An interactive debugging approach based on time-traveling queries

Maximilian Ignacio Willembrinck Santander

To cite this version:

Maximilian Ignacio Willembrinck Santander. An interactive debugging approach based on time-traveling queries. Programming Languages [cs.PL]. Université de Lille, 2023. English. NNT : 2023ULILB031 . tel-04512544v2

HAL Id: tel-04512544

https://theses.hal.science/tel-04512544v2

Submitted on 20 Mar 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université de Lille, Faculté des sciences et technologies

École doctorale MADIS Lille

An Interactive Debugging Approach Based on Time-traveling Queries

Une Approche de Débogage Interactive basée sur des Time-traveling Queries

THÈSE

présentée et soutenue publiquement le 21 novembre 2023

pour l'obtention du

Doctorat de l'Université de Lille

(spécialité informatique)

par

Maximilian Ignacio Willembrinck Santander

Composition du jury

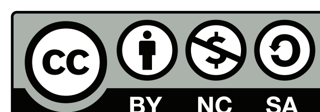
Président :	Walter Rudametkin	Professeur – Université de Rennes
Rapporteurs :	Elisa Gonzalez Boix Christophe Dony	Professeure – Vrije Universiteit Brussel Professeur – Université de Montpellier
Examinateur :	Éric Le Pors	Docteur – Thales
Directrice de thèse :	Anne Etien	Professeure – Université de Lille
Co-Encadrant de thèse :	Steven Costiou	Chargé de recherche – Inria

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
Centre Inria de l'Université de Lille

Inria Université de Lille Evref CRIStAL Centre de Recherche en Informatique, Signal et Automatique de Lille

Copyright © 2023 by Maximilian Ignacio Willembrinck Santander

This work is licensed under a Creative Commons "Attribution-NonCommercial-ShareAlike 3.0 Unported" license.



Acknowledgments

In the following lines, I would like to express my gratitude to everyone who played a crucial role in this challenging yet enriching experience.

First and foremost, I would like to thank and acknowledge the hard work put in by my supervisors, Anne Etien and Steven Costiou. Responsible for most of what I have learned and guiding me during this process, supporting me anytime, all the time, and without hesitation. Thanks to Stéphane Ducasse for his help and for kindly accepting me to be part of the wonderful Evref (former RMoD) team. I would like to thank the thesis reviewers and members of the jury Elisa Gonzalez Boix and Christophe Dony. Thank you for reading this dissertation and giving me valuable feedback. I would like to thanks the jury members Éric Le Pors and Walter Rudametkin for accepting to be part of my Ph.D. committee.

Now, the Evref team. Friendly people, always willing to help and share their expertise in the Pharo language. Thanks to the new and old members of the team for helpful exchanges: Adrien, Aless, Christophe, Clotilde, Esteban, Gabriel, Guillermo, Honoré, Iona, Marcus, Milton, Nahuel, Nicolas, Nour, Oleksandr, Pablo, Pierre, Santiago, Sebastián, Soufyane, Théo, Valentin, and Younoussa. Special thanks to Larisa Safina for her invaluable support during these difficult last months.

I would like to acknowledge the special people who made this endeavor possible. Manuel Valenzuela, and María Cecilia Rivara. Without their support and faith in me, this thesis would have never occurred. Thanks to Carolina Hernández, who started it all. I am here thanks to you. You tutored me and played a key role in my thesis and Pharo journey, for which I am forever grateful.

Last but not least, there is my family. My parents, Marcelo Willembrinck and María Cristina Santander, and my brothers Christian and Arnold.

Dear parents and brothers,

This work is for you.

Abstract

Debugging is an indispensable part of software development, often consuming a significant amount of time and resources. Efficiently debugging a program requires program comprehension. To acquire it, developers explore the program execution, a task often performed using interactive debuggers. Unfortunately, exploring a program's execution through standard interactive debuggers is a tedious and costly task.

We propose Time-traveling Queries (TTQs) to ease interactive program exploration. TTQs is a mechanism that automatically explores program executions using a time-traveling debugger to collect execution data. This mechanism enables a new debugging approach, where the collected data is used to time-travel through execution states, facilitating the exploration of program executions. We built a set of key TTQs based on typical questions developers ask when trying to understand programs. We conducted a user study with 34 participants to evaluate the impact of our queries on program comprehension activities. Results show that, compared to traditional debugging tools, TTQs significantly improve developers' precision while reducing the time and effort required to perform program comprehension tasks, allowing for a more efficient debugging process.

While our TTQs-based debugging approach shows promising results in improving interactive debugging problems, it requires a time-traveling debugger to operate. However, debuggers of such kind are not available in all systems. In systems where multiple programs run in shared memory, time-traveling debuggers face the challenge of precisely scoping time-travel operations. This presents a critical difficulty that limits the applicability of time-traveling debuggers to these systems. To investigate this difficulty, we study the essential properties required to apply time-traveling solutions to this type of system. From these properties, we introduced two time-traveling debuggers for Pharo, a shared memory system. We evaluated the importance of these properties through two experiments, demonstrating their significance in performing precise time-traveling operations. Furthermore, we analyze existing solutions to assess their applicability to shared memory systems.

Keywords: Debugging, Time-traveling Debuggers, Reversible Debuggers, Reverse and Replay, Time-traveling Queries

Résumé

Le debugging est une activité incontournable du développement logiciel. Il s'agit d'une tâche ardue qui nécessite une grande quantité de temps et de ressources, et elle exige que les développeurs aient une compréhension approfondie de leurs programmes. Pour ce faire, les développeurs explorent l'exécution de leurs programmes à l'aide de debuggers interactifs. Cependant, cette activité est souvent répétitive et coûteuse.

Afin de simplifier l'exploration interactive des programmes, nous proposons les Time-traveling Queries (TTQs). TTQs est un mécanisme qui explore automatiquement les exécutions de programmes en utilisant un Time-traveling Debugger pour recueillir des données d'exécution. Ce mécanisme permet une nouvelle approche de debugging, où les données collectées permettent de voyager dans le temps à travers les états d'exécution, facilitant ainsi l'exploration des exécutions de programmes. Nous avons élaboré un ensemble de questions clés basées sur des questions typiques que les développeurs se posent lorsqu'ils essaient de comprendre les programmes. Pour évaluer l'impact de nos requêtes sur les activités de compréhension des programmes, nous avons mené une étude empirique impliquant 34 utilisateurs. Nos résultats montrent que, par rapport aux outils de debugging traditionnels, les TTQs améliorent considérablement la précision des développeurs tout en réduisant le temps et les efforts nécessaires pour mener à bien les tâches de compréhension des programmes.

Malgré les résultats prometteurs que notre approche a montrés, elle présente une exigence cruciale: elle nécessite l'utilisation d'un Time-traveling Debugger pour être opérationnelle. Cependant, il est important de noter que les debuggers de ce genre ne sont pas disponibles pour tous les systèmes. Dans les systèmes où plusieurs programmes sont exécutés dans une mémoire partagée, les Time-traveling Debuggers se heurtent à la complexité de délimiter les effets en mémoire causés par les opérations de voyage dans le temps. Pour résoudre ce problème, nous avons mené une étude approfondie des propriétés essentielles requises pour adapter les solutions de voyage dans le temps à ce type de système. En utilisant ces propriétés comme guide, nous avons introduit deux Time-traveling Debuggers spécifiquement conçus pour Pharo, un système qui repose sur une mémoire partagée. Nous avons ensuite évalué l'importance de ces propriétés au travers de deux expériences, démontrant ainsi leur rôle crucial dans la réalisation d'opérations précises de voyage dans le temps. Enfin, nous avons procédé à une analyse approfondie des solutions existantes pour évaluer leur applicabilité dans le contexte des systèmes à mémoire partagée.

Mots-clés: Debugging, Time-traveling Debuggers, Debuggers Réversibles, Inverser et Rejouer, Time-traveling Queries

Contents

Introduction 1

1 Introduction 3
1.1 Context . . . 3
1.2 Problems Statement . . . 5
1.2.1 Motivation: Problems in debugging and live exploration of program executions . . . 5
1.2.2 Summarized main research problems . . . 8
1.2.3 Research questions . . . 8
1.3 Our Proposition: Time-traveling Queries . . . 9
1.3.1 Time-traveling queries, in a nutshell . . . 9
1.3.2 Improving the debugging process . . . 9
1.4 Contributions . . . 10
1.5 Publications and Awards . . . 10
1.6 Thesis Outline . . . 12
1.7 Conclusion . . . 13

2 Background and State of the Art 15
2.1 Debugging in Software Development . . . 15
2.1.1 Tools for debugging software: Debuggers . . . 15
2.1.2 Debugging approaches offered by debuggers . . . 16
2.1.3 Debugging by exploring executions history . . . 17
2.2 Time-traveling Debuggers . . . 18
2.3 State-of-the-art Time-traveling Techniques . . . 19
2.3.1 Techniques used for reversal . . . 19
2.3.2 Techniques used for replay . . . 20
2.4 Complementary Debugging Capabilities . . . 21
2.4.1 Scriptable debugging . . . 21
2.4.2 Query-based debugging . . . 22
2.5 Comparison of Debuggers' Features . . . 23
2.6 Conclusion . . . 25

I Time-travel Debugging in Shared Memory Systems 27

3 Selective Time-travel in Shared Memory 29
3.1 Introduction . . . 29
3.2 Motivation . . . 31
3.3 Problems of Imprecise Memory Scoping In Reversal Operations . 33

3.3.1 Side effects 33
3.3.2 Towards selective time-travel 33
3.4 Properties for Selective Time-travel 34
3.5 A Selective Time-travel Back End 35
3.5.1 Time-travel back end overview 36
3.5.2 Back end components and properties support 36
3.6 Conclusion 38

4 Implementation 39
4.1 The Standard Pharo Debugger: An Overview 39
4.2 Seeker: A Time-traveling Debugger for Single-threaded Programs 41
4.2.1 Seeker, in a nutshell 41
4.2.2 GUI Mode: Time-travel for the standard debugger 42
4.2.3 Debugging programmatically in headless mode 44
4.2.4 Seeker implementation 45
4.2.5 Components of the time-travel back end 47
4.2.6 Seeker time-travel operations explained 51
4.2.7 Configurable properties support 53
4.3 Executor: A Time-traveling Debugger for Multithreaded Programs 54
4.3.1 Executor implementation 54
4.3.2 Configurable properties support 55
4.4 Implementation Discussion 55
4.4.1 Recorded data overview 55
4.4.2 Validation of the selectiveness of our implementation . . . 56
4.4.3 Reversal by full-replay 58
4.4.4 System calls logging 59
4.4.5 Limitations 59
4.4.6 System stability assumption 60
4.5 Conclusion 60

5 Evaluation 61
5.1 Experiment Goals and Research Question 61
5.2 Evaluation of the Properties for Single-threaded Executions 62
5.2.1 Experiments general procedure 63
5.2.2 Evaluation of the properties on a crafted program 65
5.2.3 Evaluation on multiple programs running real code 69
5.3 Evaluation of the Properties for Multithreaded Executions 72
5.3.1 Experiment procedure 74
5.3.2 Experiment results and analysis 74
5.4 Results Conclusion 76
5.5 Threats to Validity 77
5.6 Related Work 78
5.6.1 Properties support in time-traveling debuggers 78

5.6.2 Properties support in time-travel techniques 80
5.7 Conclusion 81

II A New Debugging Approach 83

6 Time-traveling Queries: Improving Interactive Debugging 85
6.1 Improving on Interactive Debuggers Problems 85
6.2 Time-traveling Queries 86
6.2.1 Time-traveling Queries definition and execution 86
6.3 Off-the-shelf Time-traveling Queries 88
6.3.1 Key Time-traveling Queries 89
6.3.2 Executing queries 90
6.4 Time-traveling Queries Implementation 92
6.4.1 Time-traveling queries requirements 92
6.4.2 Query implementation 93
6.4.3 ProgramStates class 96
6.4.4 Modifications of Seeker to support Time-traveling Queries 96
6.4.5 Implementation of key Time-traveling Queries 97
6.4.6 User-defined time-traveling queries 100
6.5 Conclusion 100

7 Evaluation of the TTQ-based Debugging Approach 101
7.1 Empirical Evaluation 101
7.1.1 Objectives of the experiment 102
7.1.2 Experimental design 103
7.2 Results and Discussion 106
7.2.1 Experiment results 106
7.2.2 Post-study survey 109
7.2.3 Discussion on participant's experience impact on the results 110
7.2.4 Threats to validity 111
7.3 Conclusion 112

8 Time-traveling Queries for Specialized Debugging 113
8.1 Specialized Debugging Tools 113
8.2 A Real World Scenario: Debugging a Meta Compiler 114
8.3 Identifying False Positives in String-Symbol Comparisons 115
8.4 Domain-Specific Queries for the Moose Platform 118
8.5 Queries of Object-centric Debugging 120
8.6 Reproducing the Moldable Debugger Experiments 125
8.7 Conclusion 129

Thesis Conclusion **131**

9 Conclusion and Future Work **133**
9.1 Conclusion 133
9.2 Future Work 135
9.2.1 Enhancing performance in our time-travel solution 135
9.2.2 Leveraging our proposed debugging approach 136

Bibliography **139**

List of Figures

1.1 The iterative process of debugging. 4
1.2 Exploring an execution by using breakpoints and stepping. 6
1.3 Breakpoints scripted as automatic sequences of steps. 7

2.1 Time-traveling and back-in-time debuggers. 17

3.1 Debugger state is lost when revering in shared memory. 31
3.2 Time-travel back end for multithreaded executions in shared memory. 36
3.3 The TCU stepping pipeline. 38

4.1 The Pharo StDebugger main GUI elements. 40
4.2 Seeker GUI. 43
4.3 Integration of Seeker as an extension plug-in of the StDebugger. . 46
4.4 Seeker implementation. 47

6.1 Time-traveling query collecting execution data. 88
6.2 Exploring an execution using time-traveling queries results. 89
6.3 Integration of time-traveling queries into the debugger GUI. . . . 91
6.4 Scripting pane in Seeker to write time-traveling queries. 91
6.5 Class diagram of the *Time-traveling Queries* mechanism. 94

7.1 Debugger UI layout of used for the experiment. 102
7.2 Participant scores. 107
7.3 Participants time. 107
7.4 Participants debugging actions. 108
7.5 Experiment results. 109
7.6 Histogram of participants' years of experience in Pharo. 111

8.1 Example of inspecting a query result item. 117
8.2 Visualization of the detected string/symbol equality. 117
8.3 Seeker UI showing object-centric operations. 123
8.4 Object-centric time-traveling queries menu of the enhanced inspector. 124

List of Tables

2.1 Debuggers features comparison. 24

4.1 Seeker debugging API 45
4.2 Seeker initialization API 46
4.3 The CurrentState API. 48
4.4 CurrentState API methods for obtaining message-sends data . . . 49
4.5 CurrentState API methods for obtaining assignments data. 50

5.1 Experiment 1 criteria approval summary. 67
5.2 Experiment 1 measurements: step number. 67
5.3 Experiment 1 measurements: program counter and instruction code. 68
5.4 Experiment 1 measurements: local variable value. 68
5.5 Experiment 1 measurements: global value. 69
5.6 Experiment 2 results. 71
5.7 Experiment 3 measurements for configurations 1 to 8. 75
5.8 Experiment 3 measurements for configurations 9 to 16. 76
5.9 Time-traveling debuggers comparison 79

7.1 Tasks in the controlled experiment. 105
7.2 H_0 rejection table with Wilcoxon signed-rank test values. 109
7.3 Tool reception rating of the post-study survey. 109
7.4 Participants' confidence in their answers. 110
7.5 Participants' perceived difficulty of each sequence. 110
7.6 Results according to the experiment order. 112

Introduction

Chapter 1
Introduction

Contents

1.1 Context 3
1.2 Problems Statement 5
1.3 Our Proposition: Time-traveling Queries 9
1.4 Contributions 10
1.5 Publications and Awards 10
1.6 Thesis Outline 12
1.7 Conclusion 13

Efficiently debugging a program requires program comprehension. To acquire it, developers explore the program execution, a task often performed using interactive debuggers. Unfortunately, exploring a program's execution through standard interactive debuggers is a tedious and costly task. In this chapter, we present our proposal for addressing the challenges of using interactive debuggers and provide an overview of our thesis.

1.1 Context

In software development, the process of identifying and fixing program errors is known as debugging. Debugging is an iterative process: developers first make an observation and then formulate a hypothesis about the cause of the failure (Figure 1.1). To test their hypotheses, they try to reproduce the bug by observing data and behavior supporting such hypotheses. Facing a wrong hypothesis forces developers to formulate new, more refined ones, iteratively narrowing down the possible cause [Barr 2014, O'Dell 2017, Phang 2013, Spinellis 2018, Zeller 2009].

Formulating hypotheses requires understanding programs. Typically, developers ask themselves questions about the execution of their program [Sillito 2008], *e.g., why is this variable in an incorrect state?* Then, they try to answer these questions by exploring that execution.

Exploring program executions is important to produce good hypotheses, especially when facing unfamiliar bugs [O'Dell 2017], and it is commonly

performed using interactive debuggers. However, it is not an easy task. Traditionally, this is done by selectively stepping executions, instruction by instruction. It is a manual operation, and there is a risk of stepping too far and, therefore, missing a critical piece of information [Barr 2014]. In addition, stepping is a generic operation that does not directly translate questions asked by developers to test a hypothesis into a stepping sequence (*i.e., how many steps should we perform to find that information?*). To enhance program execution exploration and, thus, debugging, we argue that we need a mechanism that transforms a question formulated by the developer into a debugging action that collects relevant execution data of a program.

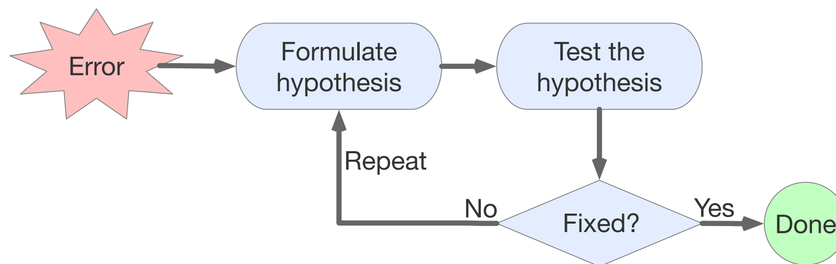


Figure 1.1: The iterative process of debugging.

To this end, we propose *Time-traveling Queries* (*TTQs*). TTQs are expressions that ask for specific information about a program execution. The time-traveling debugger answers the query by executing the program instructions, one at a time, automatically traversing all its states while retrieving the required execution data. The queried data is collected to produce a *query result set*. The produced results are then used by developers to time-travel, *i.e.,* advance or reverse the execution of the program to the point in the execution history where the result data were retrieved. TTQs would provide direct access to relevant debugging data and would also enable the creation of more specialized TTQ-based debugging tools.

To conduct our investigation in this thesis, we utilized the Pharo programming language [Black 2009]. Pharo provides strong reflective capabilities, which is advantageous for building sophisticated debugging tools, allowing us to inspect and modify the language itself and essentially any object of the system.

Terminology: Execution and time-travel

We use the term *execution* to refer to the execution of a list of instructions. Therefore, a *thread execution* is the execution of the instructions of a thread, and a *program execution* is the execution of the program's threads. That is, a program execution encompasses the execution of all instructions of its threads.

We consider program executions as sequences of program states. A program state *"consists of the values of the program variables, as well as the current*

execution position (formally, the program counter). Each state determines subsequent states, up to the final state..." [Zeller 2009]. The program state includes both: the state of the execution's dynamically allocated memory - commonly known as *the heap* - and the state of the execution *stack* (the state of all the program threads).

We use the term *execution state* as a synonym of program state.

To simplify our explanations, we observe and analyze program executions at a bytecode level granularity, breaking down executions into individual bytecode instructions and stepping through them.

The term *execution step* or *step* refers to the execution of one instruction of a program, and a step advances the execution from one state to another.

We use the term *time-travel* to refer to the features provided by debuggers that allow developers to advance or reverse the state of a program to specific points of its execution history. These features include operations such as step-back, which restores the program to the state it was before executing the last instruction, and completely reversing a program to its initial state. For these debuggers, we also consider forward-stepping (advancing the program state by performing steps) as a time-travel operation.

1.2 Problems Statement

Next, we outline the specific problems that arise when using standard interactive debuggers and identify the primary research problems addressed in this thesis.

1.2.1 Motivation: Problems in debugging and live exploration of program executions

The *simplified scientific method* [Zeller 2009, Spinellis 2018] is a common debugging method. It consists in formulating hypotheses regarding the cause of a bug. Then, developers selectively observe their program execution to confirm or discard those hypotheses. Ultimately, the correct hypothesis is confirmed, and the bug is found. It is an iterative process in which developers systematically test and observe their program to understand it better. The more they understand, the more they clarify their hypotheses and the more they narrow down the cause of the bug.

The most standard tools and techniques shipped with every debugger are breakpoints and instruction stepping (Figure 1.2). Developers use breakpoints to break the execution, then observe the state of the interrupted program. They decide to either resume the execution until the next breakpoint or to step forward one program instruction to observe the evolution of the program state [Zeller 2009]. They repeat these operations until they find the information they were looking for or until the program ends.

These debugging tools present the following problems:

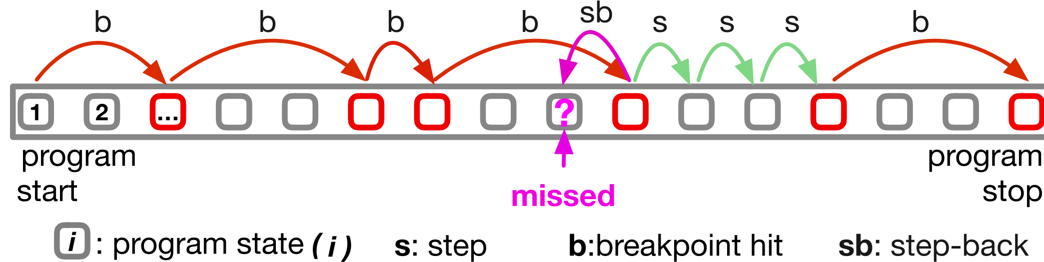


Figure 1.2: Exploring an execution with breakpoints and manual backward and forward stepping.

- **P1. Debugging questions translation difficulty.** Developers' debugging questions cannot be easily translated into sequences of breakpoints and stepping actions.
- **P2. Require prior knowledge of the program.** Choosing efficiently where to put breakpoints requires already understanding parts of the program. Developers, therefore, have to perform preliminary investigations of the program [Ressia 2012], *e.g.,* through source code reading.
- **P3. Manual/Tedious.** Developers have to manually choose where to put breakpoints and to step the execution when it breaks. They manually advance through many irrelevant breakpoint hits and then perform numerous stepping operations until reaching the point of interest.
- **P4. Missing critical points.** It is common to miss a critical point in the execution [Barr 2014], *e.g.,* the *missed* program state in Figure 1.2. Developers have to restart and explore the execution again to look for the information they missed.

With *time-traveling debuggers*, developers travel backward and forward in their program execution. For example, in Figure 1.2 a *step-back* operation allows developers to travel back in time to observe an execution point they missed with the standard stepping. Because of that, if developers stepped one step too far and missed an important piece of information, they can immediately step back and observe that information. However, looking for a piece of information by stepping back and forth in a recorded execution is also a manual operation. Without additional means to explore recorded executions, it is as tedious as standard breakpoints and stepping.

Using scriptable debuggers (Figure 1.3), developers program sequences of steps to automatically explore an execution and build problem-specific debugging tools [Dupriez 2019]. Every state of the execution can be attained, but what to do for each state (observing, collecting data...) must be specified in the scripts. This

implies that developers already gathered a sufficient understanding of the program to know what to look for to write scripts. On top of that, they must translate their debugging questions into debugging scripts. Developers must also understand and reason within several abstraction domains, including the program itself, the scripting API, etc.

When developers seek answers to debugging questions using conventional tools, they face the challenge of translating their questions into sequences of debugging actions. This process of translation is not straightforward and creates an abstraction gap To alleviate this issue, domain-specific debugging tools offer debugging actions that are closely aligned with the application domain [Chiş 2014]. However, this specialization comes with a trade-off: it sacrifices versatility, demanding tailored debugging tools for each unique domain. Furthermore, debugging tools's implementations are inflexible, as they are not typically designed to be extended. We summarize these challenges in the following problem

- **P5. Generic and inflexible debugging tools.** Extending or creating new specialized debugging tools to address domain-specific issues is challenging due to the inflexible architecture of debugging tools.

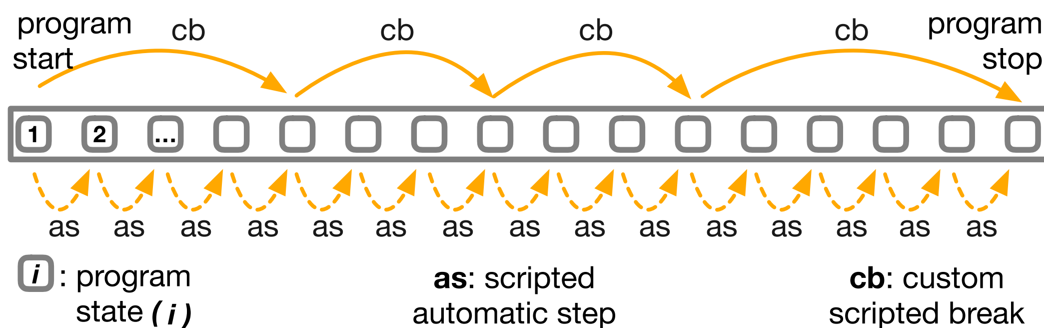


Figure 1.3: Breakpoints scripted as automatic sequences of steps.

We argue that an approach that combines time-traveling debuggers and queries would improve all listed problems. Debugging can be made easier by using specialized queries derived from debugging questions. The time-traveling debugger would take the query and automatically collect the relevant debugging data to answer the question. This approach can lead to new program exploration capabilities by automating tedious and error-prone tasks and providing direct access to pertinent debugging data.

Specialized queries would not only close the abstraction gap but would facilitate the creation of new domain-specific debugging tools, as debugging tools using this approach could be extended simply by defining new queries.

Since time-traveling debuggers are the cornerstone of this dissertation, having such a debugger is a critical requirement. Certain programming languages such as Lisp [McCarthy 1960], Pharo [Black 2009], Python [Python 2023], SELF [Ungar 1987], and Smalltalk [Goldberg 1984], run programs within a shared memory framework. Developers using these languages or working with similar *shared memory systems* will find an obstacle while trying to make use of a time-traveling debugger, as to the best of our knowledge, there are no solutions that completely support these systems. This introduces the following problem:

- **P6. Absence of time-traveling debuggers for shared memory systems.** There are no dedicated time-traveling debuggers for shared memory systems, and this problem is not addressed in the literature.

1.2.2 Summarized main research problems

From the described set of problems, we state our two research problems (*RPs*) of this thesis. We list them in the order we addressed them, each one in its dedicated part of this dissertation:

RP1, addressed in Part 2: There is no time-traveling debugger available for shared memory systems, and this problem is not addressed in the literature (From P6).

RP2, addressed in Part 3: Accessing specific program execution data to answer debugging questions is difficult when using interactive debuggers (From P1, P2, P3, P4, P5).

1.2.3 Research questions

Therefore, in the scope of our work, we investigate the following research questions (*RQs*):

RQ1: What properties a time-traveling debugger should possess to support shared memory systems? (Addressing P6) (Chapters 3, 4, 5)

RQ2: Can we express debugging questions as *queries* over programs executions to obtain relevant debugging information? (Addressing P1, P2, P3, P4) (Chapter 6)

RQ3: How does this query-based mechanism improve the debugging experience? (Chapter 7)

RQ4: Is this new debugging mechanism extensible to tackle domain and problem-specific debugging scenarios? (Addressing P5) (Chapter 8)

1.3 Our Proposition: Time-traveling Queries

We propose combining time-traveling debugging with scriptable debugging techniques to express program comprehension questions as queries over program executions. We call these queries *Time-traveling Queries* (TTQs), which introduce a novel TTQ-based debugging approach and tools.

1.3.1 Time-traveling queries, in a nutshell

A TTQ is a query over a program execution, a mechanism that automatically explores program executions to selectively collect execution data. This data is used to time-travel through execution states, facilitating the exploration of program executions by granting direct access to program states which are relevant to the query.

TTQs are defined based on developer debugging questions and are integrated into debuggers. Developers can write personalized queries for their particular debugging and program exploration needs, enabling new debugging tools.

TTQs and TTQs-based debugging tools introduce a new interactive debugging approach that closes the gap between expressing developer questions and finding relevant debugging data during a standard online debugging workflow.

1.3.2 Improving the debugging process with TTQs and TTQs-based tools

TTQs explore the whole program execution to extract execution information. This information is presented to developers, who are able to time-travel, in the program execution, to the point where that information was obtained. There, developers can observe the information in its original context. They can deepen their understanding of the execution by time-traveling to another result or by performing standard forward or backward steps.

