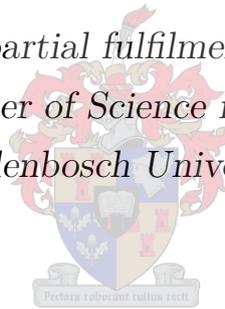


**Combining reverse debugging and
live programming towards visual thinking
in computer programming**

by

Abraham Liebrecht Coetzee

*Thesis presented in partial fulfilment of the requirements
for the degree of Master of Science in Computer Science at
Stellenbosch University*



Computer Science Division
Department of Mathematical Sciences
Stellenbosch University
Private Bag X1, 7602 Matieland, South Africa.

Supervisors:

Prof L. van Zijl
Dr M.R. Hoffmann

March 2015

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: 20th February, 2015

Copyright © 2015 Stellenbosch University
All rights reserved.

Abstract

Combining reverse debugging and live programming towards visual thinking in computer programming

A. L. Coetzee

Computer Science Division

Department of Mathematical Sciences

Stellenbosch University

Private Bag X1, 7602 Matieland, South Africa.

Thesis: MSc Computer Science

February 2015

Interaction plays a key role in the process of learning, and a learner's abilities are enhanced when multiple cognitive functions work in parallel, especially those related to language and visuals. Time is the most fundamental variable that governs the interaction between programmer and computer, and the substantial temporal separation of cause and effect leads to poor mental models. Furthermore, programmers do not have means by which to express their mental models.

The feasibility of combining reverse debugging and live programming was therefore investigated. This combination was found to be feasible, and a reverse debugger with higher levels of liveness was created for the Python programming language. It establishes a foundation for combining language and visual models as aids in computer programming education.

Uittreksel

Kombinasie van terug-in-tyd ontfouting en lewendige programmering tot bevordering van visuele denke in rekenaarprogrammering

A. L. Coetzee

*Afdeling Rekenaarwetenskap
Departement van Wiskundige Wetenskappe
Universiteit van Stellenbosch
Privaatsak X1, 7602 Matieland, Suid Afrika.*

Tesis: MSc Rekenaarwetenskap

Februarie 2015

Interaksie speel 'n belangrike rol in die proses van leer, en 'n leerder se vermoëns verbeter wanneer verskeie kognitiewe funksies in parallel opereer, veral dié wat verwant is aan taal en visuele denke. Tyd is die mees fundamentele veranderlike wat die interaksie tussen programmeerder en rekenaar reguleer, en die aansienlike temporele skeiding tussen oorsaak en gevolg lei tot swak kognitiewe modelle. Programmeerders het boonop nie middelle om kognitiewe modelle te artikuleer nie.

Die uitvoerbaarheid van 'n kombinasie van terug-in-tyd ontfouting en lewendige programmering was daarom ondersoek. Daar was bevind dat so 'n kombinasie moontlik is, en 'n terug-in-tyd ontfouting met hoër vlakke van lewendigheid was geskep vir die Python programmeringstaal. Dit vestig 'n fondament om taal en visuele modelle te kombineer as hulpmiddels in rekenaarprogrammering onderwys.

Acknowledgements

I am indebted to:

- My Lord Jesus the Great Physician, for healing my back after a simple prayer, when no one else could. And for inexpressibly more.
- My supervisor, Professor Lynette van Zijl, and co-supervisor, Doctor McElory Hoffmann, for their expertise, assistance and encouragement.
- My seraphic wife, Ilana Coetzee, whose fierce devotion to my well-being, unyielding support and gentle presence, makes everything feel right as rain.
- My family, for their endless love, patience, prayer and guidance.
- The MIH Media Lab, for the financial assistance. As well as for believing in me, and for the good coffee, the great people, and the spectacular fuball.
- The National Research Foundation (NRF), for the financial assistance towards this research, which is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Contents

| | |
|--|-------------|
| Declaration | i |
| Abstract | ii |
| Uittreksel | iii |
| Acknowledgements | iv |
| Contents | v |
| List of Figures | viii |
| List of Listings | ix |
| List of Acronyms | x |
| 1 Introduction | 1 |
| Programming should be natural | 1 |
| Visual ways of thinking are indispensable | 1 |
| Language and visuals must be combined | 2 |
| Interactive, customisable tools are required | 2 |
| Interaction is governed by time | 3 |
| The objective of this study | 4 |
| Thesis outline | 4 |
| 2 Background | 6 |
| 2.1 Reverse debugging | 8 |
| 2.1.1 Motivation | 9 |
| 2.1.2 Omniscience | 10 |
| 2.1.3 Reverse execution | 11 |
| 2.1.4 Benefits | 13 |
| 2.2 Live programming | 13 |

| | | |
|----------|--|-----------|
| 2.2.1 | Levels of liveness | 14 |
| 2.2.2 | Non-transient code | 15 |
| 2.2.3 | Hot swapping | 16 |
| 2.2.4 | Approach | 18 |
| 2.3 | Conclusion | 19 |
| 3 | Reverse debugging | 20 |
| 3.1 | Programming language | 20 |
| 3.2 | Reverse mechanism | 21 |
| 3.3 | Snapshots | 22 |
| 3.3.1 | Using Python | 23 |
| 3.3.2 | Using fork | 24 |
| 3.4 | Replay | 26 |
| 3.4.1 | Through recording | 27 |
| 3.4.2 | Through snapshots | 28 |
| 3.4.3 | Intercept mechanism | 29 |
| 3.5 | External state | 31 |
| 3.6 | Inter-process communication | 33 |
| 3.7 | Back to the future | 34 |
| 3.8 | Control of execution | 36 |
| 3.8.1 | bdb | 36 |
| 3.8.2 | pdb | 37 |
| 3.8.3 | epdb | 38 |
| 3.9 | The bidirectional ldb system | 42 |
| 3.9.1 | Breakpoint management | 42 |
| 3.9.2 | Intercept mechanism | 44 |
| 3.9.3 | External state | 47 |
| 3.9.4 | Inter-process communication | 48 |
| 3.9.5 | Back to the future | 51 |
| 3.9.6 | Conclusion | 53 |
| 4 | Live programming | 55 |
| 4.1 | User interaction | 55 |
| 4.1.1 | Awareness of change | 55 |
| 4.1.2 | An IDE | 56 |
| 4.2 | Changes to consider | 59 |

| | | |
|----------|---|-----------|
| 4.2.1 | Meaningful changes | 59 |
| 4.2.2 | Effective changes | 60 |
| 4.3 | Change propagation strategies | 61 |
| 4.3.1 | Execute the entire new program | 61 |
| 4.3.2 | Execute up to the same position | 62 |
| 4.3.3 | Execute up to the point of edit | 66 |
| 4.3.4 | Avoid unnecessary execution | 67 |
| 5 | Conclusions and Future Work | 74 |
| 5.1 | Overview | 74 |
| 5.1.1 | Motivation | 74 |
| 5.1.2 | Approach used | 75 |
| 5.1.3 | Reverse debugger | 75 |
| 5.1.4 | Higher levels of liveness | 76 |
| 5.1.5 | Summary | 78 |
| 5.2 | Objectives achieved | 79 |
| 5.3 | Shortcomings | 79 |
| 5.3.1 | Replacement objects | 80 |
| 5.3.2 | Control of time | 80 |
| 5.3.3 | Resource management | 80 |
| 5.3.4 | Change propagation | 81 |
| 5.4 | Future work | 83 |
| 5.4.1 | Architecture | 83 |
| 5.4.2 | Platform independence | 84 |
| 5.4.3 | Minimising execution | 84 |
| 5.4.4 | Usability studies | 85 |
| 5.5 | Summary | 85 |
| | List of References | 87 |

List of Figures

| | | |
|------|--|----|
| 1.1 | Pre-attentive visual communication of relationships | 3 |
| 3.1 | The snapshot-and-replay mechanism, for simulating reversal . . . | 23 |
| 3.2 | Pygraph uses the <code>f_trace</code> attribute of a stack frame for call graph visualisations | 24 |
| 3.3 | Orphan and zombie processes in epdb | 26 |
| 3.4 | Snapshots after all nondeterministic instructions, facilitates deterministic replay | 29 |
| 3.5 | Back to the future through forward activation | 35 |
| 3.6 | The mechanics of the <code>rnext</code> dictionary | 39 |
| 3.7 | Frame count activation | 41 |
| 3.8 | Setting up communication with socket pairs | 49 |
| 3.9 | Inter-process communication after multiple snapshots have been made | 50 |
| 3.10 | The steps involved in activating a previous snapshot | 50 |
| 3.11 | When forking, each parent process blocks as the snapshot, and its child continues | 51 |
| 3.12 | The state of processes after activation of a snapshot | 51 |
| 3.13 | The command line interface of ldb when tracing Listing 3.9 | 54 |
| 4.1 | The Graphical User Interface (GUI) of ldb | 57 |
| 4.2 | Returning to an earlier change point | 70 |
| 4.3 | An improved way of returning to an earlier change point | 72 |

List of Listings

| | | |
|------|--|----|
| 3.1 | Generating pseudo-random numbers, based on a seed | 26 |
| 3.2 | Replacement of <code>random.seed()</code> to record behaviour for replay | 28 |
| 3.3 | Replacement of <code>random.seed()</code> to make a subsequent snapshot | 28 |
| 3.4 | Replacement functions in <code>epdb</code> do not mimic the originals . . | 30 |
| 3.5 | Names are not identifiers | 44 |
| 3.6 | A name can refer to different objects, in different namespaces . | 44 |
| 3.7 | A module to replace the <code>time</code> module of the Python Standard Library (PSL) | 47 |
| 3.8 | Exploring nondeterministic behaviour without changing the code | 52 |
| 3.9 | A typical program that a novice programmer might create . . | 53 |
| 4.1 | A change can affect the program both directly and indirectly . | 60 |
| 4.2 | In a dynamic language, much information is only available at runtime | 60 |
| 4.3 | The same point of execution cannot necessarily be reached . . | 63 |
| 4.4 | Nondeterministic instructions should not be replayed after the point of change | 63 |
| 4.5 | A program can display the correct behaviour by chance | 64 |
| 4.6 | The same line number can be reached in different ways | 65 |
| 4.7 | Code executed within a trace function, does not trigger trace events | 68 |
| 4.8 | Previous instructions do not have to be re-executed | 69 |
| 4.9 | The point where a code block is defined, has to be returned to | 71 |
| 4.10 | The place to return to when lines are changed, is where the outermost parent entity is defined in the top-level namespace . | 72 |
| 5.1 | It might be possible to replace objects directly | 85 |

List of Acronyms

| | |
|---------------|--|
| ASCII | American Standard Code for Information Interchange |
| AST | Abstract Syntax Tree |
| CST | Concrete Syntax Tree |
| GUI | Graphical User Interface |
| HCI | Human–Computer Interaction |
| IC | Instruction Count |
| IDE | Integrated Development Environment |
| I/O | Input/Output |
| IPC | Inter-Process Communication |
| LTS | Long Term Support |
| MOOC | Massive Open Online Course |
| OOP | Object-Oriented Programming |
| POE | Point Of Execution |
| PSL | Python Standard Library |
| QWAN | Quality Without A Name |
| REPL | Read-Eval-Print Loop |
| TOPLAP | Terrestrial Organisation for the Promotion of Live Audiovisual Programming |
| UDS | Unix Domain Sockets |
| UUID | Universally Unique Identifier |
| VPL | Visual Programming Language |

Chapter 1

Introduction

Humans, not machines, should be central to man-machine interaction as machines are tools that serve us, not the other way around. To make interaction as natural as possible, computers should utilise the entire range of faculties employed by humans.

Programming should be natural Advances in computer programming could have the greatest impact on Human-Computer Interaction (HCI) as programming is the most general and powerful way of controlling computers. Kay et al anticipate “one of the 21st century destinies for personal computing: a real computer literacy that is analogous to the reading and writing fluencies of print literacy, where all users will be able to understand and make ideas from dynamic computer representations” [48]. However, programming is currently a complex mathematical exercise as it requires the human to encode situations in an unnatural, strictly logical fashion. Programming is also difficult because the computation is not visible, only the result. Kay et al conclude that it “will require a new approach to programming”. What should this approach be?

Visual ways of thinking are indispensable A number of theories in the field of psychology [31, 43, 59] indicate that the optimal comprehension and recall of information is obtained when multiple cognitive abilities work in parallel, especially language and visuals. Language, numbers and visuals are each more suited to different ways of communication, and the importance of oracy, literacy and numeracy have long been recognised. In our current age of big data, the importance of graphicacy is being realised as well, “the intellectual skill necessary for the communication of relationships which cannot be successfully communicated by words or mathematical notation alone” [28]. An extensive survey by the mathematician Jacques Hadamard in the early 1900s,

found that many of the greatest thinkers, such as Albert Einstein, “avoid not only the use of mental words, but also, just as I do, the mental use of algebraic or any other precise signs . . . they use vague images” [44]. Richard Feynman also, having once thought that “thinking is nothing but talking to yourself inside”, came to see that “thoughts can be visual as well as verbal” [39]. Visual ways of thinking are indispensable.

Language and visuals must be combined Apart from their impact on productivity, tools play an important role in thought. “The power of the unaided mind is highly overrated. . . . The real powers come from devising external aids that enhance cognitive abilities. How have we increased memory, thought, and reasoning? By the inventions of external aids” [57]. Dijkstra supports this, saying that “Nearly all computing scientists I know well will agree without hesitation . . . The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities” [34]. The tool programmers generally use to make computation visible, is to instrument their code to print out the values of variables. Textual output is however not ideal when the programmer is looking for patterns in trying to understand how an algorithm causes data to change over time – text and visuals are more suited to different programming situations [69], and our visual system is superior at pre-attentive pattern recognition, as can be seen in [Figure 1.1](#). Therefore, bringing visuals into computer programming, *alongside* numbers and words, holds much promise. Existing textual programming languages generally ignore the programmer’s visual faculties, and Visual Programming Languages (VPLs) make too little use of the programmer’s language faculties. A satisfactory combination of language- and visual-based interaction does not exist.

Interactive, customisable tools are required Trying to use visuals alongside language gives rise to the well-established but growing fields of information and algorithm visualisation. However, visuals by themselves, even dynamic visuals which unfold with the computation, are not sufficient. The amount of control and feedback that a programmer has while programming should be maximised, as greater comprehension is attained when a person can interact with the visuals with which they are presented, and when the person has *con-*

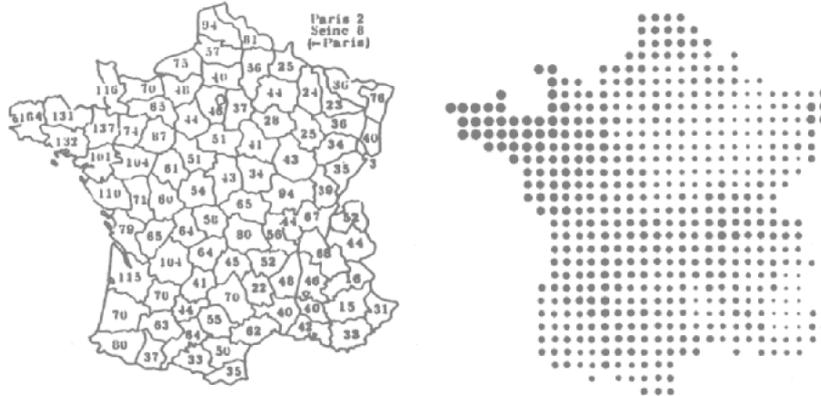


Figure 1.1: “Reading the left-hand image requires the viewer to search the image for the lowest and highest values, and the short-term memorization of the general layout of the numbers. On the right, a qualitative understanding of the image is immediately conveyed” [41, p. 34]

trol over the data on which a visual is based [46]. Programmers should be able to express their own mental model to aid their thinking, and so make the code habitable for them in the Christopher Alexander sense¹ [42]. It suggests the need for a visualisation framework, as part of the Integrated Development Environment (**IDE**), to allow for the personalisation of any visual representations which the programmer would like to employ. “Ways of supporting a programmer’s own system of imagery, integrated with the text of source code” [54], should be pursued. People who are learning to program would find it most valuable, as interaction plays a key role in the process of learning, especially the learning of language [31, 68].

Interaction is governed by time As a program executes, any visuals that are connected to the code should change with the program. This inherently dynamic nature of an executing program highlights *time* as the most fundamental variable, yet programmers do not have much control of time – a programmer can step forward during program execution by making use of the debugger found in most **IDEs**, but is not able to step backwards, which a reverse debug-

¹Alexander sought to understand how buildings might best enhance life for their inhabitants. He found the solution in the Quality Without A Name (**QWAN**), the objective essence of beauty. The buildings that possessed it had characteristics in common, which could be described by patterns. A generative grammar, called a pattern language, could be used by the inhabitants themselves, to reproduce the **QWAN**. The **QWAN** should similarly be pursued for and by programmers, the inhabitants of code.

ger would allow them to do. Without a reverse debugger, the programmer has to manually go through the entire stop-rerun-navigate cycle to simulate a single backwards step, which introduces a substantial temporal gap. Furthermore, the code is not connected to the running program so the programmer also has to manually go through the stop-rerun-navigate cycle before code changes have any effect. As programming consists of changing code, this is done regularly. This “cause/effect chasm seriously undermines the programmer’s efforts to construct a robust mental model” [36]. Incorporating visuals would not furnish the programmer with more control of time, nor remove the temporal chasm or reduce its negative impact, and would therefore not be of much value. Programmers should consequently first be given control of time – from being able to change the speed at which a program executes (and consequently the speed at which connected visuals animate), to being able to reverse program execution to go ‘back in time’, which a reverse debugger would allow. The conversation between programmer and computer would become more lively, which is also what ‘live programming’ aims to achieve by reducing the delay between changing the code of an executing program and seeing the effects of those changes. However “existing live programming experiences are still not very useful” [53] and no system exists which allows for both reverse debugging and live programming in a general programming language, indicating the need for continued research in this area.

The objective The aim of this study is therefore to investigate the feasibility of combining reverse debugging with live programming, to maximise control of time and minimise the temporal separation of cause and effect. Such a combination will establish a foundation for pursuing visual ways of thinking in computer programming. If a reverse debugger with higher levels of liveness should prove feasible, the goal will be to develop such a system. If, however, such a system proves infeasible, the fundamental obstacles will be investigated and explained, to aid future research.

Thesis outline Back-in-time debugging and live programming are each explained in more detail in [chapter 2](#), with reference to other work that has been done. The need for each, and the reasons for attempting to combine them, is also discussed. The design and implementation of a reverse debugger is

detailed in [chapter 3](#). The enhancement of the reverse debugger with higher levels of liveness, is considered in [chapter 4](#). The conclusions and suggestions for future work are discussed in [chapter 5](#).

Chapter 2

Background

Imagine how counter-productive it would be if, each time someone was asked to repeat their last sentence, the entire conversation had to be repeated. Or worse, if the entire conversation had to be repeated each time a different person was about to speak. Unfortunately that is exactly what happens in most current computer programming environments.

There is a difference between the manufacturing of an appliance in a factory, and the use thereof in a home. Similarly, it is important to differentiate between the process of creating a computer program, and simply running a completed program. A programmer, like an engineer or artist, has to work in an environment which supports their needs while ‘manufacturing’ a program. Programming is a type of conversation between the programmer and the computer; a constant interaction with, and reaction to, each other [61, p. 1]. A computer program executes forwards in time, but when a cause of an effect is investigated in a running program, such as when tracing a bug, a programmer has to search backwards in time. However, computers do not allow backwards movement in execution time, which means that reversing has to be simulated by the programmer, who has to restart the program and navigate to the point just before – in other words, the entire conversation needs to be repeated each time a backwards step is taken.

Furthermore, when a programmer ‘talks’ to the computer by changing the code which produced a running program, the computer does not ‘talk’ back to provide feedback about whether the change has had any effect – the running program is separate from the code. In this situation too, the programmer first has to repeat the conversation by manually going through the process of stopping the running program, recompiling, restarting the program and recreating the previous state, to see the results of the changes, if any. As this entire process is regularly repeated, it is one of the most time-consuming parts

of programming, and drives research in two directions looking for ways that it might be alleviated. The first is for the programmer to be able to step backwards in time while the program that they are currently working on executes. The second is into higher levels of liveness during programming, where the delay between changing code and seeing the result thereof, is minimised. Although reverse debugging and live programming may at first appear to be mutually exclusive, they are not – whereas reverse debugging is concerned with issues before a possible code change, live programming is concerned with what happens after that change. Both are concerned with reducing the temporal separation between code and its effects.

Reducing the amount of time spent on programming would not be the only or even greatest benefit of having a live system that can also reverse. Interaction plays a key role in the process of language acquisition [31, 68]. Studies in the field of computer algorithm visualisation have found that greater comprehension is attained when one has control over the data on which a visual is based, and when one can interact with the visuals [46]. This suggests the need for a customisable system in the IDE whereby a programmer can construct their own visualisations that are connected to the instructions or data of a program, to be able to ‘see’ program state and come to understand an algorithm or piece of code. The output should change or animate as the program executes. However, computer programs execute at tremendous speed. Consequently the output would change too quickly for the programmer to grasp the behaviour of the program. “What the programmer actually wants is to watch the computation unfolding smoothly over time, changing slowly, gently, predictably and meaningfully, and being presented in an appropriate visual representation.” [64, p. 17]. The programmer should therefore be able to slow down the output or connected visuals, and even pause or repeatedly reverse and ‘replay’ it, to consume it at their own pace.

