministic and without side effects. The module *random* has only 2 non-deterministic functions and non with side effects. On the other hand, most of the functions of the *os* module are non-detrministic or have side effects. Figure 4.5-4.7 show the number of different kinds of functions for three modules.

CHAPTER 5

Performance

Lienhard[LGN08] notes that the snapshot & replay approach is slow, because the debugger has to re-execute the program partly. I don't agree, because replaying can be even faster than in logging-based debuggers, e.g., when the user wants to recover a position where the debugger has made a snapshot. In this case, it has to close its actual process and create a new process from the snapshot, which becomes the debuggee. Creating a new processes with copy-on-write is so fast, that it takes usually less than 1ms. On the other hand, the seek time of a hard drive takes typically a few milliseconds for spinning hard drives or about 0.1ms for SSDs. The replaying time in the best case is much lower than most users are able to perceive, which is about 100-200ms.

In the worst case, the performance of an execution command should end after a certain amount of time and therefore, the debugger should give some upper bound on the execution time of the command. If this time is less than the user is able to perceive, then there is at least no performance reason to avoid using a reversible debugger. The purpose of this section is to show that it is in principle possible to achieve this performance goal. However, I don't want to give benchmarks on epdb, because the real execution time depends so much on the hardware and epdb isn't performance optimized yet.

Epdb distinguishes two modes, which are visible to the user, the normal and the redo mode. So we have to look at the performance of the debugger in each mode. In normal mode, the debugger has a lot of overhead. Most important is the additional work it needs to count the number of instructions, since epdb implements this in pure Python. This slows down the execution. Then, there is some overhead to save non-deterministic effects and to make snapshots and to track the execution path history. A reversible debugger is slower in this respect. However, the execution is slowed down by some constant[1] extra time per instruction.

[1] I haven't added the time for the resource management here, because this time is very application dependent

In the situation in redo mode, assuming the timeline has been executed until the program has finished and every instruction is executed in redo mode, the debugger only has to restore a snapshot and run some instructions forward. Therefore, the activation time of the new process takes some constant amount of time, but running forward an arbitrary number of instructions in redo mode takes constant time too! This is because the debugger makes a snapshot every time the execution time counter passes one second. The debugger never executes the last instruction which exceeds the one second limit in redo mode, but instead activates the next snapshot. The execution time in redo mode is always below one second plus the time needed to activate the snapshot. Of course there is some additional overhead to recover the external state, but in the special case of a program, which doesn't make use of external resources, the execution time of every navigation command in redo mode is constant or in other words $O(1)$. The execution time of a program in redo mode can be even faster than the program execution of a program without a debugger. The constant execution time in redo mode assumes however, that there is no overhead by having multiple snapshots, which is unfortunately not the case. However, it is in principle possible to limit the number of processes by saving the snapshots to disk and restoring them from there, but epdb does not implement snapshots like that.

5.1 Instruction Counting

Python does not count the instructions by default. To do this, I implemented instruction counting using the trace function. Every time an instruction is executed, the trace function is called, which increments the instruction counter by one. Consequently, it is possible to add instruction counting without changing the source code of the Python interpreter. However, it has significant influence on the execution time. To estimate the performance impact of instruction counting, I implemented a program that does nothing else than executing a Python script with instruction counting with the above described approach. Then, I ran the microbenchmark program *pybench*, that comes with the Python interpreter, with and without instruction counting. The tests were run on a AMD Athlon(tm) 7550 Dual-Core 64bit - Processor with 2.5 GHz. The Python interpreter used was 3.1.1+ on an Ubuntu Karmic Koala.

The tests in *pybench* were repeated multiple times. Figure 5.1 shows the relation between the average execution time with instruction counting and without instruction counting. As you can see from the diagram, with instruction counting the tests run most of the time about 15 times slower. For one extreme example it runs even 110 times slower.