We argue that TTQs will enable the creation of new debugging tools for in-depth live program exploration. Developers will directly use pre-existing queries available on the shelves for general-purpose debugging or express their own questions as programmatic queries. New specialized debugging tools will integrate and bring ready-to-use specialized TTQs completely tailored to different debugging needs. Program exploration will require less preliminary investigation and consequently improve developers' debugging efficiency.

1.4 Contributions

This dissertation presents the following main contributions.

1. A time-traveling debugger solution for shared memory systems. We identified essential properties that time-traveling debuggers require to support time-travel operations in shared memory systems.

2. Time-traveling Queries. A debugging mechanism that enables a novel interactive program exploration approach. It allows developers to *jump* directly to relevant program states during their debugging sessions, helping them to find answers to their debugging questions. We present a list of key ready-to-use TTQs, based on *common debugging questions*, to improve the debugging experience and an evaluation based on a controlled experiment. We present a set of new specialized debugging tools based on TTQs as well.

3. A TTQ-enhanced time-traveling debugger for Pharo. A time-traveling debugger for Pharo, enhancing the standard Pharo debugger with TTQs capabilities and enabling the creation of new TTQ-based debugging tools.

1.5 Publications and Awards

In this section, we list the publications and achievements attained in the context of this thesis.

Publications

Time-Traveling Debugging Queries: Faster Program Exploration.
[Willembrinck 2021] - Conference Paper
Maximilian Willembrinck, Steven Costiou, Anne Etien, Stéphane Ducasse.
International Conference on Software Quality, Reliability, and Security, Dec 2021, Hainan Island, China.
https://inria.hal.science/hal-03463047.

Time-Traveling Queries for Faster Debugging and Program Comprehension.
[Willembrinck 2022a] - Poster
Maximilian Willembrinck, Steven Costiou, Anne Etien, Stéphane Ducasse.
Journées Nationales du Génie de la Programmation et du Logiciel 2022, Jun 2022, Vannes, France.
https://inria.hal.science/hal-03738585.

Towards Object-Centric Time-Traveling Debuggers.
[Willembrinck 2022b] - Conference Paper
Maximilian Willembrinck, Steven Costiou, Adrien Vanègue, Anne Etien.
International Workshop on Smalltalk Technologies: IWST 22, Aug 2022, Novi Sad, Serbia.
https://inria.hal.science/hal-03825736.

Time-Traveling Queries: Extensible Tools for Faster Program Comprehension.
Maximilian Willembrinck, Valentin Bourcier, Adrien Vanègue, Stéphane Ducasse, Anne Etien, Steven Costiou.
Submitted 15/09/2023 to Journal of Object Technology.

Reverse and Replay Debugging in Shared Memory Systems.
Maximilian Willembrinck, Steven Costiou, Guillermo Polito, Anne Etien.
Submitted 16/09/2023 to Journal of Computer Languages.

Time-Traveling Object-Centric Breakpoints.
Valentin Bourcier, Steven Costiou, Maximilian Willembrinck, Adrien Vanègue, Anne Etien.
Submitted 16/09/2023 to Journal of Computer Languages.

Awards

Best poster award:
Time-Traveling Queries for Faster Debugging and Program Comprehension.
Journées Nationales du Génie de la Programmation et du Logiciel 2022, Jun 2022, Vannes, France.
https://gdr-gpl.cnrs.fr/node/502

1st place in the Innovation Technology Awards 17th Edition:
Time-Traveling Queries for Faster Debugging And Program Comprehension.
Innovation Technology Awards 17th Edition, at Esug 2022, August 2022, Novi Sad, Serbia.
https://esug.github.io/2022-Conference/awardsSubmissions.html

1.6 Thesis Outline

This dissertation is structured as follows:

- Introduction.
  - In Chapter 1, we described the difficulties of exploring a program execution using standard debugging methods and proposed a novel debugging and program exploration approach based on TTQs. Then, we listed the research questions addressed in this dissertation. We concluded by listing our publications and outlining the contents of the dissertation.
  - In Chapter 2, we explore the background of our work, including time-traveling debuggers and the current state of the art.
- Part I - Time-travel in Shared Memory Systems.
  - Chapter 3 presents a comprehensive analysis of the challenges in applying conventional time-traveling debugging solutions to shared memory systems. We identify 4 essential properties that debuggers must possess to overcome those challenges, and we propose a time-travel solution based on these properties.
  - Chapter 4 describes our time-traveling debugger implementations based on the identified properties.
  - In Chapter 5, we use our implementations to evaluate the importance of the identified properties for time-traveling debuggers in shared memory systems.
- Part II - A New Debugging Approach.
  - Chapter 6 describes our solution to the second research problem: We introduce Time-traveling Queries and our proposed debugging approach based on TTQs. We describe the implementation of our solution and its integration with our time-traveling debugger.
  - In Chapter 7, we present the evaluation of TTQs, where we show the results of our user study that measures how a TTQs-based debugging approach improves the debugging experience vs. a standard one.
  - In Chapter 8, we describe our explorations on the extensibility and applications of TTQs. We show how we used them to create new TTQ-enhanced specialized and domain-specific debugging tools.
- Thesis Conclusion.
  - In Chapter 9, we conclude this dissertation and discuss future work.

1.7 Conclusion

In this chapter, we provided an overlook of our endeavor to improve the debugging landscape. We have described the research problems at hand and presented our proposed solutions. Our efforts are concentrated on two primary areas: enabling time-traveling debugging for shared memory systems, addressed in Part I, and improving the interactive debugging experience, in Part II. Additionally, we outlined the dissertation chapters that delve deeper into challenges in each respective area.

CHAPTER 2
Background and State of the Art

Contents

2.1 Debugging in Software Development 15
2.2 Time-traveling Debuggers 18
2.3 State-of-the-art Time-traveling Techniques 19
2.4 Complementary Debugging Capabilities 21
2.5 Comparison of Debuggers' Features 23
2.6 Conclusion 25

In this chapter, we present the surrounding context and different domains covered by our work. We explain the landscape of debugging tools while defining the terms we use throughout the discourse. We describe debuggers and implemented techniques. Then, we summarize by comparing debuggers in terms of the important aspects related to the subjects addressed in this thesis.

2.1 Debugging in Software Development

In the complex world of software development, the process of detecting and resolving errors remains crucial. Errors within a program can hinder its intended functionality and disrupt the user experience. These errors are commonly known as *bugs*, and the process of identifying and fixing bugs is known as *debugging*.

In this section, we discuss the importance of debugging tools and the different approaches that these tools offer. We then explore examples of debuggers following these approaches and provide definitions to contextualize our work in time-traveling debugging.

2.1.1 Tools for debugging software: Debuggers

Debugging is a difficult and costly activity, in some cases accounting for 50% of development time [McConnell 2004, Spinellis 2018, Tassey 2002, Zeller 2009]. Therefore, tools that assist in debugging are extremely valuable. To gain insight into a program, developers can make use of several tools that provide access

to a great deal of static and dynamic program data [Phang 2013, Pothier 2007, Sillito 2008].

While static analysis tools help uncover potential issues before running programs' code, they are not designed to actively debug and fix runtime errors or investigate the behavior of a program during execution. Developers make use of dynamic analysis tools such as *debuggers* to debug and investigate a program's runtime behavior.

2.1.2 Debugging approaches offered by debuggers

Debuggers offer two main distinct debugging approaches: post-mortem and interactive. The *post-mortem debugging approach* consists of analyzing a program's execution after its conclusion. Debuggers offering this approach are known as ***post-mortem debuggers*** or *offline debuggers*. In contrast, the *interactive debugging approach* consists of analyzing a running program while controlling its execution. Debuggers offering this approach are known as ***interactive debuggers*** or *online debuggers*.

2.1.2.1 Post-mortem debuggers

Post-mortem debuggers work with recorded program data after a program has completed its execution, such as crash dumps or execution logs. These debuggers are valuable for post-mortem analysis to understand the causes of crashes or failures that have already occurred. They do not provide real-time interaction with the running program but instead help developers analyze the state of the program at the time of the crash or the events leading to the failure [Hofer 2006, Ko 2008, Lewis 2003, Lienhard 2008, Lienhard 2009, Pothier 2007].

These debuggers offer an important advantage over standard interactive debuggers when dealing with hard-to-find bugs. If the program under investigation behaves non-deterministically, then restarting the debug process and the program to investigate the causes of the bug might not reproduce the failure. Furthermore, other errors might manifest that are different than the initial one that prompted the debug session [Engblom 2012]. With post-mortem debuggers, the execution is finished and recorded, making it easier for developers to investigate this type of bug.

2.1.2.2 Interactive debuggers

Interactive debuggers, which our thesis centers on, enable developers to control the execution of a running program and inspect its internal state. With these debuggers, developers can set breakpoints, navigate through code, and analyze variables and data structures to gain insight into how a program operates and to identify potential errors.

Interactive debuggers come in various forms, such as the ones included in integrated development environments (IDEs), command-line debuggers, and graphical user interface (GUI) debuggers.

Nowadays, many popular programming languages and environments are shipped with an interactive debugger as the default debugging tool. Examples include *Gdb* [Richard Stallman 2003] (GNU Debugger) for C/C++ programs, the Java debugger [Oracle 2023], Chrome's debugger for Javascript [Google 2023], the Pharo debugger [Black 2009], *Pdb* [PDB 2023] for Python, and Visual Studio's debugger [Microsoft 2023] for various programming languages.

2.1.3 Debugging by exploring executions history

We call *exploring execution history* (EEH) to the capability provided by certain debuggers to explore program executions and observe program states in their execution history. To the best of our knowledge, there is no consensus in the literature on the specific meaning of key terminology related to the aforementioned capability. To set a clear and precise foundation for explanations, we present the following definitions, which we will utilize throughout this dissertation (Figure 2.1).

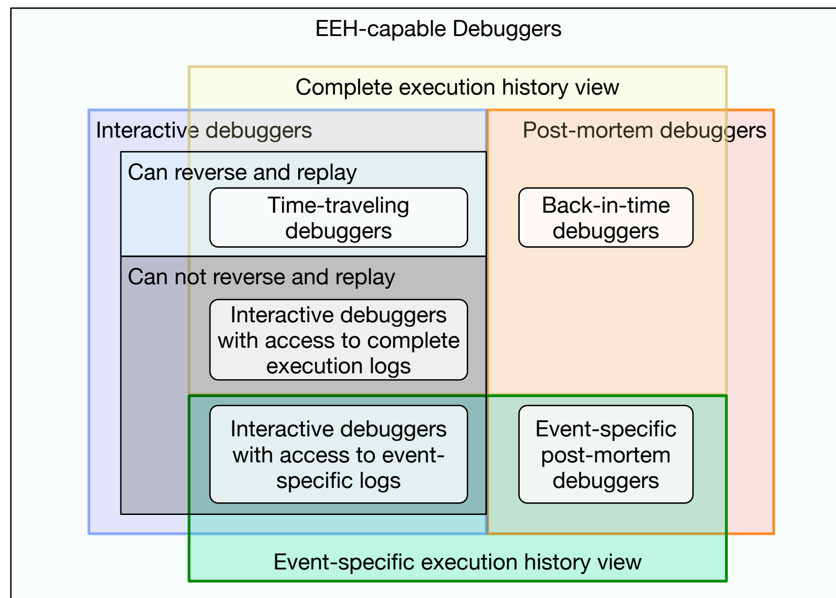


Figure 2.1: Graphical representation of our definitions of time-traveling and back-in-time debuggers based on capabilities and debugging approaches.

We use the term *back-in-time debuggers* when referring to post-mortem debuggers that provide the EEH capability. Back-in-time debuggers logs contain all the data required to allow developers to see a complete view of the state of the program at any point in its execution history. Examples of these debuggers, often

named *omniscient debuggers* in the literature, include [Lewis 2003, Pothier 2007, Pothier 2011]. Other post-mortem debuggers [Ko 2004, Ko 2008, Lauwaerts 2023], in contrast to our definition of back-in-time debuggers, do not provide a complete view of all the states of an execution, specializing instead on logging specific program events to observe particular program behaviors.

We use the term *time-traveling debuggers* when referring to interactive debuggers providing the EEH capability. The main distinction with back-in-time debuggers is that time-traveling debuggers allow developers to control and reverse the running debugged execution. This way, developers can access and explore all the program states of the running execution following an interactive debugging workflow.

As our thesis centers on interactive debuggers, from here on, our discussions will gravitate around time-traveling debuggers primarily.

2.2 Time-traveling Debuggers

These tools offer developers the ability to reverse a debugged program to any point in its execution history, which helps to find the cause of a failure. Time-traveling debuggers enable the reproduction of subtle bugs by guaranteeing the execution of otherwise non-deterministic ones [Arya 2017].

Main characteristics defining time-traveling debuggers

In the world of time-traveling debuggers, there are numerous variations that differ in their specialized debugging features and inner mechanisms.

We state what we consider to be the main common aspects that define a time-traveling debugger. In this dissertation, we consider that time-traveling debuggers present the following specific characteristics:

1. They are interactive.

2. They are able to reverse a program to a previous point in their execution history.

3. They are able to replay a program *deterministically*: On every replay of the program, the program is stepped through the same executed instructions. The program goes through the same program states during each replay.

While solutions presenting only one or two of these points deviate from our definition of time-traveling debuggers, we nonetheless examine them when discussing the corresponding characteristics in the dissertation.

2.3 State-of-the-art Time-traveling Techniques

Different time-traveling debugger solutions implement a variety of techniques for time-traveling. These techniques depend on many factors, including the target programming language of the debugged program, the runtime, the operative system, and the target abstraction level the debugger operates (software level, hardware level, and combinations). Despite the multitude of variations addressing these factors, these techniques often share a common base. We describe the base techniques commonly used by current time-traveling debuggers.

2.3.1 Techniques used for reversal

We refer as *reversal* to the feature of time-traveling debuggers that allows developers to step backward in the execution history of a program to examine its past states and diagnose issues. To achieve this, time-traveling debuggers record program data during the execution of the debugged program. Then, the debugger uses these records to reconstruct the target program state [Engblom 2012]. We identify the following reconstruction techniques used to implement the reversal feature.

Snapshot-based execution reconstruction

Process snapshotting *i.e.,* capturing a process' state, is a commonly used reversal technique. It is used as part of the reversal mechanism in many time-traveling solutions [O'Callahan 2017, Phang 2013, UDB 2023, Vilk 2018]. These solutions rely on the capability of the system to create snapshots of the debugged processes.

To create snapshots, the debugged process is suspended, and *forked i.e.,* a copy of the suspended process is made, and then it is stored by the debugger. Each snapshot captures the entire application state (*i.e.*, both heap and the stack) of the debugged program. The application state can then be returned to a previous state by resuming the process of one of the stored snapshots.

[Barr 2014, Barr 2016] takes advantage of managed runtimes features to distinguish application memory from the rest of the runtime. This allows them to scope down the snapshotted elements, avoiding taking snapshots of the complete process (which would include the runtime itself) and instead only include the crucial application state.

Replay-based execution reconstruction

Replay-based execution reconstruction consists of reconstructing a past state by executing forward from some saved state [Engblom 2012]. This prevents the need to create snapshots of every state of the program.

Depending on the implementation, debuggers choose to sparsely capture snapshots on specific program events (*e.g.,* method calls), repeatedly after a certain amount of time has passed, or both [Engblom 2012, UDB 2023]. Since not all execution states are recorded (*i.e.,* there is no snapshot taken after each step), to reverse an execution to a point a few steps forward after the state captured in a snapshot, the execution is advanced deterministically from that snapshot, thus reconstructing the target execution state.

This reconstruction technique only needs to record the information that cannot be reconstructed from loading the past state. This diminishes the need for snapshots and allows for smaller logs in comparison to a record-all approach.

In practice, most implementations follow this approach [Arya 2017, Barr 2014, Barr 2016, King 2005, Montesinos 2008, O'Callahan 2017, Phang 2013, Pothier 2011, UDB 2023, Vilk 2018].

2.3.2 Techniques used for replay

When debugging a program by exploring its execution back and forth, the program under investigation is required to be deterministic. However, there are many factors that can introduce non-determinism into program executions, such as system calls *e.g.,* obtaining the system time or generating random numbers. In such a situation, attempting to reproduce an error by rerunning the program is unlikely to work (*i.e.,* not reproducing the bug), or even encountering different errors than the initial one [Engblom 2012].

Each time-traveling debugger has a particular implementation to achieve the deterministic replay of a program. However, most implementations [Arya 2017, Barr 2014, Barr 2016, King 2005, Montesinos 2008, O'Callahan 2017, Phang 2013, Pothier 2011, UDB 2023, Vilk 2018] adhere to the following base technique:

- When executing the debugged program, the debugger logs the result of the functions that introduce non-deterministic behavior in the program. Then, during replay, the recorded functions are not executed, and their results are synthesized from the logs [Arya 2017, Barr 2014, Barr 2016, King 2005, Montesinos 2008, O'Callahan 2017, Phang 2013, Pothier 2011, UDB 2023, Vilk 2018].

- To address non-determinism in multithreaded or multiprocess programs, the debuggers enforce *lock-based synchronization.* This is typically implemented with a shared lock for synchronization [Arya 2017, Barr 2014, Barr 2016, King 2005, Montesinos 2008, O'Callahan 2017, Phang 2013, Pothier 2011, UDB 2023, Vilk 2018]. Parallel and concurrent executions are forced to execute their instructions in a sequential manner. Then, the debugger records this sequential order. During replay, the debugger enforces the recorded order.

Next, we describe different works that, while not consisting of debuggers, propose mechanisms for deterministic execution replay. [Devietti 2009, Montesinos 2008, Hammond 2004, Herlihy 1993, Huang 2013] propositions focus on the techniques and schemes that could be used to implement deterministic replay features for debuggers.

Delorean [Montesinos 2008] offers a scheme to reproduce executions efficiently. Their proposition uses an approach that achieves the deterministic re-execution of every instruction block (chunk) for a time-travel operation in all the recorded threads, respecting the order of execution of these chunks.

CLAP [Huang 2013] achieves a deterministic final output of computations on every replay by preserving the causality of actions. The approach only records the interleaving events that, if replayed in a different order, would affect the outcome of computations.

Transactional Memory [Hammond 2004, Herlihy 1993] is a transaction-based alternative to lock-based synchronization. It is used to make multiprocess executions behave similarly to deterministic serialization while keeping parallelism. This technique is used by the work presented in DMP [Devietti 2009]. DMP handles multiprocessing non-determinism differently by performing deterministic thread or process interleaving upfront (during the first execution of the program) without the need to record logs or force processes to execute instructions one at a time.

2.4 Complementary Debugging Capabilities

Debuggers often offer specialized debugging capabilities. As described in Chapter 1, our proposition to improve interactive debugging problems involves combining time-traveling, scripting, and querying capabilities. In this section, we focus on the latter ones: *scripting* and *querying*.

2.4.1 Scriptable debugging

We refer to debuggers possessing scripting capabilities as *scriptable debuggers*.

These debuggers are tools that provide the capability for developers to automate and customize the debugging process using scripts. Scriptable debuggers offer an interface that allows developers to write scripts or code that interact with the debugger's functionality, enabling them to perform complex debugging tasks and analyses without manual intervention.

Using scriptable debuggers, developers can automate repetitive debugging tasks. This includes tasks such as setting up breakpoints, analyzing data, and running tests in a specific sequence. Developers can use scripting to perform analyses on program execution, identify patterns, extract specific information from the program's state, and perform custom actions when certain conditions are met. We describe next a few examples of scriptable debuggers.

GDB [Richard Stallman 2003] (GNU Debugger) allows developers to script and automate debugging tasks using the Python and GUILE[1] programming languages. LLDB (Low-Level Debugger), commonly used on macOS and certain Linux platforms, supports scripting using the Python programming language. Sindarin [Dupriez 2019] for Pharo offers a scripting API for the Pharo debugger that eases the expression and automation of different strategies developers pursue during their debugging sessions. Visual Studio is an IDE that supports various programming languages and provides its own debugger. By using the Development Tools Environment (DTE) libraries, developers can write programs or debugger extensions to automate certain debugging actions. WinDbg is a debugger for Windows and supports scripting using JavaScript.

2.4.2 Query-based debugging

Query-based debugging, also known as *interrogative debugging*, refers to using querying mechanisms to obtain specific insights about the program's execution [Ko 2004, Lencevicius 1997, Phang 2013, Torres Lopez 2021]. We describe next work that uses queries in the context of debugging.

[Lencevicius 1997, Lencevicius 1999, Lencevicius 2003] present query-based debugging solutions for verifying relationships among objects. [Lencevicius 1997] is built for the SELF programming language and [Lencevicius 1999, Lencevicius 2003] for Java. [Lencevicius 1997] defines a querying notation that is then used in [Lencevicius 1999, Lencevicius 2003]. The queries are used to express object relationships of a debugged program. When the expressed relationship is violated, the debugger breaks the execution, starting an interactive debugging session.

Expositor [Phang 2013] is a time-traveling debugger with querying support. One of their main contributions is their abstraction of the execution trace, which is a time-indexed sequence of program state views. Programmers can manipulate traces as if they were simple lists with operations such as map and filter. Queries are written using their own DSL.

STIQ [Pothier 2011] is a back-in-time debugger for Java that provides a scalable solution that is able to deal with large execution traces. The querying part of the work focuses on efficiently building and indexing the trace, which is used for querying execution data. There is no specialized query expressing mechanism in the proposal. To query program data, developers are required to programmatically iterate through the trace elements to find the debugging information they need.

[Goldsmith 2005] introduces PTQL, a SQL-like query language over program traces, and PARTIQLE, a special Java compiler. In their work, given a PTQL query and a Java program, PARTIQLE instruments the program to execute the query on the running program. Queries are used to log events when the query condition is

[1] More information available at gdb site https://sourceware.org/gdb/onlinedocs/gdb/Extending-GDB.html

met or to break and debug the program execution. The queries in this solution are executed independently without requiring a specialized debugger. The Java debugger is mentioned as a complementary tool that is open when certain query conditions are met.

Whyline [Ko 2004, Ko 2008] work presents post-mortem debuggers that provide contextual queries to aid in program comprehension for developers. These queries help with hypothesis formulation by taking the query-writing burden off of the developer. However, Whyline implementations are tailored to specific debugging targets (Alice [Alice 2004] and Java [Oracle 2023]), making it challenging to apply the solution in a different context.

[Torres Lopez 2021] presents a combination of interactive and query-based debugging to support finding concurrency bugs on actor-based programs. Their solution, like in Whyline, aims to fill the gap between the developers' interpretation of a failure and speculations of where the root cause of the bug is. Developers select questions from a set of predefined questions about the code and the program's execution. The questions were designed based on concepts of the actor model, *i.e.,* actors, turns, messages, and promises. The debugger then computes the answers by analyzing a recorded trace of events about the program execution.

2.5 Comparison of Debuggers' Features

Throughout this chapter, we have provided an overview of various debuggers that offer different debugging capabilities and debugging approaches. Although even more numerous solutions exist in the research landscape regarding these topics, we chose to focus on reviewing the ones that best represent the crucial aspects of this thesis.

In this section, we analyze the previously reviewed debuggers based on their features and their relevance to the two research problems outlined in Chapter 1. Regarding the first research problem, we investigate the *suitability of time-traveling debuggers for shared memory systems*. For the second problem, we have briefly introduced our proposed solution to enhance interactive debugging by combining *time-traveling debugging*, *scripting*, and *querying capabilities*. Therefore, in Table 2.1, we compare the reviewed debuggers in terms of the following debugging features: time-traveling, scripting, and querying. For time-traveling debuggers, we also include their compatibility with shared memory systems.

About compatibility with shared memory systems, [Arya 2017, Barr 2014, Barr 2016, King 2005, O'Callahan 2017, Phang 2013, UDB 2023, Vilk 2018] time-traveling debuggers are not designed for such a system (Table 2.1). The debugger in [Lencevicius 1997] runs in a shared memory system (SELF). However, it does not have time-traveling capabilities. In the same manner, Sindarin [Dupriez 2019] runs

Debugger	Time-traveling	Scripting	Querying	Compatible with Shared Mem. Systems
Expositor [Phang 2013]	✓	✓	✓	✗
FReD [Arya 2017]	✓	✓	✗	✗
Jardis [Barr 2016]	✓	✗	✗	✗
[Lencevicius 1997]	✗	✗	✓	N/A
[Lencevicius 2003]	✗	✗	✓	N/A
McFly [Vilk 2018]	✓	✗	✗	✗
RR [O'Callahan 2017]	✓	✓	✗	✗
Sindarin [Dupriez 2019]	✗	✓	✗	N/A
STIQ [Pothier 2011]	✗	✓	✓	N/A
Tardis [Barr 2014]	✓	✗	✗	✗
[Torres Lopez 2021]	✗	✗	✓	N/A
TTVM [King 2005]	✓	✗	✗	✗
UDB [UDB 2023]	✓	✓	✗	✗
Whyline 2004 [Ko 2004]	✗	✗	✓	N/A
Whyline 2008 [Ko 2008]	✗	✗	✓	N/A

Table 2.1: Debuggers features comparison. We compare the reviewed debuggers in terms of the following debugging features: time-traveling, scripting, and querying. For time-traveling debuggers, we include their compatibility with shared memory systems.

in a shared memory system (Pharo). However, it has no time-traveling capabilities. In this thesis, we aim to fill this void by presenting our time-traveling debugging solution for shared memory systems in Part I.

In the subject of scripting, [Arya 2017, Dupriez 2019, O'Callahan 2017, Phang 2013, Pothier 2011, UDB 2023] provide their own scripting API. While these debuggers allow for the automatization of debugging tasks, developers still need to learn their specific scripting API.

Regarding querying, the mechanism provided by [Lencevicius 1997, Lencevicius 2003, Phang 2013, Pothier 2011] involves expressing queries through DSL expressions. [Ko 2004, Ko 2008, Torres Lopez 2021] offer a different approach where the debugger suggests queries instead of developers expressing them. While easy to use and specialized for specific debugging scenarios, the approach does not allow developers to express new queries. In this thesis, we aim to improve on these aspects, seeking to minimize querying expression complexities due to the introduction of new DSLs while allowing developers to express new queries, which is addressed in Part II.

In terms of combined features, [Arya 2017, Barr 2016, O'Callahan 2017, UDB 2023] time-traveling debuggers offer scripting but not querying capabilities. Expositor is the only time-traveling debugger that combines time-traveling, scripting, and querying. Expositor combines scripting and time-travel debugging

to allow programmers to automate complex debugging tasks. Expositor uses UDB [UDB 2023] as an execution logging backend, which grants time-traveling capabilities. The querying mechanism allows developers to abstract from the scripting API and logs implementation. Developers use common list operations such as map and filter to obtain specific execution data. However, as discussed, developers still require knowledge of Expositor APIs and their queries DSL to write them.

2.6 Conclusion

To improve the debugging landscape, our thesis addresses two research problems (Chapter 1). First, we pointed out the lack of time-traveling debuggers for shared memory systems and set it out as our first objective to settle. Table 2.1 shows that none of the reviewed solutions are applicable to these kinds of systems.

Secondly, we pointed out the many difficulties of using standard interactive debuggers. We described how debugging features such as scripting, time-traveling, and querying can each be used to improve these problems. We briefly outlined our proposition, time-traveling queries, to solve these difficulties by combining these features. While Table 2.1 shows that there are debuggers presenting some of these helpful features, only one of them combines them all (Expositor). However, as we mentioned, developers still require knowledge of Expositor APIs and their queries DSL to write them. In our *Time-traveling Queries* proposal, we aim to improve on this issue.

Part I

Time-travel Debugging in Shared Memory Systems

CHAPTER 3

Selective Time-travel in Shared Memory

Contents

3.1 Introduction 29
3.2 Motivation 31
3.3 Problems of Imprecise Memory Scoping In Reversal Operations 33
3.4 Properties for Selective Time-travel 34
3.5 A Selective Time-travel Back End 35
3.6 Conclusion 38

Time-traveling debuggers are important tools for program comprehension and debugging. They allow developers to roll back a program execution to specific points in its past, helping them to reproduce subtle bugs and understand the causes of failures. However, implementations suffer from reversal imprecision in the presence of shared memory. In shared-memory systems, where multiple programs live in the same memory space, these imprecisions can lead to the reversal of excessive or insufficient memory. This hinders the applicability of time-traveling debuggers in these systems.

In this chapter, we present a comprehensive analysis of the problems encountered when applying conventional approaches to shared memory systems. We identify the properties of time-travel back ends that allow for precise scope reversal of executions in shared-memory systems and their deterministic replay. We propose a back end where the support for each property can be optionally enabled or disabled.