Using customisable visuals to observe the program state while manipulating it in this way, would result in a powerful environment in which to learn to program. Two of Bruner’s [31] modes of representation – language-based *symbolic* and image-based *iconic* – would be working in parallel. Interacting with the visuals should also result in changes to code, covering Bruner’s final mode as well, namely the interactive *enactive* mode. The programmer would be able to create a custom VPL alongside the text. However, the visuals should not

merely be a different representation of the structural elements of code such as loops or conditionals, as in most VPLs. Instead, they should also be able to connect to the data on which the program operates. A customisable, interactive, textual and visual system would allow the computer to ‘share its thoughts’ with the programmer in the way they best understand.¹

Furthermore, being able to change the code of the running program and immediately see the result of that change, would mean that writing, executing and debugging code are no longer separate activities. One could “construct an entire program during its own execution” [64, p. 24]. This will facilitate understanding by solidifying the connection between code and its execution, as the substantial temporal separation of cause and effect hinders the formation of robust mental models [36]. Additionally, seeing a customised visual representation of program state means no brain power has to be spent on maintaining it in the mind, freeing the programmer to focus on more important issues, such as internalising programming concepts, and program design. McLean *et al* [55, p. 5] conclude that there is a need for research in this area: “Visualisation is central to live coding ... Visualisation of live code however remains under-investigated in terms of the psychology of programming”.

Although a live, interactive, customisable, visual system is the direction I have had in mind, I have not focused on visualisation. Eisenstadt’s analysis of debugging experiences “pinpoints a winning niche for future tools: data-gathering or traversal methods to resolve large cause/effect chasms” [36]. Burckhardt *et al* [32, p. 9] also recently suggested that “future work may look at how live programming and step-wise debugging can work together”. This is what I have explored instead, through the combination of reverse debugging and higher levels of liveness, each of which is discussed below. The control of time that the programmer has may serve as a foundation on which to build a visual system in future.

2.1 Reverse debugging

A programmer has a mental idea or written specification on which he bases his code, and when the code is run, it becomes clear whether the program

¹Victor [20] provides a brilliant proof of concept to demonstrate one way that such a system could be realised.

operates correctly according to that idea. It often does not, and the reason for it has to be found. This is called debugging, which makes up a large part of programming.

2.1.1 Motivation

When a program is executed forwards in time, the data on which the program state is based, and the process by which it changes, usually cannot be seen. Some types of programs, such as virtual worlds or physical simulations, do inherently display more of the state of the program. For these types of programs, changing code and re-executing, interacting with the program or even just letting the program run, results in feedback which gives some degree of insight into the code. However, not all parts of these programs are tied to some form of output, and not all programs inherently visualise their state in that way. Therefore, a programmer generally cannot tell when a program deviates from its intended behaviour, and usually only detects that it has deviated at some later point in time. The computer or programming environment does not allow stepping backwards or viewing program state history, hence the programmer has to reason backwards from effect to cause. The process takes time and is complicated, especially in larger programs and when the programmer is inexperienced. Programmers often try to make computation visible by instrumenting code with statements which print out or log the values of variables. This form of debugging is called ‘print and peruse’ or ‘dump and diff’ [36]. If the instrumentation code has been placed correctly, the programmer hopes to be able to use the output the next time the program is run, to observe where it deviates from the intended behaviour. The code usually has to be changed and the program rerun for each step backwards in the causal chain anyway, as it is rare to go from an incorrect effect to its cause in a single step. The cyclic nature of this form of debugging is evident in that it requires the program to be stopped and re-executed after each instrumentation. This is not ideal, due to the extra time it takes to reproduce state, and because state cannot always be reproduced, which results in “large temporal or spatial chasms between the root cause and the symptom” [36]. Furthermore, the programmer has to remember to remove the instrumentation code as it is not actually part of the program, but merely exists to serve the programmer while developing the application.

Computer programming is embedded in time, and consequently requires the control of time [15]. It would be better if the environment in which the program is being ‘manufactured’, allowed the programmer to slow down the program, and even pause or repeatedly reverse and ‘replay’ it. The closest a programmer gets to being able to pause the execution of a program, is by setting a breakpoint when debugging, then stepping forward from there. If, however, the last couple of lines of code were not completely understood, as is often the case especially when learning to program, there is no way to have them execute again, other than by going through the stop-rerun-navigate cycle. The programmer needs to be able to *reverse* to the correct point, to repeatedly observe what that section of code is doing.

2.1.2 Omniscience

The computer, having forward-executed the program up to some point, should really be able to provide us with information about it. Such an improvement is possible, and is found in what are called omniscient debuggers [62]. They are usually classified as reverse debuggers, although they might more accurately be described as “history logging” debuggers [30, p. 299], as they merely record information during execution to view or query later, rather than allow the programmer to actually step backwards in time in an executing program. “Omniscient” comes from the fact that the entire state history of the program, having been recorded, is available to the debugger after execution. There is then no need to rerun the program, and no need for manual code instrumentation.

Software-based omniscient debugging started with the 1969 EXDAMS system where it was called “debug-time history-playback” [29]. The GNU debugger, GDB, has supported omniscient debugging since 2009, with its ‘process record and replay’ feature [7]. TotalView [5], UndoDB [18] and Chronon [9] appear to be the best omniscient debuggers currently available, but are commercial systems. TOD, for Java, appears to be the best open-source alternative, which makes use of partial deterministic replay, as well as partial trace capturing and a distributed database to enable the recording of the large volumes of information involved [62].

According to a 2013 study by the Judge Business School at Cambridge University, programmers spend 26% less time on debugging when using omni-

scient debugging tools, which would translate into global savings of \$41 billion annually, should all programmers use them [24]. However, as the overhead of recording and querying the entire state history of a program is usually prohibitively high, omniscient debuggers are still not commonly found in modern IDEs [62, p. 15].

2.1.3 Reverse execution

Debuggers that do not merely allow navigation of a recording, but are actually able to step backwards in execution time, also exist. They can more accurately be described as back-in-time, time-travel, bidirectional or reverse debuggers.

The first such system was the 1981 COPE prototype [27], that stored all program state in its file system. It could periodically save all changes to files, which it called a partial checkpointing strategy, to use when reversing. As only the changes were saved, an `undo` command allowed reversal only in the increments recorded during forward-execution. It did not undo any changes to the external environment when reversing.

The 1988 IGOR prototype [38] also allowed for reverse execution through what were called both checkpoints and snapshots. It focused on the C programming language, executing on the DUNE distributed operating system. Changes to the compiler, operating system kernel, linker and loader were required, and even then, reversing was not always possible. The snapshot mechanism consisted of saving the program counter, memory pages, registers and file pointer data, to a file. To reverse a step, the closest previous snapshot was used to reconstruct the program at an earlier point in time, and the user then had to manually forward-navigate to the intended destination, using an extremely slow interpreter. The environment state also had to be manually recovered when reversing, though it could be automated to some degree.

The 1988 Recap system [60] was the first to use the `fork()` system call available on Unix-like operating systems, to create snapshots by duplicating the calling process. The snapshot would then block, waiting for instructions. Forking is efficient due to the copy-on-write mechanism. During forward-execution, Recap recorded the behaviour of nondeterministic events. Reversing was achieved by continuing from an earlier snapshot process, and forward-executing behind-the-scenes. The recorded events were used, if necessary, to ensure deterministic replay, but it did not undo changes to the external

environment. It could not precisely define a specific point in the program execution [56, p. 84], so reversed in intervals defined by time. The use of a software-based instruction counter, to reference a specific point in the program execution, was first proposed in 1989 [56]. An instruction counter is a form of program counter, but instead of referring to the memory address of an instruction, it is simply an integer that is used to count the number of instructions that are executed, so is called the Instruction Count (IC).

A 1993 debugger [67] for specifically the standard ML functional programming language and SML/NJ compiler, also used a snapshot and replay mechanism. Snapshots were provided by the compiler through its first class continuations, which also allowed for program execution to be redirected. It could not undo changes to the environment when reversing.

Bdb [30], released in 2000, was the first practical reverse execution prototype that did not require expensive support, according to a 2012 review [37, p. 2] of the history of reverse debugging. It focused on the C and C++ programming languages, and used `fork()` for snapshots, as well as logging to ensure deterministic replay. However, it did not address reversing changes to the environment, and it sometimes made use of multiple re-execution passes. It should not be confused with the Python debugger framework with the same name, so is referred to as the *Boothe bdb debugger* in this study.

In 2012, GDB appeared to be “the only open-source and free solution for reverse debugging” [37, p. 3], having supported reverse execution since 2009 [8]. It also uses `fork()` to create snapshots that can be returned to directly, but when reversing, it undoes the effects of a single machine instruction at a time, making it extremely slow. It does not undo changes to the environment either [6].

UndoDB [18], one of the commercial omniscient debuggers, also makes use of `fork()` for snapshots, but only does so for querying the past, not for re-execution from an earlier point in time.

The 2011 epdb prototype [63] is the most recent, open-source, reverse debugger, and is written in the Python programming language. It is built on the Python Standard Library (PSL) bdb and pdb classes, and implements the Boothe bdb debugger methodology, but contains additional side effect management functionality. In this study it is used as a starting point for a reverse debugger, although a number of changes have to be made to allow for higher

levels of liveness and future visualisation functionality, which [chapter 3](#) describes in detail.

2.1.4 Benefits

There are a number of benefits in using a reverse debugger rather than an omniscient debugger, specifically for beginner programmers. The first is that a reverse debugger is less costly in terms of both hardware and runtime overhead, as only nondeterministic events need to be managed, not the entire trace history. Although more expensive hardware might be a suitable requirement for professional programmers, an inexpensive workstation should be sufficient for a student who wants to learn to program. The second benefit of working with a step-wise rather than omniscient debugger is that one sees the computation unfolding when navigating either forwards or backwards in time, which gives a student a better grasp on exactly how the computer executes code. Some omniscient debuggers are more capable of directly revealing causes [[62](#), p. 7], which, though a useful tool for a professional programmer, does not provide the learning programmer with a solid foundation in procedural thinking. Another benefit to reverse debugging is that the program continues to execute, which means that it is able to continue reacting to different inputs. This is useful to both professional programmers testing their code, and to those learning how to write programs that are stable in a wide variety of situations.

The ability to move backwards in a live program is also where live programming comes into play – when reverse execution is used to trace a bug, the ideal system would allow the programmer to change the code of the executing program and have the program immediately reflect that change, so that the program may continue executing as before.

2.2 Live programming

Live programming is specifically about the liveness or immediacy of feedback provided to a programmer when a computer program *is being created*. The hope is that it would allow “programmers to edit the code of a running program and immediately see the effect of the code changes” [[32](#), p. 1].

2.2.1 Levels of liveness

Tanimoto defined four degrees or levels of liveness, that describe the immediacy of feedback provided by visual programming systems [66]. These levels can be broadened to apply to all software development environments [69]:

- Level 1 liveness means that no feedback is provided by a running program when its code is changed.
- Level 2 liveness means that feedback is provided on demand.
- Level 3 liveness means automatic feedback when program code is changed.
- Level 4 liveness builds on level 3 by automatically responding to other events as well, not just code changes.

Current development environments mostly have level 2 liveness – when code is changed, the programmer has to request feedback from the computer by going through the stop-compile-run-navigate cycle. Level 3+ liveness creates a more fluent conversation between programmer and computer. It is exemplified by what is found in a typical spreadsheet where, when a value of a cell changes, any dependent cells automatically update. Being able to change code and have the executing program reflect that change immediately in the way a spreadsheet does, drastically reduces the programming time, as nothing has to be requested. Getting immediate feedback means it is also a good way to gain understanding on, or tweak, elements of the program code such as the value of a specific variable. It cuts down on the amount of speculation involved as changes can then be explored and seen, rather than imagined. Good examples of exploring what programming would be like if it were like a spreadsheet, include Subtext [35], LambdaCalc [61] and Euclase [58].

Though the liveness seen in a spreadsheet is what we are after, spreadsheets fall under the declarative programming paradigm. As computers need to be instructed in an imperative way at the base level, all declarative languages (sets of declarative instructions) eventually have to be mapped to imperative instructions by their developers. This explains why students are generally taught about the imperative paradigm first, to learn about the procedural way that computers ‘think’. It would therefore be better to have higher levels of liveness in a language which makes use of the imperative paradigm, but a multi-paradigm language would be ideal.

An important question to ask is, how can a change to code be propagated in a running program so as to minimise the amount of time spent waiting for

the program to reflect it?

2.2.2 Non-transient code

To clarify the type of programming that I have in mind, as well as to explain the difference between live programming and live coding, a distinction needs to be drawn between what has been called transient or ephemeral code [23], and non-transient or long-lived code.

Programming consists of writing computer code, and the purpose of code is to create artefacts or effects, when it is run. Most code is written as non-transient code so that it can be run many times over, always creating predictably-similar effects. The code is stored in some form, which is called a computer program, and run whenever the effects that it produces are desired, ranging from simple calculations to websites to computer operating systems.

Code is not always stored as a program for repeated use, as there are situations in which code is written to produce an effect only once; in other words, situations where repeatability is not important, and the code is linked to a specific point in time. One example of this is when a small piece of code is run to see what it does, often by making use of a Read-Eval-Print Loop (**REPL**) prompt. When the line of code is run and understood, it has served its purpose in giving insight to the programmer, and the code is then discarded. Another example of throw-away, time-dependent code is when a programmer-performer writes code to create the audiovisuals during a live performance, which is called *live coding* [33]. The “live” here refers to the performance, rather than to how responsive the code of a program continues to be during its execution. The performer appears on stage with a computer, which he uses as his instrument to create the music (for example), by writing computer code instead of simply playing a musical instrument as is traditionally done. It is a musical performance, so the code is constantly changed to change the music. The desired effect is that enjoyable music is produced for the audience to listen to, or even dance to at an ‘algorave’ [22]. As the music starts, progresses and again winds down to silence, the corresponding code which produces the music gets written, changed and again discarded. Instead of performing a pre-written score of music, the musician creates the music by improvising. The desired effect is only the music at that specific moment in time, and the code merely serves as the means, so that the code itself is of little importance. A future performance

will again be differently improvised, so there is no desire or need to keep the code. Instead, the music might be recorded to listen to again. Examples of performances can be found on the website of the Terrestrial Organisation for the Promotion of Live Audiovisual Programming (**TOPLAP**) [3]. This form of code, which gets discarded after it has produced the desired, time-dependent effect, is called transient or ephemeral code [23].

Transient code is not the type of code I have had in mind. Instead, my focus has been on the production of non-transient code, which is stored, and which always produces the same or predictably-similar effects when run – in other words, time-independent code. The reason for it is the repeatability of non-transient code. I see the combination of live programming and reverse debugging being used in the learning of programming, where it is important for students to learn to connect code to the effects it produces, and this is done through repetition, as when learning any complex skill. As non-transient code is also much more widespread in the form of software applications, it is important to focus on it when educating new programmers.

2.2.3 Hot swapping

Another distinction that needs to be drawn is between live programming and ‘hot swapping’ of code, also called ‘edit and continue’. Changing some code while a program is running, so that identical, subsequent interaction with the program results in a different response, is called hot swapping as code is ‘swapped’ or modified while the program is ‘hot’ or running.

Hot swapping of code only affects the future of an executing program. It can be useful, and is especially suited to applications that operate in a cyclic way, such as animations or games where new values are calculated on every iteration of a loop. A change then simply results in different behaviour on the next iteration of the loop. Hot swapping is often used during live coding to change the audiovisuals, perhaps on the next beat. As an example of how hot swapping might be used when developing a game, consider a simple side-scrolling game where a character is moved through a virtual world. For a programmer who is developing such a game, it might be possible to change code while the game is running, so that the next time an action like jumping is performed, a different effect is observed – the character might now be able to jump only half as high as before. There would have been no need to go through

the stop-compile-run-navigate cycle, so reducing development time and facilitating quick and easy tweaking of game elements or mechanics – Victor [15] provides an excellent demonstration of hot swapping when developing a basic game. Epic Games’ latest game development engine, Unreal Engine 4, also incorporates some hot swapping [19, 21]. Consider what the senior technical artist and level designer has to say about it: “With this kind of direct editing, we get a massive productivity bonus because it lets [designers] figure out how they want something to work exactly. We’re not going to have this iterative process where I spend all night writing the code, you get to see it the next time there’s a build and go ‘No, that’s not right at all,’ so it really cuts down on that kind of loop. By making our tools as intuitive and user-friendly as possible, we cut down on a lot of the development iteration loss” [14].

However, as with transient code, there is a time-dependence factor to hot swapping. A program in which a hot swap has been performed, responds differently before and after a certain point in time – performing an action before the code was changed, and performing the same action afterwards, does not produce the same result. There might then still exist side effects in the running program or its environment, from the earlier code and interaction with the program, for which the current code cannot account. For instance, a game character might have been able to access an area of the game only as a result of his previous ability to jump twice as high. Now that the code has been changed, he can no longer access this area of the game *should the game be played from the start*. Yet, because a hot swap of code was performed, the character is currently still in that area as a hot swap only affects the future, not the past. A simple example of a remaining effect in the environment after a hot swap, rather than in the program itself, might be a file on the file system – if at the earlier point in time there had been code which created a file, and this code was removed in the change, the file would still exist as an effect for which the later code does not account. Although hot swapping can be useful, effects that remain after a code change is unacceptable when focusing on programming education. When learning to program, the code should instead always reflect and be linked to the effects, so that a student might internalise that connection over time, learning to effectively write code which would produce a desired outcome. That is the aim of live programming: for the running program, its code, and the produced effects, to be connected

at all times.

2.2.4 Approach

Having clarified that change should be propagated in such a way that the program state always reflects the state of the code, the question should again be asked, how can a change to code be propagated in a running program so as to minimise the amount of time spent waiting for the program to reflect it? It is a difficult problem to solve in a general programming language. There are a number of ways to approach it, such as simply re-executing the program and deterministically replaying previous inputs, designing the programming language to manage state as a first-class entity, or some form of dependency analysis for change propagation. The exploration of these ideas has generally focused on design at the language level, rather than only on the environment [52, 53]. Edwards states that “The fundamental reason IDEs have dead-ended is that they are constrained by the syntax and semantics of our programming languages” [10]. Alan Kay, the man behind concepts like the Dynabook, Smalltalk, OOP and more [47], thinks “that a large part of programming language design . . . is treating the language and how it is worked with as *user interface design*” (emphasis in original) [48, p. 3]. Though I believe that much progress will be made through the design of new programming language models, as the “design of the language is just as critical to the programmer’s way of thinking as the design of the environment” [16], designing new models is no mean task, especially models that focus on time-travel and change propagation. However, as live programming is “emerging as the next big step in programming environments” [53, p. 9] and “is an idea whose time has come” [32, p. 10], there have been a number of experiments. Notable attempts, most of which have focused on design at the language level, include ALVIS LIVE! [45], SuperGlue [52], LPX [53], Acar’s Self-Adjusting Computation [25] and more recently Circa [40], Moon [50] and some of Victor’s ideas [15, 16]. However, McDirmid [53, p. 9] recently concluded that “existing live programming experiences are still not very useful”, indicating the continuing need for research in this area. Instead of focusing on the design of new programming languages, I chose to investigate how live programming might be brought to an existing programming language.

2.3 Conclusion

The only combination of reverse-step debugging and live programming, appears to be the 2006 ALVIS LIVE! development environment, which has seen promising results with novice programmers [45]. It brings together direct manipulation of visuals, immediate feedback, and reverse execution with control of speed. Unfortunately, it uses a simple, pseudocode-like language, as individual commands are undone behind-the-scenes through the execution of another command, similar to how a `pop` might be used to undo a previous `push` to a stack. The language has been further restricted to support only single-procedure algorithms that involve array iteration.

The combination of reverse debugging and live programming, to form a foundation on which to pursue the integration of language-based and image-based ways of thinking for computer programming education, appears to be promising. There exists a need for such a combination in a general programming language, and its feasibility will be investigated in this study.

Chapter 3

Reverse debugging

In this chapter, I document the design of my reverse debugging system, and give details of its implementation. I have called my live debugger *ldb* in a similar vein to the naming of the PSL base debugging framework *bdb*, the PSL debugger *pdb*, the Boothe *bdb* debugger [30] and a prototype reverse debugger *epdb*, all of which I have used as a starting point. Sabin [63] covers a number of aspects of reverse debugging, some of which I will revisit, to explain reverse debugging as it pertains to *ldb*, which differs from these other systems in a number of ways. As *ldb* is an experimental combination of reverse debugging and live programming, it is important to document in detail the way it works, so that subsequent systems can easily expand upon it. The general principles which have guided its design are covered by section 3.1 through section 3.8, and section 3.9 details how *ldb* differs from previous systems.

3.1 Programming language

Instead of focusing on the design of new programming languages and paradigms as is currently done in most live programming research, I chose to investigate to what degree reverse debugging and live programming might be brought to an existing programming language, and one which is already used both for professional work and in education. It would result in earlier and wider adoption of a live, back-in-time system, should it be possible and prove to be useful. I consequently decided on Python¹.