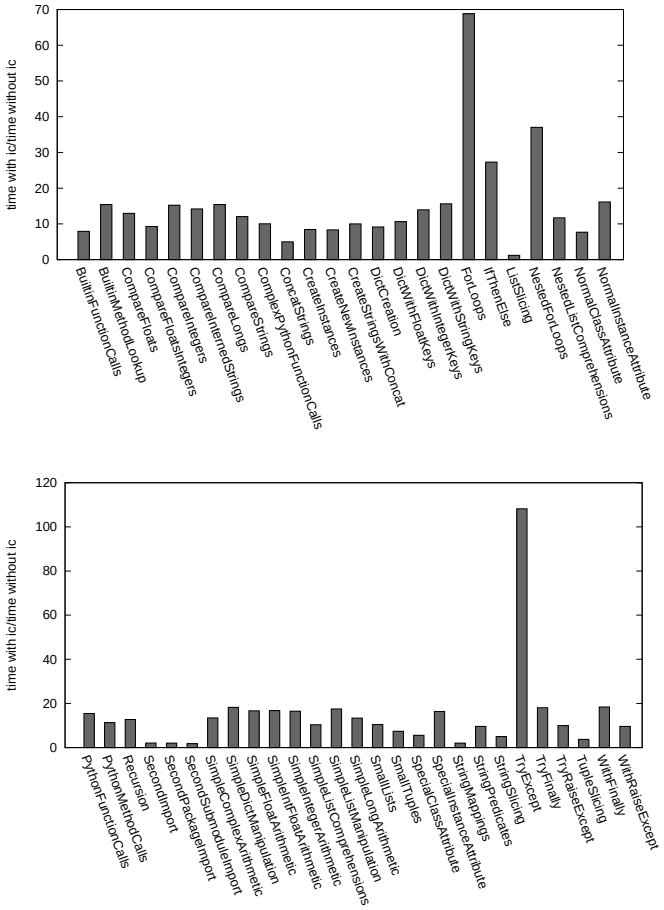

Figure 5.1: Average time used relation between with and without instruction counting

5.2 Snapshot Performance

Unix and similar systems use a method called **copy-on-write**[Bac86] for process creation, which reduces the memory the newly created process uses. When *fork()* creates a new process using copy-on-write, the new process uses the same memory. This works because the memory of the parent and the child process is initially the same. However, if one of those processes writes to a memory page, this memory page is then different for the child and parent process. Therefore, the operating system copies this page to another place, so that each process has now its own memory page.

Smith and Maquire[SJ88] did some experiments on how much a process changes during its whole lifetime and came to the conclusion that about 50 percent of the pages kept the same. As most snapshots in epdb don't span across the whole lifetime of the process, I expect that snapshots in epdb do a little bit better.

Copy-on-write process creation is really fast and typically takes less than 1 ms. Waiting for a process to finish takes a bit longer, but is usually done in less than 10 ms.

More interesting than the speed of the execution of *fork()* is the amount of processes the system can handle simultaneously. It is well-known that a fork bomb, like in Listing 5.1, will make a system completely unresponsive. I tried to estimate how many simultaneously processes the operating system could handle with the program shown in Listing 5.2. This program measures how much time per created process it takes if the program creates a bunch of processes. Figure 5.2 shows the result of this measurement. As one can see, the number of processes the program creates hardly affects the process creation time of a process, at least at these numbers of simultaneous created child processes.

The program also measures how much extra time it takes to not only create the processes, but also to wait for them afterwards. In order to do this, each process keeps itself long enough alive so that the parent process has time to create all processes, before the subprocesses finish. The child processes accomplish this waiting by using *time.sleep(5)* to stop the process for 5 seconds. After the parent process has created all child processes, it then waits for all of them to finish and measure the time up to this point. As each process waits 5 seconds, it therefore subtracts this amount of time to get the additional time the process needs to create all processes and to wait for their termination. Figure 5.3 visualizes the result of this measurement. The experiment was executed on an AMD Athlon(tm) 7550 Dual-Core Processor with 2.5 GHz and 4 GB of RAM and an Ubuntu Lucid Lynx Linux Distribution. As one can see from the diagram, the overhead explodes at some number of processes. This is where the machine typically becomes completely unresponsive and the only way to recover it, is to reboot the computer. From the diagram we, can extract an estimate of how many snapshots the debugger can handle. I tried to debug a program with epdb which made about 1000 snapshots and was still able to work and debug reasonably well on my computer. As the program makes about one snapshot per second[1], it is possible to debug programs which require about 16 minutes of execution time. This is enough for many applications, e.g., a program which handles a page request of a web server should typically respond in a second or even less. However some longer running programs such as server programs, may need a much longer execution time. There is still some work required to handle bigger numbers of snapshots. One approach would be to swap the processes to disk, and then afterwards to terminate them, and then only reactivate the process when it gets activated. The operating system does swapping, but it doesn't kill the process, which means that it has to continuously swap in and swap out, and this takes a lot of time.