3.1 Introduction

A key challenge for time-traveling debuggers in shared memory systems is identifying a precise scope for time-traveling operations (Section 3.2). State-of-the-art time-traveling solutions scope the reversal to an entire process or system. These solutions rely on the capability of the system to create snapshots

of the debugged processes to implement checkpoints [Barr 2016, Barr 2014, O'Callahan 2017, Phang 2013, UDB 2023]. Therefore, they cannot be applied to a subset of the system execution, *e.g.,* individual threads running in the shared memory (Section 3.3).

Snapshots capture the entire application state (*i.e.*, both heap and thread state) and produce *over-reversals* when restored, *i.e.,* the reversal of more memory elements other than the ones of the debugged program. Moreover, attempts to reduce the reversal scope by manually or automatically anticipating which parts of the system state (*e.g.,* packages, classes, objects) are affected by an execution would produce *under-reversals*, *i.e.,* the reversal of fewer memory elements than the ones required to reproduce the program behavior.

In this chapter, we investigate the essential properties that a time-travel back end must possess to enable *selective time-traveling*, ensuring the precise scoping of reverse operations in shared-memory systems (Section 3.4). To this end, we designed a time-travel back end where the support for each property is configurable (Section 3.5).

Terminology

We define *shared memory systems* as systems or environments that act as hosts running multiple executions (*i.e.,* multiple programs) interacting with each other and using a shared memory environment provided by those hosts [Black 2009, Goldberg 1983, Ingalls 2008, Python 2023, Ungar 1987]. For example, Pharo's runtime [Black 2009] is used by developers to build and run all their programs in the same shared memory. The Pharo IDE, its libraries, tools, and end-user programs run in the same shared memory. Any program might read and write shared state and refer to objects from everywhere in the system.

We use the term *snapshot* to refer to a serialization of a complete operative system (OS) process, or a serialization of memory regions within the process, depending on the surrounding context. In object-oriented languages, this would correspond to the serialization of object graphs. A snapshot of a shared memory system process includes the complete state of the system, *i.e.,* it includes its heap and the stack of its runtime.

We define *reversal operation* as an operation that restores a program's state back to a past point in its execution history. We define *replay operation* as an operation that steps a program's execution, ensuring a deterministic and reproducible behavior. A *time-travel* operation consists of a combination of reverse and replay operations, and we use these terms without implicating any particular implementation or technique.

We use the term *time-travel back end* to refer to the underlying mechanism and its components that enable the time-travel functionality. It is essentially the part of the debugger that handles the reverse and replay of program executions.

3.2 Motivation

In this section, we present our motivation for a selective time-traveling solution by illustrating the problems of snapshot-based implementations in a shared memory system.

Time-travel debugging in shared memory systems

Let us consider the following example, illustrated in Figure 3.1. We debug a test case in Pharo, where the debugged program and the debugger run in the same memory space. We open the debugger and see the debugged program state.

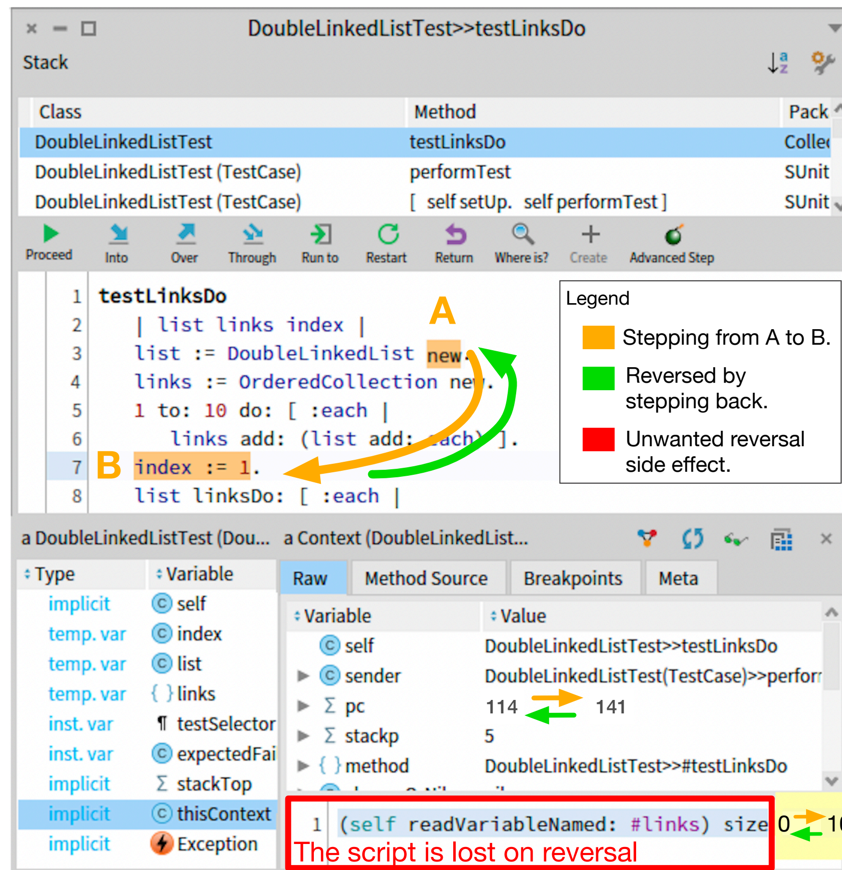


Figure 3.1: Debugger state before/after some manual steps. We write debugging scripts to obtain execution data and step the execution from PC 114 to 141, and with our script, we observe that the size of the collection changed from 0 to 10. While stepping back, the snapshot-based reversal mechanism reverses not only the debugged program but also the debugger state, erasing our debugging script.

To observe how the program evolves, we step through its instructions and write a script in the debugger (scripting area enclosed in red in Figure 3.1) to obtain

execution data. Then, we decide to step back the execution to observe a past execution state.

State-of-the-art time-traveling debuggers often rely on snapshots [Barr 2016, Barr 2014, Devietti 2009, King 2005, Montesinos 2008, O'Callahan 2017, Phang 2013, Pothier 2011, UDB 2023, Vilk 2018], but these snapshot-based solutions are *non-selective*. This means that when using them, the reversal operation reverses everything. Without the ability to selectively choose what to reverse, the debugger state is also reversed. In that case, we lose our debugging script, as the state of the scripting pane (and everything performed by the script) is also reversed by the snapshot.

This effect extends to any program running in Pharo. When reloading a snapshot, everything done in the system after that snapshot is reversed to the state captured by that snapshot. If breakpoints were placed during this debugging session, they would be lost. If we were listening to music with a multimedia player in Pharo, the current song would be interrupted, and the music player would reverse back to the song playing at snapshot time. If we made changes to the source code, they would be reversed as the code editor is also a program running within Pharo.

Therefore, using snapshot-based solutions, the state of the debugged program is always synchronized with the state of all programs running in shared memory systems. While this is a desirable property for some systems [Arya 2017], in Pharo (and in shared memory systems in general), we lose the power of our tools and programs, whose state cannot be separated from the state of the debugged program in snapshots. This means that snapshot-based solutions are not applicable in practice for implementing and using time-traveling debuggers in shared memory systems.

We need to be able to time-travel the execution of a debugged program while preserving the state of non-related programs running in the same system. Consequently, we require a back end that allows developers to:

1. Reverse the state of the thread of a particular execution and its effects while controlling undesired side effects of reversal operations, *i.e.,* having an appropriate scope for selecting and restoring memory state.

2. Perform deterministic replay of the thread to reproduce the debugged execution behavior consistently.

To the best of our knowledge, there are currently no time-traveling solutions designed specifically for shared memory systems. Our work explores this subject and suggests a solution that is tailored to meet the requirements of these particular systems. Furthermore, we discuss the key requirements for the development of such solutions.

3.3 Problems of Imprecise Memory Scoping In Reversal Operations

Without a precise memory scope for reversing an execution, reversal operations affect memory state in the system unrelated to the debugged program, producing side effects. In this section, we identify these undesired effects and describe how our work addresses scoping problems in shared memory.

3.3.1 Side effects

Over-reversing (OR) is a scoping problem that occurs when reversal operations reverse more than the required state. Consequently, the operation ends up affecting other elements in the shared memory system. This issue typically happens if a snapshot-based reversal approach is used since process restoration would affect all threads and memory elements.

Modifying memory used by other programs will introduce alterations in their execution, risking unexpected system behaviors (as in the example in Section 3.2) and potentially breaking their executions completely. The greater the excess of unrelated memory modified, the more alterations are introduced. Therefore, a reversal operation should restrict the scope of affected memory to the memory related to the debugged execution.

Under-reversing (UR) is a scoping problem that occurs when a thread reversal operation does not reverse enough state. The execution effects of the thread are not totally undone, which leads to an incomplete reversal. This happens, for example, if the reversal solution would reverse only the program counter of a stack frame.

An incomplete reversal scope will not reverse parts of the debugged program state. Consequently, when developers go backward, they reach a different past state than the one that happened. For example, when debugging a program that consumes a stream, if the reversal operation does not reverse the state of the stream, developers would be unable to correctly reproduce the program behavior as the execution would restart with the stream already consumed.

3.3.2 Towards selective time-travel

The premise of snapshot-based time-travel solutions is that debugged executions' dedicated memory is isolated from the rest system and contained within OS processes [Arya 2017, King 2005, O'Callahan 2017, Phang 2013, Savidis 2021, Wallaby.js 2023]. Such memory isolation for executions is not present in shared memory systems. To reverse a program to a previous state, snapshots are usually taken prior to the program's execution. Without clear isolation of a program's memory, any prediction of the affected system state made during snapshot creation may not be accurate due to the unpredictable changes performed by the program

execution. This applies indifferently whether it is a manual selection made by the developers or produced by a programmatic process. There is always the risk that the debugged execution will modify memory outside the selected snapshotted state, and the risk of not including all the appropriate state in the snapshot, which would lead to an incomplete reversal.

By our definition, a selective reversal scope ensures the reversal of the exact changes performed by a particular thread execution without producing OR or UR. To identify the precise scope for reversal operations, one could monitor every change in memory performed by the debugged program. This allows the reversal operation to affect precisely the changes made by the debugged program. Therefore, to overcome snapshot-based scoping imprecisions, we track all writings performed by the debugged program and ensure that reversal operations only affect memory within the scope.

However, this *writing-tracking* approach relates closely to implementation concerns and does not explain why other solutions cannot perform selective reversal operations. The approach also does not relate directly to the deterministic replay of a program, which is also an essential part of time-travel operations. Our work aims to define a basic framework that will enable us to connect these issues closely together. To this end, we followed the *writing-tracking* approach to identify the precise scope for operations as a starting point of our study. From there, we identified and studied a set of properties that time-travel solutions must possess to support selective time-traveling in shared memory systems. The properties are agnostic of the used approach or implementation details. These properties cover all aspects of time-traveling, including reversal and replay operations, on programs running in shared memory systems.

3.4 Properties for Selective Time-travel

In this section, we identify the essential properties that a time-travel back end must expose to support execution reversal and replay of single-threaded and multithreaded programs running in shared memory systems.

There are three distinct actions involved in time-traveling operations: reversing the state of the execution threads, restoring the changes made by their execution in the shared memory, and reproducing the same program behavior. To solve snapshot-based solution limitations, selective time-traveling operations require the back end to perform these actions on individual threads, *e.g.,* reversing the state of a thread should not reverse another thread's state. From these actions, we defined three properties required for selective time-traveling operations on single-threaded executions, and an additional one to support also multithreaded executions.

Property 1: Thread State Reversal (TSR). The time-travel back end is able to restore the state of a specific thread. This implies that other threads in the system

are left unaffected.

The time-travel back end must be able to recover the execution stack of a thread, *i.e.,* the thread state. For instance, reverting a thread's state to the beginning of a stack frame (*i.e.,* to the beginning of a function call) by restoring the local variables to their default initial value and the program counter of the stack frame.

Property 2: Execution Effects Reversal (EER). The time-travel back end is able to track changes in memory and undo the effects of the execution of a specific thread.

The execution of a thread performs modifications on the shared memory, having effects on the system memory. These effects must be reverted when performing a reversal operation.

Property 3: *Non-Determinism Sources* (NDS) Handling. The time-travel back end is able to enforce the deterministic replay of every instruction of a thread, reproducing the same thread execution behavior on every replay, considering a consistent initial state.

We consider instructions that potentially behave differently when being replayed as *NDS*. Replaying an execution that contains NDSs will produce program behaviors that follow different execution paths on each replay, making it non-reproducible. This impairs the capability to debug hard-to-reproduce failures [O'Callahan 2017]. Examples of NDS include typical system calls such as obtaining the system time, generating random numbers, and reading from a file or a socket.

Property 4: Deterministic Context Switching (DCS). The time-travel back end is able to execute every instruction of the concurrent threads of the debugged execution in a consistent order on every replay.

In concurrent systems running multiple threads, the interleaving of threads, and hence the execution order of the instructions of a multithreaded program, is usually non-deterministic. The host system decides the scheduling order to execute threads, which can create race conditions in the debugged program execution. This non-determinism source is a factor external to the instructions of the debugged execution. However, it must be handled by the time-travel back end to ensure deterministic execution replay.

3.5 A Selective Time-travel Back End

In this section, we propose a back end that supports selective time-travel for multithreaded programs. We designed a time-travel back end that allows for enabling or disabling support for each property individually. This configurable

design allows us to perform experiments to evaluate the impact of these properties on time-travel operations.

3.5.1 Time-travel back end overview

Our design comprises two types of components: The *Execution Control Unit* (ECU) and the *Thread Control Unit* (TCU). TCUs provide individual thread time-travel capabilities. There is one TCU for each thread of the debugged program's execution. A single ECU manages and coordinates the TCUs to enable time-travel operations on particular multithreaded execution.

Figure 3.2 illustrates a shared memory system running multiple executions. In this example, *Program 1* execution corresponds to the execution of the time-travel back end, and *Program 2* is the debugged program. The ECU controls *Program 2* execution by monitoring its corresponding threads (*Thread1* and *Thread2*) with individual TCUs. The figure also shows other programs running in the system, which are not monitored by the back end.

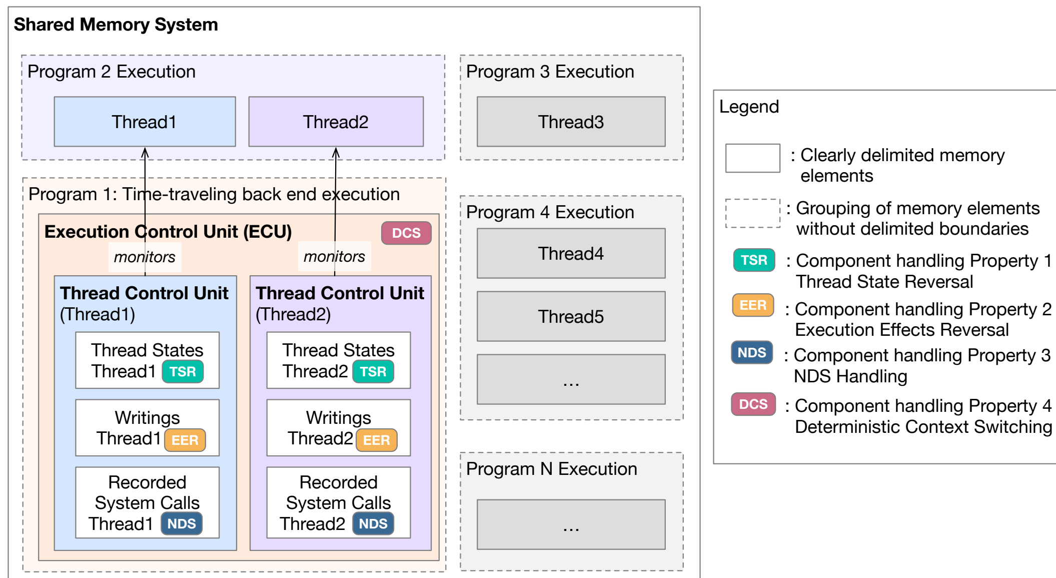


Figure 3.2: Back end design supporting multithreaded programs in shared memory systems. The figure shows which components support each one of the identified properties that enable selective time-traveling operations.

3.5.2 Back end components and properties support

The *Execution Control Unit* is the main component of the time-travel back end. To provide support for reverse and replay of multithreaded programs, it performs 3 essential tasks:

1. It generates and provides a program timestamp that serves to unambiguously identify the precise moment in the program execution history when events take place.

2. It enforces the order of thread interleaving (DCS support).

3. It coordinates actions required by each TCU to perform reverse and replay operations (TSR, EER, and NDS support).

The back end exhibits two distinct stepping behaviors based on whether the debugged program is being executed for the first time or during any subsequent replay. To control multithreaded program executions, during the first time executing the program, the ECU decides and logs which TCU will execute the next instruction. During replays, the interleaving order is reproduced from the logs, providing support for the Deterministic Context Switching property. When the support for DCS is disabled, no logs are produced.

During the first execution of the debugged program, the TCUs record program data. At each step, they capture and store execution data, including thread states, writing operations, and system calls returned values. All recorded data is logged along with the program timestamp corresponding to the moment of the capture. For each step performed by any TCU, the ECU updates the program timestamp, allowing every record to have a unique timestamp corresponding to an instruction of the program. During subsequent replays of the program, the TCUs synthesize parts of the execution using the recorded data. This synthesis process allows for precise and consistent execution replay, even in scenarios involving system calls.

For each step of the debugged program thread executed by the TCU, the current instruction is analyzed and processed following *the stepping pipeline* (Figure 3.3). Instructions processed by the pipeline will trigger the activation of corresponding back end components. First, for any instruction identified as a *writing operation*, the component handling EER records information to enable the capability of reversing the writing action. Then, if the instruction is recognized as a system call, the NDS component records information to allow for consistent replay.

The TCU enables the reversal and deterministic replay of a particular thread execution by coordinating the actions of its components, which we named after the properties they provide support for:

1. `TSR`: Provides support for the Thread State Reversal property. When enabled, this component stores the thread's state (*i.e.,* the execution stack of a thread) of a program during its execution, and restores the thread's state when a reversal operation is performed. When disabled, there is no recording of the thread's state, and the reversal leaves the thread unmodified.

2. `EER`: Provides support for the Execution Effects Reversal property. When enabled, this component logs and undoes writing operations performed

by the thread. When disabled, no writing operations are logged during execution, and consequently, writings are not undone during reversal operations.

3. `NDS`: Provides support for the NDS Handling property. When enabled, this component logs system calls executed by the thread, registering their return value so they can be deterministically replayed. When disabled, no logs are produced.

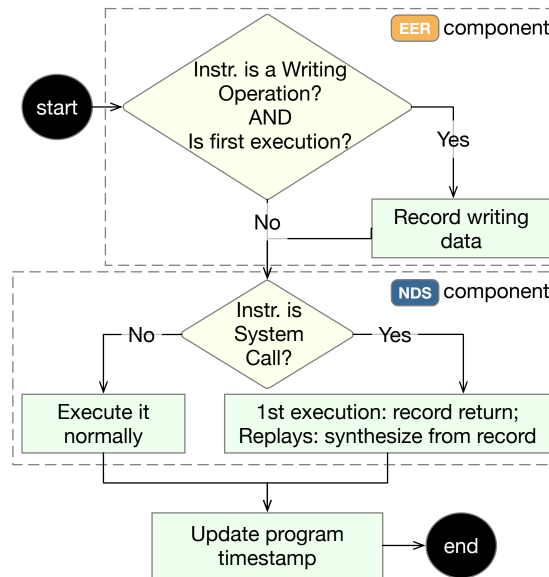


Figure 3.3: The TCU *Stepping Pipeline.* To offer the required time-traveling capabilities, the TCU advances a thread execution one instruction at a time *i.e.,* performing a step. For every step, the stepping pipeline is applied to ensure the appropriate reversal of the execution effects (EER) and deterministic replay of the execution (NDS).

3.6 Conclusion

Time-traveling debuggers are already part of the software industry. However, existing solutions are limited in their applicability, specifically when it comes to systems that involve multiple executions within a shared memory environment. This limitation arises due to the inherent reliance on snapshots as a fundamental component of the reversal mechanism.

In this chapter, we identified 4 *properties* that a time-travel back end should possess to overcome the challenging aspects of scoping reversal operations and for deterministic replay. From these properties, we proposed a selective time-travel back end design that is able to reverse and deterministically replay multithreaded program executions in shared memory systems while controlling the side effects on the rest of the system.

CHAPTER 4
Implementation

Contents

4.1 The Standard Pharo Debugger: An Overview 39
4.2 Seeker: A Time-traveling Debugger for Single-threaded Programs 41
4.3 Executor: A Time-traveling Debugger for Multithreaded Programs 54
4.4 Implementation Discussion 55
4.5 Conclusion 60

In this chapter, we present Seeker, a new time-traveling debugger for single-threaded executions in shared memory systems, and Executor, a time-traveling debugger prototype for multithreaded program executions.

We developed our time-traveling debuggers in Pharo, a shared memory system. We followed an incremental development process, starting with the creation of ***Seeker***. Its time-travel back end is specifically designed for single-threaded programs. It includes a user-friendly graphical user interface (GUI) and scripting capabilities. Our second implementation, ***Executor***, is built upon Seeker. Reusing Seeker's time-travel back end to handle each thread of a multithreaded execution, Executor serves as a prototype debugger for multithreaded programs in shared memory systems.

In the upcoming sections, we begin with an overview of the standard Pharo debugger to provide context for our implementations. Then, we will present the details of our implementations and how they meet the properties for selective time-travel for single and multithreaded programs.

4.1 The Standard Pharo Debugger: An Overview

In this section, we describe the standard debugging features, workflow, and the debugger user interface to contextualize our implementations.

The *StDebugger* (Figure 4.1) is an interactive debugger for Pharo programs. To start debugging and exploring a program, developers use breakpoints to automatically break the program at certain points in its execution. When a

breakpoint is hit, the StDebugger obtains control of the program execution, and its GUI is shown.

The debugger shows the Stack (Figure 4.1 a), a visual representation of what, in general terminology, is known as the execution call stack. Developers manually step through the execution by using the commands (Figure 4.1 b). After any debugging action, the code presenter (Figure 4.1 c) is updated and highlights the source code that will be executed in the next step. Developers observe execution data using the inspector (Figure 4.1 d) in addition to the stack.

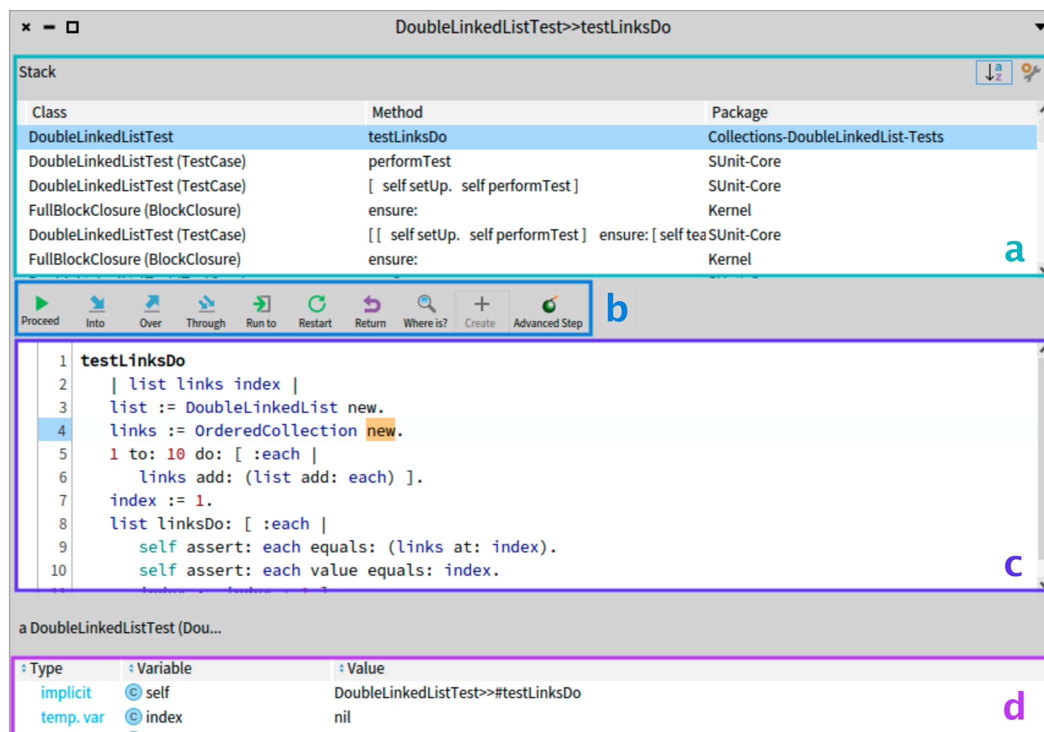


Figure 4.1: The Pharo StDebugger main GUI elements. Composed by the stack (a), the debugging commands toolbar (b), the code presenter (c), and the debugger inspector (d).

Standard debugging commands of the StDebugger

The debugging commands offered by the StDebugger in the commands toolbar (Figure 4.1 b) are the following ones:

Proceed: Resumes the normal execution of the debugged process until a new breakpoint is hit or the program is finished.

Into: Steps into method invocations. Interprets the bytecode instructions of the program until a message is about to be sent, a method is activated (*i.e.,* after

the message send), a method is about to return, or an assignment is about to be performed.

Over: Steps through the code, avoiding entering method invocations or closure evaluations. Interprets the bytecode instructions of the program. Stops only for the following conditions in the current `Context`[1] or its sender context: until a message is about to be sent, a method is about to return, or an assignment is about to be performed.

Through: Steps through the code, avoiding entering method invocations. It enters `BlockClosures`[2] defined in the current activated method. Interprets the bytecode instructions of the program. Stops when a message is about to be sent, a method is about to return, or an assignment is about to be performed in specific contexts. These contexts are: the current context, any `BlockClosure` context originating from the current context, or its sender context.

RunTo: Executes instructions up to the code under the caret or until the current method returns.

Restart: Goes back to the start of the context currently selected in the stack, typically the current executing one, reinitializing its local variables.

4.2 Seeker: A Time-traveling Debugger for Single-threaded Programs

In this section, we describe our time-traveling debugger that integrates into the standard Pharo debugging workflow while adding support for selective reversal and deterministic replay for single-threaded programs.

4.2.1 Seeker, in a nutshell

Seeker is an interactive scriptable time-traveling debugger for Pharo. It integrates seamlessly with the standard debugger, allowing developers to debug live program executions using new time-travel capabilities. With Seeker's scriptable API, debugging tasks can be automated, eliminating the need for many time-consuming manual actions.

[1] In Pharo, a `Context` is an object representing what is commonly known as a stack frame, containing data of the execution state of a method and a pointer to the calling frame (the sender `Context`)

[2] In Pharo, a `BlockClosure` is an object that represents closures, used as anonymous functions or code blocks.

Seeker can be executed embedded in the standard debugger *GUI Mode* or as a standalone programmatic debugger *Headless Mode*. GUI Mode provides a graphical interface that is complementary to the standard debugger, granting access to the new time-travel features. Headless Mode allows developers to programmatically control the debugger, enabling scripting and automatization of repetitive debugging tasks.

4.2.2 GUI Mode: Time-travel for the standard debugger

As the standard debugger for Pharo, the *StDebugger* is the main debugging tool available for Pharo developers. StDebugger users are already familiarized with its GUI and usage, and therefore, we wanted to include the new features without disrupting their usual debugging workflow. For this, we added Seeker GUI (Figure 4.2) using the Pharo debugger extension system.

4.2.2.1 Seeker enhanced debugging commands

Our reversal mechanism relies on the recorded data of a program *i.e.,* a *trace*, to perform reversal operations. This trace contains records of writing operations performed by an execution. The trace remembers values before assignments or methods that perform writing so the modified values can be restored later. The standard debugging commands advance the program without generating such a trace. Consequently, Seeker rewrites these commands, ensuring that the trace is produced when the execution is advanced. These are the differences over the standard debugging commands.

Proceed (Overridden): In addition to the original behavior, it also deletes recorded program data produced by the time-travel mechanism.

Into, Over, Though (Overridden): They show the original behavior. In addition, and transparently to the user, the debugger steps the bytecodes one by one. On each step, the trace is created, and a timestamp is generated by counting the number of executed bytecodes.

RunTo (Overridden): In addition to the original behavior, the enhanced RunTo can also target code that was already executed. Developers time-travel backward in time by placing the caret on any visible code in any of the contexts in the stack.

Restart (Overridden): In addition to the original behavior, the enhanced Restart also adjusts the counter of executed bytecodes and reverts any changes performed by the execution since the activation of the selected context. This includes reverting any changes performed by the debugged program on the method arguments, objects instance variables, globals, etc.