Python is already used in many undergraduate computer science degree programs internationally, and there appears to be a growing trend away from languages like C towards languages like Python and VPLs like Alice and

¹Available at <http://www.python.org>

Scratch [26, p. 42]. In 2011, inspired by Khan Academy, Sebastian Thrun and Peter Norvig presented the Stanford University course, *Introduction to Artificial Intelligence*, online. As the first really Massive Open Online Course (MOOC) with over 160 000 students, it started an online education revolution [17]. Thrun, who founded Google X and led the development of the Google self-driving car [2], then formed the Udacity organisation [17], which has used Python as instructional programming language from the start [11, 12, 13]. Norvig, who has been a Director of Research at Google Inc. since 2005 [1], also recommends Python as a first programming language [4]. Its dynamic typing, use of whitespace, expressiveness and other features results in it being more intuitive to new programmers. This results in greater productivity and higher quality work when using it rather than another high-level language such as Java or C++ [51, 65]. Python is also a powerful, multi-paradigm language used professionally in many domains, making it a good choice.

When referring to Python, I am either speaking about the Python programming language, or its reference implementation, which is CPython. I focused on the latest version of Python available when this project was started, Python 3.2, as well as the then-latest Long Term Support (LTS) version of the Ubuntu operating system, 12.04.

3.2 Reverse mechanism

How can a reverse execution debugger that is interactive, not merely omniscient, be implemented in and for Python? As the computational resources that are available to novice programmers are limited, and as the live programming features will require the greater share of that computational power due to regular re-execution of parts of the program, the reverse debugger needs to use a mechanism which is sufficiently lightweight. It also needs to be fast, so that the system continues to reduce the temporal gap once the live programming features are integrated.

Reversing by undoing the results of single instructions, as in GDB [6], leads to much overhead. As most instructions are not reversible, the ability to reverse requires that data be recorded before each instruction executes. Depending on the number of instructions to reverse, it could be faster to merely simulate reversing, by restarting the program and forward-executing

to the correct position. This would need to happen behind-the-scenes, so that it would appear to the user as if the program simply reversed. To complete the effect, the program would also need to re-execute deterministically – that is, as if it is merely replaying a recording of the previous execution.

This restart-and-replay mechanism could be improved by making use of checkpoints [27, p. 9] or snapshots [38, p. 113]. A snapshot can be thought of as a position in time where the state of the program has been captured, much like automatic saving when playing a computer game. Reversing is then similar to reloading, taking the program back to that earlier snapshot. A simplistic snapshot model for reversing might make a snapshot for every instruction, so that reversing is directly activating a specific snapshot, but this would result in too much overhead. A better balance of resources while still placing an upper bound on the amount of time it takes to reverse, is to only create a snapshot periodically. When the user reverses to a specific point, the program continues from the preceding snapshot, and the instructions which lie between the snapshot and the point where the user intended to reverse to, are automatically replayed behind-the-scenes. This is called ‘checkpoint and replay’ [60, 67] or snapshot-and-replay [18].

Snapshots cannot be made in the middle of the execution of an atomic instruction, so have to be made between instructions. This is the reason why snapshots appear between instructions in illustrations in this study, as in [Figure 3.1](#). However, the snapshots are usually referred to as ‘at’ an **IC**, or even metonymously referred to by the **IC**, as in “activate **IC** 7”. The number could refer to either the previous or next instruction. In this study and in `ldb`, as in `epdb`, it refers to the next instruction, meaning that “snapshot at **IC** 1” refers to a snapshot *before* the first instruction, not after [49, p. 27], and that stepping forward from that snapshot would result in the first instruction being executed again. In other words, it is set to one less than the number of instructions that have already been executed, as it refers to the next instruction that would execute should the program continue.

3.3 Snapshots

There are different ways in which snapshots could possibly be made and activated, to facilitate reversing.

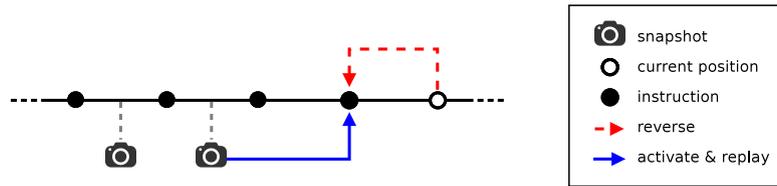


Figure 3.1: The snapshot-and-replay mechanism, for simulating reversal

3.3.1 Using Python

Python itself does not allow the creation of snapshots due to a number of limitations. It is not possible to save the call stack, or to replace it with an earlier copy to continue the user program at an earlier point in the execution. It is also not possible to adequately manipulate the current stack, as most of the attributes of a stack frame are read-only, as indicated under section 3.2 of the Python language reference². Much of CPython is written in the C programming language, and the C source code file³ for a frame object, `frameobject.c`, shows that only `f_trace` and `f_lineno` can be set. The `f_trace` attribute can be used to specify a trace hook or callback function for certain trace events, which enables the creation of applications such as memory profilers, code coverage tools, and call graph loggers such as Pycallgraph⁴, shown in Figure 3.2. It is also used for implementing debuggers, as explained in section 3.8, but does not help with the creation of snapshots. The value of `f_lineno` can be changed, as is done in the pdb `jump` command, to redirected the interpreter to next execute a different line. This might seem like a possible way to reverse if the program state could also be restored, but there are a number of situations in which even this redirect cannot be done without problems, or at all. For example, jumping into a `try` suite, `for` or `while` loop, or to a different code block such as into or out of a function, is not allowed at all.

Even if a different mechanism could be used to manipulate the stack, the management of state would still be problematic. If all the necessary state information could be saved periodically, by copying the entire `locals` and `globals` dictionaries, the overhead would be extremely high. It would also be difficult and costly to save only changes to the dictionaries, as there will be many changes, and restoring them would require applying each in turn,

²Available at <http://docs.python.org/3.2/>

³Available at <http://github.com/python>

⁴Available at <http://pycallgraph.slowchop.com>

As any number of copy processes can then exist, it is necessary to distinguish between waiting snapshot processes and the active process with which the user interacts at any point in time, which is called the ‘debuggee’ in epdb [63, p. 39], but will be called the *main process* hereafter. All snapshots, although copy processes, act slightly differently to the main debugger process in that, instead of waiting for commands from the user, they wait for commands from the main process. The user therefore does not know about these other processes; they merely exist behind-the-scenes to provide the ability to reverse. When a backwards step is taken, the main debugger process activates the closest snapshot process before the intended destination, and provides it with replay information, before shutting itself down. When a snapshot is activated, it does not take over as the main debugger process. The snapshot was made at that point in the program execution for a reason, so the snapshot should continue to exist there. For this to be achieved, the snapshot forks again when it is told to activate [60]. One of these two processes takes over as the main debugger process, and the other continues to block at that point as the snapshot. In this way, a snapshot is retained at that position for later use, and a new main process is also created for user interaction.

The many different processes and their relations to each other, have to be carefully managed, to prevent the formation of both orphan and defunct or zombie processes. A process, called a parent process, should manage the processes which it creates, called its children processes. A *zombie* is a process that has terminated, but whose exit status has not been read by its parent. Some resources continue to be used by the process, until its parent performs a `wait` to read its exit status, which is referred to as ‘reaping’ the process. An *orphan* is an active process whose parent process has terminated, which means that the parent process will never reap it. It is adopted by the first process created when the computer boots – the `init` process with identifier 1. Orphans are sometimes intentionally created to continue execution in the background, and are then referred to as *daemon* processes. The incorrect management of processes leads to zombies and unintentional orphans, which waste system resources and unnecessarily limit the number of processes allowed. In epdb, both orphan and zombie processes are formed, which continue to exist when the program is interrupted, as shown in [Figure 3.3](#).

Ldb carefully manages the creation and termination of processes by using

```
$ ps -ef | egrep "python3|PID" | grep -v grep
      PID  PPID  TTY  CMD
12800      1    ?  python3 epdb.py examples/example5.py
12807 12800    ?  [python3] <defunct>
12813 12800    ?  python3 epdb.py examples/example5.py
```

Figure 3.3: Orphan and zombie processes in epdb

a single-child process model, which is explained in [section 3.9.4](#).

3.4 Replay

Snapshot-and-replay serves to simulate reversing by making use of an earlier snapshot, combined with automatic, behind-the-scenes, replay-like forward-execution to reach the intended destination. To have it appear to the user as if a simple backwards step was taken when reversing, the intermediate instructions have to be replayed in such a way that the program reaches the same state as before.

Instructions that depend only on the internal state of the program, are called deterministic instructions. An example is an instruction that adds two numbers. Given the same program state as before, deterministic instructions automatically execute in exactly the same way each time, so they do not pose a problem during replay. A nondeterministic instruction, however, depends on the state of the environment, so usually results in different behaviour when re-executed.

For example, consider the nondeterministic `seed()` function in the `random` pseudo-random number generator module of the `PSL`. A call to `random.seed()`, as in [Listing 3.1](#), sets the seed of the generator by using a randomness source provided by the operating system – usually the current system time. The seed determines the sequence of numbers that the generator then deterministically generates.

```
1 | random.seed()
2 | x = random.random()
3 | print(x)      # 0.912413526333249
```

Listing 3.1: Generating pseudo-random numbers, based on a seed

When the user reverses at a later point in the program, but the pro-

gram continues from a snapshot before these instructions, they are re-executed behind-the-scenes. The seed is consequently set to a different value because the system time has changed, and the generator then returns different values than it did before. The user would consequently notice that the program had done more than simply reverse a step, as instructed.

A mechanism is therefore required to manage nondeterministic instructions, so that returning to a previous point in time would also return the user program to the same state as before.

There are a number of different ways in which nondeterministic functions can be managed. When the debugger compiles the user program to an Abstract Syntax Tree (AST) or code object, it could make changes to that object before executing it. Parts of the user program could then be replaced with code that behaves differently, so that behaviour can be reproduced on subsequent executions. As everything in Python is a first-class object, Python also allows for the runtime replacement of objects, and allows for the module loading mechanism to be customised as well. Objects could thereby be replaced only if and as necessary, which would be faster.

3.4.1 Through recording

One way to allow for the deterministic replay of nondeterministic instructions, would be to record the behaviour of their first execution and merely replay it in future. That is exactly how omniscient debuggers work, as explained in [section 2.1.2](#).

Such a replay strategy can be demonstrated through an example similar to one in `epdb` [63, p. 52]. To enable deterministic replay of the code shown in [Listing 3.1](#), the `random.seed()` function could be replaced with a custom one that records the state of the generator during normal execution. When it is called again during replay, the call is intercepted, and the state of the generator is simply set to its recorded value, as in [Listing 3.2](#). The subsequent instructions in [Listing 3.1](#) would then have the same behaviour as before.

```
1 | def seed () :
2 |     if mode == 'normal':
3 |         original_random.seed()
4 |         store['random.seed'][IC] = original_random.getstate()
5 |     elif mode == 'replay' :
6 |         original_random.setstate(store['random.seed'][IC])
```

Listing 3.2: Replacement of `random.seed()` to record behaviour for replay

The downside of this way of handling nondeterminism, is that all nondeterministic functions will need to be replaced individually – it is not possible to automate the process for the entire PSL simply by storing and restoring the return values of all nondeterministic functions. This can be seen by the fact that the nondeterministic function in Listing 3.2, `random.seed()`, is not the function being manipulated to save and restore state. Furthermore, the earlier snapshot that gets activated for replay, has no knowledge of the data that was recorded in its future. Therefore, this method requires some type of storage mechanism, for both the recorded data and the information related to its replay.

Epdb contains much code for proxy objects to save such nondeterministic side effect data in its server process, but the code does not actually get used. Instead, epdb makes use of snapshots, as explained next.

3.4.2 Through snapshots

Using snapshots to implicitly save the program state, is a much simpler way to implement a replay mechanism. All nondeterministic instructions can be intercepted, for a snapshot to be made directly after their first execution, as in Listing 3.3.

```
1 | def seed () :
2 |     original_random.seed()
3 |     debugger.make_snapshot = True
```

Listing 3.3: Replacement of `random.seed()` to make a subsequent snapshot

When the user chooses to reverse, the snapshot from which to replay would then be directly after the closest previous nondeterministic instruction, as can be seen in Figure 3.4. The snapshot to activate might even appear at a later

point (though still before the intended destination), if snapshots are also made to place an upper bound on replay time. As all instructions to be replayed are purely deterministic, the problem of recording and correctly replaying the behaviour of nondeterministic instructions then falls away.

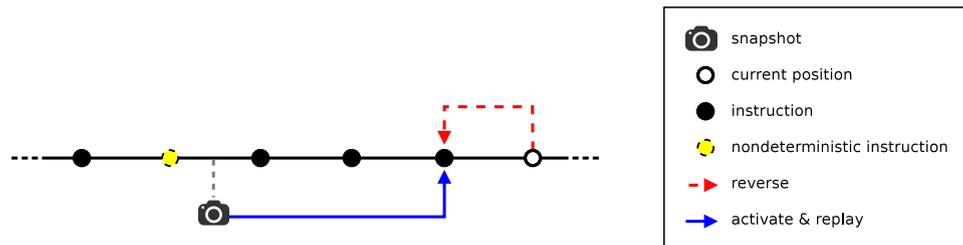


Figure 3.4: Snapshots after all nondeterministic instructions, facilitates deterministic replay

More snapshots are made when using this technique. This is generally not a problem, as forking is a lightweight operation, and large programs need not be catered for in the educational assistance scenario. It does become a problem when a great number of snapshots are made, leading to a type of fork bomb. To manage the number of snapshots, different strategies could be combined with the snapshot strategy, such as exponential checkpoint thinning [30, p. 306], or the recording strategy explained in [section 3.4.1](#). Recording could also prove to be useful when program state needs to be serialized, or when visually animating the execution of a section of code which includes a nondeterministic instruction. Future work might look at the best ways for these two techniques to work in parallel.

3.4.3 Intercept mechanism

A consequence of any replacement mechanism, is that the replacement objects would be different to the originals. This is demonstrated in [Listing 3.4](#) by using the replaced `random.seed()` function in `epdb` as an example. The user program might access the attributes of the replacement objects, as well as manipulate them in any number of unforeseen ways. The replacement objects therefore have to be carefully designed to be similar to the originals, as far as is possible in both appearance and functionality, while providing the augmented behaviour.

```
1 | import random
2 | print(random.seed.__doc__) # None
```

Listing 3.4: Replacement functions in epdb do not mimic the originals

In epdb, contrary to the explanation given by its author, “if a patch module provides a symbol, which does not exist in the original module” [63, p. 37], it will *not* be ignored, and *is* accessible by the program at `<module>.<symbol>`. The reason is that the replacement mechanism works by executing the replacement module in the namespace of the original module. This overwrites any symbols already found in the symbol table of the original module, which is how the replacement is made possible, but it also enters any symbols not yet found there. Although generally not a serious issue, care has to be taken in the replacement modules, so that none of the symbols of the original module are unintentionally replaced, leading to unexpected behaviour in the user program.

Another consequence of the epdb mechanism is a name conflict problem, which the Boothe bdb debugger also suffered from [30, p. 300]. The epdb mechanism replaces the module for the entire system, so that there is no longer a distinction between the module which the debugger uses, and the module which the user program uses. This leads to problems, as the augmented behaviour of the module is only meant for the user program, not the debugger. The debugger has to make use of many of the functions that need to be replaced, such as some built-in functions like `open`, and modules like `time`, from the PSL. When the the user program calls the functions of the `time` module, for instance, it has to trigger the creation of a snapshot by the debugger to manage the nondeterministic behaviour, as explained in section 3.4.2. However, when the debugger uses the `time` functions, which it does to keep track of how much time has elapsed since the previous snapshot was made, it should not trigger the creation of more snapshots. Epdb tries to circumvent the problem by checking the name of the file from which a call originates, and avoids making the snapshot if the call appears to come from an epdb file. This blacklist approach results in unexpected behaviour when a user file has the same name as an epdb file, and when extending the epdb system with more files. As epdb was meant to be extensible, especially the replacement mechanism, this approach is not ideal.

The ldb replacement mechanism does not suffer from the name conflict problem. It also addresses the similarity of replacement objects to the originals, which is discussed in more detail in [section 3.9.2](#).

3.5 External state

A program runs in its environment and interacts with it, changing it and being changed by it as it executes over a certain period of time, which means that the environment can be a part of a program in an indirect way. As control of time is required during programming, the environment has to be managed as well, if it affects or is affected by a program's execution.

The environment has an effect on a program through nondeterministic instructions. These have been dealt with in [section 3.4](#), through the recording and snapshot strategies.

A program can also have an effect on its environment. For example, a file might be created when a program is run. The intentional, non-transient effects that a software application produces, should remain when it is closed. However, while a program is being manufactured, its effects should usually not remain. For example, if a file is created when a program runs, and the file was not removed when the program ends, a new file will be created each time. That is not ideal, as the programmer typically runs the program many times during development, which would result in the creation of many unnecessary files. Although such automatic management of side effects is not provided by IDEs in general, it would be useful. In a reverse debugging environment where time is explicitly controlled, it is a requirement. When stepping backwards, the environment should be as it was before. For example, a user program might read from a file before finally deleting it. Should the file not be restored when the user reverses, the program will break when moving forward the second time, when it attempts to read from the file which no longer exists. Reverse debugging therefore requires that a program is always connected to its side effects, which is also one of the aims of live programming, as explained in [section 2.2.3](#). A single mechanism might therefore serve both.

There are many different types of resources that may be affected by a running program, such as files, databases, displays, or the standard output stream. Epdb defines a resource management system where each type of resource has

a corresponding manager [63, p. 6]. For example, each file is managed by a `FileResourceManager` instance. Each manager has a `save` and `restore` function, which can be called each time the state of the resource changes during the execution of a program, to have the manager save and restore the different states. The manager identifies each state with a Universally Unique Identifier (`UUID`). Replacement objects, similar to those discussed in [section 3.4.1](#), are used to intercept instructions that would have an effect on the environment. When the user program changes a resource during normal execution, the replacement object calls the `save` function of the manager directly thereafter, for the manager to save the new state of the resource. The manager returns the `UUID` that it has associated with the new state. The debugger then connects the `UUID` to the current `IC` for that specific resource, so that the state can be restored according to the position in the program.

When a user navigates through time, the resources only have to be restored once. The speed of restoration depends only on the number and size of resources to restore, so is generally fast. The debugger uses the `IC` to check for corresponding resource states that should be restored. It passes the `UUID`s which identify those states, to the manager instances, for the resources to be restored to those states.

In `epdb`, the actual resource states are saved in files by using the `shelve` module. When the user moves either forwards or backwards in time, a new main process is created when a snapshot is activated. During the redo and replay modes, instructions that affect the environment are intercepted so that they are not executed, and the new main process restores the resources right before stopping at the destination `IC`. It therefore needs to know what the resource states were at the destination `IC`, which was only determined in its future. The `IC` to `UUID` mappings are therefore stored on the server. Each timeline has its own mapping dictionary, so that resource states may differ between timelines. As `epdb` is a prototype system, it provides resource management only for the `print`, `input` and `open` built-in functions. The `open` function replacement returns a `FileProxy` instance, to replace the file-like object that the built-in `open` function normally returns. The `epdb FileProxy` class manages only the `read`, `write` and `close` functions of a file, and the creation of files are not undone.

`Ldb` uses the same framework for resource management as `epdb`, but to

minimise the overhead of the system, `ldb` does not store resource data on the server, and restores resources at a different time, which is discussed in [section 3.9.3](#).

3.6 Inter-process communication

An Inter-Process Communication (**IPC**) method is needed between processes, for sending data such as messages. Two-way communication of data might be required, so signals are not an adequate solution. Sockets and pipes (queues are implemented using pipes) are better candidates.

Sockets work with binary data, so messages have to be converted to bytes before being sent over a socket connection. They have to be converted using a specific serialization format or encoding scheme, so that the receiving end knows how to decode them again. In `epdb`, messages are only basic instructions, so can be constructed with alphanumeric characters and spaces. For example, the message “`runic 33`” might tell a snapshot to activate and have the new main process execute up to instruction number 33. The American Standard Code for Information Interchange (**ASCII**) character-encoding scheme is therefore used by `epdb`, which results in each encoded character using one byte.

As the data that is received over the socket connection is merely a stream of bytes that might represent multiple messages, a mechanism also has to be chosen to indicate the boundaries of single messages, in much the same way that words in written English are separated with a space. As `epdb` employs stream-oriented sockets, a 30-byte fixed-length message scheme is used to demarcate messages. As each character takes up one byte, it only allows for messages up to 30 characters long, which is an unnecessary limitation. It also results in overhead as messages are padded so that they are always 30 characters long. If stream-oriented sockets are required, variable length messages would be better.

The sending of variable length messages can be done by prefixing each message with its length, using a much smaller fixed number of bytes for the length data. The length data is not encoded using **ASCII**, but is merely the decimal number in binary format, otherwise a number such as 13 would require two bytes – one for each character – instead of one. If the length of the subsequent

message is always communicated using x bytes, variable-length messages up to $2^{8x}-1$ bytes long can be sent this way, with much less overhead. As an example, consider two messages being sent, “hi” and “exit”, and 1 byte being used for message lengths, which translates into a $2^{8x}-1$ or 255-character limit for messages. Each message is converted to bytes using the `encode()` function of Python string objects, with `ASCII` as the encoding scheme. The respective message lengths of 2 and 4 are then prepended to each message in binary format, and everything is sent over the socket connection. It arrives at the receiving end as: “0000001001101000011010010000010001100101011110000110100101110100” (colour used only for readability). The receiving end reads 1 byte, as that is the agreed-upon size of the field which contains the length of the subsequent message. The byte has the value 00000010, which is the number 2. It then reads the next 2 bytes and decodes it using `ASCII` to get the message “hi”. Again it reads 1 byte and 00000100 is the number 4. It then reads and decodes 4 bytes to get the message “exit”.