[1] this is at least the case if the debugger does not make snapshots to handle non-determinism

Listing 5.1: forkbomb.py

```
import os

while True:
    os.fork()
```

Listing 5.2: maxfork.py

```
for maxnoproc in range(100,1000,100):
    starttime = time.time()
    for i in range(maxnoproc):
        id = os.fork()
        if id == 0:
          time.sleep(5)
          break
    else:
        finishtime = time.time()
        createtime = (finishtime-starttime)/maxnoproc)
        for i in range(maxnoproc):
            os.wait()
        waitfinishtime = time.time()
        overhead = (waitfinishtime-starttime-5)/maxnoproc
        print("Create time:", createtime)
        print("Overhead:", overhead)
        continue
    break
```

5.3 Epdb Performance

The performance of epdb is dependant on many factors. I compared execution time and memory usage, when the program is run inside the debugger and when it is executed without the debugger. I chose two benchmarks – *fankuch.py* and *nbody.py* – from the computer language benchmark game[CLB11]. The benchmark *gcd.py* calculates the greatest common divisor for two very large integers using Euclidean algorithm. These three tests should measure typical programs.

The benchmark *call_snap.py* illustrates the case, when a program repeatedly calls a non-deterministic function. The program calls a patched function in a for-loop for 500 times, which instructs the debugger to make a snapshot. Therefore, running the program without a debugger has not much overhead. However, when running it with epdb, the debugger has to make a snapshot every time it encounters the function call.

The program *create_array.py* creates a very large list of integers. It is an example where debugger execution almost performs as well as native execution. To show, that it

Figure 5.2: process creation

also performs well with snapshots, the program also makes a snapshot before and after the creation of the list.

For all programs I measured execution time and memory usage and the number of snapshots made when run under the debugger. For measuring execution time, I used the *time* command to run the programs and took the real execution time. The reason I chose real time over user time is that it is better comparable with the measurement when the debugger runs the program. The *time* command doesn't work in conjunction with epdb, because epdb doesn't terminate after it has executed every instruction. Therefore, I implemented time measuring for epdb. In normal mode, epdb always measures the time it needs to run *step*, *next* or *continue* commands. Thus, I started the program with epdb, issued the *continue* command and then took the time epdb put out.

To measure the memory usage I used *free* command, which shows the actual memory usage of the whole system. I measured the memory usage before I started the program and before it ends and took the difference. To achieve measuring the memory usage before the end of the program, I injected an additional *input()* command which halts the program until the user presses the return key. This makes it possible to measure the memory usage again and then subtract the initial memory usage from it to receive the memory usage of the program. Measuring the memory usage with debugger works similarly, although I didn't need to inject an *input()* statement, because the debugger doesn't terminate the program when it finishes. Although using *free* measures the memory used by the whole system, it allows to factor in that epdb uses multiple processes,

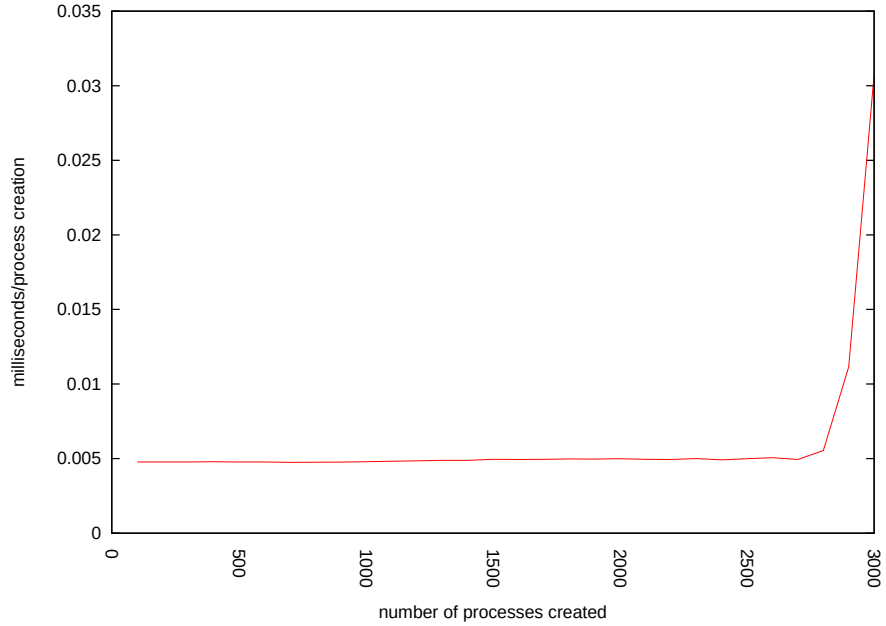

Figure 5.3: fork() overhead

which share memory pages. The problem that the measurement is susceptible by other processes can be reduced by executing the tests multiple times.