4.2.2.2 New time-travel debugging commands

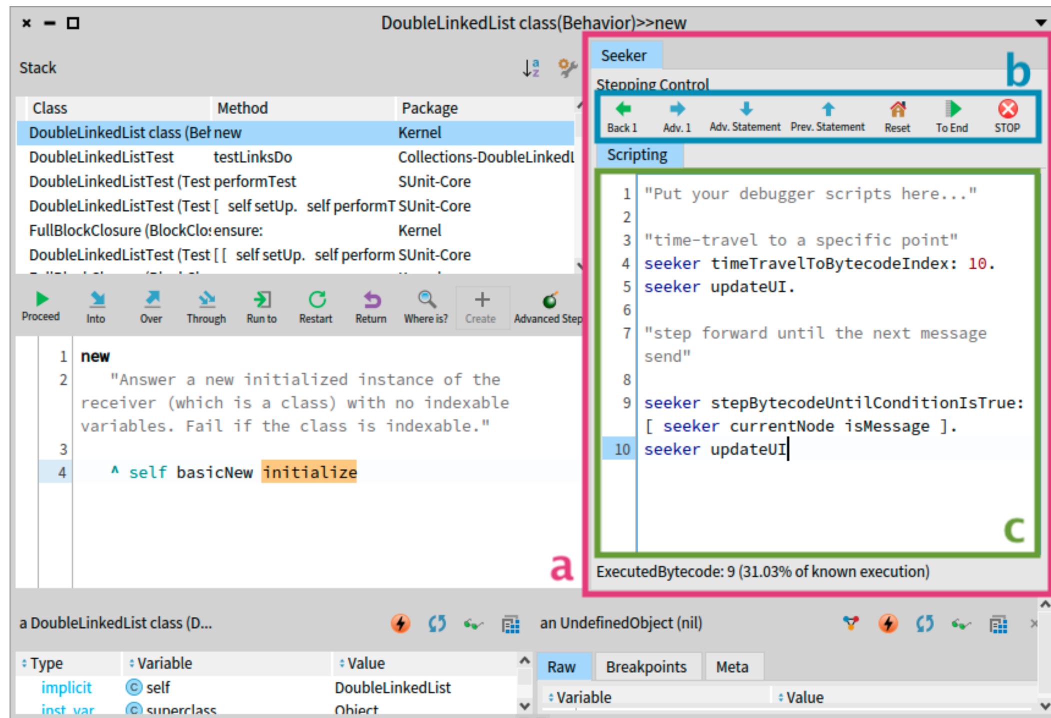


Figure 4.2: Seeker GUI as an extension of the standard Pharo debugger (a). The GUI includes the new time-travel commands (b), and the scripting panel (c).

Seeker includes a new set of time-travel debugging features, which are also included in the GUI (Figure 4.2 b):

1. **Back 1:** Reverses the execution state to the state it was before the execution of the last executed bytecode.

2. **Adv. 1:** Interprets one bytecode instruction of the program and stops.

3. **Adv. Statement:** Advances the execution until reaching a different statement in the current context.

4. **Prev. Statement:** Reverses the execution to the previous statement in the current context.

5. **Reset:** Executing a Reset command is equivalent to performing a Restart command while selecting the program's initial context. Reverses the execution by restarting the initial context of the program while undoing all the changes performed by the execution. Resets the counter of executed bytecodes to 0.

6. **ToEnd:** Automatically steps through the bytecodes, stopping right before the step that would terminate the execution.

7. **STOP:** A fail-safe mechanism to interrupt debugging scripts that take too long to complete or that might never finish. If developers write and execute a debugging script with an infinite loop of steppings or time-travels, they can press the STOP button. A flag is then set to stop stepping iterations forcefully. With this feature, developers can regain control of the debugger without the need for more extreme measures, such as closing and reopening the debugger or manually terminating the process.

4.2.2.3 The scripting panel

Seeker includes a code presenter where developers can write their debugging scripts (Figure 4.2 c).

The code presenter offers the `seeker` variable, which is bound to the `SeekerDebugger` object, and developers use it to programmatically manipulate the debugger during their usual GUI-based debugging workflow.

4.2.3 Debugging programmatically with Headless Mode

All debugging commands and features of GUI mode are also available in headless mode.

Developers use headless mode to code routines to automate debugging tasks, as they do with typical scriptable debuggers. The difference is that Seeker offers the means to reverse executions and their effects on the global state, giving developers the tools to observe programs behavior and their effects on the system at any point in their execution history.

```
1 observedObject := Smalltalk globals at: #aGlobalObject.
2 seeker := SeekerDebugger headlessDebugBlock: myProgramBlock.
3 [seeker executionIsFinished] whileFalse: [
4   (seeker currentState isMessageSend and: [
5     seeker currentState messageReceiver == observedObject ]) ifTrue: [
6       transcript show: 'method called for observed object: #' ,
7         seeker currentState methodAboutToExecute name;cr]. "prints the method name"
8          seeker stepBytecode]
9 seeker restart "reverses the execution and its effects"
```

Listing 4.1: Example of using the headless mode to debug a program. We use the debugger to automatically step the program while printing a message when any method of a particular object is called. Then, we reverse all changes performed by the program.

In the example in Listing 4.1, we want to observe how a program interacts with a particular object. First, we store the object in a local variable (Line 1). `myProgramBlock` is a closure with a program that calls methods of the object.

We start a programmatic debugging session by initializing Seeker to debug the program in headless mode (Line 2). To observe how the program interacts with the object, we write a debugging script to list the called methods (Lines 3 to 8). We use the `currentState` object to acquire execution data of the program. If the program state is a message-send (Line 4) and the receiver of the message is our observed object (Line 5), we log the method name (Lines 6 and 7). Then, we step the execution once (Line 8), and the script is repeated until the program ends. After listing the information, we restart the program by using the `restart` method (Line 9). This reverts the changes produced by the execution of the program, leaving the debugger and the program ready to replay the execution. This allows developers to run more debugging scripts to collect additional execution data.

4.2.4 Seeker implementation

We describe our debugger main class and its integration with the StDebugger. Finally, we describe how we implemented each component of the selective time-travel back end design.

`SeekerDebugger` is the main class of our debugger. It provides an API to perform programmatic time-travel debugging operations on a program process (Table 4.1).

Method	Description
currentState()	Returns a CurrentState object, which is used to obtain execution data of the debugged process.
restart()	Performs the restart operation.
stepInto()	Performs a stepInto operation.
stepOneBytecode()	Executes the next bytecode of the debugged process, advancing the debugged program.
stepOver()	Performs a stepOver operation.
stepThrough()	Performs a stepThrough operation.
stepUnitConditionIsTrue:(Block)	Automatically steps the program until the argument block evaluates to `true`.
timeTravelToStepNumber:(int)	Performs a time-travel (reversing or advancing the program) to the specified step number.

Table 4.1: Methods of the debugging API of SeekerDebugger, the main object of our time-traveling debugger implementation.

`SeekerDebugger` class is used in both modes: Headless and GUI. To initiate a headless session, developers initialize the debugger object using any of its initialization methods (Table 4.2). For GUI mode sessions, Seeker uses the *plug-ins* system of the Pharo debugging framework to embed itself in the StDebugger view, and its integration is accomplished by the `SeekerDebuggerPresenter` class (Figure 4.3).

Class-side Method	Description
headlessDebug:(Process)	Returns a SeekerDebugger object to debug the execution of the suspended process in the argument.
headlessDebugBlock:(Block)	Returns a SeekerDebugger object to debug the execution of the block argument.
headlessDebug:selector:withArgs: (Object, Symbol, Array)	Returns a SeekerDebugger object to debug the execution of a method call of a specific object with the specified selector and the provided arguments.

Table 4.2: Class side methods of the initialization API of SeekerDebugger.

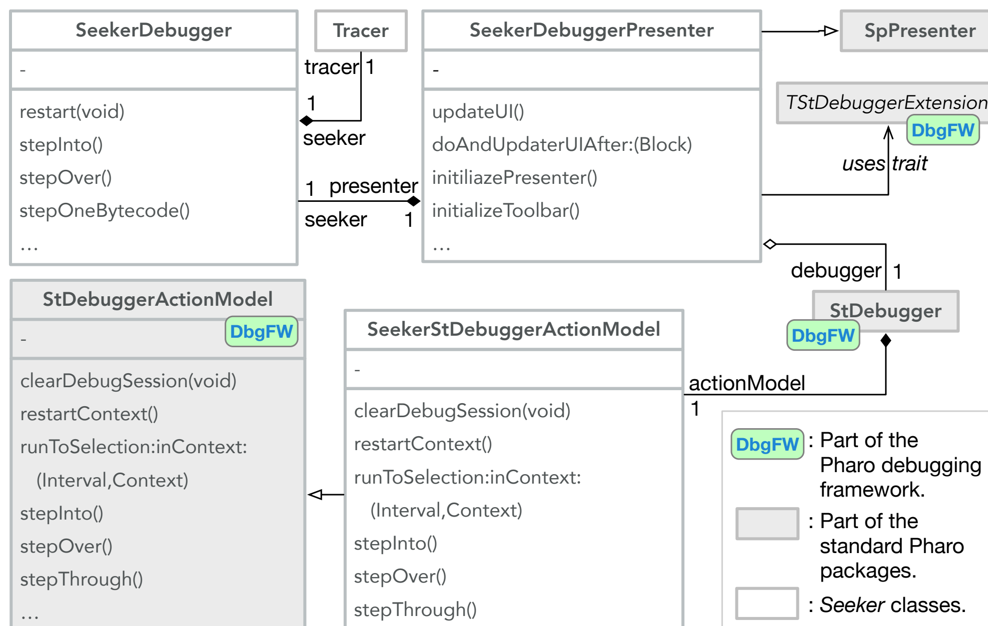


Figure 4.3: Integration of Seeker as an extension plug-in of the StDebugger. *DbgFW*: Classes belonging to the Pharo debugging framework in the standard packages.

`SeekerDebuggerPresenter` defines the graphic elements of the user interface (*i.e.*, the toolbar with the new time-travel commands and scripting panel) and its integration in the StDebugger graphic interface.

The debugger initialization events take place in sequence. When the StDebugger is initialized, it initializes a `SeekerDebuggerPresenter` object. Then, the presenter creates a `SeekerDebugger` object initialized to perform time-travel debugging operations on the same debugged process handled by the StDebugger.

We ensure that the UI is always synchronized with the actual debugging session state. Whenever a debugging operation is performed (*e.g.*, a time-travel or program restart), `SeekerDebugger` notifies the `SeekerDebuggerPresenter` object. Then, the presenter notifies the StDebugger that the display needs to be

refreshed.

SeekerStDebuggerActionModel is the class we implemented to override the standard debugging commands and make them compatible with the time-travel mechanism, as described in 4.2.2.1.

While SeekerDebugger exposes the public API to access time-travel functionalities, the time-travel logic is contained in the classes composing the time-travel back end, which we describe next.

4.2.5 Components of the time-travel back end

We describe here in detail the implementation of the time-travel back end and how each component provides support for the identified properties for selective time-travel of single-threaded executions.

Tracer: In Seeker implementation (Figure 4.4), the Tracer corresponds to the main class of the *Thread Control Unit* described in Chapter 3, Section 3.5.

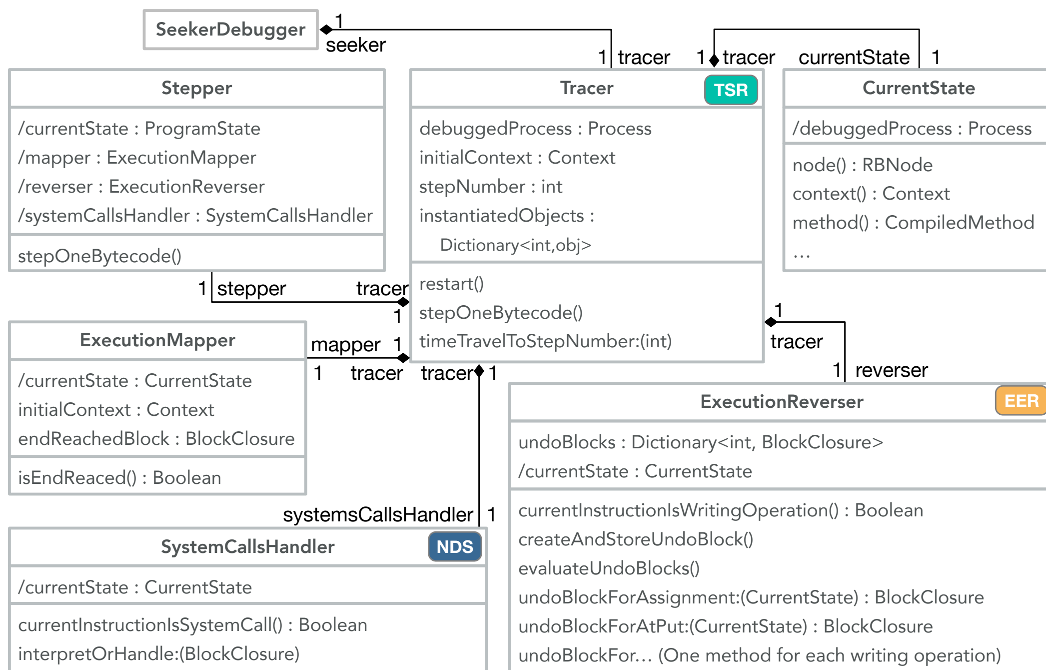


Figure 4.4: The time-travel back end of Seeker, where the Tracer is the TCU controlling the other components. *TSR*: component handing Thread State Reversal. *EER*: component handing Execution Effects Reversal. *NDS*: component handling Non-Determinism Sources.

As the main object of the back end, the Tracer coordinates the other TCU components to create a program trace whose records are used to reverse the debugged program and also to perform the deterministic replay of the program.

During Seeker initialization, the Tracer object is instantiated and initialized. When initialized, it stores the debugged process in an instance variable. It also stores a reference to the top `Context` of the suspended process in the `initialContext` variable. Additionally, it sets the `stepNumber` (which is used as a timestamp) to zero.

CurrentState: `CurrentState` objects offer a high-level and uniform API to access execution data (Table 4.3, Table 4.5, and Table 4.4).

Method	**Description**
arguments() : Array	Returns an array containing the objects in the arguments of the current activated method.
context() : Context	Returns the suspended `Context` object, *i.e.,* the context of the activated method at the top of the stack of the debugged process.
isAssignment() : Boolean	Returns true if the current bytecode instruction is an assignment, false otherwise.
isMessageSend() : Boolean	Returns true if the bytecode instruction is a message send, false otherwise.
method() : CompiledMethod	Returns the CompiledMethod of the suspended context.
node() : RBProgramNode	Returns the AST node corresponding to the current bytecode instruction of the suspended context.
receiver() : Object	Returns the receiver object of the current activated method context in the top of the stack.
receiverClass() : Class	Returns the class of the receiver object.
receiverClassName() : Symbol	Returns the name of the class of the receiver object.
receiverPackage() : RPackage	Returns the object representing the package of the receiver class.
selector() : Symbol	Returns the selector of the activated method of the suspended context.
willCreateBlock() : Boolean	Returns true if the current bytecode instruction will create a `BlockClosure`.
willReturn() : Boolean	Returns true if the current bytecode is a return instruction.

Table 4.3: Methods of the CurrentState API to obtain general execution data of the debugged program.

The execution data available through the CurrentState API is obtained from the debugged process using Pharo reflective methods, as shown in Listing 4.2. The `context` method (Lines 1 and 2) returns the suspended `Context` object, i.e., the context of the activated method at the top of the stack of the debugged process. The rest of the listed methods use internally the `context` method to obtain other execution information.

```
1  CurrentState >> context
2    ^ debuggedProcess suspendedContext
3
4  CurrentState >> method
5    ^ self context method
6
7  CurrentState >> node
8    ^ self method sourceNodeForPC: self context pc
9
10 CurrentState >> selector
11   ^ self method selector
12
13 CurrentState >> receiver
14   ^ self context receiver
```

Listing 4.2: Code extract of the implementation of the CurrentState class, showing how the debugged program data is retrieved from the debugged `Process` object.

CurrentState objects simplify the task of retrieving execution data by offering a selection of simple accessor methods that hide most of the complexities of Pharo reflective API. Developers using CurrentState do not need to know the complex and lower-level internals of the interpretation model. For example, if the debugged program is about to execute a method, to obtain the method that will be executed, developers invoke `CurrentState » methodAboutToExecute` (Table 4.4), which will return the corresponding compiled method. To achieve the same without using CurrentState, developers need to write knowledge-demanding routines such as the one in the code described in Listing 4.3, Lines 8 to 14.

Method	Description
isInstantiationMessage : Boolean	Returns true if the method about to execute is an instantiation primitive.
methodAboutToExecute() : CompiledMethod	Returns the CompiledMethod that will be executed by the message-send.
messageReceiver() : Object	Returns the object that will receive the message.
messageSelector() : Symbol	Returns the selector of the message being sent.
messageArguments() : Array	Returns an array with the arguments of the message.
classAboutToBeInstantiated() : Class	Returns the class of the object that will be instantiated by the instantiation message.

Table 4.4: Methods of the CurrentState API to obtain instruction-specific program execution data related to message-sends.

The CurrentState API prevents developers from writing repetitive and error-prone code. For example, to access the receiver object of a message-send without using CurrentState, developers need to calculate the offset of the object position in the context, as it is done in Listing 4.3, Line 5. The calls in such a line of code do not effectively portray the operation the code is trying to achieve, risking the usage of heuristics or incorrect method calls, producing a wrong result.

```
CurrentState >> isMessageSend
 ^ self node isMessage

CurrentState >> messageReceiver
 ^ self context at: self context basicSize - self node arguments size

CurrentState >> methodAboutToExecute
 | methodAboutToExecute msgNode messageReceiverClass |
 msgNode := self node."a RBMessageNode"
 messageReceiverClass := self messageReceiver class.
 msgNode receiver isSuperVariable ifTrue: [
   messageReceiverClass := messageReceiverClass superclass ].
   methodAboutToExecute := messageReceiverClass lookupSelector: node selector.
 ^ methodAboutToExecute
```

Listing 4.3: Code extract of the CurrentState class, showing the implementation of methods to obtain execution data related to message-sends.

The CurrentState API also exposes methods to obtain program execution data related to assignment instructions (Table 4.5).

Method	Description
assignmentVariable() : Variable	Returns the `Variable` object in the left side of the assignment.
assignmentVariableName() : Symbol	Returns the name of the assignment variable.
assignmentCurrentValue() : Object	Returns the current value of the assignment variable, *i.e.,* before the assignment occurs.
assignmentNextValue() : Object	Returns the value about to be assigned to the variable.

Table 4.5: Methods of the CurrentState API to obtain instruction-specific program execution data related to assignments.

Stepper: The `Stepper` controls the `ExecutionReverser` to revert the effects of execution. The stepper also uses the `SystemCallsHandler` component to enforce deterministic replay. This corresponds to the implementation of the stepping pipeline described in Chapter 3, Section 5.

ExecutionMapper: Provides a safety mechanism to prevent performing steps that would terminate the execution.

Since the debugger can only reverse the state of live executions, we need to ensure that the steps we perform will not terminate the process. If an execution is advanced carelessly, it might produce the termination of the debugged process. For example, in Pharo, the origin context of a *DoIt* execution evaluates the selected code first and then evaluates code that terminates the process. When our debugger detects a DoIt execution, the ExecutionMapper creates a block that evaluates to true if the next instruction is the one that terminates the process. This block, named

`isEndReached`, is used by the Stepper to decide if steps can be performed or not.

Additionally, the ExecutionMapper provides a safety mechanism to prevent restarting an execution from an *unsafe context*. We consider *safe initial context* as the ones that will not trigger the termination of the process on restart, due to Pharo `BlockClosure»ensure:` and unwinding mechanisms. When developers start a time-travel debugging session, the ExecutionMapper ensures that a safe initial context is chosen as a restart point for the debugged program. If an unsafe context is detected, the execution mapper automatically takes a few steps forward until it detects a safe context at the top of the stack and sets it as the new initial context. When the execution mapper updates the initial context this way, it also updates the `isEndReached` block to prevent returning from the new safe initial context.

ExecutionReverser: Its role is to reverse the effects of a thread execution, which corresponds to the writings performed by *writing operations*. Writing operations consist of assignments, writing primitives, and specific other writing methods (*e.g.*, file system writing). When a writing operation is detected during a stepping operation, the `ExecutionReverser` component creates and stores `BlockClosures` that, when evaluated, they undo the change performed by the writing. We call these closures *UndoBlocks*.

Not all writing operations are recorded. We register only writings to objects that were not instantiated by the debugged thread. We avoid logging writings performed to local variables of the thread stack, and to objects instantiated by the debugged process. As our implementation reversal always restarts the execution from the beginning, local variables always have the default value and objects are also initialized with default values during replay.

4.2.6 Seeker time-travel operations explained

In the following, we describe how the components of the back end execute deterministic time-travel operations.

4.2.6.1 Performing an execution reversal

To reverse the state of the debugged process, all contexts are popped off from the execution stack until the initial context reaches the top of the stack. Afterward, the initial context is reset to its default values. As a result, the process state is reinitialized with the reinitialized initial context on top of the stack, which satisfies property 1 (Thread State Reversal). The effects of the process execution are reversed by undoing writing operations recorded by the `ExecutionReverser`.

Undoing writing operations: To undo the effects of an execution, the `ExecutionReverser` evaluates the stored undo blocks in reverse order. This

undoes all the thread execution effects, which satisfies property 2 (Execution Effects Reversal).

Creating undo blocks for writing operations: We create undo blocks dynamically when a writing operation is executed. Our implementation supports assignments and writing primitives. Each primitive and each specific writing method are handled as a different case.

For each different writing case, we developed a specific method that creates an undo block for that case. When we detect a writing operation, the corresponding specific method is executed before the writing is performed, and uses execution data to build an undo block to revert the writing operation.

To illustrate how the undoing works, let us consider the following example that shows how a particular writing operation is undone. We will focus on the writing primitive `Object»at:put:`, which stores a value in the receiver at the indicated indexable element position (similar to performing `receiver[anIndex] = aValue` assignments on arrays in many languages).

Let *obj* be an object and its fourth indexable element value is 0. The debugged program is about to execute the writing operation `obj at:4 put:2`, which is detected by our back end.

Listing 4.4 shows the method that builds `UndoBlocks` when the `Object»at:put:` writing primitive is executed. The method takes as a parameter a CurrentState object to access the current execution data of the writing operation (Line 1). From that execution data, we recover the receiver that will be written to (Line 3), the memory index to which a new value will be written (Line 4) and the original value located at that memory index before writing (Line 5). In our example, these values are `obj`, `4` and `0` respectively.

We then create an undo block (Line 6) that writes back the original value to the memory index. On a reversal operation, the ExecutionReverser evaluates the produced undo block, which performs `obj at:4 put:0`, reversing the writing.

```
1 ExecutionReverser class >> undoBlockForAtPut: programState
2   | receiver index originalValue |
3   receiver := programState messageReceiver.
4   index := programState messageArgumentAt: 1.
5   originalValue := receiver at: index.
6   ^ [receiver at: index put: originalValue] "<-undo block"
```

Listing 4.4: Undo block creation method for `Object»at:put:` writing primitive. The creation method is executed before the writing is performed.

4.2.6.2 Performing deterministic replay

We explain here how our implementation provides deterministic replay by handling system calls, satisfying property 3 (NDS Handling).

Identifying NDS: We manually identified a set of methods that can introduce a NDS. When going through the stepping pipeline, each instruction is analyzed to detect calls to these methods.

Enforcing deterministic replay: Whenever a system call is about to be executed, two courses of action can be taken depending if the program is running for the first time or if the program is replaying that first execution:

1. When running the first time: the instruction is executed, and its return value is logged.
2. When replaying: the instruction is skipped by increasing the program counter of the current context, pointing to the instruction immediately following the message-send. The return value is synthesized by writing the logged value to the current context at the position where the real return value should be set.

Instantiation primitives and non-deterministic object identities: When Pharo objects are instantiated, they are assigned an identity hash by the VM. Re-executing instantiation instructions creates objects with new identity hashes on every replay. Because identity hashes are non-deterministic, certain data structures (*e.g.*, identity sets) that collect these objects will store them in a different order on each replay. This problem then manifests as an NDS when these data structures are iterated.

We handle this NDS as a particular system call. During the first play, all instantiated objects are registered by the `TCU`, and we add an entry to the NDS Handling log. This entry is composed of the instantiation timestamp and of a closure that, when evaluated after an instantiation, will enforce the recorded identity on the new instantiated object. During replay, if an instantiation instruction is detected, it is executed as usual, generating a new instance with a non-deterministic identity hash. Then, we evaluate the logged closure to enforce the object identity to match the one recorded from the first execution.

4.2.7 Configurable properties support

To provide configurable support of properties, each responsible component possesses a slot named `enabled`. Disabled components will not produce any records of the program, prevent any reversal action of the component, and will always perform the *first execution* behavior, even during replays. In particular, when support for TSR is disabled, the Tracer will not perform the reversal of the debugged process for reversal operations. When support for EER is disabled, the ExecutionReverser will not produce writing logs and will not execute any undo block for reversal operations. When support for NDS is disabled, the SystemCallsHandler will not produce logs and, during replay, will always execute the system calls.

4.3 Executor: A Time-traveling Debugger for Multi-threaded Programs

Executor is a prototype time-traveling debugger for shared memory systems, designed specifically for multithreaded programs. Executor does not incorporate a graphical user interface and therefore is used programmatically.

4.3.1 Executor implementation

Executor re-uses the totality of the classes of Seeker. It introduces a new `Executor` class and adds a new slot to the Tracer class. The class `Executor` corresponds to the ECU of the design to support multithreaded program.

Executor controls each thread of a multithreaded program with a dedicated `TCU`. While each `TCU` contains its individual `stepNumber` (thread instruction counter), Executors's `timestamp` is a shared counter that is increased when any of the `TCU` step number is increased.

4.3.1.1 Executor in action

To debug programs, Executor exposes an API to perform essential time-travel debugging operations through the methods: `step`, `restart`, and `timeTravelTo:`.

At the time of writing, our prototype does not support programs that dynamically create threads. As a result, developers must manually create TCUs for each thread in the debugged program in order to start a debugging session using Executor.

When developers call the `step` method, Executor performs one step (a single bytecode instruction) in only one of the `TCU`s. This ensures that the instructions in the threads are executed sequentially.

During a stepping operation, Executor randomly selects a TCU to advance by one instruction, assuming equal priority for all threads. The index of the chosen thread is recorded in the logs of Executor for the current `timestamp`. This selection process occurs only during the first execution of the debugged program. During replay, the choice of which thread to advance is retrieved from the logs, adhering to the property of Deterministic Context Switching (Property 4).

On every Executor step, the `timestamp` is increased by one, and only the stepped `TCU` increases its step number by one. Additionally, we adapted the Tracer class to hold additional data that is required by Executor for multithreaded time-travel operations. The Tracer now registers on each step, the correspondence of its step number and the ECU timestamp.Invoking the `restart` method resets the execution of the debugged multithreaded program. It resets the ECU `timestamp` and utilizes each TCU to reverse their respective thread states and execution effects.

The TCUs thread state reversal is performed in the same manner as in Seeker. However, the execution effects reversal is handled differently. To reverse the effects of a multithreaded execution, Executor makes a new list of undo blocks by merging the blocks of every TCU. These undo blocks are then sorted by their corresponding ECU timestamp. As each TCU labels its undo block using the TCU step number, Executor uses the logged correspondence of the TCUs step number to find the corresponding ECU timestamp. Then, Executor evaluates the undo blocks of the merged list in reverse order, reversing the effects of the multithreaded execution.

The `timeTravelTo:` method is implemented by first invoking the `restart` method of Executor, followed by stepping Executor until the target program timestamp is reached.

4.3.2 Configurable properties support

To provide configurable support of properties, the same behavior described in Subsection 4.2.7 is applied to every TCU. When support for DCS is disabled, Executor will not record the interleaving order of threads and, during replay, will always decide the interleaving order randomly.

4.4 Implementation Discussion

In this section, we provide further notes to complement our implementation description. We elaborate on the decisions we made during the implementation process, including considerations and alternatives. Furthermore, we address the limitations of our prototypes.

4.4.1 Recorded data overview

This is the summary of the data logged by our implementations.

- **Thread interleaving order:** Recorded by the `ECU` to enforce deterministic context switching.

- **No thread state recorded:** Once a `TCU` is initialized, we store only a reference to the initial `Context`. We use the `Context` API to reset it, recovering its local variables initial values.

- **Instantiated objects by each thread:** Each `TCU` registers instantiated object identities and the corresponding instantiation timestamp to enforce their deterministic replay. The registered identities are also used to identify if the object was instantiated by the thread and decide if a writing operation should generate or not an `UndoBlock`.

- **UndoBlocks:** For every writing operation to objects instantiated by other threads, an `UndoBlock` is registered by the `TCU` along with the corresponding timestamp.
- **Non-deterministic system calls:** We log non-deterministic system calls return value and the corresponding timestamp for deterministic replay.

4.4.2 Validation of the selectiveness of our implementation

We argue that our implemented reversal mechanism is selective *i.e.,* it does not produce Over-reversal or Under-reversal. To the best of our knowledge, there is no standard conclusive mechanism to prove such a statement. Instead, we support this claim by explaining and providing evidence on how our implementation does not produce over or under-reversal within the boundaries of its limitations (Subsection 4.4.5).