Ldb initially made use of anonymous, duplex pipes for `IPC`, with the variable message length scheme described above, but now uses lightweight Unix Domain Sockets (`UDS`) instead, as described in [section 3.9.4](#).

3.7 Back to the future

In `epdb`, timelines are possible execution paths through the program [63, p. 6]. The potential need for timelines is introduced when a user reverses past a *nondeterministic* instruction. When the user moves forward again, there is ambiguity about whether they would like to reproduce the behaviour of the previous execution of the nondeterministic instruction, or would prefer to have the command re-execute. The way `epdb` resolves this is by allowing for both of these options through timelines. After reversing, when moving forward again, the previous execution is reproduced in what is called ‘redo mode’ [63, p. 9]. A new timeline can be created for all commands after the current point to actually re-execute instead.

Note that this is different to replay of deterministic instructions, which happens behind-the-scenes, without the knowledge of the user. The starting point for replay is always the closest previous snapshot to the intended destination. If snapshots have been made after every nondeterministic instruc-

tion, there will be only deterministic instructions to replay, as the snapshot to activate will appear later than the previous nondeterministic instruction, as discussed in [section 3.4.2](#). Redo mode in epdb is when the user has *deliberately* reversed. Timelines are used to manage how the user moves forward again *past previously-executed nondeterministic instructions*. In other words, it deals with selecting between going back to the same future, or with creating a possibly-different future.

Snapshots are created when a program is first executed up to a certain point. If the user has reversed past a specific snapshot to a previous IC, and chosen to move forward again to an IC ahead of it, epdb will not re-execute the in-between instructions, but will activate this ‘future’ snapshot instead, as illustrated in [Figure 3.5](#). This is called *forward activation* [63, p. 30]. If there were nondeterministic instructions between the two ICs, they appear to have executed the same way as before, but in reality, they did not re-execute at all; a future snapshot has merely been activated.

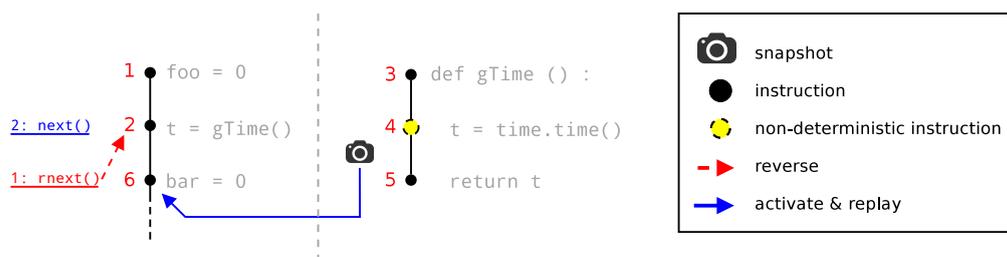


Figure 3.5: Forward activation. If the user performs an `rnext()` at IC 6 to go to IC 2, then performs a `next()`, the snapshot after IC 4 will be forward-activated for replay, and the nondeterministic instruction at IC 4 is not executed again

Although a clever mechanism for returning to the future, it means that earlier snapshots need to know about later snapshots, to be able to tell them to activate. However, as the earlier snapshots were created when first executing the program, by forking at those earlier points in time, snapshots have no knowledge of future snapshots. To work around this, an independent central process, called the ‘controller’ in epdb [63, p. 39], is required to manage communication between all snapshots, which in epdb is called the global snapshot strategy [63, p. 25]. All snapshot-related commands are then rerouted through the controller, and all timeline-related data is stored separately by a server process, which slows down the debugger. Although it might be useful

in a limited set of circumstances, multiple execution paths that *simultaneously* exist is not an indispensable feature in a reverse debugger that allows for easy time-travel; increasing the speed of the debugger is more important, so that the system remains usable when the live programming features are implemented, to reduce the temporal separation of cause and effect as much as possible. Furthermore, using forward activation to go back to the future has a negative effect on visualisation, as intermediate instructions can be skipped [63, p. 31], and generally are skipped when using the `next` or `continue` commands when `epdb` is in redo mode. `Ldb` therefore has no redo mode or timelines, and the implications and workarounds are discussed in [section 3.9.5](#).

3.8 Control of execution

As the mechanisms involved in reverse debugging have been dealt with, a way to allow for the controlled execution of a user program in Python, is considered next.

3.8.1 `bdb`

`Bdb` is the base debugger framework class in the [PSL](#). The way it provides a debugging framework is by registering itself with the interpreter as a trace hook, so that control passes to it before certain tracing events during the execution of the program. `Bdb` contains a number of abstract functions for implementation by derived classes. Based on the trace event, the derived class can then manage what should happen at that point, using its implementation of the abstract functions. If it returns control to the interpreter, the instruction is executed. The next trace event then takes place and the cycle is repeated.

The way in which `bdb` registers itself as a trace hook, is through the `settrace()` function of the [PSL `sys` module](#). This sets the `f_trace` attribute of the stack frame so that a `bdb` function is called before every trace event in that frame, as mentioned in [section 3.3](#). The explanation given for each trace event type in the documentation for the `sys` module⁵, is as follows:

| | |
|-------------|---|
| line | The interpreter is about to execute a new line of code or re-execute the condition of a loop. |
|-------------|---|

⁵Available at <http://docs.python.org/3.2/library/sys.html>

| | |
|--------------------|--|
| call | A function is called (or some other code block entered). |
| return | A function (or other code block) is about to return. |
| exception | An exception has occurred. |
| c_call | A C function is about to be called. |
| c_return | A C function has returned. |
| c_exception | A C function has raised an exception. |

More accurately, a `call` event is triggered whenever a new namespace is entered, and a `return` event when that namespace is again exited. This not only happens for function and nested function calls, but also when a module is imported, for class definitions, lambda expressions, list comprehensions, generators, generator expressions, and in `exec` and `eval` statements. More than one line event can also be triggered for a single line in the source code, such as when two function calls appear on the same line.

The events to which the `bdb` function responds, are `line`, `call`, `return` and `exception`. It checks whether it needs to stop for user interaction, based on certain stop conditions, then calls the four abstract `user_<event>` methods if it should, where `<event>` is one of the four trace event types. The `c_call`, `c_exception` and `c_return` events are ignored, as `bdb` serves to debug Python code, not C code.

`Bdb` stops for user interaction when certain conditions are met, such as when an exception is raised, or when the end of the program is reached. The following conditions also cause the program to stop, depending on the command that is has received from the derived class:

| | |
|-----------------|--|
| step | reaching the next instruction. |
| next | reaching the next instruction at the same (else a lesser) call stack height. |
| continue | reaching the next line which has a breakpoint set. |

The next instruction could appear in a different scope, if the statement was the last one in that scope, or if an exception was thrown.

3.8.2 pdb

`Pdb`, another `PSL` class, extends the `bdb` class and implements the abstract functions. By also extending the `cmd` module of the `PSL`, which is a framework for line-oriented command interpreters, `pdb` enables user interaction by

providing a prompt when a stop condition is met, in the terminal wherein the debugger was started. It makes the standard debugging commands, like `step`, `next` and `continue`, available to the user. By providing a mechanism for the evaluation of arbitrary Python code in the context of the top stack frame at any point, it allows for the exploration of specific program behaviour without changing program code, which is discussed in more detail in [section 3.9.5](#).

3.8.3 epdb

Epdb is built on pdb and therefore bdb, but implements extra functionality for reverse debugging. It provides a number of other commands, some of which have the form `r<command>`, as they are the reverse counterparts of forward-execution commands. For example, `rstep` is the counterpart of `step`.

Bookkeeping

For normal forward-execution, epdb uses the bdb stop conditions. However, a number of changes have to be made to implement reverse debugging.

To allow for reversing to a specific position in the program, a debugger has to count instructions, so that the position can be identified [56]. However, bdb only moves forward during execution, so does not have to keep track of the `IC`. Consequently, bdb is optimised to defer to the `user_<event>` functions only if it has to stop. To implement reverse functionality, the bdb method therefore has to be adapted. In epdb, the trace hook function of bdb is changed so that it does not check for stop conditions, but instead passes control to the extended debugger via the `user_<event>` functions, which manages the stop condition checks instead [63, pp. 18–19].

As a result of the change to the bdb method, the call stack height has to be recorded for use by the `next` command. The height is measured in frames, so the value of the counter that keeps track of it, is called the *frame count*, whereas it was called the call depth counter in the Boothe bdb debugger [30, p. 300]. The counter is incremented on `call` events and decremented on `return` events. During normal forward-execution, the frame count is noted when the `next` command is given, and the debugger stops again on the next `line` event, after an instruction is reached with either the same or a lesser frame count.

For reversing, there are other conditions that cause epdb to stop for user

interaction as well, such as when the start of the program is reached. The stop conditions for the `rnext` and `rcontinue` commands warrant special consideration.

For the stop condition of the `rnext` command, a stack is used in the form of a python list. The current `IC` is pushed to the list on every `call` event. On every `return` event, the corresponding `IC` is again popped from the list, and is entered into an `rnext_dict` dictionary with the current `IC + 1` used as the key. In this way, corresponding `IC` pairs are recorded, which describe a `call` and the `IC` that the `call` returns to, as illustrated in Figure 3.6. This information is used when an `rnext` command has been given, to determine the destination `IC`, which is consequently used to determine the closest previous snapshot to activate for replay.

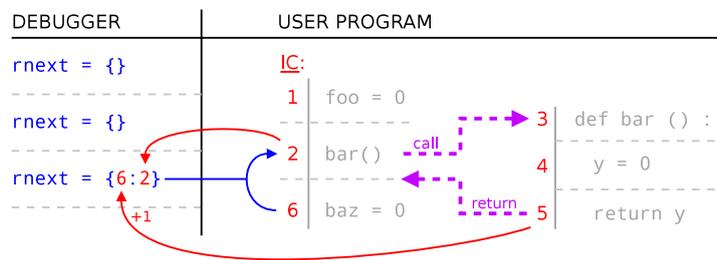


Figure 3.6: The mechanics of the `rnext` dictionary

The stop condition of the `rcontinue` command works in a similar way. Execution of source code is not concerned with line numbers. Earlier lines of code are often executed later, such as when a function is defined near the top of a file, but the code which calls it appears lower down. A line of code, such as a line in a loop structure, may also be visited many times during execution. `ICs`, not lines, are used to identify positions in a program for a reverse debugger too, so snapshots are connected to `ICs`, and the replay mechanism stops at specific `ICs` as well. However, the user works with *line numbers* in the `IDE`, not `ICs`. When the user sets a breakpoint, they specify the line on which that breakpoint should reside.

A mapping of lines to `ICs` is therefore required, so that breakpoints can be connected to `ICs`. `Epdb` stores this data in its `continue_dict` dictionary, which is populated during forward-execution. When the `rcontinue` command is given, the closest previous `IC` with a breakpoint is found, and its closest previous snapshot is activated for replay.

For long-running programs, the overhead of recording such mapping data, with respect to memory or storage space, is high. The cost with respect to execution time also becomes prohibitively high in a long-running program, as the mapping data is recorded at every instruction. The speed of forward execution is usually more important than the speed of reverse execution. The Boothe bdb debugger, which focused on long-running programs, therefore used a different strategy for `rcontinue`, which involved multiple re-executions of some of the intervals between snapshots [30, p. 306].

As epdb records the required information during normal execution, as described above, the stop conditions for the standard epdb reverse commands are relatively straight-forward:

| | |
|------------------|--|
| rstep | reaching a specific IC , namely the current IC - 1. |
| rnext | reaching a specific IC , namely the previous one with the same frame count. If it is not found in the <code>rnext_dict</code> dictionary when using the current IC as key, it is the current IC - 1. |
| rcontinue | reaching a specific IC , namely the closest previous IC where a breakpoint is found, as recorded in the <code>continue_dict</code> dictionary. If there are no previous breakpoints, IC 1 is activated. |

Breakpoints can be set at any point during the execution of a program. Ideally, the breakpoints should still exist when an earlier snapshot is activated, even if they were set at a later time. To facilitate this, breakpoints are stored on the server in epdb. The bdb breakpoint system is also adapted, so that interaction with breakpoints results in interaction with the server instead, through a number of proxy objects. As the debugger has to check for breakpoints at every **IC** as part of the stop condition tests, it slows down the debugger. Although multiple execution paths can exist simultaneously in epdb, the breakpoint mechanism does not currently give each timeline its own set of breakpoints [63, p. 47].

Snapshot activation

There are three ways in which a snapshot can be activated in epdb, described by three activation types, namely counting activation, frame count activation and continue activation [63, p. 42].

For the reverse commands, a specific **IC** is the intended destination, as described in the stop condition list above. The closest previous snapshot before (or at) that **IC** is found and activated, with directions to forward-execute up to the given **IC** if not already there. This is called counting activation, as the stop condition depends on the instruction *count*.

Epdb also allows the programmer to go back to the future after reversing, in its redo mode. If no snapshots exist between the current **IC** and the later destination **IC**, the program simply forward-executes using the normal stop conditions, as before. However, if one or more snapshots do exist between the current **IC** and the destination **IC**, epdb uses forward-activation with one of the activation types. The closest previous snapshot to the destination **IC** is found, and activated with directions that depend on the command.

For the `step` command, it simply activates the snapshot, which will be at the current **IC** + 1, so this is also counting activation.

Frame count activation is used when a `next` command is given. The **IC** to stop at is the next one with the same frame count (or lower), which is not necessarily the current **IC** + 1. Neither the snapshot to activate nor the current main process has knowledge about what **IC** to stop at, as that information is only available in their futures. The snapshot might also have a higher frame count than the destination **IC**. The snapshot is therefore activated and directed to execute up to the same frame count as that of the current main process. This is called frame count activation, illustrated in [Figure 3.7](#).

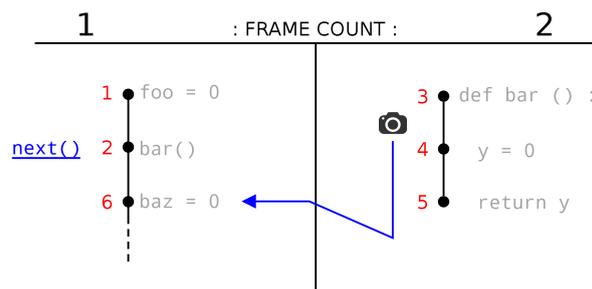


Figure 3.7: Frame count activation

Continue activation works in the same way, except that the destination **IC** depends on the next breakpoint. The closest previous snapshot to the destination **IC**, or the latest snapshot in the timeline if there is none, is activated, and directed to `continue`.

3.9 The bidirectional ldb system

Like epdb, ldb uses the bdb and pdb methodologies for implementing a debugger in Python. Ldb uses the Boothe bdb debugger approach for reverse functionality, and epdb partly served as the template for implementing it in Python. Ldb also uses the epdb approach to resource management and breakpoints, with some adjustments.

Some of the reasons why ldb is a complete rewrite and not simply an extension of epdb, have already been mentioned. Another reason is that, due to the complicated use of the proxy pattern, the multiprocess nature of the debugger, and the presence of much code that is never used, epdb results in unexpected behaviour that is difficult to resolve when trying to extend the system.

To resolve the difficulties that were discussed in [section 3.7](#), ldb was created without a redo mode, and without the concept of timelines. This facilitates future visualisation features, and results in a less complex overall design as well as an increase in speed through the removal of the controller process. Without a redo mode as in epdb, ldb does not require forward activation, so only counting activation is needed. Whenever a snapshot is activated, it consequently does not have a future to go back to, so the `next` and `continue` commands, which in the redo mode of epdb require frame count activation and continue activation respectively, still work in the same way. However, without a redo mode, nondeterministic instructions cannot be deterministically re-executed *when a program returns to its previously executed future*, which is discussed in [section 3.9.5](#). Without timelines, multiple execution paths that *simultaneously* exist are also not possible.

Some more reasons why ldb is not simply an extension of epdb, as well as the other ways in which ldb differs from bdb, pdb, the Boothe bdb debugger and epdb, are discussed in further detail below.

3.9.1 Breakpoint management

The breakpoint management of ldb differs from that in bdb and epdb. Bdb does have local (to the process) breakpoint management, but as epdb and ldb provide reverse functionality, breakpoints set in the future would be lost when the main process ends when reversing, if a breakpoint manager is used that is

only local to the process. In ldb, breakpoints would also be lost when the main debugger process is ended and a new one started, which might need to happen as part of the live programming mechanism. This means that a server for breakpoints is required, but is not sufficient unless it is persistent. Interaction with breakpoints also has to be fast, as breakpoints are used at every IC as part of the stop condition checks. The breakpoint management of epdb is therefore improved in ldb, to increase the speed of the system, by keeping both a local and server copy of the breakpoint dictionaries. When a breakpoint is created or changed, both the local and server copies are affected, but only the local copy is otherwise referred to. When a snapshot is activated, the local breakpoint manager receives an update from the server, before execution continues. This allows for both speed and persistent breakpoints even when reversing to an earlier point in time. The breakpoint management found in bdb has therefore been changed and moved to the `BreakpointManager` class, which manages `Breakpoint` instances. The breakpoint functions of bdb have been adapted to interact with the new breakpoint management class. The server in ldb is simply a separate process, created not through forking of the main debugger process as in epdb as none of the debugger functionality is required by the server, but as an independent process by using the `multiprocessing` package of the PSL. Communication between the server process and the main debugger process uses the same socket communication class as the rest of the ldb system, as detailed in [section 3.9.4](#).

Ldb requires a Graphical User Interface (GUI) for implementing higher levels of liveness, which is discussed in [section 4.1.2](#). It might appear possible to treat the GUI process as the server, as it is already persistent, and breakpoints will be manipulated through the GUI. However, when a snapshot is activated and it creates a new main process, the process has to request the most recent breakpoint dictionaries from the server before it continues, which requires an immediate response. The GUI might send other commands to the main debugger process in the interval between the activation of the snapshot by the current main process, and the new main process continuing to consume the commands from the GUI. This means that a race condition is introduced – when the new main process requests the breakpoint data from the server, other commands might have already been sent by the GUI, so that the data that the main process receives is not the breakpoint data, but a command.

This could be circumvented by using an extra socket only for breakpoint communication, but to allow for persistent breakpoints even on the command line when the GUI is not used, a dedicated process for the server is a better design.

3.9.2 Intercept mechanism

As described in [section 3.4.3](#), the system-wide modification of objects has to be avoided. The ldb object replacement mechanism for intercepting calls to nondeterministic instructions, which does not suffer from the same problems, is discussed below.

In Python, a distinction is made between an identifier and a name. An object in memory has a single identifier, which is a unique integer during the lifetime of the object, accessible via the builtin-in `id()` function. In CPython, the identifier of an object is its memory address. However, a single object might have many names that refer to it, as can be seen in [Listing 3.5](#).

```
1 | first = [1]
2 | second = first
3 | id (first)      # 30464768
4 | id (second)    # 30464768
5 | second.append(5)
6 | print (first)  # [1, 5]
```

Listing 3.5: Names are not identifiers

Having more than one name that is bound to the same object, has to be avoided, to prevent modifications to the original objects by the replacement mechanism. Fortunately, it is also possible for a single name to refer to different objects, if that name appears in two different *namespaces*, as in [Listing 3.6](#).

```
1 | var = 1
2 | def func () :
3 |     var = 5
4 |     print (var)      # 5
5 | func()
6 | print (var)         # 1
```

Listing 3.6: A name can refer to different objects, in different namespaces

To avoid the system-wide modification of objects and leave the originals intact, only the namespace of the user program should be changed, so that

a name in that namespace can refer to a different object than the original, creating an isolated sandbox in which the user program is executed. This is possible, as the `exec` function which is used by the debugger to execute the user code, can be provided with global and local dictionaries which form the namespace in which the user code is run. By using the same names, but changing what those names refer to in the namespace of the user code, different behaviour can be given to the user code without the original objects being changed.

A naive approach would be to replace the references directly, with replacement objects that are defined in the debugger. However, an object has access to the surrounding scope of the place where it is defined, and the replacements are defined in the debugger instead of the sandbox in which the user program is executed. The `__globals__` attribute of an object, which is a dictionary that defines its global namespace, cannot simply be replaced, as it is read-only. The items in the dictionary can be manipulated, but to remove all the references to debugger objects in the surrounding scope, would again require a blacklist approach. The references would have to be updated whenever the debugger code is changed.

To resolve the scope problem, replacement objects that are based on the original objects could be created, by using the standard types defined by the interpreter, such as `FunctionType` for functions. A different dictionary can then be provided, for the replacement objects to use as their global namespace. A simpler method is to define each replacement object in a limited scope, such as within a module containing no initialisation code. That module could then return the replacement, which is possible, as everything in Python is a first-class object.

The replacement objects would not automatically mimic the object they are replacing – attributes like the docstring would be different. To fix this for functions, a decorator can be used, such as the `wraps` decorator, provided by the `functools` module of the `PSL`. The attributes of the function will then be identical to those of the original function, as far as possible.