Table 5.1 shows the averaged results of the measurements, where I executed each benchamrk five times. In the case of *fankuch.py*, *nbody.py*, *gcd.py* the execution time using epdb is higher – approximately 1000 times. The additional memory these programs need is about 10-20 times higher. The program *call_snap.py* doesn't take particularly long to execute, i.e., it is about 300 times slower, which is low compared to the other measurements. However, the debugger makes a snapshot in every iteration of the loop. Therefore, epdb has to make a snapshot in every iteration, and thus it needs lots of additional memory, i.e., it is about 300 times higher. On the other hand, the program *create_array.py* doesn't have a lot of overhead at all, when running under the debugger. It is only 1.05 times slower and needs 1.02 times more memory. The reason for the good results for execution time is, that there are only very few atomic instructions for the debugger, which cause overhead. These instructions have long execution time, which increases the execution time for the execution without a debugger as well. The memory usage of the program is very high even without using a debugger and thus the memory usage of the debugger itself doesn't contribute much to the overall memory usage compared to the memory usage without a debugger. The large list doesn't need more memory when running under epdb, because epdb makes use of the copy-on-write mechanism of *fork()*. Therefore, it doesn't need extra memory to save the array in another snapshot.

Table 5.1: Benchmarks for epdb

benchmark	parameter	execution time (s)	execution time with debugger (s)	number of snapshots made	memory usage	memory usage with debugger
fankuch.py	6	0.034	37.7	13	3MB	44MB
nbody.py	500	0.051	366.4	65	3MB	106MB
gcd.py	-	0.115	49.5	10	3MB	37MB
call_snap.py	-	0.029	8.6	501	3MB	1010MB
create_array.py	-	0.93	0.98	3	766MB	785MB

Although epdb is considerable slower than running without a debugger, there is quite a lot of room for improvement. Especially rewriting the debugger in C should make it much faster, because there would be no more additional Python instructions every time the debugger encounters an atomic instruction. Especially instructions like incrementing the instruction counter are very slow in Python, because Python uses its own implementation for integers, which allows integers to get arbitrarily large. However, this reduces the execution time for integer operations. I think the memory usage of epdb is reasonable. Most of the overhead comes from the additional processes epdb uses, which doesn't increase much, when the program runs longer.

CHAPTER 6

Applications

People, who have developed reversible debuggers have complained that nobody uses their decent debugging tools, e.g., Lewis writes on the homepage [Lew07] of odb:

> The ODB is as close to a silver bullet as you can get. Why don't people use it?
>
> I don't get it. :-(

Lieberman, a developer of ZStep, has written some text[Lie97] about people not using good debugging tools.

I don't want to join their complaints. Instead, I want to give examples of how a user might want to use epdb. In this section, I want to risk looking in the future, which means that I will write about features and additional tools which are not available yet.

6.1 Web Applications

Web application development is an important branch of software engineering. There are many tools and frameworks available for Python to ease the development of web applications, e.g., Django, Pylons, Turbogears. Typically, these web frameworks provide their own debugging mode, where they show some debugging information when the program fails, but usually don't allow to control the program execution.

I think using epdb to debug web applications can work very well. Web applications have usually a short running time, because the user expects a web page to load fast. Therefore the limited amount of snapshots of epdb doesn't play a big role. They also use a limited amount of resources; often a database and files are the only resources they use. Some databases like Oracle or CouchDB already support accessing old versions of the data records. If the database doesn't allow this, it is maybe possible to write

an object relational mapper which accomplishes this. Object relational mappers are very common in web applications too. File access could be implemented by using a special file system, or the framework could add some implementation for file access to accomplish reversible debuggable file access. Running the program with its resources in epdb should work without any bigger problems. What remains left is the interaction with the user, preferably with a graphical user interface. Here the framework could present the user the interface via the browser and the use of AJAX. When the user requests a page, the web server would start the Python program under debugging control. If it runs successfully, the web server sends the webpage to the browser to render it. Otherwise, it sends a webpage to the browser, which contains a graphical user interface to debug the program. This user interface would of course also allow the user to debug the program in reverse and it would be possible to run the program either deterministically or non-deterministically by making use of timelines. Of course, debugging a web application using this approach should only be enabled in debug mode.

Another problem which debuggers of web applications face is that web applications typically use some sort of template language. Often there is a bug in a template instead of the code itself. The *render()* method of the template typically raises an exception, but this doesn't give the programmer much insight into what went wrong in the program. Thus it should be possible to inject a debugger inside the debugger, which works for the template language. Using the patching mechanism of epdb, this should be possible to implement.