To address OR concerns, we show that our reversal mechanism does not affect unnecessary state *i.e.,* state that was not modified by the program execution. To address UR concerns, we show that the scope of our reversal mechanism is the minimum that includes all the state affected by the program execution *i.e.,* excluding anything else from the scope would lead to an incomplete reversal since some state modified by the program would not be reversed.

As discussed in Section 3.4, the reversal operation involves reverting the thread state, and the execution effects (*i.e.,* writings to the global state). We analyze both of these actions individually in terms of OR and UR concerns.

4.4.2.1 No OR for thread state reversal.

To avoid OR, the reversal operation scope should include the state modified by the process execution or less. To address this concern, we list all the elements affected by the *thread state reversal* operation and then verify that they were all modified by the execution.

In Pharo, the thread state is the state of the Process object that is executing the program. Process objects contain the following slots: `effectiveProcess`, `env`, `level`, `myList`, `name`, `priority`, `suspendedContext`, and `terminating`.

In 4.2.6.1, we outlined how our implementation reverses the thread state. This is achieved by calling the `popTo:` method of the debugged process and passing the initial context stored by the `Tracer` as an argument. As a result, the `suspendedContext` of the process is set back to the initial context. Therefore, `suspendedContext` is the only slot of the process affected by the reversal operation.

Afterward, the method `privRefresh` of the initial context is called, which reinitializes the context to default values. Context objects contain the following

slots: `sender`, `pc`, `stackp`, `method`, `closureOrNil`, and `receiver`. The execution of the program modifies only the `pc` and `stackp` slots of the initial context, and for reversal, calling the `privRefresh` method of the context resets the values of these slots, and does not affect anything else.

In summary, our thread state reversal mechanism reverses specifically the suspended context slot of the process and the aforementioned slots of the initial context. Since all these elements are modified by the execution, there is no OR.

4.4.2.2 No OR for execution effects reversal

In a similar manner to the previous point, to address this concern, we list all the elements affected by the *execution effects reversal* operation and verify that they were all modified by the execution.

Our implementation steps through every bytecode instruction of the debugged program. If a writing operation is performed by the program, the instruction will go through the stepping pipeline. We manually identified a set of writing operations in Pharo, which are detected during the pipeline.

We detect these writing operations, producing and registering actions to undo each particular change. This means that by construction, there is no potential OR as the undo actions write back exactly what was written in the first place by the debugged program. The complete code of the undo actions is publicly available in our debugger project repository [3] where the reader can verify that these actions write back the same state that was originally modified.

Although identifying Pharo's writing operations manually can be prone to errors, it is not a concern for OR. If we had missed any writing operations, there would be even fewer undo actions performed during reversal. Consequently, there is no OR.

4.4.2.3 No UR for thread state reversal

To avoid UR, the reversal operation scope should include at least all the state modified by the process execution. To address this concern, we first list all memory elements of the thread state that are written by the debugged execution and show that our mechanism reverses all these elements.

From the 8 slots of Process described in 4.4.2.1, we list whether these slots of the process running the debugged program are modified and how they are reversed. The slots `effectiveProcess`, `env`, `name` are not modified, and therefore, they do not require to be reversed.

The slot `level` is used by Pharo instrumentation libraries (`MetaLink`), a case of our listed limitations on reflective programs. In non-instrumented programs, `level` is left unmodified, and there is no need to reverse it.

[3] Thesis GitHub repository:
https://github.com/Willembrinck/2023-Selective-Time-Traveling-Thesis

The `myList` slot holds a semaphore that is modified when the process is suspended or resumed. In our implementation, the debugged process (*debuggee*) is always held suspended, and it is advanced by interpreting its instructions by the debugger. The debugger does not change the debuggee suspension state at any point. The debuggee can still attempt to suspend itself (*e.g.,* the debugged program invokes a *wait* operation). However, as a result of the Pharo code interpretation mechanism, it is the debugger process that is suspended and not the debuggee. Resuming the debuggee resumes the debugger process instead, and in practice, the debuggee suspended state remains unaffected. Consequently, as `myList` slot is never changed due to the execution of the debugged program, there is no need to reverse it.

The debugged program itself can change its own `priority` slot. In such a case, it would be registered as a writing operation, and the change would be reversed.

Our debugger prevents stepping instructions that will terminate the process, so the `terminating` slot remains constant.

The only process slot that requires to be modified during the thread state reversal is `suspendedContext`. To this end, the reversal mechanism assigns back the initial context.

The initial context itself is also modified due to the execution of the program, and therefore, any change on it requires to be reversed. The initial context slots `closureOrNil`, `method` `sender`, and `receiver` remain constant during the program execution, so no there is no need to reverse them.

Only the `pc` and `stackp` slots are modified by the execution, and they are both reversed, as we explained previously. Consequently, as all the modified slots are reversed, there is no UR.

4.4.2.4 No UR for execution effects reversal

We reverse all necessary writings. Our implementation detects all writings the debugged program performs based on a list of identified writing operations. This can be verified from the provided code. However, and to the best of our knowledge, the list of Pharo primitives that perform writing is not officially documented. This means that we can not categorically assess the completeness of our list. Should the list be incomplete, it is not a concern for our proposed design but an implementation detail. For example, if an official list is released, it would be only a matter of expanding our list of identified writing operations without incurring any change to our propositions.

4.4.3 Reversal by full-replay

For our reversal mechanism, we do not store any thread state. Instead, we only store a reference to the initial context. To recover the initial thread state, we use the

`Context` class API, particularly the `reset` method. Then, we reverse the effects of the execution by evaluating all recorded undo blocks. This completely restarts the execution. Then, we replay the program instructions deterministically from the beginning to reach intermediate states.

This full-replay decision simplifies the implementation, as we do not need to store intermediate thread states or decide when to capture them. We do not require to manipulate the execution stack manually to restore stored contexts. Additionally, we do not have to calculate which undo blocks should be evaluated and do not need to update TCUs step numbers to match the target timestamp. The trade-off is that without intermediate checkpoints, each time-travel operation, especially targeting a timestamp near the end of the program execution, will incur an overhead that negatively impacts the debugger performance.

4.4.4 System calls logging

Our implementation does not log the number of instructions performed by the system call. In Pharo, calls to primitives can fail, executing application-level code instead, comprising an arbitrary quantity of instructions and potentially performing more system calls. We solve that by registering primitives as system calls. When any system call is detected, we increase the tracer step number by one, as if every system call invocation takes one step to complete, and we create a log entry. These logged entries comprise the step number of the call and a closure that captures the system call returned value and returns this value when evaluated.

4.4.5 Limitations

Our prototype implementations present the following limitations:

1. Similarly to current time-travel solutions, our implementation cannot detect changes in the shared memory originating from unmonitored external threads. Such changes might interfere with the reverse-replay operations if they modify parts of the shared memory used by the monitored threads.

2. Although our prototype handles many commonly used system calls (including random number generation, system time, and objects' identities), the total number of covered system calls is not complete.

3. Not all reflective operations are supported, and their support is a work-in-process. Currently, programs that directly modify their process could interfere with the precision of the reversal mechanism.

4.4.6 System stability assumption

While our implementation is designed to perform reverse and deterministic replay, the deterministic aspect of replays also relies on the stability of the system. For example, replaying an execution will unavoidably create new instances of objects. Since there is no explicit memory management in Pharo, a replay operation risks errors such as running out of memory under certain system conditions. Therefore, we assume that the system remains stable, where no events external to the debugged program or reverse-replay debugger will interfere with the debugging session.

4.5 Conclusion

In this chapter, we described Seeker and Executor, our implementations based on our proposed time-travel back end design. We explained in detail how Seeker and Executor provide configurable support for the identified properties. In the next chapter, we use these implementations to evaluate the significance of the properties for selective time-travel by performing experiments. In these experiments, we explore the potential outcomes of utilizing time-travel back ends that support and that do not support each property in terms of deterministic replay of programs in shared memory.

CHAPTER 5
Evaluation

Contents

5.1 Experiment Goals and Research Question 61
5.2 Evaluation of the Properties for Single-threaded Executions . . . 62
5.3 Evaluation of the Properties for Multithreaded Executions . . . 72
5.4 Results Conclusion 76
5.5 Threats to Validity 77
5.6 Related Work 78
5.7 Conclusion 81

In previous chapters, we explored the challenges that time-traveling debuggers encounter regarding shared memory systems, where multiple programs run concurrently in the same memory space. We identified four properties required for effective scoping of reversal operations, three for single-threaded programs and an additional one to support multithreaded programs. Building upon these properties, we proposed a selective time-travel back end design and two implementations: Seeker and Executor. In this chapter, we evaluate the effects of the identified properties for selective time-traveling in shared memory systems, report our findings, and discuss related work.

5.1 Experiment Goals and Research Question

In this section, we introduce our experiments focused on exploring the significance of the identified properties in attaining selective time-travel in shared memory systems. Our aim is to determine the potential consequences if time-travel back ends lack any of these properties. Specifically, we want to investigate whether a debugger can accurately reverse program execution without these properties and if a debugged program can display deterministic replay behavior without them. To accomplish this, we begin by defining the terminology we used in this experiment.

We consider that a replayed program *behaves deterministically* if given a fixed initial state, in each replay, the program always executes the same number of instructions in the same order and produces the same output. We consider

that a time-travel operation is *correct* if the operation reaches program states deterministically. Correct time-travel operations allow the time-travel back end to replicate a program's behavior during replay, while incorrect operations would introduce inconsistent replay behavior.

We investigated the following research question:

RQ: What is the impact of the selective time-travel properties on the correctness of time-travel operations for program executions in shared memory systems?

To study this question, we performed three experiments. In experiments #1 and #2, we used Seeker to evaluate the properties 1 TSR, 2 EER, and 3 NDS and their impact on the deterministic reproduction of executions of single-threaded programs. Then, in experiment #3, we used Executor to evaluate the impact of all properties on the deterministic reproduction of executions of multithreaded programs.

5.2 Evaluation of the Properties for Single-threaded Executions

We performed two experiments to analyze the impact of the properties on the correctness of time-travel operations of single-threaded programs in shared memory systems. With these experiments, we aim to observe if debuggers without support for any of the properties allow for correct time-travel operations.

In our first experiment, we performed controlled time-travel operations on a crafted program that modifies local and shared memory. We check for the correctness of time-travel operations with 8 property configurations, each enabling all, part, or none of the properties. Our results show that selective time-travel operations are always correct when all properties are enabled and incorrect otherwise.

In our second experiment, we reuse our 8 property configurations to perform controlled time-travel operations on 119 unit tests from the Pharo system. Contrary to the first experiment, these 119 unit tests execute real code. We ran the experiment 100 times to tame potential non-deterministic effects, e.g., that could make time-travel operations randomly correct or incorrect. Our results also show that time-travel operations are always correct for all programs when all properties are enabled and that disabling any property would produce incorrect time-travel operations on many programs.

Both experiments' results show that selective time-travel operations are correct only when all properties are enabled. We argue that time-travel back ends failing to support any of these properties would produce incorrect time-travel operations.

5.2.1 Experiments general procedure

For experiments with single-threaded programs, we use Seeker, our selective time-traveling debugger, to run the programs. However, for experiments with multithreaded programs, we use Executor instead. To observe how each property affects the deterministic behavior of the program, we configure the debugger to toggle support for individual properties. To determine if the execution behaves deterministically under support for specific properties, we compared data collected during the initial program execution with the data from its replay. We collect experiment-specific data by taking measurements of variables within the execution and objects within the global state. If the collected data from the replay are equal to the collected data from the initial execution, then the execution behavior is considered deterministic. In such a case, the configuration of enabled and disabled properties allowed for correct time-travel operations.

To collect measurements and results for each experiment, we first declare the debugger configurations to be evaluated. Configurations specify the specific properties that the debugger will support for the evaluation. For example, a configuration labeled as *(110)* enables support of property 1 (TSR) and property 2 (EER) but disables the support of property 3 (NDS) in the debugger.

We studied all possible combinations: C1: (111), C2: (110), C3: (101), C4: (100), C5: (011), C6: (010), C7: (001), C8: (000). For each configuration, we followed these steps:

1. We initialize the debugger with the control program.

2. Set up the initial state of the observed global state.

3. Perform a sampling run (the original run), in which we collect measurements of the debugged program at different points of its execution.

4. Reverse the program according to the support of TSR and EER properties defined by the configuration.

5. Perform a new sampling run (referred to as the replay run) enabling support for NDSs, as defined by the configuration.

6. Compare the collected measurements obtained from the original and replay runs.

For the analysis of results, we define the following criteria for a *correct time-travel operation*:

1. Execution Reversal: The program can be reversed to a target past execution state, reverting modifications in the global state. The execution reversal criterion is evaluated by testing the equality of measurements of the

debugged program and the observed global state at the beginning of the first execution with measurements obtained after reversing the program execution.

2. Deterministic Replay (*DR*): The program is replayed deterministically after reversal. The deterministic replay criterion is evaluated by testing the equality of measurements of the debugged program taken at different points during its first execution, compared to measures taken at the same points during replay.

General control measures To ensure a controlled evaluation of the properties, we ran the experiments on programs that alter known global state and whose NDSs are accounted for by the time-travel back end.

We define the *global state* as all the objects of the system that are not instantiated by the debugged execution. We consider the debugger as part of the global state. We refer to the objects instantiated by the debugged execution as *local objects*.

When observing changes occurring in local objects and objects of the global state, there is ambiguity in knowing if the change occurred due to the debugged program execution or external executions. We explain how we solved this ambiguity by first explaining the possible writing interactions.

There are 4 writing interactions that can occur when debugging a program in a shared memory system.

1. The debugged execution modifies local objects.

2. The debugged execution modifies objects of the global state.

3. The executions different from the debugged one modify local objects of the debugged execution.

4. The executions different from the debugged one modify objects of the global state.

For each experiment, we need to observe changes in the global state introduced by the debugged execution and time-travel operations, which corresponds to interaction (2). However, global state objects can be created and modified by executions in the system other than the debugged one, *i.e.,* by interaction (4). This problem makes observation unreliable as we cannot tell from the measurements if the change observed in one of these objects occurred due to the debugged execution or another execution in the system.

To control this issue, we define a subset of objects of the global state for each experiment that we are certain will not be altered by any other execution. We call these objects the *observed global state*. By focusing only on these objects, we

rule out changes produced by interaction (4). This allows us to observe changes produced exclusively by interaction (2).

Tracking changes in local objects is of no concern since there is no ambiguity. Changes in these objects are always performed by interaction (1), and interaction (3) is ruled out. This is by design, as we did not introduce any other execution that affects objects local to the debugged execution. In addition, a fair assumption to make is that the Pharo runtime does not modify random objects's instance variables at arbitrary moments.

5.2.2 Evaluation of the properties on a crafted program

In our first experiment, our objective is to closely observe and understand the impact of the properties (or their absence) on the behavior of a specific and comprehensible program.

Specific control measures We implemented control measures to observe changes in the global state and designed a program that introduces a non-deterministic behavior.

The observed global state consists of a unique control object, which is instantiated before each experiment and is passed as a parameter to the program that modifies its value.

The *control program* (Listing 5.1) we created involves modifying local and global state (*i.e.*, the variable in the control object of the observed global state), and performing system calls (such as object instantiation and obtaining random numbers) to introduce non-deterministic behavior. To explain the program, we break down the code into 4 *segments*.

In the first segment (Lines 1 to 6), the debugged program performs initialization actions on the local and global state. The debugged program receives an object of the global state as an argument (Line 1). Then, the debugged program initializes a local variable and increases the value stored in the global object by 1 (Lines 4 and 5). We use special methods to mark the end of each segment, which we call *marker methods* (Lines 6, 11, and 17). These methods do not perform any action, and we use them to identify the moments to take measurements of the debugged execution. When our experiment framework detects that any of these marker methods is invoked, a measurement is taken and logged.

In the second segment (Lines 7 to 11), the program makes modifications to local and global state based on a non-deterministic system call. The program generates a random number invoking a non-deterministic method (Lines 7 and 8). Then, the program modifies local and global state increasing stored values by the obtained random number (Lines 9 and 10)

In the third segment (Lines 12 to 16), the program produces a random number based on instantiation-related non-determinism. First, one object is created (Line

12), whose identity hash is non-deterministic. Then, 100 objects are instantiated and added into a set, along with the first object (Lines 13 to 14). In Line 16, we obtain a number that is non-deterministic since sets internally store objects in a particular order based on their identity hash.

In the fourth segment (Lines 17 to 20), the local and global state is modified according to the non-deterministic number obtained in segment 3.

```
1  TtdValidationProgram class >> runWithGlobal: global
2    | localVar randomNumber set obj number |
3    localVar:=1.
4    localVar := localVar + 1.
5    global val: global val + 1.
6    self firstMarker. "Start of segment 2"
7    randomNumber := SharedRandom globalGenerator
8      nextInteger: 9999.
9    localVar:= localVar + randomNumber.
10   global val: global val + randomNumber.
11   self secondMarker. "Start of segment 3"
12   obj := Object new.
13   set := IdentitySet new.
14   1 to: 100 do: [ :i| set add: Object new ].
15   set add: obj.
16   number := set asArray indexOf: obj.
17   self thirdMarker. "Start of segment 4"
18   localVar := localVar + number.
19   global val: global val + number.
20   ^ {localVar . global} "program end"
```

Listing 5.1: Control program for the experiment. Markers methods (Lines 6, 11, 17) are used to trigger the collection of measurements of the execution state.

5.2.2.1 Data collection

We take measurements of the execution state by registering the following values:

1. The current program counter of the debugged thread,
2. the number of steps performed by the debugger,
3. the source code of the current instruction,
4. the observed global state value and
5. the value of the local variable of the program.

We collect measurements on the following points in the program execution: the program start, invocation of `firstMarker`, `secondMarker` and `thirdMarker` methods, and at the end. We compare the measurements taken at each marker during the original run with those of the replay run, checking for equality.

5.2.2.2 Experiment results and analysis

To report the experiment results, we provide first a summary of these results followed by a more detailed explanation.

Overall, the only configuration performing a correct reversal and deterministic replay of the debugged program is C1 (all properties support is enabled), where there is no criterion violation, as shown in Table 5.1.

Criterion	C1 (111)	C2 (110)	C3 (101)	C4 (100)	C5 (011)	C6 (010)	C7 (001)	C8 (000)
1. Execution Reversal	✓	✓	✗	✗	✗	✗	✗	✗
2. Deterministic Replay	✓	✗	✗	✗	✗	✗	✗	✗

Table 5.1: Criteria approval summary. We observe that the absence of support for any property led to a criterion violation, producing incorrect time-travel operations.

Table 5.2 shows the replay behavior of the step number measured at each marker for the original execution (`o_` prefix) and its replay (`r_` prefix). Table 5.3 shows the replay behavior of the measured instruction program counter and the corresponding source code. Table 5.4 shows the replay behavior of the measured local variable value. Table 5.5 shows the replay behavior of the measured global value.

Measure Label	C1 (111)	C2 (110)	C3 (101)	C4 (100)	C5 (011)	C6 (010)	C7 (001)	C8 (000)
o_start	1	1	1	1	1	1	1	1
r_start	1	1	1	1	1	1	1	1
o_firstMrk	18	18	18	18	**18**	**18**	**18**	**18**
r_firstMrk	18	18	18	18	**1**	**1**	**1**	**1**
o_secMrk	163	163	163	163	**163**	**163**	**163**	**163**
r_secMrk	163	163	163	163	**1**	**1**	**1**	**1**
o_thirdMrk	8175	**8150**	7688	**7652**	**7300**	**7489**	**7413**	**7673**
r_thirdMrk	8175	**7548**	7688	**7177**	**1**	**1**	**1**	**1**
o_end	8194	**8169**	7707	**7671**	**7319**	**7508**	**7432**	**7692**
r_end	8194	**7567**	7707	**7196**	**1**	**1**	**1**	**1**

Table 5.2: Step number measured for each configuration at every execution marker. Red values show that, for some configurations, the markers are reached in a different step number after restarting the program.

For all configurations without support for property 1 TSR (C5, C6, C7, C8), the reversal operation does not affect the thread state, violating criterion 2.

- Table 5.2 shows that the reversal operation reverted the step number to 1, but the replay did not perform any step. This is observed by comparing the measures taken during the first execution (labeled with prefix *o_*) against their counterpart measured during replay (labeled with prefix *r_*).

Measure	C1 to C4 (111),(110),(101),(100)		C5 to C8 (011),(010),(001),(000)	
Label	pc	code	pc	code
o_start	153	localVar:=1	**153**	**localVar:=1**
r_start	153	localVar:=1	**241**	**^{localVar . global}**
o_firstMrk	167	self firstMarker	**167**	**self firstMarker**
r_firstMrk	167	self firstMarker	**241**	**^{localVar . global}**
o_secMrk	186	self secondMarker	**186**	**self secondMarker**
r_secMrk	186	self secondMarker	**241**	**^{localVar . global}**
o_thirdMrk	224	self thirdMarker	**224**	**self thirdMarker**
r_thirdMrk	224	self thirdMarker	**241**	**^{localVar . global}**
o_end	241	^{localVar . global}	241	^{localVar . global}
r_end	241	^{localVar . global}	241	^{localVar . global}

Table 5.3: Program counter and corresponding code measured at every execution marker for all configurations. The table shows in red that configurations where TSR is disabled (C5 to C8) remained in the ending state of the first execution and were not able to reverse or replay the thread.

Measure Label	C1 (111)	C2 (110)	C3 (101)	C4 (100)	C5 (011)	C6 (010)	C7 (001)	C8 (000)
o_start	nil	nil	nil	nil	**nil**	**nil**	**nil**	**nil**
r_start	nil	nil	nil	nil	**36**	**42**	**38**	**69**
o_firstMrk	2	2	2	2	**2**	**2**	**2**	**2**
r_firstMrk	2	2	2	2	**36**	**42**	**38**	**69**
o_secMrk	27	6	5	**32**	**32**	**32**	**32**	**48**
r_secMrk	27	6	5	**12**	**36**	**42**	**38**	**69**
o_thirdMrk	27	6	5	**32**	**32**	**32**	**32**	**48**
r_thirdMrk	27	6	5	**12**	**36**	**42**	**38**	**69**
o_end	31	**10**	11	**44**	36	42	38	69
r_end	31	**27**	11	**24**	36	42	38	69

Table 5.4: Local variable value measured for each configuration at every execution marker. Red values show that, for some configurations, the value behaves non-deterministically after restarting the program.

- Table 5.3 also shows that the execution did not reverse the thread state. This is observed by comparing first execution measurements against the replay measurements of the *pc* and *code*. These measurements show that the execution remained in the last instruction of the first run, and the reversal and replay operations had no effect.

- Similarly, Table 5.4 also shows that the local variable value remained constant after the first execution.

- Finally, Table 5.5 shows that even when configuration C5 and C6 reversed

Measure Label	C1 (111)	C2 (110)	C3 (101)	C4 (100)	C5 (011)	C6 (010)	C7 (001)	C8 (000)
o_start	0	0	**0**	**0**	0	0	**0**	**0**
r_start	0	0	**10**	**43**	0	0	**37**	**68**
o_firstMrk	1	1	**1**	**1**	**1**	**1**	**1**	**1**
r_firstMrk	1	1	**11**	**44**	**0**	**0**	**37**	**68**
o_secMrk	26	5	**4**	**31**	**31**	**31**	**31**	**47**
r_secMrk	26	5	**14**	**54**	**0**	**0**	**37**	**68**
o_thirdMrk	26	5	**4**	**31**	**31**	**31**	**31**	**47**
r_thirdMrk	26	5	**14**	**54**	**0**	**0**	**37**	**68**
o_end	30	**9**	**10**	**43**	**35**	**41**	37	68
r_end	30	**26**	**20**	**66**	**0**	**0**	37	68

Table 5.5: Global value measured for each configuration at every execution marker. Red values show that, for some configurations, the global value behaves non-deterministically after restarting the program.

the global value to 0, it remained constant during replay.

These results show that the reversal operation and replay did not produce the re-execution of the debugged program.

In configurations without support for property 2 EER (C3, C4, C7, C8), the value of the global object was compromised. We observe that the initial global object value of the replay run corresponds to the ending value of the original run (Table 5.5), violating criterion 1. The global keeps incrementing during the replay run, which violates criterion 2.

In configurations without support for property 3 NDS (C2, C4, C6, C8), the local variable of the program, the number of steps, and the global value were compromised. In Table 5.4, the measured local variable values are different for the original and the replay runs. This happens because the local variable value is increased by values produced by non-deterministic system calls (During the second and third segments of the control program). The global variable in Table 5.5 is the same. Even though configurations C2 and C6 correctly reverted the global value for replay (which is observed in labels `o_start` and `r_start` in Table 5.5), the subsequent modifications of the value behaved non-deterministically. Finally, due to the effect of NDS in the program, the step number registered is different for the replays under the configurations C2, C4, C6, and C8. Therefore, these configurations violate criterion 2.

5.2.3 Evaluation on multiple programs running real code

To improve the generalization of our results, we performed a second experiment on realistic debugging scenarios. In this experiment, we study the overall effects of the properties across multiple programs based on real code from the Pharo codebase.

These control programs are extracted from the Pharo tests suite, which covers the standard packages commonly utilized by Pharo developers. By taking code from real Pharo tests, our results are not limited to isolated or ad-hoc examples and are more representative of the code frequently encountered in user programs in real-world scenarios.

Specific control measures In Pharo, test cases are designed to initialize state through the `setUp` method and then execute the test case itself using the `performTest` method. Before running each experiment, our experiment framework instantiates these test cases and executes the `setUp` method, which sets the test's instance variables to an initial value. Then, for each experiment, we create a new debugged program execution that invokes the `performTest` method of the test case. As neither the test instance nor its instance variable are created by the debugged program, we use them to simulate the global state of the debugged program. In this experiment, we scope the *observed global state* to the instance variables of the instantiated test case.

We selected multiple test cases from the standard packages, aiming to maximize the number of collected results while preventing possible bias introduced by the filtering process. From all the test cases of the Pharo standard packages, we chose tests that define and modify instance variables. To obtain a set of tests that can be executed multiple times in a short period, we excluded tests that took more than 5 seconds to complete when run with Seeker. We randomly selected 150 of these tests. Then, from these tests, we excluded the ones that involved known unsupported cases (since the debugger is still a prototype), such as tests that create new processes, perform reflective operations, use sockets, or involve file system writing. Additionally, we discarded tests that exhibited unhandled exceptions or erratic behavior, as they could interfere with the execution of the experiment. The final selection consists of 119 test cases to experiment on.

To tame potential non-deterministic effects, e.g., that could make time-travel operations randomly correct or incorrect, we ran the experiment 100 times.

5.2.3.1 Experimental procedure

In this experiment, we followed the first experience procedure (5.2.2.1) while implementing the following actions for each program.

We take measurements at the start and the end of their execution (*i.e.,* the start and end of the execution of the `performTest` method). In contrast to the first experiment, we do not take measurements on intermediate execution states because there are no common points in their executions to be considered execution markers to perform these measurements. The measured execution data is the step number and the observed global state, *i.e.,* the test instance variables. After each configuration is executed for the program, we register in our result the name of the test case, the configuration, and 3 booleans expressing: If the test presented a

deterministic step number, if the global state is reversed, and if the observed global state exhibited deterministic replay.

To measure the observed global state, we create a dictionary whose keys are the names of the instance variables of the tests, and the values correspond to a copy of the variable's value. To make a copy, we use the method `veryDeepCopy` of the `Object` API, which makes a copy of an object and the entire tree of objects it points to. Simply put, we consider that two object trees match if their root objects have the same identity or are of the same class and their slots contain matching object trees.

5.2.3.2 Experiment results and analysis

Table 5.6 shows that for the measured executions, the property NDS had little impact on the overall correctness of time-travel operations (C1 vs. C2 presenting a decrease in the correctness of 6%, and C3 vs. C4 differing for only 1%). This is expected since tests are typically designed to reproduce the same output every time, *i.e.,* to present an assertion-free execution, in a controlled setup. Therefore, their implementation is biased toward deterministic behavior.

Incorrectness mainly resulted from not undoing effects on the global state (C1 vs. C3 presenting a decrease in the correctness of 61% and C2 vs. C4 dropping by 54%).

Config.	Det. Step Number	Global Reversed	Det. Global	Correct
C1 (111)	11900 (100%)	11900 (100%)	11900 (100%)	11900 (100%)
C2 (110)	11129 (94%)	11900 (100%)	11200 (94%)	11129 (94%)
C3 (101)	8400 (71%)	5000 (42%)	10100 (85%)	4700 (39%)
C4 (100)	7887 (66%)	4900 (41%)	9500 (80%)	4752 (40%)
C5 (011)	0 (0%)	11900 (100%)	5000 (42%)	0 (0%)
C6 (010)	0 (0%)	11900 (100%)	4900 (41%)	0 (0%)
C7 (001)	0 (0%)	5000 (42%)	11900 (100%)	0 (0%)
C8 (000)	0 (0%)	4900 (41%)	11900 (100%)	0 (0%)

Table 5.6: Experiment results showing the number of program executions presenting deterministic behavior on the measured values. The last column shows the overall correctness of time-travel operations for each configuration. We ran 119 tests for each configuration and repeated the experiment 100 times, resulting in 11900 program executions in total.