Modules that the user program imports, have to be replaced as well. For the debugger to accurately reflect the way that the user program would execute without the debugger, the replacement must only happen when a module is imported, not when the debugger is initialising. This can be accomplished

by assigning a custom importer to the `__import__` attribute of the global namespace of the user program. The importer will be called when any module is imported. If a module should not be replaced, the importer reverts to the built-in import mechanism. If a module should be replaced, like the `time` and `random` modules, the importer returns a replacement object, which can be either a class or a module.

Although a module cannot be extended like a class, a *class* can be used as a replacement for a module, if it extends the standard interpreter `ModuleType`. The `wraps` decorator of the `functools` module can again be used for each replacement function in the class. The magic `__getattr__` method of class instances, can be used to refer all references to symbols that have not been replaced, to the original module.

The other potential mechanism for providing a replacement object, is for the importer to return a different *module* when an import is attempted. Unfortunately, even when a module is imported by the custom importer, from the context of the user program, Python makes the module available to the entire system by placing it in the `sys.modules` dictionary, which overwrites the original module. This is prevented by first importing the replacement module under a different name, then changing its reference in `sys.modules` to the original name. When the replacement module is returned by the importer, the user program uses it instead of the original. To prevent any missing attributes, replacement modules should import into their own namespace the entire namespace of the module to be replaced, and overwrite specific symbols by defining new ones with the same name. The `wraps` decorator can again be used for the functions to mimic the functions they replace. A replacement for the `time` module of the `PSL`, is given in [Listing 3.7](#).

```
1 | from time import *
2 | import time as _originalModule
3 | from functools import wraps as _wraps
4 | from lib.storage import snapshotControl as _ssc
5 |
6 | @_wraps(_originalModule.time)
7 | def time () :
8 |     value = _originalModule.time()
9 |     _ssc.make_snapshot = True # make a snapshot at the next IC
10 |     return value
```

Listing 3.7: A module to replace the `time` module of the `PSL`

As most replacement modules will look almost identical if the snapshot strategy is used for dealing with nondeterminism, the process could be automated. It would aid in preventing the unintentional replacement of symbols, as discussed in [section 3.4.3](#).

3.9.3 External state

Ldb uses the `epdb` resource management framework, but allows for the creation of files to be undone as well. Although `ldb` does have a server for storing persistent breakpoints, `ldb` does not use the server to keep track of resources, to minimise overhead. Therefore, when a snapshot is activated and a new main process is created, it has no knowledge of the states of resources at the destination `IC`. The way `ldb` solves this problem without having to use the server, is by having the previous main process restore the states of the resources *before* activation of the snapshot. It restores the states as they were at the `IC of the snapshot`, instead of the destination `IC`. As a result of the snapshot strategy for handling nondeterministic effects, the instructions that will execute during replay, which lie between the snapshot and the destination `IC`, are guaranteed to be deterministic. It means that, if there are any instructions that change resources, they will change the resources in the same way as before.

There is one behaviour that is important to point out for all reverse debugging systems that make use of forked processes, as it results in bugs that are difficult to trace. Forked processes share the file offset value of open file descriptors. When one of the processes writes to a file, the file offset is changed for the other process as well, which can lead to an incorrect state for file re-

sources. When a snapshot is activated, such as when the user reverses, the file offset will not necessarily be what it was when the snapshot was created, but will be what it was after the other process had manipulated the file. When the resource manager writes to the file to restore it to its previous state, the content that has been written to the file since the creation of the snapshot, is not necessarily overwritten. This has been fixed in `ldb`.

3.9.4 Inter-process communication

`Ldb` initially used pipes for `IPC`, by using the `multiprocessing` package of the `PSL`. However, pipes appear to operate at about the same speed as `UDS`, and `UDS` facilitates future expansion, so the switch to `UDS` was made. `Ldb` uses the more lightweight `AF_UNIX` address family for faster local communication using unnamed socket pairs. `Ldb` also first used the variable message length scheme described in [section 3.6](#), but now uses `SOCK_SEQPACKET` sockets instead, which are similar to stream-oriented sockets but handle message boundaries automatically. For serialization, `ldb` uses the `PSL pickle` module, so messages can consist of more than the characters in the `ASCII` encoding scheme, and even Python objects can be directly sent over the socket connection. `Ldb` groups all of this socket communication functionality together in its higher-level `Socket_Communication` wrapper class, which is used as the base class for all communication in the `ldb` system.

Socket communication setup

`Ldb` does not use a controller process as in `epdb`, and so requires a different mechanism for all the processes to be able to communicate, if necessary. The way that the communication is set up, is by creating a pair of connected sockets. After forking, the parent process is the snapshot and the child process is the main process. Although it is a duplex connection, the snapshot blocks while listening for messages using one of the sockets, so it will be called the *listening* socket end. The child process, using the socket at the other end of the connection, only initialises communication with the snapshot when it needs to, so it will be called the *sending* socket end. As the connection has to be created before forking, each process initially has both sides of the connection, but then closes the side they do not use, which [Figure 3.8](#) illustrates.

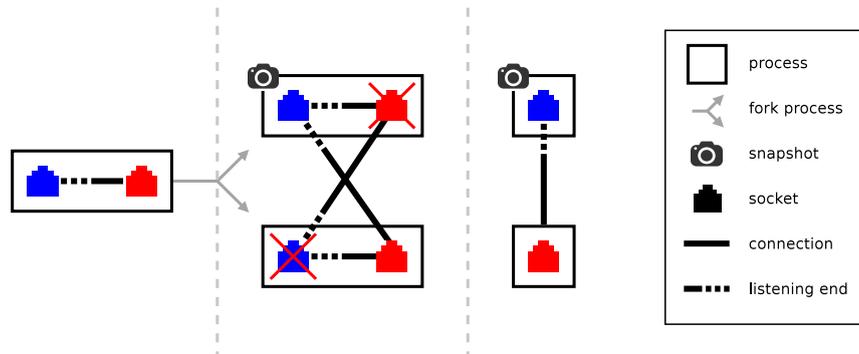


Figure 3.8: Setting up socket communication. A socket pair is created before forking. After forking, the redundant sockets are deleted so that only 1 socket remains in each process

Each socket gets wrapped in a higher-level `Snapshot` class, which acts as the communication contract between snapshots. The `Snapshotting` class manages all snapshot-related functionality. It keeps track of `Snapshot` instances in its `snapshots` dictionary, indexed by the `IC` that each snapshot is blocking at. When a snapshot is about to be created, a new `Snapshot` instance which contains the sending socket is placed in the `snapshots` dictionary at the current `IC`. The child process can then message the snapshot process using that entry. A later call to activate the snapshot made at instruction 4 might then look as follows: `snapshots[4].activate()`. This calls the `activate` method of the `Snapshot` instance at index 4 of the `snapshots` dictionary, which sends the “activate” message, using the sending socket. The snapshot process which was made earlier when the program was at instruction 4, is always listening on the listening socket, so immediately receives the message.

Each time a process forks, the child is a copy of the process, so still has the `snapshots` dictionary containing all the sending sockets connected to the receiving sockets of previous snapshots. In this way, each process can communicate with every earlier process. This is illustrated in [Figure 3.9](#).

Snapshot activation

Having no redo mode in ldb means that, when a past snapshot is activated when a user reverses, there is no more need for the existing future snapshots, and they should be instructed to shut down, to free up resources. However, having no controller in ldb means that a past snapshot has no knowledge of

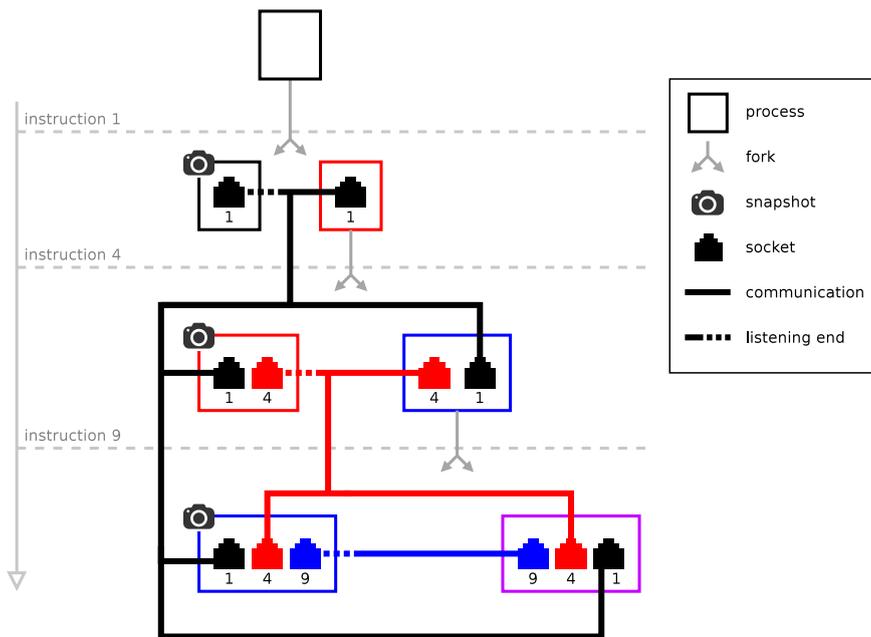


Figure 3.9: IPC after multiple snapshots have been made

future snapshots, and therefore cannot tell them to shut down. This means that as part of the process of activating an earlier snapshot, the main process has to manage the shutting down of all processes which come after that earlier snapshot, including itself. It sends a shut down message to all intermediate snapshots, and after sending an activate message to the snapshot, it uses the `exit` system call to shut itself down as well. When the earlier snapshot then activates, it forks to create a new process to take over as the main process. This procedure is illustrated in Figure 3.10.

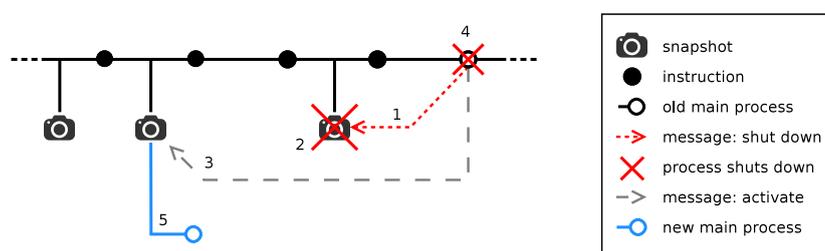


Figure 3.10: The steps involved in activating a previous snapshot

As processes have specific ways of operating on Unix-like systems, it is important that the child process is the one to continue as the main process, not the parent process. In ldb it ensures that every process has a maximum of

one direct child. It also guards against orphan processes that would be created were the parent to continue as the main process, but later shut down when an earlier child snapshot process is activated. [Figure 3.11](#) is a more accurate portrayal of snapshot creation in ldb.

Before any snapshot either shuts down or activates, it waits for its child process to shut down first, by using the `wait` system call. In [Figure 3.11](#), if process `p1` is activated, it waits for `c1` to shut down before it reaps it and continues, but `c1` in turn waits for `c2` to `exit` before it will shut down. When `c2` does `exit`, the exiting of processes cascades up, which ensures that no descendant processes exist before `p1` will continue, to guard against the formation of defunct or zombie processes.

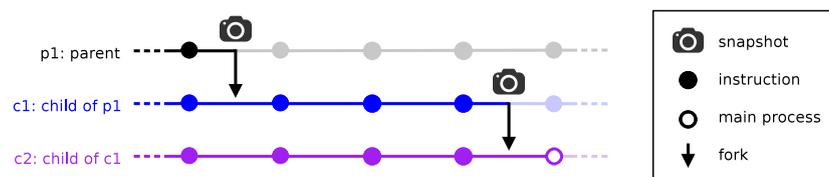


Figure 3.11: When forking, each parent process blocks as the snapshot, and its child continues

Process `p1` will then create a new child to take over as the main process, and again start blocking, waiting for commands from any descendants. This is illustrated in [Figure 3.12](#).

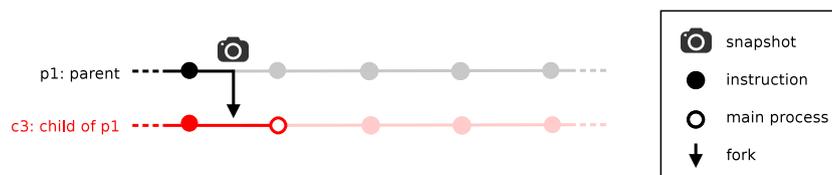


Figure 3.12: The state of processes after the `p1` process of [Figure 3.11](#) has been activated

3.9.5 Back to the future

As explained in [section 3.7](#), ldb has no redo mode, as found in epdb. Reverse debugging is usually used only to trace the cause of effects, so the programmer is mostly interested in moving backwards up the causal chain. If the user does

move forward again, having no redo mode simply means that the user, after having reversed *past a nondeterministic instruction*, might not automatically step forward in the exact same way. It will only be an issue if the user wants exactly the same behaviour as before. This will not often be the case, so should not be a problem in practice, especially in programming education. Changing code is what programming consists of, and a program is regularly re-executed as part of the development cycle anyway. So if deterministic replay of nondeterministic instructions does become important to the learner for some unforeseen reason, the behaviour of that specific instruction could be hard-coded while the user explores it. The speed by which that can be done and again undone, will also be much reduced by having a live system.

There is also a way to explore a specific behaviour of a nondeterministic instruction without changing the program code, should the programmer want to avoid the hard-coding of specific values as part of the development process. Commands can be given to the interpreter at any point where the execution of a program has been paused, through the prompt that the debugger provides. This functionality comes from `pdb`, as mentioned in [section 3.8.2](#). It allows the programmer to change the program state after the nondeterministic instruction has executed. The user could simply run a command which affects the program state in such a way that it is as if the nondeterministic instruction displayed a specific behaviour. For example, consider a program that uses the `time()` function in the `time` module of the `PSL`, to return the operating system time, before printing it:

```
1 | x = time.time()
2 | print(x)
```

Listing 3.8: Exploring nondeterministic behaviour without changing the code

The user might want to test the behaviour of the `print()` function when the `time()` function specifically returns a value of 1234567890, without hard-coding that value into the program code. To do this, the program can be paused directly after the `x = time.time()` nondeterministic instruction has executed, and the value of `x` can then be replaced in the running program by typing `!x = 1234567890` at the debugger prompt. When the user then steps forward, the `print` instruction is executed with that specific value for `x`, without a change to the program code.

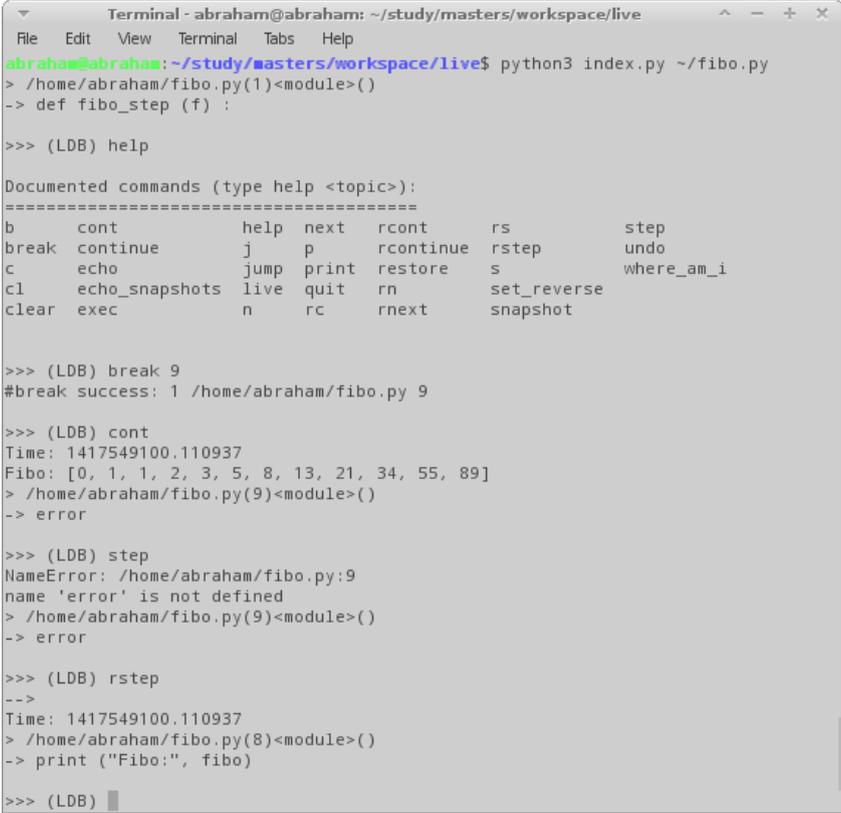
3.9.6 Conclusion

The reverse debugging functionality of ldb, whose design and implementation has been discussed in this chapter, was informed by the PSL bdb and pdb classes, the Boothe bdb debugger [30] and the epdb prototype [63]. Epdb could not be extended directly, due to the state of the code base – parts were no longer used, and the complex interactions between the proxy objects in different processes, resulted in unexpected behaviour that could not be resolved. Epdb also contained functionality which ldb did not require, namely its redo mode and timelines, with accompanying activation modes. Therefore, ldb was built from the ground up, using bdb, pdb, the Boothe bdb debugger and especially epdb as reference. A number of changes were made, not only to fix defects like orphan and zombie processes and the name conflict problem of the intercept mechanism in epdb, but to reduce the complexity of the code base, and to allow for future visualisation, which is where this project is headed. The speed of the system was also improved in a number of ways, to facilitate the integration of live programming features while still reducing the temporal gap as much as possible.

An example of what interaction with ldb is like when using the command line interface, is shown in Figure 3.13, where the program given in Listing 3.9 is being debugged.

```
1 | def fibo_step (f) :
2 |     return f.append(f[-2] + f[-1])
3 | fibo = [0, 1]
4 | import time
5 | print ("Time:", time.time())
6 | for i in range(10) :
7 |     fibo_step(fibo)
8 | print ("Fibo:", fibo)
9 | error
```

Listing 3.9: A typical program that a novice programmer might create



```
Terminal - abraham@abraham: ~/study/masters/workspace/live
File Edit View Terminal Tabs Help
abraham@abraham:~/study/masters/workspace/live$ python3 index.py ~/fibonacci.py
> /home/abraham/fibonacci.py(1)<module>()
-> def fibo_step (f) :

>>> (LDB) help

Documented commands (type help <topic>):
=====
b      cont          help  next  rcont  rs      step
break continue      j     p     rcontinue rstep  undo
c      echo           j     p     print  restore s      where_am_i
cl     echo_snapshots  live  quit  rn      set_reverse
clear  exec            n     rc    rnnext  snapshot

>>> (LDB) break 9
#break success: 1 /home/abraham/fibonacci.py 9

>>> (LDB) cont
Time: 1417549100.110937
Fibo: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
> /home/abraham/fibonacci.py(9)<module>()
-> error

>>> (LDB) step
NameError: /home/abraham/fibonacci.py:9
name 'error' is not defined
> /home/abraham/fibonacci.py(9)<module>()
-> error

>>> (LDB) rstep
-->
Time: 1417549100.110937
> /home/abraham/fibonacci.py(8)<module>()
-> print ("Fibo:", fibo)

>>> (LDB) █
```

Figure 3.13: The command line interface of lldb when tracing Listing 3.9

Chapter 4

Live programming

Higher levels of liveness are about changes to code, and minimising the amount of time that passes before the user receives feedback about the effect of the change, as explained in [section 2.2](#). How might the reverse debugger bring higher levels of liveness to a program that is under its control?

4.1 User interaction

Reverse debugging functionality only requires that a file be specified, for its execution to be managed. In `ldb`, the user is then able to control the debugger with commands at the interactive prompt. However, live programming begins with a change to the source code, which implies the use of some mechanism whereby the code can be changed. This mechanism can take many different forms, such as the visual object-manipulation mechanisms found in [VPLs](#), or structured editors which limit the allowed changes to the grammar of the programming language. It usually takes the form of an unstructured text editor.

4.1.1 Awareness of change

Whatever editor is chosen, when the code is changed, the debugger has to be made aware of it. A user might prefer to use a specific editor, which could be allowed if the debugger is able to watch for changes to files on the file system. This was the first strategy of `ldb`, by using the `inotify` monitoring service provided by the linux operating system. However, such a strategy requires that the files actually change on the file system, which only happens when the user saves a file. Higher levels of liveness specify that the user should

receive feedback about their changes *as they edit* the code, not only when they perform some action to request feedback.

Higher levels of liveness further imply that the debugger should not even wait until the user has finished editing a specific symbol, before calculating the effects. For example, the intention of the user might be to change the name of a variable from `var` to `long_var`, and the user does this by prepending each character of the string `long_` in turn. As the user is typing, it is impossible to know what the intention of the user is – the user might have the name `lovar` in mind instead. Only registering the change after a short interval of inactivity might save computational overhead, by preventing many intermediate changes from being propagated. However a fast typing speed of 240 characters per minute would still mean that the interval has to be longer than 0.25 seconds to prevent change propagation while the user is editing a single, multi-character symbol. That could already be too long an interval for the user to wait at the end of the edit, before the system even begins to propagate the change. Novice programmers are also likely to have a slower typing speed, not only as a result of having spent less time using a keyboard, but because they have to deliberately reason about each change, more than an experienced programmer would. Therefore, the better strategy seems to be to propagate all changes immediately, which requires that the debugger be made aware of every single change. Future usability studies are required to evaluate the best strategy for balancing the speed of feedback with the computational overhead – each user should perhaps be able to customise the behaviour of the system, in this regard.