6.2 Smart Phone Development

There are lots of small applications, which are often called apps, for smart phones. Most of these apps are written higher level programming framework. Developers usually test their software on a virtual machine, before deploying it to the phone. The host machine is usually much more powerful in terms of disk space, main memory and CPU speed. Therefore, it is reasonably to use a virtual machine, which does some additional work and allows reversible debugging of applications.

The developer of most apps usually use a higher programming framework, which also allows a limited amount of API calls to access resources. Therefore it seems feasible to develop reversible debuggable versions of these calls for the execution on the virtual machine. On the virtual machine are already only simulated devices, and this makes developing reversible debuggable resources even easier. Having a good debugging framework for smart phone development may attract many developers, which develop lots of applications. A smart phone operating system with lots of application is more likely to be a business success.

6.3 Visualization of Algorithms

I believe using a debugger is a great way to get a better understanding of how a program works and a reversible debugger makes this process even more enjoyable. Often speakers want to present some algorithm and illustrate how it works. It would be nice to visualize the data as the user steps through the program. ZStep[1] focuses on visualization, but epdb can visualize data too. Using its mechanism to patch instructions, it is possible to visualize an algorithm without changing its code.

6.4 Design and Architecture

In my opinion a well-designed program is one which is easy to test and easy to debug. Even if one doesn't want to use a debugger, e.g., because one prefers to use test driven development, the basic understanding of the debugging architecture of epdb helps the programmer to design better programs. This is because epdb emphasizes the reproduction of certain states. This is important in design because a bug, which is not reproducible, is very difficult to fix. Therefore, the first step in fixing a bug is to reproduce it. The remaining part is straight forward at least for an experienced programmer who knows the code.

[1] see Section 7.1

CHAPTER 7

Related Work

Although reversible debugging has certainly various benefits, some authors have argued against it. For example, Rosenberg [Ros96] argued against reversible debugging by calling reverse execution a much-requested but dubious feature. He also believes that this feature is so much work and still fraught with so much error that it is not worth the engineering investment. He also gives some basic algorithm to implement reverse execution. However, he does not mention copy-on-write optimizations

There is still a controversy between logging-based debuggers and replay-based debuggers. Lienhard[LGN08] argues that the disadvantage of replay-based debuggers is that moving backwards in time can be very slow. However, I can't confirm this, because by using continuous snapshotting every second, the debugger can bound the amount of time it needs to run backwards.

Feldman and Brown[FB88] give an alternative implementation of efficient snapshots without using *fork()*. Mellor-Crummey and LeBlanc[MCL89] discuss a software instruction counter, which usually does not have more than 10% overhead.

Pan and Linton[PL88] describe how to use *fork()* to create new checkpoints (i.e. snapshots). They also describe the use of an event log, which the debugger uses during replay when accessing the shared memory or when it replays system calls.

Kessler[Kes90] gives an approach to implement fast breakpoints using self-modifying code. Netzer and Weaver[NW94] show an efficient adaptive tracing strategy for logging-based debuggers. Demetrescu and Finocchi[DF04] describe the Leonardo virtual machine which allows directing and checkpointing, which is useful for reversible debugging. Nitin[KNM06] shows how the virtual machine Xen could be used to implement kernel debugging.

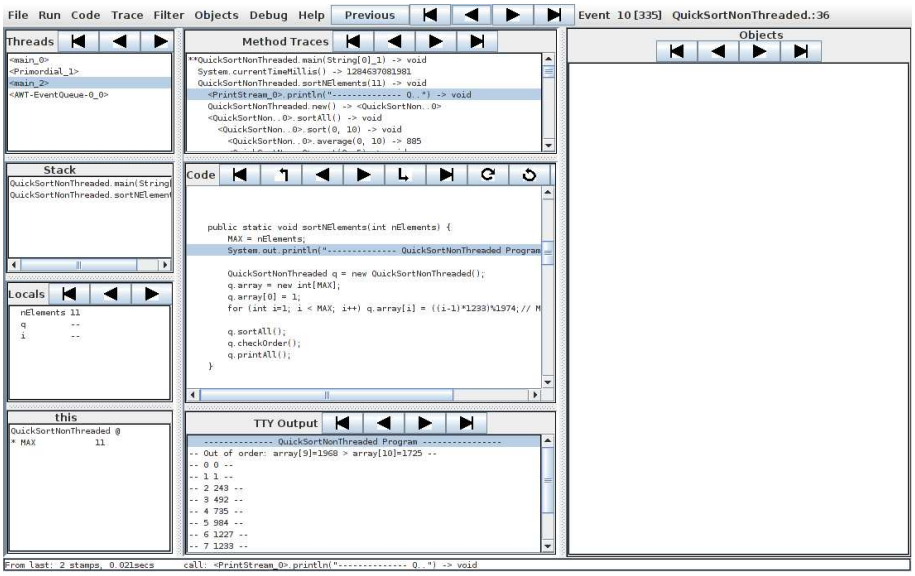

Figure 7.1: Omniscient Debugger (odb)

7.1 Other Reversible Debuggers

Others already have developed reversible debuggers. However, they work differently than epdb and target different programming languages.