5.3 Evaluation of the Properties for Multithreaded Executions

In this experiment, we evaluate the 4 properties needed for selective time-travel and how they affect the deterministic behavior of a crafted multithreaded program.

Specific control measures We next explain our control multithreaded program and why it serves to evaluate the properties.

We created a program that executes two threads (Listing 5.2). Each thread attempts concurrently to increment a shared counter several times. Without any time-travel back end, our crafted program behaves non-deterministically due to a race condition and non-deterministic thread interleaving.

```
1  Experiment3 >> controlRun
2    | sharedCounter program |
3    sharedCounter := 0 .
4    2 timesRepeat: [ | threadBlock |
5      threadBlock := [
6        "Increment the sharedCounter 1000 times"
7        1000 timesRepeat: [
8        | temp1 |
9        temp1 := sharedCounter. "We introduce a race condition"
10       self busywaitMilliseconds: 5.
11       sharedCounter := temp1 + 1 ] ].
12     "Start the execution of the block in a concurrent thread"
13     threadBlock fork].
14   self busywaitMilliseconds: 20000."Wait enough so all threads are finished"
15   ^ sharedCounter
16
17 Experiment3 >> busywaitMilliseconds: ms
18   | targetTime |
19   targetTime := ms + Time millisecondClockValue.
20   [ Time millisecondClockValue >= targetTime ] whileFalse: [ #doNothing ]
```

Listing 5.2: Control multithreaded program for the experiment and a waiting mechanism. The program creates two threads, each executing the same method that increases a shared counter concurrently. The race condition occurs due to Lines 9 to 11, during which other threads can alter the value of the shared counter.

The program creates two threads using a 2 `timesRepeat:` loop (Line 4) to define and start each thread. The code executed by each thread is defined in Lines 5 to 11. Then, each thread is started (Line 13), executing concurrently.

When the program is executed, both threads of the program attempt to increase the value of a shared variable concurrently. To increase the shared counter, the program threads first make a local copy of the value (Line 9). Then, the thread waits for 5 milliseconds (Line 10). Once the wait is finished, the thread assigns to the shared counter the local value plus 1 (Line 11). This incrementing process is

repeated a thousand times.

If the program had performed the increments as atomic operations and without using local copies of the shared counter, then the output of this program running two threads would be 2000. However, the program does not include atomic operations or use synchronization mechanisms, introducing a race condition. Both threads executed concurrently create a local copy of the shared counter and then wait. Consequently, any increment produced by one thread can be overridden by the other, which affects the output of the program. In our exploratory tests, the program executed this way outputs random values close to 1000 (*e.g.,* $1021, 1013, 1010, 1049$) as most increments of one thread are overridden by the other.

The randomness of the output values occurs due to the non-deterministic order in which the threads execute their instructions, *i.e.,* their interleaving. Making the interleaving deterministic would still produce a resulting value near 1000. However, the output would remain consistent on each replay.

We consider that this program offers a fitting scenario to evaluate the impact of property 4 DCS, as the consequences of non-deterministic interleaving are easily observable. We expect that when support for property 4 DCS is enabled, the program outputs consistently the same result on each replay.

Additionally, the program threads perform system calls (calling **`Time`** `millisecondClockValue`, inside the `busywaitMilliseconds:` method, in Lines 21 and 22). This makes it appropriate to evaluate property 3 NDS, which addresses non-determinism introduced by system calls. The program also modifies shared state, corresponding to the shared counter variable (in Line 3), which we use to evaluate property 2 EER. Finally, property 1 TSR can also be evaluated since the program execution modifies the threads' state.

We implemented the `busywaitMilliseconds:` method (Listing 5.2, Line 19) because our prototype does not yet support synchronization primitives. This waiting mechanism does not rely on Semaphores or other not yet-supported calls, and we use it in our program to make threads wait for a number of milliseconds.

Loading the program in Executor required us to modify the code related to the creation and execution of its threads. (Listing 5.3) shows how we initialized Executor with the program.

To initialize Executor, we first define the code of the program threads (Lines 5 to 9). Then, we used SeekerDebugger to provide TCUs for each thread of the debugged program (Line 10).

```
1  Experiment3 >> loadExecutorWithProgram
2    | sharedCounter threadBlocks |
3    sharedCounter := 0 .
4    threadBlocks := 1 to: 2 collect: [ :i | "collects 2 blocks"
5      [ "a block that increments the sharedCounter 1000 times"
6        1000 timesRepeat: [ | temp1 |
7        temp1 := sharedCounter.
8        self busywaitMilliseconds: 5.
9        sharedCounter := temp1 + 1 ] ] .
10     ^ Executor newFor: (threadBlocks collect: [ :tb | SeekerDebugger headlessDebugBlock: tb])
```

Listing 5.3: Initialization of Executor with the program. We use SeekerDebugger to provide the TCUs for Executor.

The modified version of the program retains the same characteristics that made the original one fit for the evaluation of the properties. We verified this by running the new program with Executor, with all properties support disabled, and obtaining the same behavior with resulting shared counter values close to 1000.

An important difference in this modified program is that the shared counter is not part of the debugged execution. The debugged program consists only of the execution of the threads (Lines 5 to 9), and the shared counter is defined in the external scope (Line 3). Therefore, the shared counter is not a local variable of the debugged program (it is already defined before the debugged program starts). This change is crucial because it allows us to use this variable as the observed global state for measurements.

5.3.1 Experiment procedure

In this experiment, the debugger configurations include the 4 properties, *e.g.,* a configuration labeled as (1101) enables support of properties 1 (TSR), 2 (EER), and 4 (DCS) but disables support of property 3 (NDS) in the debugger. We studied all 16 possible configurations C1: (1111), C2: (1110), C3: (1101), C4: (1100), C5: (1011), C6: (1010), C7: (1001), C8: (1000), C9: (0111), C10: (0110), C11: (0101), C12: (0100), C13: (0011), C14: (0010), C15: (0001), C16: (0000).

We performed the general procedure and took measurements at the start and end of the initial execution, at the start after reversal, and at the end after replaying. We measured the following values: the shared counter, Executor timestamp, the TCU#1 step number, and TCU#2 step number.

5.3.2 Experiment results and analysis

Overall, the only configuration performing correct time-travel operations on the debugged program is C1 (all properties support is enabled), where all the measurements show deterministic behavior on the reversal and during replay, as shown in Table 5.7.

Sample	Label	C1 (1111)	C2 (1110)	C3 (1101)	C4 (1100)	C5 (1011)	C6 (1010)	C7 (1001)	C8 (1000)
shared counter	o_start	0	0	0	0	0	0	0	0
	r_start	0	0	0	0	999	997	1002	1003
	o_end	995	1003	999	993	999	997	1002	1003
	r_end	995	1004	999	999	1998	2002	2000	1999
program time stamp	o_start	1	1	1	1	1	1	1	1
	r_start	1	1	1	1	1	1	1	1
	o_end	481302	487254	510606	539646	461454	502422	509934	513990
	r_end	481302	487254	543726	537558	461454	502422	526182	540414
TCU#1 step number	o_start	1	1	1	1	1	1	1	1
	r_start	1	1	1	1	1	1	1	1
	o_end	240735	243351	255495	269823	230679	251943	255351	256191
	r_end	240735	243351	272319	267687	230679	251943	262791	269607
TCU#2 step number	o_start	1	1	1	1	1	1	1	1
	r_start	1	1	1	1	1	1	1	1
	o_end	240567	243903	255111	269823	230775	250479	254583	257799
	r_end	240567	243903	271407	269871	230775	250479	263391	270807

Table 5.7: Sampled values taken from the multithreaded program for configurations C1 to C8. Red numbers highlight samples that behave non-deterministically after the reversal and during replay.

For all configurations without support for property 1 TSR (C9 to C16 in Table 5.8), the reversal operation does not affect the threads state, and the program cannot be replayed. Consequently, we will not include these configurations in the rest of the analysis, and we will exclusively focus on the results shown in Table 5.7.

Configurations without support for property 2 EER (C5, C6, C7, C8) did not reset the shared counter on reversal. As a consequence, the counter behaved non-deterministically, nearly doubling the final value produced by the other configurations during replay.

In configurations without support for property 3 NDS (C3, C4, C7, C8), the number of performed steps behaves non-deterministically. This was expected because our *busywait* implementation relies on the system clock to define the condition that ends the wait. In C3, where all property support is enabled except for property 3, the shared counter presented deterministic behavior. However, this *(false)* positive outcome corresponds to a sampling issue. In this configuration, a replay operation will attempt to enforce the determinism of the interleaving registered from the first execution. However, since there is no support for property 3 NDS, system calls can introduce non-deterministic behavior in each respective thread. These system calls alter the number of steps that each thread take to finish, making them take fewer or more steps to complete when replayed. Since the logs register the order of interleaving along with a timestamp, if during replay the program takes more steps to complete than the last record of the log, any step performed after the last registerd step do not have interlaving order records. This

makes the execution to perform the last steps with a non-deterministic interleaving. As a result, even if property 4 DCS support is enabled, there is still a possibility of randomness being caused by interleaving due to the lack of property 3 NDS, which can impact the shared value. However, the risk is significantly reduced as the randomness only occurs in the step numbers out of the interleaving records.

Configurations without support for property 4 DCS (C2, C4, C6, C8) produced non-deterministic behavior of the shared counter. Our results show that when support for properties 1, 2, and 3 is enabled but support for property 4 is disabled (configuration C2), the shared counter presented a non-deterministic behavior due to interleaving randomness and the race condition, even if each thread performed the same number of steps during replay.

Sample	Label	C9 (0111)	C10 (0110)	C11 (0101)	C12 (0100)	C13 (0011)	C14 (0010)	C15 (0001)	C16 (0000)
shared counter	o_start	0	0	0	0	0	0	0	0
	r_start	0	0	0	0	1001	1000	997	1001
	o_end	1006	996	1003	1002	1001	1000	997	1001
	r_end	0	0	0	0	1001	1000	997	1001
program step number	o_start	1	1	1	1	1	1	1	1
	r_start	1	1	1	1	1	1	1	1
	o_end	460349	473837	491021	523829	469253	494765	535013	554837
	r_end	1	1	1	1	1	1	1	1
TCU#1 step number	o_start	1	1	1	1	1	1	1	1
	r_start	1	1	1	1	1	1	1	1
	o_end	229095	237183	247071	262479	235719	247695	267711	277167
	r_end	1	1	1	1	1	1	1	1
TCU#2 step number	o_start	1	1	1	1	1	1	1	1
	r_start	1	1	1	1	1	1	1	1
	o_end	231255	236655	243951	261351	233535	247071	267303	277671
	r_end	1	1	1	1	1	1	1	1

Table 5.8: Sampled values taken from the multithreaded program for configurations C9 to C16. Red numbers highlight samples that behave non-deterministically after the reversal and during replay.

5.4 Results Conclusion

Enabling or disabling specific properties support of the time-travel back end impacted the ability to achieve correct time-travel operations. Answering our experiment research question, only time-travel back ends with support for all properties are able to perform correct time-travel operations, allowing the debugged programs to present a deterministic reversal and replay behavior. Back ends failing to support any of the properties risk non-deterministic reversal and replay of programs, hindering the ability to reproduce the program behavior.

- A back end not supporting property 1 TSR produces an under-reversal which completely denies the ability to replay an execution.
- A back end not supporting property 2 EER produces an under-reversal by not reversing changes in the global state, which in most of our tested programs, lead to a non-deterministic replay, even in the absence of system calls.
- A back end not supporting property 3 NDS would replay executions non-deterministically in the presence of system calls, even if the execution and its effects on the global state were effectively reversed.
- Finally, a back end not supporting property 4 DCS would replay multithreaded executions non-deterministically in the presence of race conditions.

5.5 Threats to Validity

Threats to construct validity. We only test for determinism after one reversal and replay cycle. In real scenarios, determinism must be maintained throughout multiple time-travel operations. However, as backed up by our results on multiple programs, we argue that a correct reversal operation followed by a deterministic replay would always yield the same program behavior, and as the execution is completely reproduced, so will future reversal operations.

Threats to internal validity. The experiment includes a limited number of measurements for each configuration. This implies that the intermediate states of the program are not observed, making it difficult to draw definitive conclusions about the behavior of time-traveling debuggers. However, this is mitigated as in our results, non-determinism effects carried over for the rest of the execution, suggesting that if we had missed a compromised intermediate state, it would have shown up in the subsequent states. Not all global state is observed, which might imply that we missed some under or over-reversal effects. To detect the reversal of the global state, we scoped our observations to the state of particular objects, as it is not feasible to anticipate which system state is modified by the debugged programs and which system state is modified by the experiment framework program itself. To reduce this threat, we chose to use tests. Tests are typically designed in favor of avoiding changing global state, trying instead to modify only the test's instance variables or the objects these variables contain. This way, we were able to determine which state was altered by the debugged program by focusing on the test instance variables before and after running the program.

Threats to external validity. The experiments only look at a selection of Pharo programs, so they do not take into account all the complexities and variations found

in real-world software. This could imply that the findings might not apply well to different kinds of programs or programming languages. However, this is mitigated as the Pharo programming language shares many similarities with most object-oriented programming languages, and the test cases cover code commonly used by Pharo developers in their real software. We discarded several tests from the test suite that use reflective computations and perform systems calls unsupported by our current implementation. We do not see this as a significant threat, as we are evaluating the correctness of the properties and not the completeness of our implementation (which remains a prototype). We evaluated the properties mostly on tests, which are not proper real programs. However, tests provide reproducible executions that can be used as oracles of comparison. Everything can be replicated and is provided to the reader[1].

Threats to conclusion validity. The results we obtained will vary depending on the selected programs. The correctness percentages are not inherent to the evaluated properties and only reflect the impact of the time-travel operations on the selected programs. For example, and as we mentioned in 5.2.3.2, tests are designed with a bias toward deterministic replay. Experiment 2 results reflect the little impact of the properties in the selected tests. Programs introducing more system calls will undoubtedly be more affected by the property support. Such is the case of our crafted programs used for Experiments 1 and 3. As the results depend on the used programs, drawing definitive conclusions on each property might be considered a flawed approach. However, the main conclusion, where we state that the absence of support for any of the properties would yield incorrect time-travel operations, is consistent both in the theory and in the numerous results.

5.6 Related Work

In this section, we study different time-traveling debuggers from the state of the art and analyze them with respect to the properties proposed in this paper. We summarize how each debugger supports each property and outline the technique used. If a property is not supported, we discuss what is missing to satisfy it.

5.6.1 Properties support in time-traveling debuggers

Selective thread state reversal (TSR). State-of-the-art debuggers [Arya 2017, Barr 2014, Barr 2016, King 2005, Montesinos 2008, O'Callahan 2017, Phang 2013, UDB 2023, Vilk 2018] are not able to reverse selectively the state of particular threads within a process as they rely on snapshots or system memory checkpoints

[1] Complete implementation code and experiment replication instructions available at https://github.com/Willembrinck/2023-Selective-Time-Traveling-Thesis

Time-traveling Debugger	Reversal Mechanism	Multi-thread Support	Prop 1. Selective TSR	Prop 2. Selective EER	Prop 3. NDS Handling	Prop 4. DCS
Executor	TS+LU	✓	✓	✓	✓	✓
Expositor [Phang 2013]	Snapshot	✓	✗	✗	✓	✓
FReD [Arya 2017]	Snapshot	✓	✗	✗	✓	✓
Jardis [Barr 2016]	Snapshot	✗	✗	✗	✓	✗
McFly [Vilk 2018]	Snapshot	✗	✗	✗	✓	✗
RR [O'Callahan 2017]	Snapshot	✓	✗	✗	✓	✓
Seeker	TS+LU	✗	✓	✓	✓	✗
Tardis [Barr 2014]	Snapshot	✗	✗	✗	✓	✗
TTVM [King 2005]	Snapshot	✗	✗	✗	✓	✗
UDB [UDB 2023]	Snapshot	✓	✗	✗	✓	✓

Table 5.9: Time-traveling debuggers comparison. **TSR**: Thread State Reversal. **EER**: Execution Effect Reversal. **NDS**: Non Determinism Source. **DCS**: Deterministic Context Switching. **TS+LU**: Thread state snapshots, plus logging and undoing.

that comprise at least the whole application space. Adding support for this property in these solutions would require their time-travel back ends to serialize single thread state, so they can be later recovered individually.

Selective execution effects reversal (EER). State-of-the-art debuggers such as [Arya 2017, Barr 2014, Barr 2016, King 2005, Montesinos 2008, O'Callahan 2017, Phang 2013, UDB 2023, Vilk 2018] are not able to selectively reverse the effect of the execution of particular threads within a process, because they rely on snapshots to produce system memory checkpoints. To support this feature selectively, these debuggers would need to log writing events performed by each thread and then use this log to undo these writings. UDB [UDB 2023] does log necessary information for the application of this property, *i.e.,* which thread performed each change. That information is available to be queried by the developer, but it is not used for reversing particular threads.

Non-deterministic sources handling (NDS) By our definition of time-traveling debuggers, they all support NDS handling. Our implementations present a generic approach to support NDS handling, and other time-traveling debugger implementations present a variation of the same generic approach. For example, UDB [UDB 2023] logs system calls and on replay, those system calls are synthesized from the logs. RR [O'Callahan 2017] uses `ptrace`[2], a system call from Unix-like operative systems, to log system calls of debugged executions.

[2]`ptrace` is a system call found in Unix and several Unix-like operating systems. This feature allows a parent process to monitor and control the execution of another process, as well as examine and modify its core image and registers. https://www.unix.com/man-page/linux/2/ptrace/

To replay, it places breakpoints in logged system calls and uses `ptrace` to run the program until hitting these breakpoints. Once hit, the back end advances the program counter and applies the recorded register and memory changes. Overall, many time-travel solutions implement their own variation to handle NDS [Barr 2016, Barr 2014, King 2005, Phang 2013, Vilk 2018].

Deterministic context switching (DCS). Debuggers that can debug multithreading programs [Arya 2017, Devietti 2009, Montesinos 2008, O'Callahan 2017, Phang 2013, UDB 2023] support DCS. They use different techniques to reproduce the same execution order of instructions in multithreaded programs.

Some debuggers [O'Callahan 2017, UDB 2023] enforce concurrent execution and the interleaving of the debugged program threads.

UDB [UDB 2023] does it through binary instrumentation and using a global mutex. RR [O'Callahan 2017] uses ptrace to enforce one-at-time thread execution and preemptive thread switches.

5.6.2 Properties support in time-travel techniques

We discuss works that, while not consisting of debuggers, propose techniques and schemes that could be used to implement mechanisms for reversal and deterministic replay in debuggers.

The technique proposed in Delorean [Montesinos 2008] includes a system checkpointing support, *i.e.,* a hardware-level rollback recovery mechanism for shared memory multiprocessors which stores the system memory state. There is no proposition for single-thread reversal, which makes it not compliant with properties TSR and EER.

Reversible computing [Aman 2020, Ulidowski 2020] is one alternative to snapshots as a reversal technique. For sequential systems, reversibility can be understood as *recursively undo the last action* [Lanese 2018], *i.e.,* undoing computation actions starting from the last one until the first. Any implementation following this technique is able to comply with properties TSR and EER as they would allow the reversal of specific thread actions.

In the context of deterministic concurrent replay, CLAP [Huang 2013] achieves the same output of computations on every replay by preserving the causality of actions. The approach only records the interleaving events that, if replayed in a different order, would affect the outcome of computations. Such an approach does not comply with property 4 DCS, as it compromises the deterministic replay order of intermediate events *i.e.,* attempting to time-travel to one of these intermediate states would end up in a potentially different application state.

Transactional Memory [Hammond 2004, Herlihy 1993] is a transaction-based alternative to lock-based synchronization for achieving deterministic concurrent replay. It is used to make multiprocess executions behave similarly to

deterministic serialization while keeping parallelism. This technique is used by the work presented in DMP [Devietti 2009], which offers different approaches for deterministic interleaving for replay. Their proposal handles multiprocessing non-determinism in shared memory by enforcing deterministic thread/process interleaving for transactions upfront without the need to record logs or force processes to execute their instructions one at a time. Implementations based on transactional memory are unable to comply with property 4 DCS, as the technique only enforces the ordering of transactions at most, and intermediate instructions of parallel threads have no relative ordering constraints.

5.7 Conclusion

We conducted experiments to evaluate the impact of the identified properties for selective time-travel operations. We ran three experiments: one over a crafted program in which we carefully control reversal and replay effects, a second one over a suite of 119 unit tests executing real code, and a third one over a multithreaded execution.

The results show that to achieve selective time-travel operations, back ends must possess the three first properties to support single-threaded programs and expose all 4 properties to support multithreaded programs.

These properties cover all aspects of time-traveling, including reversal and replay operations, and served as a helpful analysis tool for existing time-travel solutions and their applicability on shared memory systems.

Part II

A New Debugging Approach

CHAPTER 6

Time-traveling Queries: Improving Interactive Debugging

Contents

6.1 Improving on Interactive Debuggers Problems 85
6.2 Time-traveling Queries 86
6.3 Off-the-shelf Time-traveling Queries 88
6.4 Time-traveling Queries Implementation 92
6.5 Conclusion 100

Efficiently debugging a program requires program comprehension. To acquire it, developers explore the program execution, a task often performed using interactive debuggers. However,, exploring a program execution through standard interactive debuggers is a tedious and costly task. To ease program exploration, we propose *Time-traveling Queries* (*TTQs*). TTQs is a mechanism that automatically explores program executions to collect execution data related to debugging questions. This data is used to time-travel through execution states, enabling a new debugging approach that facilitates the interactive exploration of program executions. In this chapter, we define *TTQs* and their implementation.

6.1 Improving on Interactive Debuggers Problems

After acknowledging the issues associated with interactive debuggers outlined in Chapter 1, it would be helpful to have a mechanism that bridges the gap between the questions asked during debugging and obtaining the debug data needed to answer those questions. As a first step of our investigation toward improving the interactive debugging experience, we study the following research question:

RQ: Can we express debugging questions as queries over program executions to facilitate the obtention of relevant debugging information?

In the following sections, we investigate this question, proposing a novel mechanism to improve the debugging experience and explaining how it addresses the listed problems of interactive debuggers described in Chapter 1.

6.2 Time-traveling Queries

We propose to combine time-traveling debugging with scriptable debugging techniques to express program comprehension questions as queries over program executions. We call these queries *Time-traveling Queries* (*TTQs*).

TTQs bridge the programmatic gap between developers' program comprehension questions and the search for their answers in program executions. TTQs explore the whole program execution to extract information answering these questions (addressing problems 1 and 3). This information is presented to developers, who are able to time-travel in the program execution, to the point where that information was obtained. There, developers can observe the information in its original context. They can deepen their understanding of the execution by time-traveling to other results or by performing standard forward or backward steps. Like this, developers do not need to restart the debugging processes to see information about the program's past states (Addressing problems 2 and 4).

We argue that TTQs enable in-depth live program exploration. Developers will directly use pre-existing off-the-shelves queries or express their questions as programmatic queries. Program exploration will require less preliminary investigation and consequently improve developers' debugging efficiency.

In this section, we provide a high-level description of TTQs. We describe how to define a query, how to execute that query, and how to time-travel in that query's results. To be consistent with our implementation (Section 6.4) and our evaluation (Chapter 7), we write our examples in the Pharo language.

6.2.1 Time-traveling Queries definition and execution

We consider program executions as sequences of program states, as described in Chapter 1. A TTQ is a query over a program execution that selectively collects information from every program state. It is then possible to time-travel to the execution context from which information was collected.

Defining queries. A time-traveling query is an object specifying a *data source*, a *selection predicate*, and a *projection function*.

The data source is an iterable object that represents a sequence of program states, from where to select (*i.e.,* filter) and collect (*i.e.,* transform) data. In the following Pharo script, we instantiate a query that will iterate over all the program states of a program execution (Listing 6.1). In this code, the sequence of program states is referred to as `programStates` and should be generated by the underlying debugger.

```
1 query := Query from: programStates
```

Listing 6.1: Instantiation of a Query object.

The instantiated query object is just an object containing the query definition and is not executed yet. Next, we will manipulate this query to specify which program states we are interested in, and the program data we desire to collect from those states. We do this by setting the query's *selection predicate* and *projection function* respectively.

The ***selection predicate*** is a closure that is evaluated for each item of the data source (in the same way as the OCL selection clause). This predicate is a condition to decide if a program state is of interest for a given query. When evaluated, this condition returns `true` if the program state should be selected and `false` otherwise.

In the script (Listing 6.2), we define a selection predicate, obtaining a query to select program states corresponding to message-sends.

```
2 query := query select: [ :state | state isMessageSend ].
```

Listing 6.2: We create a more refined query with a selection predicate that finds all states corresponding to message-sends.

The ***projection function*** is a closure that is evaluated for each item of the data source selected by the selection predicate. For each selected program state, the function collects specific execution data in a form specified by the developer.

In the script (Listing 6.3), we define a projection function, obtaining a query that collects the class of the receiver, the selector of the message sent, and its arguments.

```
3 query := query collect: [ :state |
4   {(#receiverClass -> state receiverClass).
5    (#selector -> state msgSelector).
6    (#args -> state arguments)} asDictionary ].
```

Listing 6.3: We create a more refined query with a projection function that records every message-send data (receiver class, selector, and arguments).

In our examples, we decomposed the query definition into several steps for illustrative purposes. However, queries can be defined in a single expression, as shown in Listing 6.4:

```
1 query := Query from: programStates
2   select: [ :state | state isMessageSend ]
3   collect: [ :state |
4     {(#receiverClass -> state receiverClass).
5      (#selector -> state msgSelector).
6      (#args -> state arguments)} asDictionary ]
```

Listing 6.4: A Query defined in a single expression. The query retrieves specific execution data of all the message-sends performed by a program.

The messages select: and collect: return a new query instance with a new selection predicate or projection function. They do not execute the query. Queries

are executed by calling the query's `asOrderedCollection` method, which triggers the iteration of the program states and the production of results. We elaborate further on this in Subsection 6.4.2.

TTQ execution. When a query is executed (Figure 6.1), the time-traveling debugger restarts and executes the program, instruction by instruction, advancing from program state to program state. For each state, the query tests its selection predicate over that state. If the state is selected, the debugger collects the data as a result item by applying the projection function to that state.

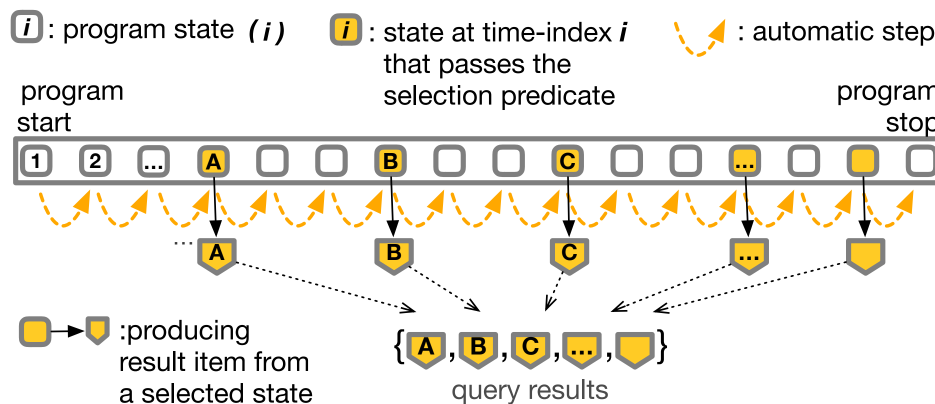


Figure 6.1: Time-traveling query collecting time-indexed data from the program states of a program execution.

Time-traveling from query results. From any result item, and at any moment when debugging, developers are free to time travel. Time-traveling to a result item restores a program execution to the program state denoted by the time-index (a timestamp) from which that item was collected (Figure 6.2). After a time travel, they can continue navigating the execution with conventional tools and techniques (*e.g.*, stepping, breakpoints, etc.) or time-travel to another result item.

6.3 Off-the-shelf Time-traveling Queries

In this section, we present a list of key queries that we elaborated from the literature. We propose these queries as a standard library for developers to explore their program execution. We describe how we implement a representative selection of these queries using the formalism described in Section 6.2.