4.1.2 An IDE

For the editor, or the **IDE** in which the editor is found, to notify the debugger of every change to code, it requires that the editor either saves the file on every code change, or that it notifies the debugger directly. Generally, editors do not save a file on every change, and do not have an event handler registration mechanism whereby another program can be notified of events. It is also an infeasible task to provide an extension for all editors or **IDEs** which users might want to use, so it was decided to implement a basic **IDE** for `ldb`. The focus of `ldb` is on education, and novice programmers have not spent much time in any **IDEs**, so requiring the use of a different editor is not of great concern.

It would provide a single interface to the novice programmer for controlling the debugger, changing the code, and seeing feedback about the execution of the program. The ldb GUI was created with the `tkinter` package of the PSL, which is a lightweight layer on top of the open-source and cross-platform Tcl/Tk GUI toolkit¹. It is shown in Figure 4.1.

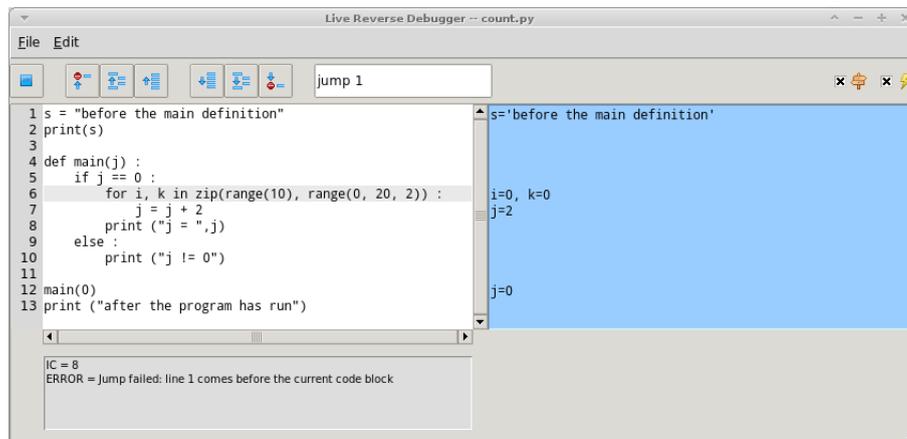


Figure 4.1: The GUI of ldb

Responsiveness

It is not possible to tell whether a statement in the user program is still being executed, or whether the program is hanging. For example, a blocking `receive` call on a socket might be faulty, or might just be waiting for data. The user interaction functionality should not become unresponsive as a result, which would happen if the same process was providing the user interaction functionality and executing the user program. This happens in `pdb`, as the `pdb` class extends the `bdb.Bdb` class to provide debugging functionality, but also extends the `cmd.Cmd` class to provide user interaction functionality. A process to interact with the user, which is separate from the process in which the user code is executed, is a better design for a system that allows for the execution of user code written in a general programming language. In ldb, the GUI is therefore run as a separate process.

Most interactive applications have a main loop which checks for user interaction events, and Tcl/Tk is no different. Event handler functions can be

¹Available at <http://www.tcl.tk>

registered, which the main loop will dispatch to for specified events, so that the GUI can react to those events. This allows for a GUI to be created that sends specific commands to the debugger when certain actions, such as the click of a button, are performed.

A communication mechanism is consequently required between the debugger and the GUI. An unnamed socket pair is created in the GUI, and when the GUI initialises the debugger as a completely separate process by using the `Process` function of the `multiprocessing` package, the listening socket is passed to the debugger. The socket wrapper class that is used for communication between the main debugger process and the snapshot processes is reused, to manage the encoding and decoding of data. The debugger, instead of waiting for the `cmd` module to provide commands, then waits for data from the GUI. Further improvements to this architecture are discussed in [section 5.4.1](#).

Execution of the user code by the debugger might take any amount of time. The GUI therefore should not block for a response from the debugger after giving it an execution command, as it would again result in an unresponsive program, and so nullify any gain from separating the GUI from the debugger process. In other words, an asynchronous communication mechanism is required. The debugger needs to be able to initiate communication with the GUI at the appropriate time, to provide information about the execution of the user program, for the GUI to process and display when it is ready to do so. To this end, a callback function is registered for the Tcl/Tk main loop to call periodically. The main loop carries on processing user interaction events until the specified amount of time has elapsed, which allows the user to stop the debugger even if it becomes unresponsive. The callback function checks the socket for any communication from the debugger, using a non-blocking mechanism. If there is information from the debugger, such as the line that the debugger has stopped at, it is processed before control returns to the main program loop. As all processing of information from the debugger results in little computational overhead, the GUI handles the processing itself. However, if it does become more computationally expensive in future as the `ldb` system is expanded, the GUI should create a new handler process and delegate the processing to it, so as not to hold up the main program loop. If data needs to be sent to the debugger in response, the socket could be passed to the new process so that it can respond to the debugger directly. The callback function

finally registers itself to be called again, after another interval of the specified amount of time has elapsed.

4.2 Changes to consider

Non-transient code is what forms the final computer program, as explained in [section 2.2.2](#). However, on the way to it, while a computer program is in the process of being created, the code is also transient – each change to the code results in a ‘new’ program. Once a GUI is in place to send data to the debugger, to notify the debugger of all changes to the code, and to receive back information about the execution of the user program, the fundamental question of live programming, asked in [section 2.2](#), must be considered – how can a change to code be propagated in a running program so as to minimise the amount of time spent waiting for the program to reflect it? A hot-swap, though useful, is not adequate when focusing on education, as explained in [section 2.2.3](#). Instead, the transformation has to be propagated in such a way that the state of the program would be as if the new program had executed.

4.2.1 Meaningful changes

Whether a change to the source code even results in a different program, can only be determined by parsing the source code text into a Concrete Syntax Tree (CST), and comparing that to the CST of the previous text, to understand the viewpoint of the compiler. However, as some elements that are found in a CST, such as comments, have no impact on the program, the ASTs should be compared instead. If the editor is not a structured editor that works with the AST directly, it can simply parse the text into an AST whenever it is changed, to determine whether a change to code results in a different program at all. If it does *not* result in a different program, the parsing effort is not wasted, as it prevents an attempt to unnecessarily propagate a change, which is an operation with higher overhead. If it *does* result in a different program, the effort is not wasted either, as it forms the first part of the compilation process anyway – the AST object is subsequently compiled to bytecode.

4.2.2 Effective changes

Each successive program that is structurally different, is a transformation of the previous program, but not all new programs would necessarily result in *effects* that are different to those of the running program. For example, changing the value of a variable which is never used, would still result in the same effects when run. In Python, an edit to the code might lead to effects in three different areas – the internal program state, the external environment, and the call stack. All three areas could be affected by changing the external environment or a symbol in the code, as can be seen in [Listing 4.1](#).

```
1 | if x == 3 or fp.read() == "external data" :  
2 |     fp.write("external effect")  
3 |     x = 9  
4 |     func()
```

Listing 4.1: A change can affect the program both directly and indirectly

When the code is edited, all three these areas should be updated for their state to be a reflection of the execution of the new program, and no longer the execution of the previous program. A complete analysis on every edit, to determine all the ways that a change could affect those areas, is not the way to approach the problem for a general, dynamic programming language – some information is available for static program analysis, but much information is only available at runtime. For example, consider the user program in [Listing 4.2](#). To what degree would the rest of the program be affected by a change to the variable `a`, on line 1? It is impossible to tell, as it depends entirely on the input that the user provides on line 2, at runtime.

```
1 | a = 5  
2 | exec(input())  
3 | x = 2 * a
```

Listing 4.2: In a dynamic language, much information is only available at runtime

The only available approach is therefore to consider *any* change to the code that does result in a new **AST**, as a change that should be propagated, although it does mean that a change is processed even when the change does not have an effect. From here on, when a change to code is referred to, it means a change which results in a new **AST**.

4.3 Change propagation strategies

Whether the effects of a change can be propagated in a running program, depends on the programming language and its implementation. In Python, some of the limitations that prevented the creation of snapshots for the reverse debugging functionality, discussed in [section 3.3.1](#), also prevent the direct manipulation of the running program – the call stack cannot be adequately manipulated or completely replaced, so changes to the execution path cannot be propagated. As an analysis of the effects of a change is also of limited use, due to the information only being available at runtime, a new program will have to be *executed* for the change to be propagated.

4.3.1 Execute the entire new program

The simplest approach is to get rid of the need for the programmer to *ask* the computer to go through the stop-run-navigate cycle, by automatically executing the new program from the start whenever the code is changed, and executing until it ends. It can be seen in the `live-py-plugin`², and in the binary search example of Victor [15]. An earlier version of `ldb` did use this strategy, as the focus of `ldb` is on education, and the overhead in terms of execution time when small programs are continually executed from the start, is not necessarily a major problem. When the code is changed, the initial bootstrapping need not be redone; the debugger simply resets the bookkeeping info, and executes the new code until it ends. Control then returns to the debugger, to allow for reversing. `Ldb` also improved this strategy by automatically restoring the resources to their initial states before the new program is executed, by using the resource management functionality of the reverse debugger.

Although this strategy is straight-forward and relatively easy to implement, its major disadvantage is that it limits the size of the programs. Even small programs in terms of lines of code, can result in computation that takes a relatively long time, so that executing all instructions from the beginning for each cycle, results in much overhead in terms of time. However, this strategy does save the cumulative time that it takes users to manually manage resources and go through the stop-run-navigate cycle, so further ways that it might be

²Available at <http://donkirkby.github.io/live-py-plugin/>

improved, are considered below.

4.3.2 Execute up to the same position

As a debugger allows the user to navigate through time, either only forwards in a traditional debugger or both forwards and backwards in a reverse debugger, the program will often be at a specific position in the execution which is not the end of the program. This point will be called the Point Of Execution (**POE**).

When the user changes the code, the program could return to the point in the execution where the code was *changed*. However, the line being changed would not be the line that the user would like to return to, in most cases. For example, one of the most common use cases would be when the user has written a function and wants to observe its behaviour when given different inputs. The user changes the inputs, and expects to see how the function handles it, which means that the program should not return to the point where the code was changed, as that point comes before the execution of the function. It would be better if the user could specify that they want the execution to stop directly after the function, as they want its feedback as fast as possible and are not yet interested in the execution behaviour up to the end of the program. The user should be able to do this by moving the **POE** to directly after the function, for the program to execute up to that point whenever the code is changed.

This introduces a problem – how should the **POE** be defined for the debugger to be able to return to it?

Line number, or IC?

A different execution run from the start, even for the same program, might lead to a different execution path being followed as a result of nondeterministic instructions. The point where the user was previously at, might therefore not be reached again. For example, in [Listing 4.3](#), the program might not return to line 2, 3 or 5 if that were the **POE**. If the **POE** were line 6, the program would be able to return to it, with the value of `x` possibly being different than before. It can also be seen that the **IC** would be different at line 6, depending on which branch of the conditional statement had executed, which means that the **POE** cannot be defined in terms of the **IC**. For now, it is instead defined by the line that the program was previously at.

```
1 | if 5 <= random.randrange(10) :
2 |     x = 3
3 |     y = 0
4 | else :
5 |     x = 7
6 | print(x)
```

Listing 4.3: The same point of execution cannot necessarily be reached

Nondeterminism

Deterministic replay of the nondeterministic instructions seems to be the solution, as the program will then constantly follow the same execution path, and the program state is retained. The user might prefer this at times, but it still does not solve the problem of returning to the same **POE**, as it only assists in retaining the program state and execution path *up to the point where the old and new programs diverge*. Instructions should not be re-executed deterministically from that point on, as it is impossible to know what impact the changes would have on the program, and the replay of previous behaviour is an assumption that the changes have had no impact. For example, if line 1 in **Listing 4.4** has just been entered, it would not make sense to replay the previous behaviour of line 2, when returning to line 3.

```
1 | del sock    # just inserted
2 | data = sock.recv()
3 | print (data)
```

Listing 4.4: Nondeterministic instructions should not be replayed after the point of change

Deterministic replay would also be a disadvantage, as only a single behaviour of every nondeterministic instruction would be observed throughout the construction of the program. It would be better for new data from the external environment to be used instead. Regular re-execution of the nondeterministic instructions would lead to many scenarios and execution paths being explored, which is similar to how the traditional development cycle works. That would aid in preventing the “even a broken clock is right twice a day” scenario, where the program appears to be correct, but it has only happened

by chance. For example, consider the program in [Listing 4.5](#). The `square` function should return the square of a given number, but instead of returning the result of the calculation, the function has a bug which causes it to always return the number 4. The program uses the `random` module to nondeterministically provide a number as input, which provides the number 2 during the first execution, by chance. The assert on line 5 does not raise an error, even though the `square` function is faulty. If, in this way, only the first behaviour shown by each nondeterministic instruction is ever used, a program would more often appear to be correct when it is not, as deterministic behaviour is not the true nature of a nondeterministic instruction. However, if the program were to be regularly ‘massaged’ through the re-execution of nondeterministic instructions, bugs would surface more often, leading to better programs.

```
1 | def square (v) :  
2 |     res = v * v  
3 |     return 4  
4 | x = random.randrange(-5, 5)  
5 | assert square(x) == x * x
```

Listing 4.5: A program can display the correct behaviour by chance

Novice programmers would not yet make use of other means – such as unit tests as part of a build process – of exposing their programs to multiple executions. The re-execution of nondeterministic instructions is therefore to be preferred over replay. Just as in the traditional development cycle, the user should explicitly hard-code specific behaviours if they want to retain them, otherwise make use of the mechanism discussed in [section 3.9.5](#), so that nondeterministic instructions do not, by default, behave deterministically.

The question of how to return to the same **POE** has therefore not yet been answered.

Execution history

A compromise might seem like a better solution, where the previous **POE** is stopped at only if it is reached, but where the program continues to execute if it is not reached. The **POE** might therefore be controlled by the user through a breakpoint. However, as the user is able to freely navigate through the program execution without setting breakpoints, and as breakpoints might already exist

at earlier points in the program, it would be better to temporarily ignore breakpoints and simply return the user to a preferred point that they can specify. It would consequently be better for the **POE** to take the form of a special breakpoint that is the only one not ignored.

However, breakpoints operate by stopping when a certain line is reached. If the **POE** were to work in the same way, it could lead to confusion, as the same line can be reached at entirely different times in the execution, such as in a loop, and by following a different execution path, such as for function calls. If the execution were to miss that line as a result of the code change, but stop at the same line at a later point, it would incorrectly communicate to the user that they are still at the same point in the execution of the program. When navigating to a breakpoint, *explicit navigation* is required, and the user therefore sees and understands that movement has taken place, which inherently includes the possibility of different execution paths leading to the same position in the source code. If, however, the user were to *remain at the same position* in the source code, it would appear to them that they had remained at the same point in the program execution as well. As that is not necessarily the case, it could confuse them.

Consider the example in [Listing 4.6](#). The first branch of the conditional statement on line 4, causes the program to reach line 2. After a code change, the program would again stop at line 2 on account of the **POE** being defined only by the line number, but the other branch of the conditional statement could have executed. The user would incorrectly believe that the execution path and program state are still the same, as it would appear to the user that they had not moved. However, a different execution path was followed in the two executions, which has led to different internal and external effects.

```
1 | def save(val) :  
2 |     print('Same POE?')  
3 |  
4 | if 0.5 < random.random() :  
5 |     x = fp.write('data')  
6 |     save(3)  
7 | else :  
8 |     save(7)
```

Listing 4.6: The same line number can be reached in different ways

Usability studies would be required to gauge the amount of confusion that this would or would not cause, to determine whether this is an acceptable way to define the point to return to. Until that can be done, this way of defining the **POE** is not used in `ldb`, as I believe it would cause much confusion for novice programmers. Consequently, the search continues for an acceptable way to define the **POE**, so that it can be returned to after a code change.

The only way to uniquely identify a point in the program execution, is to define it by its entire execution history. This can be captured through the sequence of line numbers visited to reach that point, as well as the locations of the files that those line numbers refer to, for enabling the use of multiple files as part of a single program. The **POE** can be defined as that sequence, at the point where the program has stopped. Returning to the same **POE** is therefore only possible if the line number history up to that point is not changed by a change to the code or by re-executing. It means that the same **POE** cannot be returned to when lines are deleted or inserted, or when the execution path is affected, either directly through the change itself, or indirectly if there are any nondeterministic instructions as part of the execution history. This scenario severely limits the times when the same **POE** can be returned to.

Managing user expectation

Instead of returning to the *exact* same point in terms of execution history, the program could attempt to return to a point that the user would *expect* or wish to be returned to. Extensive usability studies would have to be done to determine what the most intuitive stopping point would be in each situation for novice programmers, and how those points could be described to the system by using line numbers or **ICs**, or by other means. Until then, the best that can be done if the user is not to manually go through the stop-run-navigate cycle, is either to execute the entire program again as before, or to return the user to the point that has been *edited*, which is discussed next.

4.3.3 Execute up to the point of edit

Until the usability studies can be done, it would be better for the program to return to where the code was edited, as that seems to be the point where the old and new programs diverge. The point will be referred to as the *change*

point. The user could then decide how to deliberately move forward from there, by continuing to a breakpoint after a function call, for instance. As the user works with lines of code in the editor, the change point could simply be defined by the specific line number in the source code where the change has taken place. However, even the change point proves difficult to capture and return to.

It may be argued that the new program only has to be executed up to the change point, if that line was reached in the previous execution of the old program. If that were true, the line number to **IC** mapping that is recorded by the debugger during forward-execution for use by the `rcontinue` command, as discussed in [section 3.8.3](#), could be used to determine whether or not a line had been reached. If the line had not been reached, the new program would not have to be executed yet. If the line had been reached, it might have been reached multiple times, so the debugger would have to use the first time that the line was reached, as the stopping point for the new execution. However, the change point is not necessarily the first place where the two programs diverge, depending on the presence of nondeterministic instructions, which are likely to display different behaviours when the new program executes. It means that the change point could trigger the execution of the new program, and yet not be reached again. The point to stop at therefore becomes a subjective matter again, unless all nondeterministic instructions up to the change point are replayed. It is not as problematic to replay the behaviour of only these nondeterministic instructions, as they have not been affected by the change in the code, which only comes later. Although it does not avoid the broken clock scenario described in [section 4.3.2](#) for the nondeterministic instructions up to the change point, constantly replaying their first behaviour would be the only way of ensuring that the change point is definitely reached again.

4.3.4 Avoid unnecessary execution

As returning to the change point requires the deterministic replay of nondeterministic instructions, the behaviour of the two programs would be identical up to that point, meaning that the external program state would be affected in an identical manner as well. This leads to the idea that it might be possible to *retain* the effects of the old program, instead of trying to recreate it, if the debugger can be directed to use the new program for all subsequent execution.

Future change point

First consider the simpler situation where the change point is the **POE**, or is in the future relative to the **POE**. In Python, the debugger cannot simply be directed to continue tracing different code. A workaround that seems like it could work, without yet considering how to navigate back to the **POE** in the new program, is to do an **exec** of the new program, so that it becomes the program that is traced, but even this cannot be done directly. The debugger functionality can be divided into two parts – an *inside* and an *outside*, so to speak, which refer to their use, relative to the user program. The outside part is used *before and after* the execution of the user program, and contains functionality for setting up and tearing down both the debugger and the sandbox in which the user program will execute. As part of setting up, the debugger also puts a callback trace function in place, through `sys.settrace`. When the debugger executes the user program by using **exec**, the trace function is called for every trace event. The trace function calls a number of debugger functions, which can be grouped with the other parts of the debugger that they make use of, as the inside part, due to their use *during* the execution of the user program. These are the functions that stop for user interaction, which means that the only place for execution of the user program to stop, is *inside* the trace function. The problem with executing the new program in that situation, is that the new program does not trigger trace events, which the debugger requires. A minimal example which demonstrates this, is given in [Listing 4.7](#). When this program is run, it can be seen that only the outside **exec** triggers trace events.

```

1 | def trace (frame, event, arg) :
2 |     if event == 'line' :
3 |         print ("--- line {} ---".format(frame.f_lineno))
4 |         exec("print(3)" + "\n" + "print(4)") # lines 3 and 4
5 |     return trace
6 |
7 | import sys
8 | sys.settrace(trace)
9 | exec("print(1)" + "\n" + "print(2)") # lines 1 and 2

```

Listing 4.7: Code executed within a trace function, does not trigger trace events

The `call_tracing` function in the `sys` module is an indirect way of running

the new `exec` call while inside the old, but as the old tracing state is saved to be restored after the new `exec` returns, it continues to use resources. As a new `exec` would be done each time the code was changed, it would quickly lead to all available resources being consumed.