Odb

The Omniscient Debugger[Lew03] [LD03] is a logging-debugger for Java, which uses byte-code instrumentation of the classes when they are loaded. To debug a program in odb one should first execute the program under the debugging environment until it ends or crashes. During execution, odb creates a log file of everything that happens in the program. Everything here refers to time stamps, local variables, the state of all objects, and the tty output for every thread at any position in the program. After the program ends, the debuggers graphical user interface shows up (see Figure 7.1). It allows the programmer to examine the state of the program at any position in the program at any time, and it also allows one to query program states. The users don't have to examine the log files after they have executed them, as they can also save them and examine them later, or send them to the programmers to help them fix the observed bug.

Odb is different from epdb, in that it is a logging debugger. With epdb, you don't have to execute the program until it ends. Epdb allows the programmer to run the program up to any point, then to go some steps back and then to go some steps forward again, either in deterministic or non-deterministic fashion. Epdb also introduces resource management, which odb lacks.

Gdb

Gdb uses a technique to implement reversible debugging, which its developer name **process record & replay**. As gdb is a native debugger, it has a parent process, the debugger, and a child process which executes the program and gets directed by the parent process. With process record & replay, the parent process logs the execution of each machine instruction in the child process, together with each change in memory and registers. With this execution log, the debugger can execute the program in reverse by successively undoing each change of each logged instruction. To go forward again, it also uses the execution log to replay the state and so it then uses deterministic forward execution.

As odb, gdb has no resource management yet. However, it is more similar to epdb, in that it is an interactive debugger. A problem with gdb's approach is that running backwards a huge amount of instructions takes a very long time, because each machine instruction has to be undone. With epdb's, approach the debugger activates a snapshot which is near to the target location in the program and starts from there, which can save replaying time. In gdb, there is also no concept which distinguishes between deterministic and non-deterministic instruction execution.

ZStep

ZStep is an interactive reversible debugger visualization tool for Lisp, which allows reverse execution of code. In contrast to epdb, ZStep does not work on source line level, but on expression level, which allows the user to step over smaller pieces of code. ZStep is also a visualization tool, which allows graphical representation of the data structures or program execution, while the user steps through the program. However, ZStep has no concept of non-deterministic functions and does not have an answer to side effects. The visualization of code is something epdb supports in principle too by allowing the user to patch instructions.

EXDAMS

The oldest reversible debugger I know of is the EXtendable Debugging and Monitoring System, or EXDAMS[Bal69] for short. It was a debugger for the ancient Multics operating system. In fact, the authors didn't call it a reversible debugger, but debug-time history playback. EXDAMS is a logging debugger. The debugging system augmented the source code with additional logging statements and allowed one to view the state of the program at any position in the program.

IGOR

Igor [FB88] is a prototype reversible debugger for the DUNE distributed operating. It uses a snapshot & replay approach. The authors refer to the snapshots as checkpoints. To implement checkpoints they make a special system call *pagemod*, which returns the pages that where written since the last call. With this system call the debugger can save only those pages, which have changed since the last checkpoint. Igor also has same very basic form of resource management, i.e., it has a *prestart* routine which the debugger before executing of the program. This routine sets the state of user supplied files before executing the program. Although there are similarities with epdb resource management, IGOR does not track the different states of the files while executing. It also does not support timelines or a similar concept.

Bdb

Bdb [Boo00] is a prototype reversible debugger for C/C++ running on Digital/Compaq Alpha based Unix workstations. Bdb uses a snapshot & replay approach similar to epdb. Like epdb, it uses the system call *fork()* to create snasphots. Bdb uses a technique called exponential checkpoint thinning to reduce the number of checkpoints. With this technique the debugger only keeps snapshots at exponential intervals and thus the number of snapshots grows only logarithmically. While the program execution progresses, the debugger thins out the number of snapshots. Although it reduces the number of snapshots, it has the disadvantage that re-execution takes longer for code at the beginning of the program. In contrast to epdb, Bdb does not allow multiple timelines and it does not manage the external state.