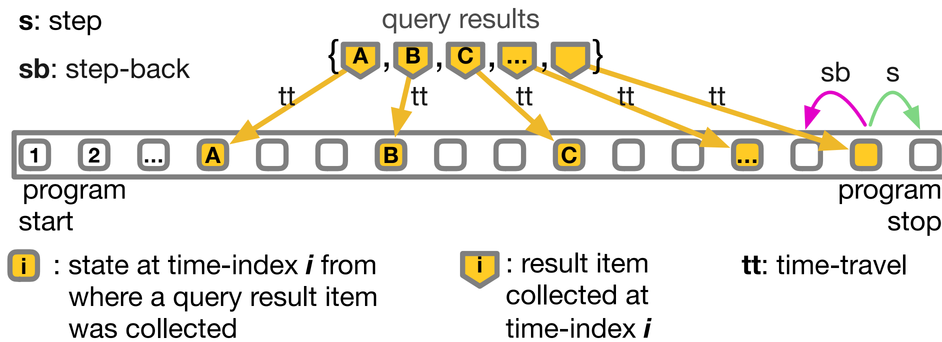


Figure 6.2: Exploring an execution by time-traveling from the result items of a query. After a time travel, developers can perform conventional stepping or another time travel.

6.3.1 Key Time-traveling Queries

We studied the key program comprehension questions that are important for developers [Sillito 2008]. We focus on questions developers ask in object-oriented programming [Kubelka 2014]. We selected 6 questions, for which we use the same numbering as in [Sillito 2008]:

13. When during the execution is this method called?
14. Where are instances of this class created?
15. Where is this variable or data structure being accessed?
19. What are the values of these arguments at run time?
20. What data is being modified in this code?
32. Under what circumstances is this method called, or an exception is thrown?

We analyzed these questions and defined 12 *TTQs*, organized into 4 categories, that aim to support answering those questions. We provide these queries so that developers do not need to write them manually to find answers to their program comprehension questions:

I. Queries over messages (Questions 13, 32)

I.1 Find all messages sent during the execution.

I.2 Find all messages sent with a given selector.

I.3 Find all received messages by any object.

II. Queries over instances creation (Questions 14, 32)

II.1 Find all instance creations.

II.2 Find all instance creations of a class with a given name.

II.3 Find all instance creations of exceptions.

III. Queries over assignments (Questions 15, 19, 20)

III.1 Find all assignments of any variable.

III.2 Find all assignments of variables with a given name.

III.3 Find all assignments of instance variables for instances of a given class.

IV. Queries over assignments for a specific object (Questions 15, 19, 20)

IV.1 Find all assignments of instance variables for the receiver of the currently executed method.

IV.2 Find all assignments of instance variables for a particular object.

IV.3 Find all assignments of a given instance variable for the receiver of the currently executed method.

6.3.2 Executing queries

We enhanced our Time-traveling debugger, *Seeker* (Chapter 4.3), with Time-traveling queries capabilities, adding a new context menu offering the key TTQs (Figure 6.3 a), and a *Query results panel* (Figure 6.3 b).

There are two ways to execute queries: by selecting an off-the-shelf query from the debugger menu or by writing a query directly in Seeker's scripting tab (Figure 6.4). Once a query is executed, the results are shown in the Query Result panel, from where the developer can directly see the queried debugging data, time-travel to the step number of any result, or perform more TTQs in the results context menu (Figure 6.3 c).

6.3.2.1 Executing queries from the debugger menu

The debugger queries menu is populated with queries defined from different sources. For the off-the-shelf query library (Section 6.3), we populate the menu from code in the query library. Alternatively, developers define *user queries*, which are automatically included in the `UserTTQ` submenu.

Seeker debugger opens with the standard debugger view on the left and the time-traveling queries view on the right (Figure 6.3).

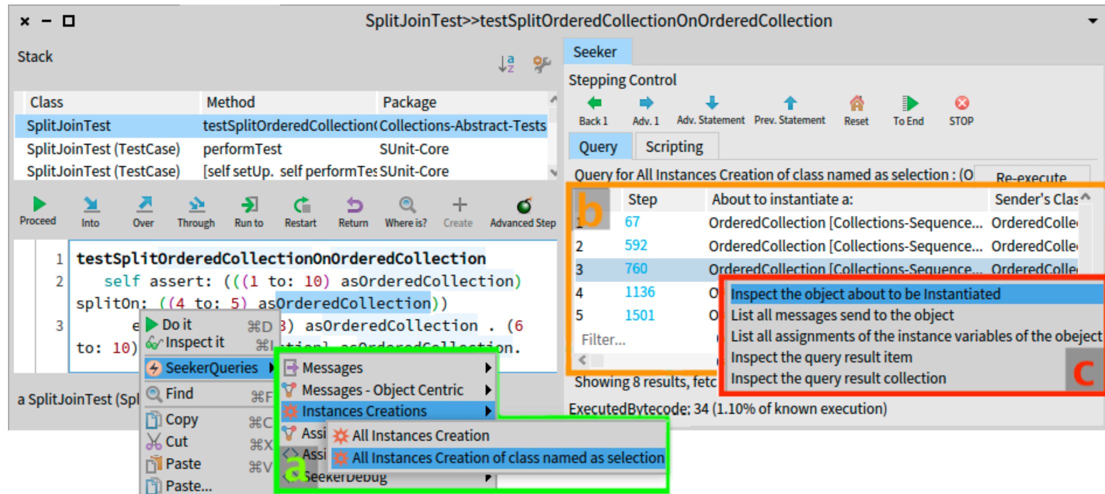


Figure 6.3: Time-traveling queries integration in the debugger. (a) Debugger queries menu. (b) Query results panel. (c) Results context menu.

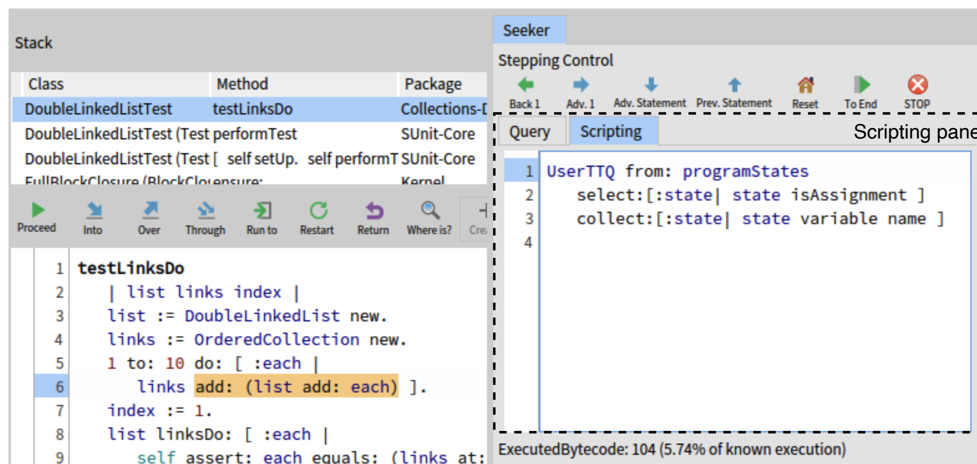


Figure 6.4: Scripting pane in Seeker to write time-traveling queries on the fly.

The code presenter (on the left) exposes a contextual *SeekerQueries* menu. In this menu, users will find queries from the query library proposed in Section 6.3 and all user-defined queries.

Some queries require parameters from the execution context. These parameters are obtained from the selected code in the debugger. For example, *II.2 Find all instance creations of a class with a given name* has a parameter to specify the class name of interest. To execute this query from the debugger's queries menu, developers first select the text containing the name of the class from the code pane, and then the debugger automatically assigns the selection to the query parameter.

Query results are displayed on the right pane of the debugger (Figure 6.3). The results are displayed according to the projection function of the executed query. In the results table, the first column labeled *step* serves as a timestamp to control time-traveling operations. When clicking on the step of a query result line, the debugger time-travels to the moment at which the associated instruction was executed and updates the debugger views (*e.g.,* code, stack) accordingly. From there, developers can perform more debugging actions (such as stepping forward and backward), time-travel again, or execute another query.

6.3.2.2 Writing Time-traveling Queries in the scripting presenter

The scripting presenter is available in the right pane of the debugger (Figure 6.4). There, developers can write and execute queries on the fly directly in the debugger. Developers can therefore ask new questions when they arise while exploring an execution, either from conventional steps or by browsing the results of a query.

To write queries in the debugger, developers have access to a variable named `programStates`, which represents the collection of all possible program states of the debugged program execution. Developers then write queries by using the `programStates` variable as the data source for the query. Figure 6.4 and Listing 6.4 show examples of scripts that use the `programStates` of the scripting presenter and define the selection predicate and projection function in a single expression.

6.4 Time-traveling Queries Implementation

In this section, we describe the implementation details of TTQs.

6.4.1 Time-traveling queries requirements

TTQs require a time-traveling debugger back end that provides the following features:

1. The debugger is able to restore a program execution state to any given time index.

2. An iterable object that represents the sequence of program states of an execution.

3. A unique time index for every executed instruction (bytecode, opcode, abstract syntax tree...), that the debugger records.

4. To support concurrent executions, the sequence of program states is well ordered:

   (a) The back end must control the execution of concurrent instructions and order their associated program state with unique and sequential time indexes.

   (b) The back end must enforce the deterministic order of the instructions of an execution during replay.

To implement our solution, we used *Seeker* (Section 4.2), which fulfills all listed requirements for single-threaded executions (requirements 1, 2, and 3) and already integrates with the Pharo debugger. We used `CurrentState` objects (Described in detail in 4.2.5) to access program execution data. We next describe only the code required to support TTQs, while the rest of the API and implementation of our time-traveling debugger are available in full detail in Chapter 4.

6.4.2 Query implementation

Query objects (Figure 6.5) define the query parameters and an iteration routine comprising three instance variables:

1. `fromSource:` Can be any iterable object (*i.e.,* responds to `do:` message in Pharo).

2. `selectionPredicate:` A block that takes a CurrentState object as an argument and returns true or false.

3. `projectionFunction:` A block that takes a CurrentState object as an argument and returns an object.

The selection predicate and projection function can alternatively use custom objects instead of blocks. Any object that responds to the `value:` message can be used as long as the method respects the parameter and the return.

The `Query` class implements the API to write selection and projection functions and for executing queries.

To create Query objects, there are 2 constructors. The first constructor (Listing 6.5) is used by developers to quickly create custom queries during their debugging sessions.

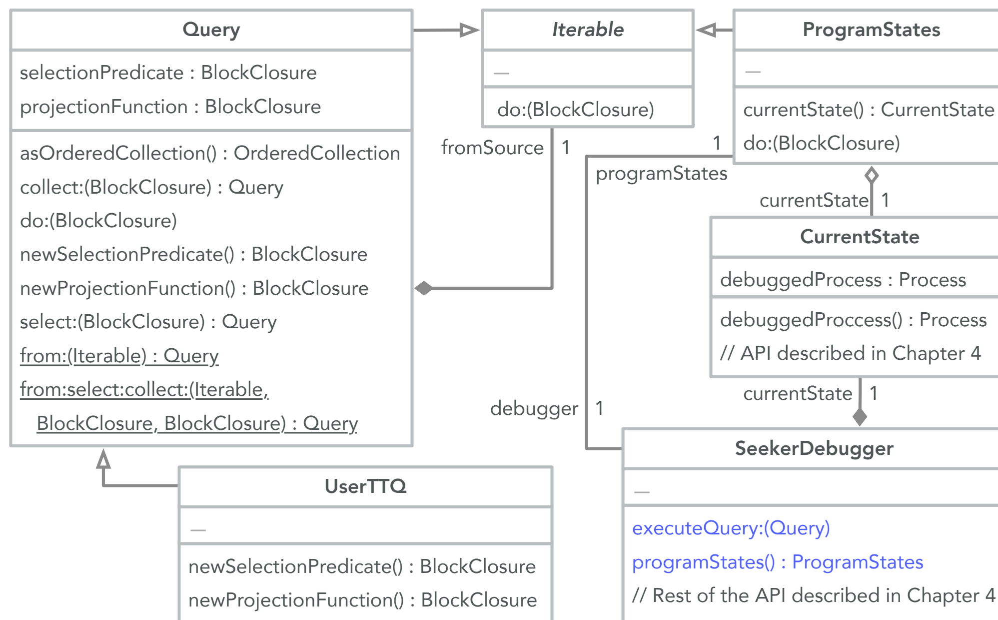


Figure 6.5: Class diagram of the *Time-traveling Queries* mechanism. To make TTQs, Query objects use `Seeker`'s time-traveling features to iterate over the states of a program. The new methods added to Seeker to enable support for TTQs are written in blue. Underlined method are defined class-side.

```
1 Query class >> from: aDataSource select: selectionPredicate collect: projectionFunction
2   ^ self new fromSource: aDataSource;
3     selectionPredicate: selectionPredicate;
4     projectionFunction: projectionFunction;
5     yourself.
6
```

Listing 6.5: Main constructor of the Query class.

```
1 Query class >> from: aDataSource
2    | query |
3    query := self new fromSource: aDataSource; yourself.
4    query selectionPredicate: query newSelectionPredicate;
5    query projectionFunction: query newProjectionFunction;
6    ^ query
```

Listing 6.6: Single-argument constructor, used by subclasses.

The second constructor (Listing 6.6) is reserved for subclasses of Query. This constructor sets the selection predicate and projection function defined in the instance of the query subclass (Lines 4 and 5). The methods `newSelectionPredicate` and `newProjectionFunction` are overridden

by developers when creating Query subclasses. We elaborate further into the subject of creating Query subclasses in Subsection 6.4.5.

Query execution and results production

The iteration routine of a query is defined in the `do:` method (Listing 6.7).

```
1 Query >> do: aBlock
2  fromSource do: [ :srcItem | (selectionPredicate value: srcItem) ifTrue: [
3   aBlock value: (projectionFunction value: srcItem) ]
4
```

Listing 6.7: `Query» do:` method iterate over the items of the `fromSource` object. The method evaluates the block argument on each selected and projected item.

The iteration of a query is performed by iterating over the items of `fromSource`, evaluating the `do:` argument block using the projection of the items that test positive for the selection predicate.

Our Query system follows a deferred execution design[1]. Therefore, the items defined by the query are computed only when required. In our implementation, the `do:` (Listing 6.7) method is the main method triggering such computation.

Since query objects only define the querying operation, the method `asOrderedCollection` (Listing 6.8) is provided for developers to obtain a computed set of results stored in an `OrderedCollection` object. This method internally uses the `do:` method (Line 4), which triggers the computation of the query results.

```
1 Query >> asOrderedCollection
2  | result |
3  result := OrderedCollection new.
4  self do: [: item | result add: item ].
5  ^ result
```

Listing 6.8: `Query » asOrderedCollection` method executes a query, producing an OrderedCollection as a result.

To populate the resulting collection, inside the `do:` method call (Listing 6.8, Line 4), the query iterates over the items of the `fromSource` (As described in Listing 6.7). Still inside the `do:` method, the query evaluates the `selectionPredicate` for each item. If the predicate evaluation returns true, the `projectionFunction` is applied to the item, and the projected output is added to the result collection as defined in the `do:` argument block (Listing 6.8, Line 4).

[1] Deferred execution is a programming concept where an operation or computation is not executed immediately but is delayed until it is explicitly requested or required. For example, Java stream operations like `map`, `filter`, and `reduce` are performed only when a terminal operation is invoked, such as `forEach` or `collect`. Other similar examples include LINQ expressions in .Net and Python Generator expressions.

6.4.3 ProgramStates class

ProgramStates objects are used as data sources for queries, and they use Seeker's time-traveling features to iterate over all the states of a program execution. We list the most important methods in Listing 6.9.

```
"Constructor"
ProgramStates class >> newFrom: aSeekerDebugger
 ^ self new debugger: aSeekerDebugger;
   currentState: aSeekerDebugger currentState);
   yourself.

"Iteration logic"
ProgramStates >> do: aBlock
 [ debugger isExecutionFinished ] whileFalse: [
  aBlock value: currentState.
  debugger stepBytecode]
```

Listing 6.9: Code extract of the implementation of the ProgramStates class. To make ProgramStates an *iterable*, it defines the method `do:`, which encapsulates the debugger stepping logic to traverse all the states of a program.

The constructor `newFrom:` is used by Seeker to return a program states object (Listing 6.10, Lines 9 and 10).

Calling the `do:` method of a ProgramsStates object will evaluate the block, passed as an argument, over every execution state of the program (Lines 8 to 11).

6.4.4 Modifications of Seeker to support TTQs

To enable the integration of TTQs in the debugger, we modified the SeekerDebugger class. In Listing 6.10, we show the most relevant methods added:

```
SeekerDebugger >>  executeQuery: aQuery
  | presentStepNumber result |
  presentStepNumber : = self stepNumber.
  self restart.
  result := aQuery asOrderedCollection.
  self timeTravelTo: presentStepNumber.
  self showInQueryResultsTab: result.

SeekerDebugger >> programStates
  ^ ProgramStates newFrom: self

SeekerDebugger >> showInQueryResultsTab: aCollection
  "Populates the table in the view with the data in aCollection"
```

Listing 6.10: New methods of SeekerDebugger to enable support for TTQs.

The listed code is a simplified version of the original, where we removed UI updating and other accessory code to improve clarity.

As Listing 6.10 shows, when we execute a query, the debugger restarts the execution from the first step (Line 4). The query is executed to produce results (Line 5). To produce results, the query internally uses the ProgramStates object to step over the entire program execution and collect the results (As described in Listing 6.8). To prevent losing the progress of our program exploration, caused by the ProgramStates stepping over all the states of the program, the debugger returns the execution to the same step number that was present before the query was executed (Line 6). Finally, the collected results are displayed in the UI (Line 7).

6.4.5 Implementation of key Time-traveling Queries

Each one of our Key TTQs is defined in a dedicated class. To implement these specialized queries, we subclass the `Query` class and override the `newSelectionPredicate` and `newProjectionFunction` methods. This way, when the single-parameter constructor is used (Listing 6.6), the Query subclass instance is initialized with the overridden selection predicate and projection functions.

In the following, we describe how we implemented two representative queries of our library from Section 6.3.1: Query *III.3* (Section 6.4.5.1) and query *I.2* (Section 6.4.5.2). In each example, we first outline the procedure and then describe the details of their selection predicate and projection function.

6.4.5.1 Finding all assignments to the instance variables of a class

This query finds all assignments to instance variables of any instance of a target class. We first subclassed the `Query` class with a new class `QueryAssignmentsOfInstVarsOfClassName`. This subclass also defines an instance variable named `targetClassName`, used to specify the target class name for logging assignments. Then, we defined a new selection predicate to filter all program states, selecting only the ones corresponding to assignments. Finally, we defined a new projection function that outputs a dictionary with information extracted from the filtered assignments. During the query initialization, we store the selected text from the code presenter into the query's `targetClassName` variable.

The selection predicate. To implement the selection predicate, we override the method `newSelectionPredicate` of `Query` (Listing 6.11). Instead of returning a block with the definition of the selection predicate, this time, the method returns a new dedicated object that encapsulates the definition (Lines 2 to 4).

```
QueryAssignmentsOfInstVarsOfClassName >> newSelectionPredicate
  ^ SelectionForQueryAssignmentsOfInstVarsOfClassName new
    targetClassName: targetClassName;
    yourself
```

Listing 6.11: Overriding the newSelectionPredicate method for the key query *III.3* `QueryAssignmentsOfInstVarsOfClassName`.

This new dedicated object acts as a block, for which it needs to implement the `value:` (Listing 6.12), and also stores the `targetClassName` in an instance variable. In the predicate defined in the `value:` method, we first use the CurrentState API (Chapter 4.2) to determine if a program state corresponds to an instance variable assignment (Lines 2 and 3). If that is the case, we compare the class name of the receiver (the object owning the instance variable) with the `targetClassName` stored in the instance variable (Line 4).

```
SelectionForQueryAssignmentsOfInstVarsOfClassName >> value: state
  ^ state isAssignment and: [
    state assignmentVariable isInstanceVariable and: [
      state receiverClass name == targetClassName ]]
```

Listing 6.12: Definition of the selection predicate in a dedicated class for the key query *III.3* `QueryAssignmentsOfInstVarsOfClassName`.

Projection function. To implement the projection function, we override the `newProjectionFunction`. In a similar manner to the selection predicate, the overridden `newProjectionFunction` returns a dedicated object that defines the projection function instead of a block.

```
QueryAssignmentsOfInstVarsOfClassName >> newProjectionFunction
  ^ ProjectionForAssignmentsOfInstVarsOfClassName new
```

Listing 6.13: Overriding the newProjectionFunction method for the key query *III.3* `QueryAssignmentsOfInstVarsOfClassName`.

The new dedicated object defines the projection logic in the `value:` method (Listing 6.14). We map specific fields of the program states into a dictionary. We use the CurrentState API to collect various information about the assignment, such as the method, class, and package where it occurs, the variable name, its current value, and the new value being assigned. Additionally, we collect a technical field named `bytecodeIndex`, which corresponds to the step number of the instruction. This field is used by the time-travel back end to navigate to the execution point where the assignment took place. For example, when the user clicks the step number of the result line, the debugger performs a time-travel to the step number registered in the result.

```
1 ProjectionForQueryAssignmentsOfInstVarsOfClassName >> value: state
2   ^ { (#selector -> state methodSelector).
3       (#class -> state receiverClass).
4       (#package -> state receiverPackage).
5       (#varName -> state variableName).
6       (#currentValue -> state readVariableValue).
7       (#newValue -> state assignmentValue).
8       (#bytecodeIndex -> state bytecodeIndex)} asDictionary
```

Listing 6.14: Definition of the projection function in a dedicated class for the key query *III.3* QueryAssignmentsOfInstVarsOfClassName.

6.4.5.2 Finding all sendings of a specific message

This query finds all occurrences of a specific message-send, represented by a selector given by the developer. When the query is initialized, we store the selected text from the code presenter into the query's selector variable.

We created a Query subclass QueryAllMessagesSentWithSelector, along with its selection predicate and projection function following the same methodology as in the previously described query.

The selection predicate. In the selection predicate (Listing 6.15), we first check if the current program state corresponds to a message-send (Line 3). In such a case, we return the result of the comparison between the message selector with the one stored in the instance variable selector of the query.

```
1 SelectionForQueryAllMessagesSentWithSelector >> value: state
2   ^ state isMessageSend and: [state messageSelector == selector]
```

Listing 6.15: Definition of the selection predicate for the key query *I.2* QueryAllMessagesSentWithSelector.

In the projection function (Listing 6.16), we collect information about the selected message-sends. Similarly, as in Listing 6.14, we collect and return this information in the form of a dictionary.

```
1 ProjectionForQueryAllMessagesSentWithSelector >> value: state
2   ^ { (#selector -> state messageSelector).
3     (#arguments -> state messageArguments).
4     (#receiver -> state messageReceiver).
5     (#bytecodeIndex -> state bytecodeIndex) } asDictionary
```

Listing 6.16: Definition of the projection function for the key query *I.2* QueryAllMessagesSentWithSelector.

6.4.6 User-defined time-traveling queries

When developers require a specialized query that is not available in the key TTQs collection, they can create their own user-defined query. Developers add new user queries to the debugger queries menu by subclassing the `UserTTQ` class. The new subclasses require overriding the methods `newSelectionPredicate` and `newProjectionFunction` in the same manner we showed for implementing the key TTQs. UserTTQs subclasses are automatically added to the UserTTQs submenu of the debugger queries menu during the debugger initialization.

6.5 Conclusion

In this chapter, we introduced a new debugging mechanism to address standard interactive debugger problems. This new mechanism, which we named *Time-traveling Queries*, combines time-traveling debuggers and queries to bring novel debugging capabilities. We have provided a description of what TTQs are. We explained how they support the debugging activity and the benefits they bring. Then, we described in depth their implementation and how they integrated them into Seeker and into the interactive debugging workflow. Finally, we presented examples of how new queries can be created by developers.

CHAPTER 7

Evaluation of the TTQ-based Debugging Approach

Contents

7.1 Empirical Evaluation 101

7.2 Results and Discussion 106

7.3 Conclusion 112

In previous chapters, we explored the problems of using standard debugging techniques, and we proposed a new debugging approach based on Time-traveling Queries. In this chapter, we evaluate our proposition by conducting a controlled experiment and report our findings.

7.1 Empirical Evaluation

To measure how our proposed TTQ-based debugging approach helps to improve the debugging process by supporting program comprehension, we investigated the following research question:

RQ: *Can we express general program comprehension questions as queries over program executions, and does that improve program exploration regarding developers' efforts, time spent, and precision?*

We ran a quantitative evaluation [Elmqvist 2015], following a repeated-measures design [Seltman 2015] with 34 participants. We asked participants to solve a set of program comprehension tasks with standard debugging tools (*i.e.,* the most common tools shipped with development environments) and another set of similar tasks using our set of queries defined in Chapter 6. For each participant, we measured for each task the time taken to solve that task, the precision of the participant's answer, and the number of debugging actions. We then compared measures using TTQs and standard debugging tools.

7.1.1 Objectives of the experiment

Our objective is to investigate if assisting program exploration with TTQs improves program comprehension compared to using standard debugging tools (abbreviated in the following as SDT). As we investigate the RQ along three dimensions - time, precision, and debugging actions - we have derived three Experimental Research Questions:
ERQ1: Do TTQs improve the precision of answers of program comprehension tasks compared to SDT?
ERQ2: Do TTQs reduce the time employed to answer program comprehension tasks compared to SDT?
ERQ3: Do TTQs reduce the number of actions performed to answer program comprehension tasks compared to SDT?

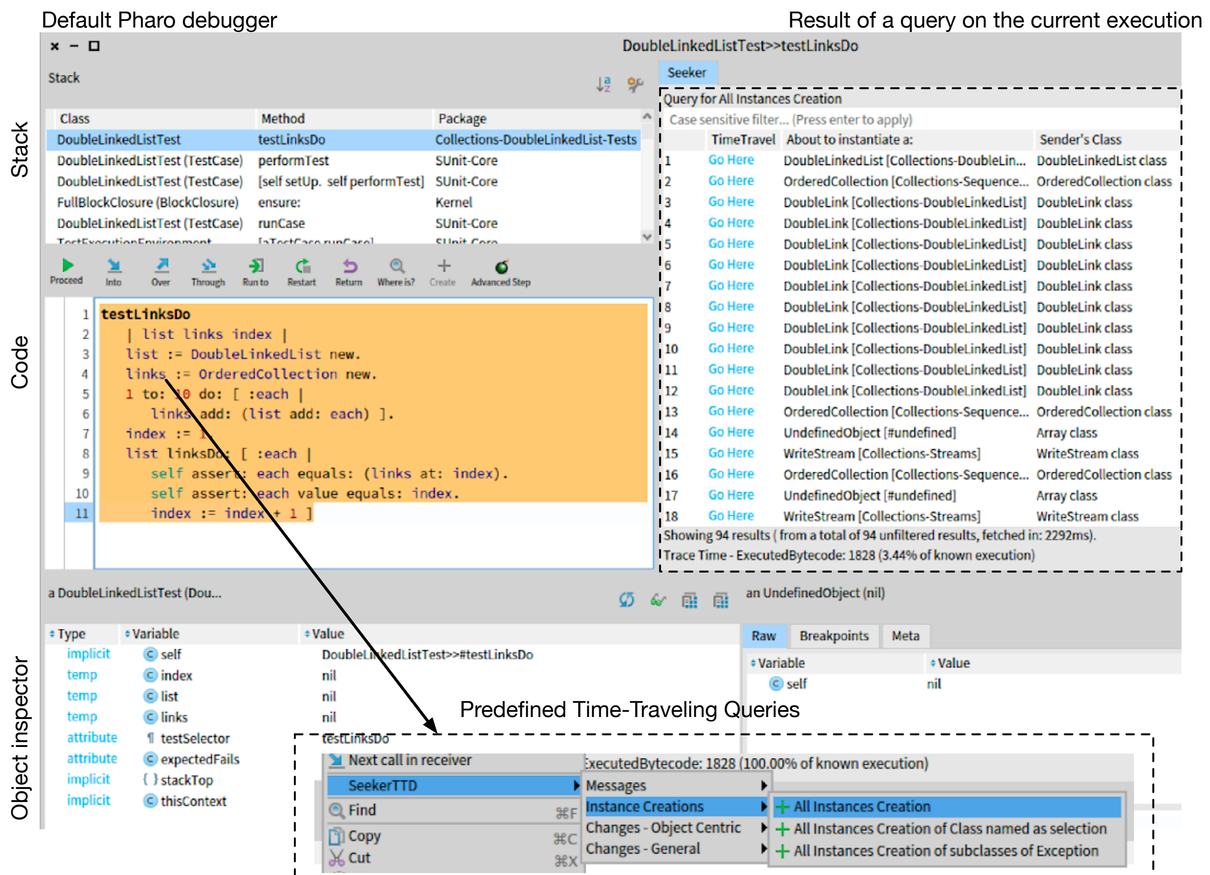


Figure 7.1: UI layout of the debugger supporting TTQs used for the experiment. The queries menu contains only the key queries. For the experiment, solely the query result panel is displayed. Stepping and scripting were disabled.