To `exec` a new program without unnecessarily holding on to resources, the current `exec` first has to be broken out of. This can be done by raising and catching a particular exception. The debugger then does an `exec` of the new program in the old context, which was retained through a handle on the context dictionaries provided to the initial `exec`. The new `exec` takes the debugger back to the first line, but it can immediately return to the `POE` before any instructions are executed, by writing to the `f_lineno` field of the stack frame, as explained in [section 3.3.1](#). In this way, the call stack and internal and external states are retained, so the instructions before the `POE` do not have to be re-executed, and nondeterministic behaviours do not have to be replayed. It leads to a significant improvement in speed, proportional to the execution time of the instructions before the `POE`. For example, the `sleep` call on line 1 of [Listing 4.8](#), which could represent 10 seconds of computational overhead, is not executed again when a change is made to line 3 when the `POE` is line 2, unlike when the new program is executed from the start.

```
1 | time.sleep(10)
2 | print('the POE')
3 | print('this is changed')
```

Listing 4.8: Previous instructions do not have to be re-executed

Past change point

The situation where the change point is in the past, relative to the `POE`, can now be considered. The debugger has to return to the earlier change point, which the reverse capabilities of `ldb` allow it to do. The closest previous snapshot to the change point is activated. It breaks out of the `exec`, and its own position is jumped back to immediately after the `exec` of the new program. The snapshot then has to forward-execute to reach the change point. The reverse debugging strategy of making a snapshot after every nondeterministic instruction, means that the instructions that remain to be executed between the positions of the snapshot and the change point, are guaranteed to be

deterministic, meaning that the change point will definitely be reached again, as shown in [Figure 4.2](#).

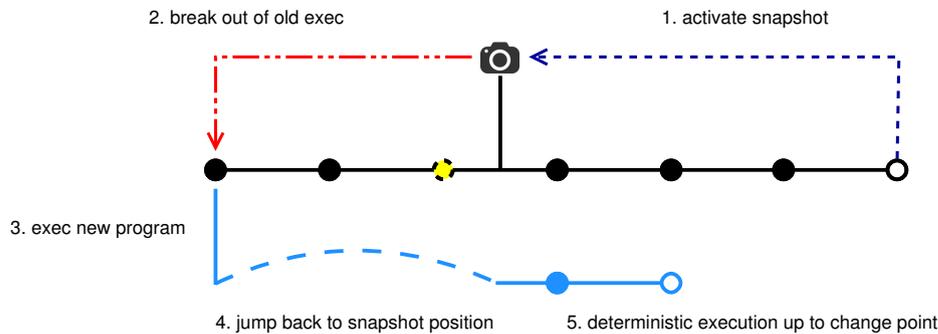


Figure 4.2: Returning to a change point that is before the current **POE**

Complications

Although this strategy allows the debugger to quickly return to the change point, without replay and without much re-execution, it cannot always do so. Having to write to the `f_lineno` field, limits the snapshot positions that can be directly jumped back to. Some compound statements such as `for` loops, `while` loops, `try` suites and `except` clauses, require special initialisation and clean-up of information on entry and/or exit, such as the value of an iterator or point to return to when leaving the statement. This is already taken care of in ldb by activating and retaining the state of a snapshot, which is a program that has already done the required initialisation. For example, even though a `finally` clause, after being directly jumped into it in pdb, causes a segmentation fault when it is exited, it does not do so in ldb. Python, however, never allows most blocks to be jumped into directly, and cannot be told of the ldb workaround so that it would know that it is safe to allow the jump. The use of `f_lineno` would therefore be only slightly less problematic here than in [section 3.3.1](#), so either only snapshots in the top-level namespace have to be activated, or the snapshot positions are not the points that should be jumped back to. If only top-level snapshots can be activated, the *closest* previous snapshot to a position is not necessarily the one to activate, which would mean that nondeterministic instructions are no longer taken care of automatically, and that re-execution

of more instructions would be required. Until a way can be found to remove the `f_lineno` jump restrictions, the best approach would be to try to use the closest previous snapshot, but jump to positions in the top-level namespace.

A further complication is the fact that the true point of change is not necessarily the line itself; even *earlier* lines can be affected. If a line appears in an entity with its own namespace, such as a function or class, and the entity is defined in another namespace at an earlier point, the earlier point will need to be returned to, to register the change in the parent namespace. If this is not done, the earlier version will continue to be used. Furthermore, entity definitions are often registered long before they are used, such as when a function is defined at the top of a file, but only called at a later point. In [Listing 4.9](#), line 1 will need to be returned to when line 2 is changed, even though the function is only used on line 5.

```
1 | def func() :  
2 |     print('this is changed')  
3 | a = 1  
4 | b = 2  
5 | func()
```

Listing 4.9: The point where a code block is defined, has to be returned to

As the parent has undergone a change, its parent has to be updated in turn, all the way back to the top-level namespace. It would have spared much execution if only the lines where the entities are defined, could be re-executed to register each change in turn, and the change point be jumped to immediately thereafter, but it is not possible either due to the limitations of writing to `f_lineno` – a code block cannot be jumped into. The point to return to therefore has to be the closest previous point to where the outermost parent entity is defined, in the top-level namespace.

How can the **IDE** tell to which entity a line belongs, so that the correct line can be returned to? Having first tried, unsuccessfully, to record the information during execution, `ldb` now makes use of the **AST**, which already has to be constructed, as explained in [section 4.2.1](#). For each entity, the lines on which it is defined and ends, are captured before execution of the new program, by using the **AST**. When a line is changed, this information is used to check whether the line is part of an entity. If it is, the line where the outermost parent entity is defined, is used as the line to return to instead. [Listing 4.10](#)

shows an example of the line, given as a comment, that a change on that line would return the debugger to.

```

1 | a = 1                                # 1
2 | def func () :                       # 2
3 |     x = 1                          # 2
4 |     class InnerClass :             # 2
5 |         def innerClassFunc (self) : # 2
6 |             z = 1                 # 2
7 | func()                             # 7

```

Listing 4.10: The place to return to when lines are changed, is where the outermost parent entity is defined in the top-level namespace

The debugger can use the same approach for the other compound statements, such as `for` and `try`. The line where the statement is defined, becomes the destination line for the debugger to return to, when a line is changed that belongs to the statement. The snapshot to activate when reversing, would be the closest previous snapshot to the first time that the destination line was executed. However, the forward jump is to the snapshot position, as was shown in [Figure 4.2](#). As the snapshot position cannot necessarily be jumped back to, and as the destination line is guaranteed to be a point that *can* be jumped to in the top-level namespace, the jump should only occur once the destination has been reached, not when the snapshot is activated. This allows the closest previous snapshot to be used, so that nondeterministic instructions are automatically taken care of and the least amount of re-execution takes place, while the debugger still traces the new program. [Figure 4.3](#) illustrates this strategy.

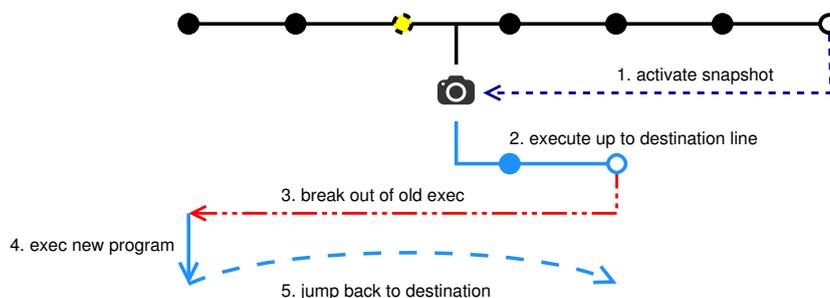


Figure 4.3: An improved way of returning to an earlier change point

As the point that the debugger returns to is different to both the POE and the change point, it makes it even more important that the position in the source code where the execution is at, at any moment, has to be clearly indicated to the user, which is done by highlighting the relevant line in the GUI, as was shown in Figure 4.1.

Chapter 5

Conclusions and Future Work

5.1 Overview

5.1.1 Motivation

This study was motivated by the desire to place humans, instead of computers, at the centre of **HCI**. As a human interacts most naturally when using language and visual faculties in parallel, these should be utilised simultaneously during programming. However, existing programming interfaces have a strong bias towards either language-based or visual-based interaction – no satisfactory combination exists. The need to maximise interaction and customisability when combining visuals and text, suggests the need for a visualisation framework to enable programmers to express their own mental models, as well as to receive feedback from the computer by way of that same channel. Such a system would be most valuable to novice programmers, as interaction and clear communication are essential to the learning process. All interaction is however fundamentally governed by time. The considerable temporal separation of cause and effect during programming, hinders the formation of robust mental models. This separation would still exist when visuals are incorporated. A solid foundation for integrating visuals consequently first requires the minimisation of delays in time, as well as the maximisation of the control of time. To this end, this study investigated the feasibility of combining reverse debugging and live programming in a general programming language, which had not been done before.

5.1.2 Approach used

My approach was to create a reverse debugger which was sufficiently lightweight to facilitate live programming features, and then to augment the debugger with higher levels of liveness. Reverse debugging was introduced in [section 2.1](#). The strategies of previous reverse debugging systems were considered. The Boothe bdb debugger approach was found to be the optimal strategy, with epdb the most recent implementation thereof. Live programming was introduced in [section 2.2](#). Consideration of other live programming prototypes showed that most approaches focus on design at the language level. It also showed that the fundamental concern of live programming remains unanswered – how a change to code can be propagated in a running program in a way that minimises the amount of time spent waiting for the program to reflect it.

5.1.3 Reverse debugger

The motivation for choosing Python as the programming language was explained. A way to implement a fast reverse mechanism in Python was considered, and the snapshot-and-replay strategy was examined. It was shown that nondeterministic instructions would need to be managed to ensure that the program truly reverses to a previous state. The replacement of objects at runtime with objects that behave differently, allowed for nondeterministic behaviour to be managed through either a recording or snapshot strategy. It was explained that the two strategies are also able to function in parallel, but that the investigation of the optimal combination would be left as future work.

A reverse debugger was created, called ldb. Reasons for creating ldb from the ground up were provided, and details were supplied about where the ldb approach differed from that of bdb, pdb and epdb. The shortfalls of previous approaches to the object replacement mechanism were covered, and an improved mechanism was presented. Refinements to the resource and breakpoint management systems were detailed. The requirements related to [IPC](#) were discussed, and the suggested improvements were implemented.

The implications were discussed of navigating back to the future after having reversed past a nondeterministic instruction. The epdb solution to the problem was explained, and identified as an area that should be approached differently, for a number of reasons. Most importantly, forward activation hin-

dered future visualisation functionality. The complex program architecture needed to be simplified, and gains in speed were also required to facilitate the integration of live programming using the limited computational resources typically available to novice programmers. The `ldb` approach solved these issues. The implications were that nondeterministic instructions could not be deterministically re-executed when deliberately returning to the future, and that multiple execution paths could not simultaneously exist. These features are however not indispensable in a reverse debugger, especially one with higher levels of liveness which would allow fast exploration of execution paths. Ways to accomplish the return to the same future in `ldb`, were covered.

5.1.4 Higher levels of liveness

Strategies were discussed for making the debugger aware of changes to the code of an executing program, and it was suggested that future usability studies be done to evaluate the optimal way for user expectation and computational overhead to be balanced.

The necessity of an `IDE` was presented, and an `IDE` was created which allows the programmer to load and execute programs, control the debugger, change the program code, and receive feedback about the program execution. User interaction problems were addressed, and the `GUI` was created so as to remain responsive even when the execution of the user program is not. Possible future improvements were considered.

The code changes to propagate in a live program were contemplated. It was shown that all changes which result in a new `AST` would have to be propagated, and the optimal way to do this was examined. It was determined that the new program would have to be executed at least in part. A number of approaches to automatic execution were considered.

The simplest approach of automatically executing the entire new program was shown to be a feasible yet non-optimal solution. It removes the need for the programmer to request feedback from the computer, resulting in a higher level of liveness. It was found that reverse debugging and live programming work well together, in that the resource management system which was required for the reverse debugger could be used to restore resources before the new program is executed.

An improvement was proposed where the new program would only exe-

cute up to the point, called the **POE**, where the previous program execution was stopped. The way to define the **POE** was considered. Nondeterministic instructions which result in different execution paths when the program is executed from the start, were explained to be a major obstacle when trying to return to that point. Deterministic replay of all nondeterministic instructions up to the **POE** was contemplated, but it was demonstrated that nondeterministic instructions could not be replayed after the point where the old and new programs diverge. An explanation of why constant replay would be a disadvantage, and why re-execution is to be preferred, was also provided. Defining the stopping point in a way similar to how breakpoints work, was consequently examined. It was argued that it would likely lead to confusion, especially for novice programmers. Usability studies were suggested to ascertain the amount of confusion that this would cause, to determine whether the point to return to should be defined in this way. Usability studies were also suggested to determine whether novice programmers might expect the program to return to specific positions, and if so, what those points would be in different situations.

Consideration was given to returning the program to the change point instead. The deterministic replay of all nondeterministic behaviour up to the change point was again found to be necessary to be able to reach the change point. As the old and new program would therefore be identical up to that point, a significant improvement could be considered – not re-executing that part of the program at all. It was indicated that this strategy would depend on being able to direct the debugger to trace the new program. As Python does not allow this directly, workarounds were investigated. Simply doing an `exec` of the new program was shown not to be possible, and it was explained that the `call_tracing` workaround would quickly consume all available resources. A strategy which broke out of the current `exec` but which retained the state of the program, did an `exec` of the new program, and jumped back to the previous position before any execution took place, was suggested. For future change points, relative to the **POE**, it appeared to hold much promise, in that it avoids re-execution of all instructions before the change point, so allows for a direct return to the **POE**. For change points in the past, relative to the **POE**, the reverse debugging functionality was ideal for allowing the program to first return to the change point before continuing with the strategy, and then to deterministically execute up to the change point. The use of `f_lineno`

to jump back to the same position after the new `exec`, limits this strategy however. Although `ldb` already takes care of the reasons for the interpreter disallowing a jump at certain times, there is no way to inform the interpreter of that, so a workaround has to be implemented. In the meantime, the strategy of only jumping to allowed positions in the top-level namespace was investigated. Lines which change object definitions were considered, and the consequent need to return to the line where the outermost object definition was registered in the top-level namespace. The restrictions on `f_lineno` again prevented only the definition lines from being re-executed before jumping back to the previous point. Different methods were investigated for identifying the definition lines in the top-level namespace, and using the `AST` was found to be the optimal approach. The break-exec-jump strategy was also slightly altered to allow for the continued use of the previous snapshot, to automatically take care of nondeterministic instructions.

5.1.5 Summary

A reverse debugger with higher levels of liveness was investigated and implemented by using the general, multi-paradigm programming language Python, which is widely used for computer programming education. Improvements to previous reverse debugger approaches were discussed and implemented, especially to facilitate the integration of live programming. The need for a reverse debugger to manage side effects was also explained and addressed. The approach used for navigation through time allows for future visualisation functionality.

The fundamental, language-independent concerns of live programming were discussed and addressed. Change propagation was investigated, and the need to consider its effects on the internal program state, the external environment and the call stack was explained. The optimal way to propagate changes in a way that covers these areas, by using the least amount of execution, was pursued. The approach of executing the entire new program was shown to be feasible, yet non-optimal. The concepts surrounding the `POE` were explained, as well as the difficulties that would be encountered when returning the new program to such a point. An optimal strategy was proposed of returning to the change point by making use of the reverse debugging functionality, and only executing from there to the `POE`. It avoids all unnecessary execution.

5.2 Objectives achieved

The aim of this study was to investigate the feasibility of combining reverse debugging with live programming. The goal was to develop such a system if the combination should prove feasible, or to investigate and explain the fundamental obstacles should it prove infeasible. This was accomplished. The combination proved to be feasible, and the ldb debugger was developed with reverse capabilities and level 4 liveness. It supports a more natural, lively conversation between programmer and computer. The mechanisms used for implementing a reverse debugger also facilitated higher levels of liveness. Although the combination proved feasible, the fundamental principles and difficulties were nevertheless explained, to aid future research.

Ldb addresses the temporal gap present in the standard development cycle by furnishing the programmer with more control of time and by reducing the temporal separation between code changes and seeing its effects. Control of time is provided by allowing the programmer to navigate both forwards and backwards during execution, with the state of the program and environment being updated accordingly. The user is able to explore different paths of execution, and observe the corresponding interaction of the program with the environment. Changes to the code of the user program that the debugger is tracing also results in an immediate response. Parts of the program do not have to be re-executed when the code change is propagated. This reduces the temporal separation of cause and effect, which aids the formation of clear mental models. Ldb should therefore be used by novice programmers especially. Ldb also creates a platform which would enhance the usefulness of visuals, and so establishes a solid foundation for pursuing visual ways of thinking in computer programming.

5.3 Shortcomings

Although the ldb system was created and the fundamental difficulties addressed, there remains work to be done.

5.3.1 Replacement objects

Replacement objects have not been provided for much nondeterministic behaviour. The suggestion was made in [section 3.9.2](#) that the process could be automated to some degree, but this has not yet been attempted.

5.3.2 Control of time

Ldb contains the mechanisms that would allow the user to control the speed of execution, as well as to repeatedly loop over any section of the program, for the programmer to exercise complete control of time and so gain the maximum amount of insight, especially once visuals are incorporated. However, the interface functionality has not yet been fleshed out to allow the user to do this.

5.3.3 Resource management

Although the ldb approach to restoring the states of resources does not make use of the server process, so as to increase the speed of the system to reduce the temporal gap, it might not always result in an increase in speed. The ldb approach means that when a snapshot is activated, the resources are restored to their states at the **IC** of the snapshot. The remaining instructions that are re-executed to reach the destination **IC** are all deterministic, and therefore change the resources in the same way as before. If the server process were utilised for storing resource data, as in epdb, the resources could be restored to their states at the destination **IC** instead, and the remaining deterministic instructions would be intercepted so that they do not change the resources during replay. Although the chance of it happening is slight, it is possible for re-execution to take much longer than communication with a server process, when large resources are managed. Testing would have to be done to determine which strategy is faster on average, given different situations, and whether the gains in speed by switching to the other strategy would be significant. The ldb approach was used, however, as novice programmers generally do not work with large resources.

5.3.4 Change propagation

While investigating change propagation strategies in [section 4.3](#), for bringing higher levels of liveness to the debugger, a number of difficulties were encountered. Although workarounds were provided for most of these, two issues have not been overcome.

Stopping point after code change

The first issue that has not been overcome, is the determination of the optimal stopping point for the execution of the new program. The presence of any nondeterministic behaviour after the change point would mean that the point to return to cannot be captured or described in a way that allows for use across different runs. Capturing that point by its line number, as with breakpoints, might lead to confusion – only a usability study would be able to determine if it would. The optimal points to stop at in different situations could perhaps be determined subjectively, which a usability study would again throw light on. As usability studies have not been done as part of this study, the optimal point at which to stop the execution of the new program, has not been determined. Consequently, `ldb` currently returns the user to the change point instead, for the user to manually navigate from there, which is not ideal.

Tracing a new program

The second issue that is not adequately addressed in the current state of the `ldb` system, is the inability in Python to instruct the debugger to continue tracing a different program from any specific point onwards. Although the reverse debugging functionality proved ideal for returning to the change point, the debugger cannot be instructed to continue executing the new program from the change point on. The `ldb` workaround of breaking out of the current `exec` but retaining the program state, doing an `exec` of the new program, and immediately jumping back to the previous position, is limited by the restrictions that the Python interpreter places on `f_lineno`. `Ldb` therefore currently returns the program to an earlier point in the top-level namespace that can be jumped to, which is not ideal.

There is another consequence to the interpreter's `f_lineno` restrictions for which no workaround has yet been implemented. Even though snapshots al-

low navigation backwards in time, not necessarily being able to jump forwards to their positions for live programming, also has an impact on their use by the reverse debugger. Although the jump limitation could be circumvented in [section 4.3.4](#) by first executing to a position in the top-level namespace, which becomes the position that is jumped to, the same cannot happen when a snapshot is activated when reversing. Once a new program is being traced by the debugger, after following the procedure that is illustrated in [Figure 4.3](#), new snapshots that are created also trace the new program, but activating an earlier snapshot reverts the debugger to the program that was being traced when the snapshot was made. The snapshot would have to break out of the `exec`, `exec` the new program, and jump back to its own position before continuing. However, the same problem is encountered as earlier, where the snapshot position cannot necessarily be jumped back to due to the limitations of `f_lineno`. This is an unsolved problem. The only solution, so far, seems to be to activate the closest previous snapshot that is at a position that *can* be jumped to, and execute to the intended destination from there. This would have an impact both on re-execution time, and on nondeterministic instructions, which are automatically handled by the snapshot strategy, as explained in [section 3.4.2](#). The recording strategy, introduced in [section 3.4.1](#), might prove useful here, though not ideal. If a different way, that does not make use of a jump, cannot be found of instructing the debugger to continue tracing a different program from a specific point, it would be necessary to patch the Python interpreter.

As the Python programming language is already used widely in computer programming education, I had hoped that patching the interpreter would not be necessary, so that this system would be easier to roll out, and consequently lead to faster adoption. However, patching of the interpreter appears to be necessary, to minimise the amount of re-execution that needs to take place, and so truly minimise the temporal separation between code changes and seeing its effects. The number of changes to make to the interpreter would be minimal. The only change would be to remove the `f_lineno` jump restrictions for registration of the new program with the debugger, as `ldb` already takes care of the reasons for these restrictions, through forward *execution* from an earlier snapshot. It would remove the issues with continuing execution of the new program from a change point that is inside compound statements, and the issues with not being able to jump to all snapshots that are still tracing

the old program, so that they can also continue with execution of the new program.

Once the best position to stop execution at can be determined through a usability study, and once the debugger either allows tracing of a different program from any point or no longer restricts jump positions for the optimal ldb workaround to be implemented, the states of the program and the environment will be able to fully reflect the new code with the minimal amount of execution.