CHAPTER 8

Further Work

In this section, I want to present interesting work which may also be useful for a reversible debugger like epdb. I also want to give a short summary of other reversible debuggers and their differences to epdb, as well as to give some ideas for additional research which would help reversible debugging.

8.1 Smart Snapshot Making

Usually the user is only interested in a small part of the program and therefore some snapshots are never activated. Such snapshots would be harmful, as they consume system resources without any need. Therefore a smart debugger could use a strategy to avoid making such snapshots in advance, but rather only when the user initiates a reverse execution command. In this case, the debugger could make a snapshot of the process and insert it into the timeline. When the user does a reverse execution command again, the reverse execution would be much faster because the debugger could use a more recent snapshot. A sophisticated strategy to make snapshots would rely on information about how users use a reversible debugger. Therefore, one could log the behavior of the users when they debug a program and then find a snapshot making strategy which would reduce the average time the user would need to wait. This approach would be even more powerful if it is combined with persistent snapshots, i.e., snapshots which store the memory to disk. Therefore these snapshots don't use a process which reduces the system performance, but on the other hand the debugger would need more time to create and to activate them than it needs to create and activate snapshots, which were created by using *fork()*.

8.2 Reversible Debuggable Libraries and Other Tools

The resource management approach of epdb works well if it is easy to save and restore the current state of the resource. Some databases already allow this, but other resources such as files, network communication libraries, and graphical tools usually don't support state saving. Therefore, it would be interesting if it is practical to implement this state saving feature into these tools and libraries. As most computers today usually have huge amounts of free disk space, it seems that marking data as deleted instead of actually deleting the data shouldn't decrease the overall performance too much. I guess the performance should be at least good enough for the debugging mode.

8.3 Native Instruction Counting and Bookkeeping

Epdb uses the trace function to implement instruction counting and to do some bookkeeping. This has the advantage that the whole debugger is written in Python, but on the other hand, it has less performance as a result. This is because the interpreter calls a Python function for each line of code. Pdb can use a lot of optimizations, because it doesn't rely on instruction counting. Therefore pdb can only trace functions, which have a breakpoint in their code. However it should be possible to put the instruction counting and bookkeeping code into the interpreter itself. Consequently, this part of the code would have to be written in C. Using this optimization, I would expect that epdb would be much faster in its execution, especially in normal mode.

CHAPTER 9

Conclusion

In this thesis, I have shown that it is possible to develop an interactive reversible debugger using the snapshot & replay approach with reasonable performance. I have also shown that it is possible to get around the problems of non-deterministic instructions and instructions with side effects, by using instruction patching, timelines and resource management. During this thesis, I have not only developed epdb, but also documented the design for a sophisticated reversible interactive snapshot & replay debugger, which others may use to implement reversible debuggers for their favorite programming language.

Reversible debugging, although a very old idea, is not very widely used yet. Nevertheless, I hope that I have made reversible debugging easier to understand with the concepts I have developed in this thesis, i.e., that a function can affect the program environment, or the execution of a function can be affected by the environment, as well. Therefore, any programming instruction can have side effects or can be non-deterministic. A function with side-effects changes the environment of the program, while the execution of a non-deterministic function depends on it. When the user replays a non-deterministic function, he can have either one of two different expectations. He could expect to have the same result of the execution as in the first run (deterministic execution), or he could expect to run the function with the new environment (non-deterministic execution).

The implementation of epdb shows that this view of reversible debugging is actually useful. Epdb introduces timelines and therefore allows unambigous reversible debugging, because it is now up to the user to decide which type of execution, deterministic or non-deterministic, he wants to use. Epdb also introduces a new stateful resource management concept. With resource management, the user can inspect the environment of the program which, because of resource management, always corresponds to the actual position inside the program. The implementation of two resource managers for epdb, one for *stdout* and one for file access, shows that the resource management concept

is actually applicable for complex resources and it is possibly useful for many other resources as well.

Bibliography

[ALS10] J. Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: The Definitive Guide: Time to Relax (Animal Guide)*. O'Reilly Media, 1 edition, 1 2010.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[Bal69] R. M. Balzer. Exdams: extendable debugging and monitoring system. In *AFIPS '69 (Spring): Proceedings of the May 14-16, 1969, spring joint computer conference*, pages 567–580, New York, NY, USA, 1969. ACM.

[Bea09] David M. Beazley. *Python Essential Reference*. Addison-Wesley, fourth edition, 2009.

[Boo00] Bob Boothe. Efficient algorithms for bidirectional debugging. In *PLDI*, pages 299–310, 2000.

[Boo03] Duncan Booth. Patterns in Python. `http://www.suttoncourtenay.org.uk/duncan/accu/pythonpatterns.html`, March 2003.

[CFC01] Shyh-Kwei Chen, W. Kent Fuchs, and Jen-Yao Chung. Reversible debugging using program instrumentation. *IEEE Trans. Software Eng.*, 27(8):715–727, 2001.

[Chi96] Ram Chillarege. What is software failure. *IEEE Trans. Reliability*, 45:354–355, Sep 1996.

[CJ03] Ron Crocker and Guy L. Steele Jr., editors. *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*. ACM, 2003.

[CLB11] The Computer Language Benchmark Game. `http://shootout.alioth.debian.org/`, January 2011.

[DF04] Camil Demetrescu and Irene Finocchi. A portable virtual machine for program debugging and directing. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *SAC*, pages 1524–1530. ACM, 2004.

[Eis97] Marc Eisenstadt. My hairiest bug war stories. *Commun. ACM*, 40(4):30–37, 1997.

[FB88] Stuart I. Feldman and Channing B. Brown. Igor: A system for program debugging via reversible execution. In *Workshop on Parallel and Distributed Debugging*, pages 112–123, 1988.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[HS02] Brent Hailpern and Padmanabhan Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[Kes90] Peter B. Kessler. Fast breakpoints: Design and implementation. In *PLDI*, pages 78–84, 1990.

[KNM06] Nitin A. Kamble, Jun Nakajima, and Asit K. Mallick. Evolution in kernel debugging using hardware virtualization with Xen. *2006 Linux Symposium, Volume Two*, 2006.

[KTD05] Toshihiko Koju, Shingo Takada, and Norihisa Doi. An efficient and generic reversible debugger using the virtual machine based approach. In Michael Hind and Jan Vitek, editors, *VEE*, pages 79–88. ACM, 2005.

[Lap92] J. C. Laprie. *Dependability : basic concepts and terminology in English, French, German, Italian and Japanese*. Springer, 1992.

[Lay09] Jeffrey B. Layton. NILFS: A file system to make SSDs scream. *Linux Magazine*, 6 2009.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java Virtual Machine. In *OOPSLA*, pages 36–44, 1998.

[LD03] Bil Lewis and Mireille Ducassé. Using events to debug Java programs backwards in time. In Crocker and Jr. [CJ03], pages 96–97.

[Lew03] Bil Lewis. Debugging backwards in time. In Michiel Ronsse and Koen De Bosschere, editors, *AADEBUG*, 2003.

[Lew07] Bil Lewis. Omniscient debugging. `http://www.lambdacs.com/debugger/`, Feb 2007.

[LGN08] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In Jan Vitek, editor, *ECOOP*, volume 5142 of *Lecture Notes in Computer Science*, pages 592–615. Springer, 2008.

[Lie97] Henry Lieberman. The debugging scandal and what to do about it (introduction to the special section). *Commun. ACM*, 40(4):26–29, 1997.

[LMS05] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.

[MAA+10] Sheila Moore, D. Adams, L. Ashdown, M. Cowan, J. Melnick, R. Moran, E. Paapanen, J. Russell, R. Strohm, and R. Ward. Oracle ® database advanced application developer's guide 11g release 2 (11.2), 2010.

[MCL89] John M. Mellor-Crummey and Thomas J. LeBlanc. A software instruction counter. In *ASPLOS*, pages 78–86, 1989.

[NW94] Robert H. B. Netzer and Mark H. Weaver. Optimal tracing and incremental reexecution for debugging long-running programs. In *PLDI*, pages 313–325, 1994.

[PL88] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution of parallel programs. In *Workshop on Parallel and Distributed Debugging*, pages 124–129, 1988.

[Ros96] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. Wiley, 1 edition, 9 1996.

[Sei09] Justin Seitz. *Gray Hat Python: Python Programming for Hackers and Reverse Engineers*. No Starch Press, San Francisco, CA, USA, 2009.

[SJ88] Jonathan M. Smith and Gerald Q. Maguire Jr. Effects of copy-on-write memory management on the response time of UNIX fork operations. *Computing Systems*, 1(3):255–278, 1988.

[Sos95] Rok Sosic. The Dynascope directing server: Design and implementation. *Computing Systems*, 8(2):107–134, 1995.

[Tas03] Gregory Tassey, editor. *The Economic Impacts of Inadequate Infrastructure for Software Testing: Final Report*. Diane Pub Co, 9 2003.

[Zel09] Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann, second edition, 6 2009.

[Zia08] Tarek Ziade. *Expert Python Programming*. Packt Publishing, 2008.