5.4 Future work

Some noteworthy future improvements are discussed in this section, while smaller existing features and experiments, such as an `undo` command which novice programmers would find useful, are documented by and in the ldb code itself.

5.4.1 Architecture

As the main debugger process currently manages the execution of the user program as well as the setting up and tearing down of the surrounding framework and the breakpoint server, when the user program hangs and is stopped, the framework and server are stopped as well. It would be better to divide the parts of the debugger, so that the *inside* and *outside* parts, described in [section 4.3.4](#), are separate processes, which would communicate over `UDS` as well. The outside part would do the setting up, tearing down, interaction with the `GUI` and management of the different processes. The inside part would manage only the execution of the user program, and would then be the main process as defined in [section 3.3.2](#), from which snapshots are made. It would execute and report back to the outside part, then block, waiting for instructions. The outside part would block for interaction from the `GUI`, unless the main process is busy executing, when it would not block but periodically check for commands from the `GUI` as well as check on the status of the main process. Should the user then reverse in the `GUI` when the user program seems to be hanging, the command can be sent to the outside process, which would simply terminate the main process and activate the previous snapshot. In this way,

even when the user program is unresponsive, reversing would still be possible, which is functionality that might often be required by novice programmers.

5.4.2 Platform independence

Ldb is currently limited to Unix-like operating systems, due to both the direct use of the `fork` command, and the use of `UDS`. A greater degree of platform independence could be gained by switching to the `multiprocessing` package of the `PSL` for snapshot creation in general, and by using the slower internet protocol suite for `IPC` when running on operating systems, such as Windows, that do not support `UDS`, and to enable remote debugging. The `multiprocessing` package is already used to start the debugger and breakpoint server as separate processes. It requires that the process which starts a new process, has to be the one to `join` it, but as this is already done in the single-child model of ldb as detailed in [section 3.9.4](#), it is not a problem. The ldb `GUI` also already makes use of the cross-platform Tcl/Tk toolkit.

5.4.3 Minimising execution

As explained in [section 4.3.4](#), in the current ldb implementation, the point to return to after a change to the code of an entity, is the closest previous point in the top-level namespace to where that entity is defined. It might be possible to improve this in a number of ways. A promising strategy, which would allow the debugger to return to a point closer to the first place where the entity is *used*, instead of where it is defined, is to jump to the line where it is defined, execute the line, then jump to the closest previous position in the top-level namespace to where it is first used.

A further improvement could be to update its definition in the top-level namespace *directly*, instead of by executing the line of its definition again, by means of a hot-swap before the first time that it is used. The code of that entity in the new program can be isolated, compiled and executed in a separate namespace, from where the code object of the new entity can be extracted and used to replace the code object of the old entity. An example of how this could work is shown in [Listing 5.1](#).

An entity has access to its surrounding namespaces when it is defined. Testing showed that the access is affected when using this strategy, even though

only the `__code__` attribute and not the `__closure__` attribute is changed. As the `__closure__` attribute is read-only, more work would need to be done to determine whether the scope of the new entity can be made to be identical to the previous scope. Tracing of the new entity also appears to be affected.

```

1 | # EXTRACT NEW OBJECT CODE
2 | objectName, endLine = CodeInfo.objectDefinedAt(line)
3 | lines = newCode.splitlines(True)
4 | newObjectCode = "".join(lines[(line-1):endLine])
5 | # COMPILE AND EXEC IT
6 | compiled = compile(newObjectCode, '<string>', 'exec')
7 | namespace = {}
8 | exec(compiled, namespace)
9 | # REPLACE OLD OBJECT WITH NEW
10 | user_context[objectName].__code__ = namespace[objectName].__code__

```

Listing 5.1: It might be possible to replace objects directly

5.4.4 Usability studies

Usability studies would answer a number of questions related to the live programming part of ldb:

- Could computational overhead be reduced by only registering a code change after a short interval of inactivity, or would that make the speed of feedback too slow and lead to frustration?
- As it would need to be approximated, what would be the most intuitive point, to novice programmers, for a program to return to after a code change?
- To what degree, if any, would novice programmers be confused if that point was defined only by a line number in a file?

5.5 Summary

This study was introduced in [chapter 1](#), and the motivation behind it was expanded in [chapter 2](#), where reverse debugging and live programming were introduced. The investigation of how they might be combined, was explained to be the main goal of this study. Consideration was given to the approaches of other reverse debugging and live programming systems, and some neces-

sary clarifications were provided. It was found that, although they could work well together, no combination of reverse debugging and live programming for a general programming language had been attempted before. The design and implementation of a robust reverse debugger, informed by the top reverse debugging approaches so far, was detailed in [chapter 3](#). It was constructed in such a way that it allowed for higher levels of liveness and future visualisation. The strategies for bringing higher levels of liveness to the reverse debugger were discussed in [chapter 4](#), and the implementation of the optimal strategy resulted in the live reverse debugger, `ldb`. The results of the study were discussed in [chapter 5](#). It showed how the mechanisms used for the reverse debugger also proved useful for implementing the live programming features, and that the objectives were achieved. Improvements and future possibilities were also explored. Although work remains to be done, `ldb` already forms a solid foundation on which to combine visuals with language, for use in computer programming education.

List of References

- [1] (1997). Peter Norvig. Available at: <http://norvig.com/bio.html>. Retrieved: 20 February 2015. 21
- [2] (2003). Sebastian Thrun. Available at: <http://robots.stanford.edu>. Retrieved: 20 February 2015. 21
- [3] (2004). TOPLAP: Terrestrial organisation for the promotion of live audiovisual programming. Available at: <http://toplap.org>. Retrieved: 20 February 2015. 16
- [4] (2006). Teach yourself programming in ten years. Available at: <http://norvig.com/21-days.html>. Retrieved: 20 February 2015. 21
- [5] (2008). TotalView. Available at: <http://www.roguewave.com/products-services/totalview>. Retrieved: 20 February 2015. 10
- [6] (2009). Debugging with GDB. Available at: <http://www.sourceware.org/gdb/onlinedocs/gdb.html>. Retrieved: 20 February 2015. 12, 21
- [7] (2009). Process record and replay. Available at: <http://sourceware.org/gdb/wiki/ProcessRecord>. Retrieved: 20 February 2015. 10
- [8] (2009). Reverse debugging with GDB. Available at: <http://sourceware.org/gdb/wiki/ReverseDebug>. Retrieved: 20 February 2015. 12
- [9] (2011). Chronon. Available at: <http://chrononsystems.com>. Retrieved: 20 February 2015. 10
- [10] (2012). Alarming development: An IDE is not enough. Available at: <http://alarmingdevelopment.org/?p=680>. Retrieved: 20 February 2015. 18

- [11] (2012). CS101 – Intro to computer science: Build a search engine & a social network. Available at: <http://www.udacity.com/course/cs101>. Retrieved: 20 February 2015. 21
- [12] (2012). CS212 – Design of computer programs: Programming principles. Available at: <http://www.udacity.com/course/cs212>. Retrieved: 20 February 2015. 21
- [13] (2012). CS373 – Artificial intelligence for robotics: Programming a robotic car. Available at: <http://www.udacity.com/course/cs373>. Retrieved: 20 February 2015. 21
- [14] (2012). Gorgeous Unreal Engine 4 brings direct programming, indirect lighting. Available at: <http://arstechnica.com/gaming/2012/06/gorgeous-unreal-engine-4-brings-direct-programming-indirect-lighting/>. Retrieved: 20 February 2015. 17
- [15] (2012). Inventing on principle. Available at: <http://worrydream.com/#!/InventingOnPrinciple>. Retrieved: 20 February 2015. 10, 17, 18, 61
- [16] (2012). Learnable programming: Designing a programming system for understanding programs. Available at: <http://worrydream.com/LearnableProgramming/>. Retrieved: 20 February 2015. 18
- [17] (2012). The Stanford education experiment could change higher learning forever. Available at: http://www.wired.com/2012/03/ff_aiclass/all/. Retrieved: 20 February 2015. 21
- [18] (2012). UndoDB. Available at: <http://undo-software.com>. Retrieved: 20 February 2015. 10, 12, 22
- [19] (2012). XboxViewTV: Unreal Engine 4 features walkthrough. Available at: <http://www.youtube.com/watch?v=VitLyrynBgU>. Retrieved: 20 February 2015. 17
- [20] (2013). Drawing dynamic visualizations. Available at: <http://worrydream.com/DrawingDynamicVisualizationsTalk>. Retrieved: 20 February 2015. 8
- [21] (2013). Epic Games: Unreal Engine 4. Available at: <http://www.unrealengine.com>. Retrieved: 20 February 2015. 17

- [22] (2013). Hacking meets clubbing with the ‘algorave’. Available at: <http://www.wired.co.uk/magazine/archive/2013/09/play/algorave>. Retrieved: 20 February 2015. 15
- [23] (2013). Transient and ephemeral code. Available at: <http://yaxu.org/transient-and-ephemeral-code/>. Retrieved: 20 February 2015. 15, 16
- [24] (2013). University of Cambridge study: Failure to adopt reverse debugging costs global economy \$41 billion annually. Available at: <http://www.roguewave.com/company/news/2013/university-of-cambridge-reverse-debugging-study>. Retrieved: 20 February 2015. 11
- [25] Acar, U.A. (2009). Self-adjusting computation. In: *Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pp. 1–6. ACM Press, New York, New York, USA. Available at: <http://www.umut-acar.org/publications/pepm2009.pdf?attredirects=0>. 18
- [26] ACM/IEEE-CS Joint Task Force on Computing Curricula (2013). Computer Science Curricula 2013. Tech. Rep., ACM Press and IEEE Computer Society Press. Available at: <http://www.acm.org/education/CS2013-final-report.pdf>. 21
- [27] Archer, Jr., J.E., Conway, R. and Schneider, F.B. (1981). User recovery and reversal in interactive systems. Tech. Rep., Cornell University, Ithaca, New York, USA. Available at: <http://ecommons.library.cornell.edu/bitstream/1813/6316/1/81-476.pdf>. 11, 22
- [28] Balchin, W.G.V. and Coleman, A.M. (1966). Graphicacy should be the fourth ace in the pack. *Cartographica: The International Journal for Geographic Information and Geovisualization*, vol. 3, no. 1, pp. 23–28. Available at: <http://utpjournals.metapress.com/openurl.asp?genre=article&id=doi:10.3138/C7Q0-MM01-6161-7315>. 1
- [29] Balzer, R.M. (1969). EXDAMS - extendable debugging and monitoring system. In: *Proceedings of the 1969 AFIPS Spring Joint Computer Conference (SJCC)*, pp. 567–580. ACM Press, New York, New York, USA.

- Available at: http://www.rand.org/content/dam/rand/pubs/research_memoranda/2009/RM5772.pdf. 10
- [30] Boothe, B. (2000). Efficient algorithms for bidirectional debugging. In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pp. 299–310. ACM Press, New York, New York, USA. Available at: <http://suif.stanford.edu/~courses/cs343-s04/papers/boothe.pdf>. 10, 12, 20, 29, 30, 38, 40, 53
- [31] Bruner, J.S. (1964). The course of cognitive growth. *American Psychologist*, vol. 19, no. 1, pp. 1–15. Available at: http://bienser.umanizales.edu.co/contenidos/lic_ingles/desarrollo_cognitivo/criterios_conceptuales/recursos_estudio/pdf/The%20course%20of%20cognitive%20growth_Bruner.pdf. 1, 3, 7
- [32] Burckhardt, S., Fahndrich, M., de Halleux, P., Kato, J., McDirmid, S., Moskal, M. and Tillmann, N. (2013). It’s alive! Continuous feedback in UI programming. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 95–104. ACM Press, New York, New York, USA. Available at: <http://research.microsoft.com/pubs/189242/pldi097-burckhardt.pdf>. 8, 13, 18
- [33] Collins, N., McLean, A., Rohrhuber, J. and Ward, A. (2003). Live coding in laptop performance. *Organised Sound*, vol. 8, no. 03, pp. 321–330. Available at: http://yaxu.org/writing/laptop_performance.pdf. 15
- [34] Dijkstra, E.W. (1982). *Selected writings on computing: A personal perspective*. Springer-Verlag New York Incorporated, New York, New York, USA. Available at: <http://www.cs.utexas.edu/users/EWD/ewd04xx/EWD498.PDF>. 2
- [35] Edwards, J. (2005). Subtext: Uncovering the simplicity of programming. In: *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, vol. 40, pp. 505–518. ACM Press, New York, New York, USA. Available at: <http://www.subtext-lang.org/OOPSLA05.pdf>. 14
- [36] Eisenstadt, M. (1997). My hairiest bug war stories. *Communications of the ACM*, vol. 40, no. 4, pp. 30–37. Available at: <http://web.media.mit.edu/~lieber/Lieberary/Softviz/CACM-Debugging/Hairiest.html>. 4, 8, 9

- [37] Engblom, J. (2012). A review of reverse debugging. In: *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference (S4D)*, pp. 28–33. ECSI, Belmont, Isère, France. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.338.3420&rep=rep1&type=pdf>. 12
- [38] Feldman, S.I. and Brown, C.B. (1988). Igor: A system for program debugging via reversible execution. In: *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD)*, pp. 112–123. ACM Press, New York, New York, USA. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.2238&rep=rep1&type=pdf>. 11, 22
- [39] Feynman, R.P. (1988). “*What do you care what other people think?*”: *Further adventures of a curious character*. WW Norton & Company. 2
- [40] Fischer, A. (2013). Introducing Circa: A dataflow-based language for live coding. In: *Proceedings of the 1st International Workshop on LIVE Programming (LIVE)*, pp. 5–8. IEEE Press, San Francisco, California, USA. Available at: <http://liveprogramming.github.io/2013/papers/circa.pdf>. 18
- [41] Fry, B.J. (2004). *Computational information design*. Ph.D. thesis, Massachusetts Institute of Technology. Available at: <http://benfry.com/phd/dissertation-110323c.pdf>. 3
- [42] Gabriel, R.P. (1996). *Patterns of software: Tales from the software community*. Oxford University Press, New York, New York, USA. Available at: <http://www.dreamsongs.com/Files/PatternsOfSoftware.pdf>. 3
- [43] Gardner, H. (1983). *Frames of mind: The theory of multiple intelligences*. Basic Books, New York, New York, USA. 1
- [44] Hadamard, J. (1945). *The mathematician’s mind: The psychology of invention in the mathematical field*, vol. 107. Princeton University Press, Princeton, New Jersey, USA. 2
- [45] Hundhausen, C.D. and Brown, J.L. (2007). What you see is what you code: A “live” algorithm development and visualization environment for novice learners. *Journal of Visual Languages & Computing (JVLC)*, vol. 18, no. 1, pp. 22–47. Available at: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.107.7938&rep=rep1&type=pdf>. 18, 19

- [46] Hundhausen, C.D., Douglas, S.A. and Stasko, J.T. (2002). A meta-study of algorithm visualization effectiveness. *Journal of Visual Languages & Computing (JVLC)*, vol. 13, no. 3, pp. 259–290. Available at: <http://www.cc.gatech.edu/~stasko/papers/jvlc02.pdf>. 3, 7
- [47] Kay, A. (1993). The early history of Smalltalk. In: *Proceedings of the 2nd ACM SIGPLAN Conference on History of Programming Languages (HOPL II)*, pp. 69–95. ACM Press, New York, New York, USA. Available at: <http://worrydream.com/EarlyHistoryOfSmalltalk/>. 18
- [48] Kay, A., Ingalls, D., Ohshima, Y., Piumarta, I. and Raab, A. (2006). Proposal to NSF: Steps toward the reinvention of programming (granted on August 31, 2006). Available at: http://www.vpri.org/pdf/rn2006002_nsfprop.pdf. 1, 18
- [49] Lam, L.C. (2001). A survey of data breakpoint and reverse execution. Tech. Rep., Experimental Computer Systems Lab, Stony Brook University, Stony Brook, New York, New York. Available at: <http://www.ecsl.cs.sunysb.edu/tr/rpe12.ps.gz>. 22
- [50] Lemma, R. and Lanza, M. (2013). Co-evolution as the key for live programming. In: *Proceedings of the 1st International Workshop on LIVE Programming (LIVE)*, pp. 9–10. IEEE Press, San Francisco, California, USA. Available at: <http://liveprogramming.github.io/2013/papers/moon.pdf>. 18
- [51] McConnell, S. (2004). *Code Complete: A practical handbook of software construction*, vol. 2. Microsoft Press, Redmond, Washington, USA. Available at: <http://www.cc2e.com>. 21
- [52] McDirmid, S. (2007). Living it up with a live programming language. In: *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pp. 623–638. ACM Press, New York, New York, USA. Available at: <http://research.microsoft.com/pubs/179365/mcdirmid07live.pdf>. 18
- [53] McDirmid, S. (2013). Usable live programming. *Proceedings of the 2013 ACM SIGPLAN Conference on New Ideas in Programming and Reflections on Software (Onward!)*. Available at:

- <http://research.microsoft.com/pubs/189802/onward011-mcdirmid.pdf>.
4, 18
- [54] McLean, A. (2011). *Artist-programmers and programming languages for the arts*. Ph.D. thesis, Goldsmiths, University of London. Available at:
<http://yaxu.org/writing/thesis.pdf>. 3
- [55] McLean, A., Griffiths, D., Collins, N. and Wiggins, G. (2010). Visualisation of live code. In: *Proceedings of the 2010 International Conference on Electronic Visualisation and the Arts (EVA London)*, pp. 26–30. British Computer Society, London, England. Available at:
<http://yaxu.org/writing/visualisation-of-live-code.pdf>. 8
- [56] Mellor-Crummey, J.M. and LeBlanc, T.J. (1989). A software instruction counter. In: *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 78–86. ACM Press, New York, New York, USA. Available at:
http://www.researchgate.net/profile/John_Mellor-Crummey/publication/234825618_A_software_instruction_counter/links/02e7e523c6b6e675c6000000.pdf. 12, 38
- [57] Norman, D.A. (1993). *Things that make us smart: Defending human attributes in the age of the machine*. Addison-Wesley, New York, New York, USA. 2
- [58] Oney, S., Myers, B.A. and Brandt, J. (2013). Euclase: A live development environment with constraints and FSMs. In: *Proceedings of the 1st International Workshop on LIVE Programming (LIVE)*, pp. 15–18. IEEE Press, San Francisco, California, USA. Available at: http://www.joelbrandt.org/publications/oney_live2013workshop_euclase.pdf. 14
- [59] Paivio, A. (1971). *Imagery and verbal processes*. Holt, Rinehart and Winston, New York, New York, USA. 1
- [60] Pan, D.Z. and Linton, M.A. (1988). Supporting reverse execution of parallel programs. In: *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging (PADD)*, pp. 124–129. ACM Press, New York, New York, USA. Available at:
<http://doi.acm.org/10.1145/68210.69227>. 11, 22, 25

- [61] Perera, R. (2013). *Interactive functional programming*. Ph.D. thesis, University of Birmingham. Available at: <http://etheses.bham.ac.uk/4209/1/Perera13PhD.pdf>. 6, 14
- [62] Pothier, G. (2011). *Towards practical omniscient debugging*. Ph.D. thesis, University of Chile. Available at: http://www.thesis.uchile.cl/tesis/uchile/2011/cf-pothier_g/pdfAmont/cf-pothier_g.pdf. 10, 11, 13
- [63] Sabin, P. (2011). *Implementing a reversible debugger for Python*. Master's thesis, Vienna University of Technology. Available at: <http://mips.complang.tuwien.ac.at/Diplomarbeiten/sabin11.pdf>. 12, 20, 25, 27, 30, 32, 34, 35, 36, 38, 40, 53
- [64] Schiffman, J. (2001). *Aesthetics of computation—Unveiling the visual machine*. Ph.D. thesis, Massachusetts Institute of Technology. Available at: <http://pubs.media.mit.edu/pubs/papers/VisualMachineThesis.pdf>. 7, 8
- [65] Stefik, A. and Siebert, S. (2013). An empirical investigation into programming language syntax. *ACM Transactions on Computing Education*, vol. 13, pp. 1–40. Available at: <http://dl.acm.org/authorize.cfm?key=6968137>. 21
- [66] Tanimoto, S.L. (1990). VIVA: A visual language for image processing. *Journal of Visual Languages & Computing (JVLC)*, vol. 1, no. 2, pp. 127–139. Available at: <http://linkinghub.elsevier.com/retrieve/pii/S1045926X05800126>. 14
- [67] Tolmach, A. and Appel, A.W. (1995). A debugger for Standard ML. *Journal of Functional Programming*, vol. 5, pp. 155–200. Available at: <http://web.cecs.pdx.edu/~apt/jfp95.ps>. 12, 22
- [68] Vygotsky, L.S. (1978). *Mind in society: The development of higher psychological processes*. Harvard University Press, Cambridge, Massachusetts. ISBN 0674576292. 3, 7
- [69] Wilcox, E., Atwood, J.W., Burnett, M.M., Cadiz, J.J. and Cook, C.R. (1997). Does continuous visual feedback aid debugging in direct-manipulation programming systems? In: *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI)*, pp. 258–265. ACM Press, New York, New York, USA. Available at: <http://old.sigchi.org/chi97/proceedings/paper/mmb.htm>. 2, 14