7.1.2 Experimental design

Our experiment has two parts. First, a task-solving portion following a repeated measures design, and immediately after that, a survey.

Experimental setup. We asked 34 participants to perform two sets of tasks with Pharo 9 under an informal time limit of 90 minutes. Participants performed the experiment remotely without supervision. A pilot participant also performed the same sets of tasks prior to the 34 subjects.

We informed participants that the Pharo images they received were instrumented to log their actions. However, they were not informed what was going to be measured, such as the number of actions they performed to resolve a task or their employed time. We suggested participants use queries during the TTQs tasks without hinting which ones and without enforcing their usage. Participants did not have to manually write or compose queries: the default debugger menu exposed queries. Figure 7.1 shows the integration of time-traveling queries and their results in the default Pharo debugger, used specifically for the experiment. Since the queries menu and the results are embedded in the debugger window, participants did not have to leave the debugger screen to perform debugging actions and navigate the results.

Each task is a program comprehension question for which participants must provide an answer. To solve a task, participants had to open a debugger on a unit test and answer 1 or 2 program comprehension questions.

The two sets of tasks are:

- The *control* set is composed of 5 tasks. We asked participants to provide an answer using exclusively standard Pharo debugging tools.
- The *TTQ* set is composed of 5 tasks. We asked participants to provide an answer using TTQs in addition to the standard Pharo debugging tools.

Each task in a set has a similar counterpart in the other set *i.e.*, we ask a similar question in an equally difficult task between the control and the TTQ sets.

The pilot first performed the tasks following the {control, TTQ} order and reported a carryover effect. The pilot reported that starting with the control set seemed to help to understand what to look for in the TTQ set when answering similar tasks. To limit this learning effect, we randomly assigned 50% of the participants to the {control, TTQ} order and the other 50% to the opposite {TTQ, control} order.

Participants. We gathered 34 participants and 1 pilot. Most of them are Pharo developers with experience ranging from a few months to 20 years (Figure 7.6). Some of them have Pharo development experience but work outside of the Pharo world. Participants had no previous experience with TTQs, and thus discovered it during the experiment. We provided them with a two-minute video on TTQs and their usage, along with TTQs reference material consisting of a 5 slide presentation.

Tasks. We defined 14 tasks (Table 7.1) based on the questions described in Chapter 6.

We made sure that each question we asked was connected to what participants saw when opening the associated test with a debugger. We also made sure that participants would not have to write too much text as an answer, *e.g.,* hundred of values.

We distributed the 14 tasks in different task groups, each group containing 5 pairs of tasks. Each pair of tasks contained one control task and one TTQ task of equivalent difficulty, both tasks targeting the same program comprehension question. We made sure that, within the groups, every task was equally distributed as a control and as a TTQ task. We then randomly assigned a task group to each participant, with an experiment order ({control, TTQ} or {TTQ, control}).

Metrics and measurements. To answer ERQ1, 2 and 3, we defined three metrics: *Score* (precision), *Time*, and *Debugging Actions*. We measured (through execution logs) and calculated these metrics 2 times for each participant: for control and TTQ tasks. Participants had no knowledge of the measured metrics, and data was collected anonymously. All participants gave their consent for the collection of the experimental data.

The *score* is the number of tasks with correct answers. It is an integer value between 0 and 5, calculated as the count of tasks with 100% answer *correctness*. The correctness C of a task t of a participant p is calculated as: $C(p,t) = (cv(p,t)/ev(t))$ where $cv(p,t)$ is the number of correct values provided in the participant's answer for task t, and $ev(t)$ is the number of expected values for task t. To reach 100% correctness, a participant's answer needs to include all the expected values. To define the list of expected values, we first performed all tasks using TTQ and recorded the results. We then compared participants' answers to this list of results. If an answer differed from our list, we analyzed it to understand why the participant arrived at that conclusion. If it could be due to a reasonable level of ambiguity of the question, then we registered it as an additional accepted correct value of the answer. Finally, tasks for which no answer was provided (*e.g.,* the participant failed to answer or did not have enough time) are counted as 0%.

Time corresponds to the time in minutes a participant took to answer a task. It is the chronological time span (obtained from logs) from the beginning of a task until it is answered. The beginning of a task corresponds to the moment a participant starts that task. Participants were not able to see a task description before manually starting it through a graphical control. The end of a task corresponds to the moment a participant provides an answer for that task. We considered that the time to write an answer did not affect our measurements. Finally, we removed periods of inactivity $>$ 5 minutes. For example, if the mouse of a participant did not move for 15 minutes, we considered that the participant was idle for 10 minutes. 2 participants fell in that case, *e.g.,* one participant had a 10 hours period without any event.

T	Test Method and Question	SQ
1	`RSMonitorEventsTest >> testNoTarget` From which domain method is the exception signaled?	S32
2	`STONJSONTest >> testUnknown` From which domain method is each exception signaled	S32
3	`MetacelloVersionNumberTestCase >> testApproxVersion02` How many times is `asMetacelloVersionNumber` called and from which method?	S13
4	`GeneratorTest >> testAtEnd` How many times is `Generator >> atEnd` called and from which methods?	S13
5	`MicToPillarBasicTest >> testHeader` How many instances of `PRHeader` are created? and from which methods?	S14
6	`MicToPillarBasicTest >> testCodeBlock` How many instances of `PRCodeblock` are created? and from which methods?	S14
7	`MicOrderedListBlockTest >> testSingleLevelList2` Which classes from the `Microdown` package are instantiated?	S14*
8	`HiRulerBuilderTest >> testCycle` Which classes from the `Hiedra` package are instantiated	S14*
9	`NSPowScaleTest >> testSqrt` What are the classes of every object receiving the `scale:` message? What are the values of the arguments in each message?	S19
10	`RSNormalizerTest >> testBasic` What are the classes of every object receiving the `color:` message? What are the values of the arguments in each message?	S19
11	`RSCameraTest >> testPosition` What instance variables of the `RSCanvas` object are modified during this test?	S20
12	`RSAttachPointTest >> testVerticalAttachPoint` What instance variables of `RSBox b1` are modified during this test?	S20
13	`OCPragmaTest >> testPragmaAfterBeforTemp` What are the different values assigned to the instance variables: `pragmas source` and `keywordsPositions` of `aRBMethod` object, during the execution?	S15
14	`ContextTest >> testSteppingReturnSelfMethod` What are the different values of the `pc` instance variable of the `newContext` object during this test?	S15*

Table 7.1: Tasks in the controlled experiment. To complete a task, developers were asked to debug the execution of a test case and answer the corresponding question. T is the task id, SQ refers to the question types of [Sillito 2008] selected in Chapter 6. (*): The task question is a variation of the original SQ.

Debugging Actions is an integer representing the sum of program exploration actions performed by a participant to answer a given task. We considered the following actions: configuring breakpoints, modifying methods, executing code, opening debuggers, stepping in the debugger, executing TTQs, time-traveling, and filtering TTQs results.

Post-study survey. We requested participants to fill out a survey after they performed the experiment. First, we gathered factual information such as the participants' professional background and programming experience. Second, we gathered subjective information through the following questions:

- *TTQ: do you find TTQs useful?*
- *TTQ: do you find TTQs intuitive?*
- *Control: what is your confidence level for your answers?*
- *Control: what would be your perceived difficulty level for completing the tasks?*
- *TTQ: what is your confidence level for your answers?*
- *TTQ: what would be your perceived difficulty level for completing the tasks with TTQs?*

Our objective in gathering these subjective data is to put in contrast how participants perceived and trusted TTQs in regards with their measured efficiency during the experiment.

7.2 Results and Discussion

In this section, we analyze the data collected from the experiment[1] and their statistical significance. We then analyze the data collected from the post-study survey.

7.2.1 Experiment results

From the experiment data, we rejected the results of two participants who did not follow the experimental protocol. Logs show these participants did not use TTQs at all. One of them also loaded external advanced tools to perform the tasks. This makes any comparison unreliable. The following analysis is, therefore, based on results from 32 participants out of the 34 who performed the experiment.

Figures 7.2, 7.3, and 7.4 show the differences for each participant respectively for the score, time, and debugging action metrics. For example, in Figure 7.2, 24 participants, over the 32, have a greater score with TTQs than with SDT, 6 have the same score, and only 2 a lower with TTQs. Compared to standard debugging tools, most participants using TTQs seem to reach a better score in less time and by performing fewer debugging actions.

Figure 7.5 shows the averages over all participants for each one of these metrics. On average and compared to standard debugging tools, participants using TTQs obtained a 39% higher score, invested 28% less time, and performed 38% less debugging actions.

[1] The anonymized data are publicly available at https://github.com/willembrinck/2021-TTQs

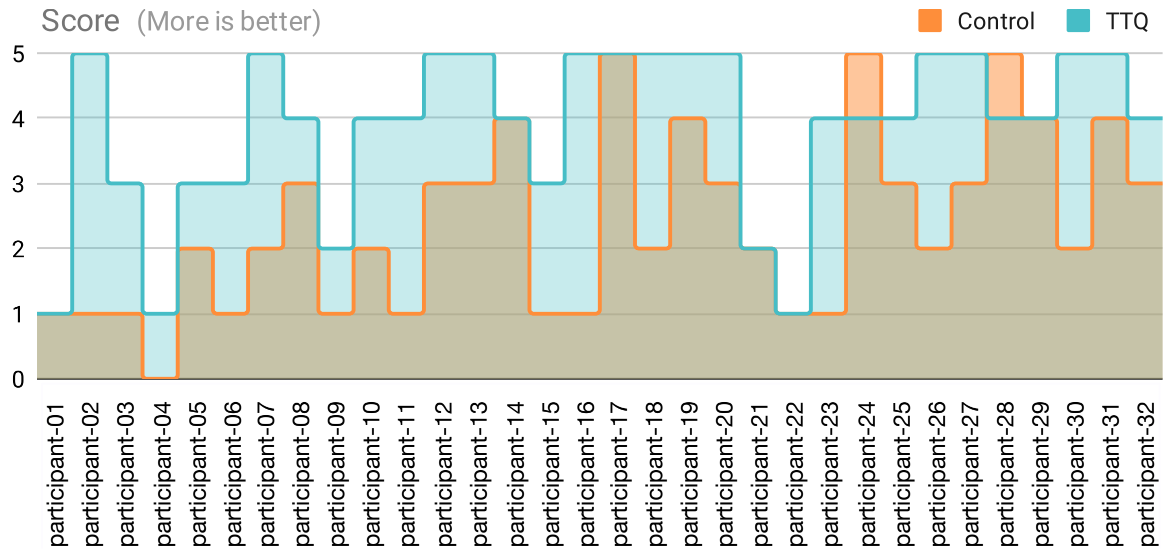


Figure 7.2: Participants scores. In the horizontal axis, participants are sorted in ascending order by their years of experience in Pharo.

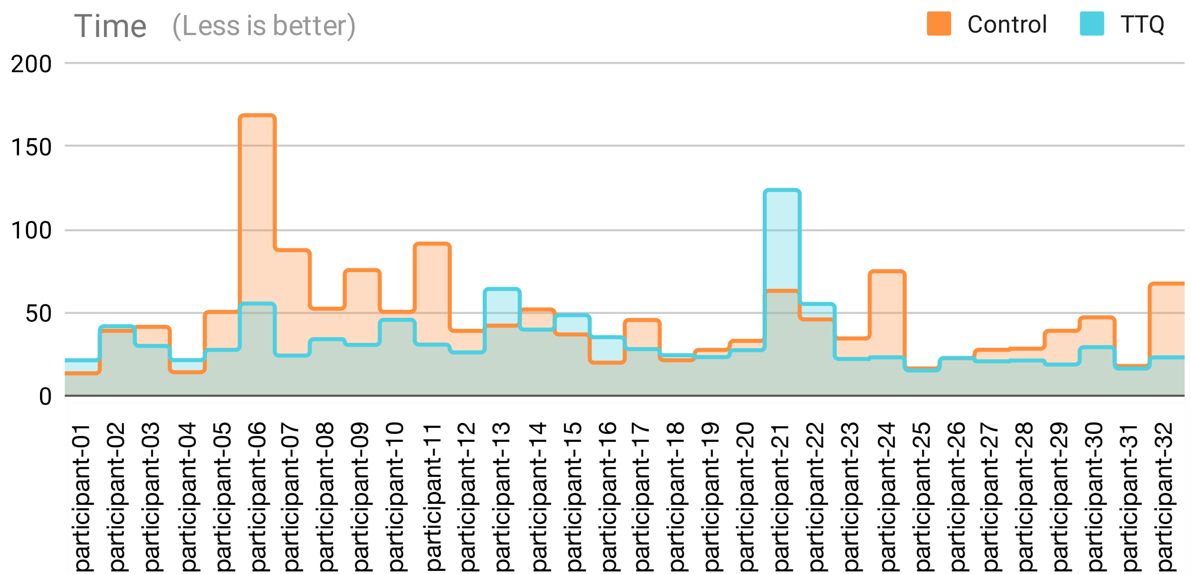


Figure 7.3: Participants total time per sequence, in minutes. In the horizontal axis, participants are sorted in ascending order by their years of experience in Pharo.

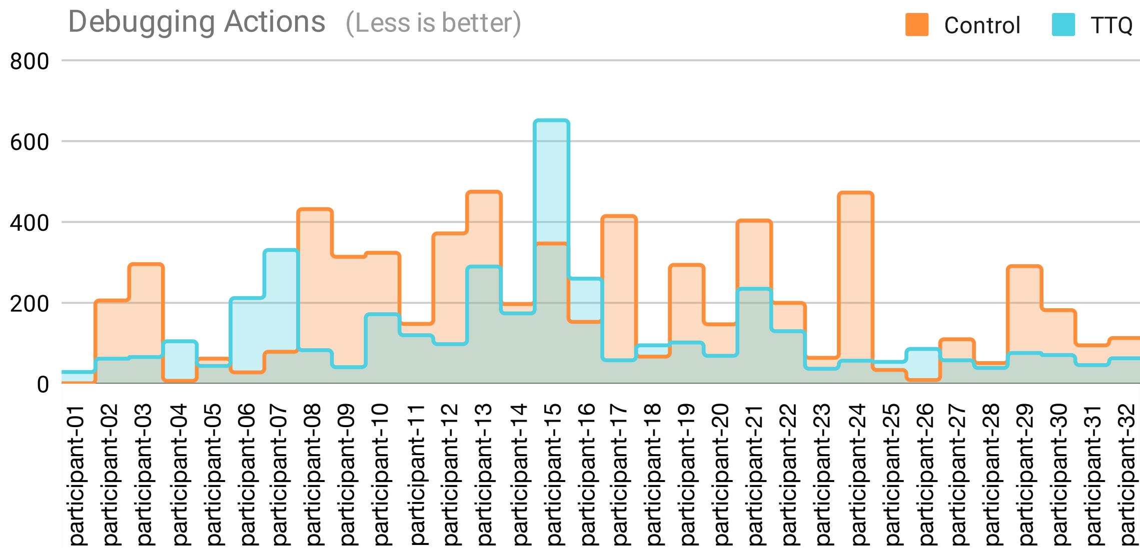


Figure 7.4: Participants total debugging actions. In the horizontal axis, participants are sorted in ascending order by their years of experience in Pharo.

To check if the differences between participants are significant, we formulate the null hypotheses corresponding to our experimental research questions ERQ1, ERQ2, and ERQ3:

> $H_0 1$ for $ERQ1$: The precision of program comprehension tasks is the same with or without TTQs.

> $H_0 2$ for $ERQ2$: The time employed solving program comprehension tasks is the same with or without TTQs.

> $H_0 3$ for $ERQ3$: The number of debugging actions to solve program comprehension tasks is the same with or without TTQs.

Due to the relatively small data sample, we cannot make assumptions about the distribution of the data. Therefore, we performed the nonparametric Wilcoxon signed-rank test to compare the paired differences of the two measurements (control and TTQ). We applied the same methodology for every formulated null hypothesis, considering the differences $TTQ - control$ per participant for each metric (Table 7.2). All p-values are < 0.05, we therefore reject all null hypotheses.

We conclude that to answer program comprehension questions, our *TTQs* improves program exploration regarding developers' efforts, time spent, and precision compared to standard debugging tools.

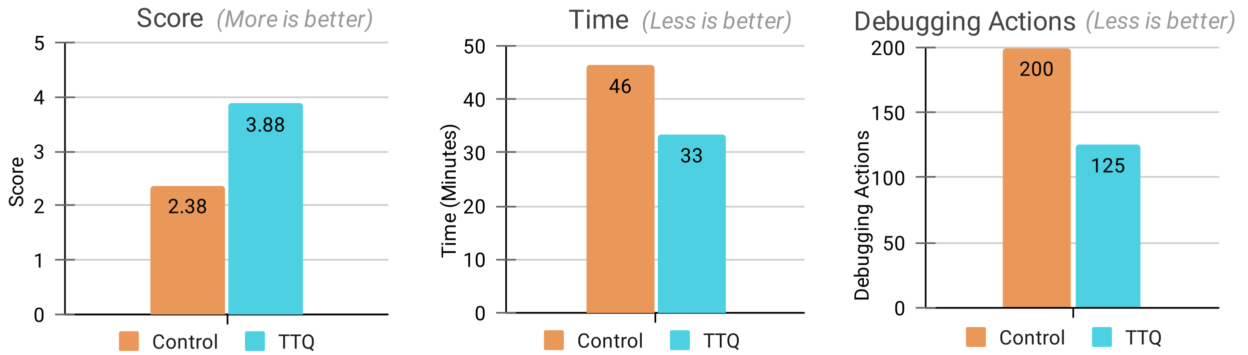


Figure 7.5: Experiment results of each research dimension, averaged.

	N	Z-value	p-value
H_0 $EQ1$ - Score	26	-4.178	<0.001
H_0 $EQ2$ - Time	32	-2.4496	0.014
H_0 $EQ3$ - Debugging Actions	32	-2.3748	0.018

Table 7.2: H_0 rejection table with Wilcoxon signed-rank test values.

7.2.2 Post-study survey

Tables 7.3, 7.4, 7.5 summarize the results of the post-study survey. Most participants found that TTQs were useful and of intuitive usage 7.3. Most participants were more confident in the precision of their answers with TTQs than with standard debugging tools 7.4. Most participants perceived the tasks as less difficult with TTQs than with standard debugging tools 7.5. This is a positive reception, considering the fact that participants were not exposed to the tool before the experiment. This suggests that to answer program comprehension questions, our tool is easy to learn and easier to use than standard debugging tools.

	TTQ Reception	
Rating (More is Better)	***Usefulness***	***Intuitive Usage***
Poor: 1	6%	3%
Fair: 2	6%	0%
Satisfactory: 3	25%	18%
Very good: 4	44%	28%
Excellent: 5	19%	50%

Table 7.3: TTQ Reception rating of the post-study survey. Participants evaluated the usefulness of the tool (Debugger with TTQs) and its intuitive usage.

	Participants' confidence in their answers	
Rating (More is Better)	***Control***	***TTQ***
Not sure at all: 1	6%	6%
2	34%	3%
3	28%	19%
4	19%	41%
They are for sure the correct ones: 5	12%	31%

Table 7.4: Participants' confidence in their answers, according to the post-study survey. Participants evaluated their confidence in the correctness of their answers when using TTQs and when not.

	Perceived difficulty of sequence	
Rating (Less is Better)	***Control***	***TTQ***
Easy: 1	0%	38%
2	12%	28%
3	22%	25%
4	41%	9%
Difficult: 5	25%	0%

Table 7.5: Participants' perceived difficulty of each sequence, according to the post-study survey. Participants evaluated the task difficulty when using TTQs and when not.

7.2.3 Discussion on participant's experience impact on the results

The question of the impact of participants' previous debugging habits and experience has to be discussed. Indeed, if participants had all one or two years of development experience in Pharo, the results could be less significative in the sense that, for someone not used to debug a program or used to a given debugger, a new tool can be as easy/difficult than a traditional one.

The participant population described in Figure 7.6 ensures that we do not have such bias. Figure 7.6 presents the years of experience in Pharo of the participants. It shows that we have nearly an equal number of participants with 0 to 4 years than 4 to 25 years of development with Pharo and related environments such as Squeak (the ancestor of Pharo). Pharo is not taught at the University close to the place where our experience happened. Therefore, having 4 or more years of Pharo developing experience exhibits a solid experience with the system, including familiarity with the standard debugging tools and debugging in general.

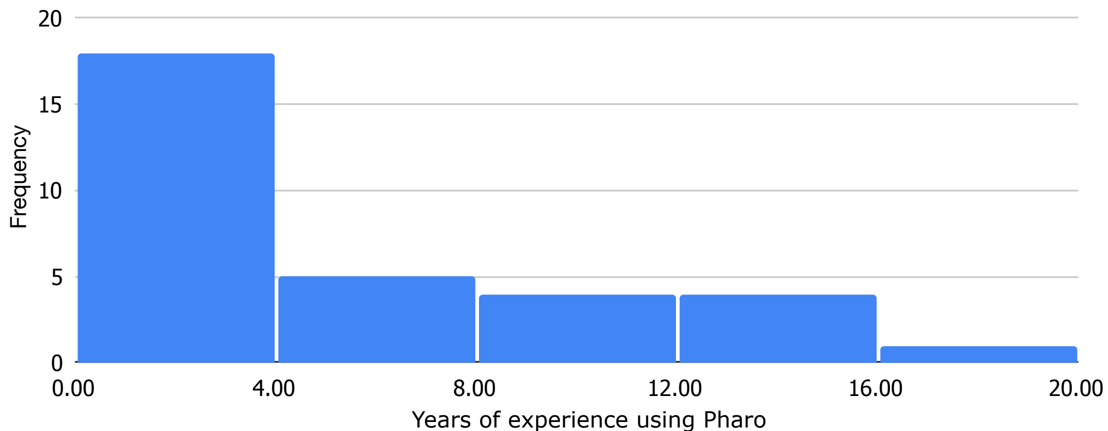


Figure 7.6: Histogram of participants' years of experience in Pharo.

Post-survey results showed that the debugging tool was trusted by most participants. However, some experienced Pharo developers manifested that *to trust the tool result, they would had to validate the results using other tools*, but in doing so, they would break the experiment protocol. This puts the participant in a problematic situation, which can potentially affect the experiment results. As stated in Section 7.2.1, we discarded one participant's results for this reason. We acknowledge the need to minimize these scenarios for future experiences.

7.2.4 Threats to validity

Answers correctness. The list of expected correct values, used to decide if a task answer was correct, was produced using TTQs in addition to participant answers in an iterative process. We tested each listed value by manually finding them in their respective test case, and consequently, we consider them as correct. However, it is not possible to prove the completeness of these lists.

Carryover effect on the experiment order. We balanced the order of the experiment ({control, TTQ} or {TTQ, control}) to avoid a learning effect between the control and the TTQ tasks. However, the data suggest a learning effect in favor of the control tasks. Table **??** presents the means of the score, time and debugging actions metrics for the two experiment orders. Participants performed better on all metrics in their control tasks with the {TTQ, control} order. In particular, they are almost 2 times faster while obtaining a slightly better precision (score) and performing slightly fewer debugging actions. This suggests a learning effect: Participants learned while doing the TTQ tasks first and, therefore, were more efficient during the following control tasks. Participants were not familiar neither with the comprehension tasks nor with the TTQs. Starting with the TTQ set, they learned both during the first part of the experiment. In the {control, TTQ} order, they have two learning phases, one per experiment.

Metric	Sequence	Sequence Order	
		Control→TTQ	***TTQ→Control***
Score	Overall	5.62	6.88
	Control	2.13	2.63
	TTQ	3.5	4.25
Time	Overall	90.4	69.2
	Control	59.9	32.9
	TTQ	30.5	36.3
Debugging Actions	Overall	307.8	342.6
	Control	206.7	192.7
	TTQ	101.1	149.9

Table 7.6: Results according to the experiment order.

Tasks equivalence. Every control task has an equivalent TTQ task in terms of difficulty. This makes it possible to compute per-participant means over the control tasks and over the TTQ tasks and then compute the means difference. However, we assessed this difficulty equivalence based on our own development experience. Formally proving this equivalence is not possible in practice. Comparing per-task means suggests this equivalence (score, time, and debugging actions). Still, there are not enough samples for each task individually to tell if this equivalence is statistically significant.

Remote participation modality. Participants went through the experience remotely. They performed the experiment in full autonomy, using their own equipment and in their own environment. We accounted for inactivity time longer than 5 minutes, but we did not monitor participants for small interruptions and distractions that might affect the results.

7.3 Conclusion

In this chapter, we described the controlled experiment we conducted to evaluate how queries help to answer common program comprehension questions. Results show that with TTQs, developers perform program comprehension tasks more accurately, faster, and with less effort than with standard debugging tools. This answers our research question. The positive feedback regarding the tool indicates that it is easy to learn and use for program comprehension compared to standard debugging tools.

CHAPTER 8

Time-traveling Queries for Specialized Debugging - TTQs Applications

Contents

8.1 Specialized Debugging Tools 113
8.2 A Real World Scenario: Debugging a Meta Compiler 114
8.3 Identifying False Positives in String-Symbol Comparisons 115
8.4 Domain-Specific Queries for the Moose Platform 118
8.5 Queries of Object-centric Debugging 120
8.6 Reproducing the Moldable Debugger Experiments 125
8.7 Conclusion 129

Time-traveling Queries (TTQs) offer a new perspective on debugging by enabling developers to easily traverse program execution history and use problem-specific queries for specialized debugging.

In this chapter, we explore the versatility of the TTQs mechanism. We show examples of how TTQs are used to solve debugging problems that are difficult to approach by using conventional tools. Then, we showcase the development of a suite of specialized debugging tools that address a selection of problem-specific scenarios.

8.1 Specialized Debugging Tools

As described in Chapter 1, developers face an abstraction gap when using traditional debugging tools to find answers to their debugging questions. To address this challenge, domain-specific debugging tools provide debugging actions that are closely aligned with the application domain, thereby reducing the gap. However, crafting specialized debugging tools for each specific domain or extending existing ones is a difficult task, which hinders the availability of these tools to address problem-specific scenarios.

In light of these concerns, we look for an approach that helps to address problem-specific debugging scenarios or that facilitates the creation of new specialized debugging tools.

We claim that TTQs offer a flexible solution for problem-specific debugging concerns. New specialized queries can be used to answer developers' problem-specific debugging questions. TTQs would also facilitate creating new or extending existing specialized debugging tools. To study our claim, we investigate the research question:

RQ: Are TTQs extensible and customizable to tackle domain and problem-specific debugging scenarios?

To conduct our investigation, we first examine how TTQs are used to solve real and specific debugging problems that are difficult to approach with conventional tools. Then, we study existing work from the literature that addresses domain and problem-specific debugging problems [Ressia 2012,Chiş 2014]. From these works, we replicated a portion of their proposals using our TTQs mechanism.

In the following sections, we describe:

1. How Seeker and TTQs were used to debug a meta compiler problematic case.
2. How we built problem-specific queries for MicroDown [Ducasse 2020], a parser and text formatter, to understand aspects of its execution and make design decisions.
3. How we built domain-specific queries for Moose [Anquetil 2020], a software analysis platform whose developers have recurring domain-specific questions.
4. How Seeker provides object-centric [Ressia 2012] debugging capabilities through TTQs.
5. How we use TTQs to reproduce experiments done by the Moldable Debugger [Chiş 2014], a framework to build domain-specific debuggers.

8.2 A Real World Scenario: Debugging a Meta Compiler using TTQs

Seeker has been used by the developers of Druid[1], a meta-compiler. When compiling, Druid uses an intermediate representation in the form of a Control Flow Graph (CFG). During the compilation process, Druid runs many optimizations and changes that CFG. Sometimes, an optimization or a combination of multiple optimization breaks the CFG and produces a compiling error. Most of the time, we cannot easily determine which optimization breaks the CFG, because when

[1] https://github.com/Alamvic/druid

we observe the error, it is already too late. The guilty optimization was applied at some point, followed by other destructive optimizations until the effects of the guilty optimization become visible.

This is tedious to debug with conventional tools. First, developers do not know which optimization introduces the bug. They have to put a lot of breakpoints and go through all of them until they find the guilty optimization. This happens even with more specialized breakpoints, such as conditional breakpoints. Second, applying optimization has effects on the CFG, and those effects are potentially destructive. This means developers cannot evaluate safely the same optimization many times during the same debugged execution. Therefore, if the developers miss the exact point where that guilty optimization is applied, *e.g.*, if they do not put the correct breakpoint or if they step too far in the debugger, they have to restart the whole debugging process.

Using our time-traveling queries library, the Druid developers accessed the history of operations applied to the CFG. They were able to travel between states, before and after each optimization. They found the point where the error appeared, and this information told them exactly where to search in the code.

In this scenario, the TTQs come as a complementary tool to the standard debugger to obtain important knowledge about a bug. In general, any time-traveling debuggers could help avoid restarting the debug process while exploring the program execution. However, this experience shows that our off-the-shelf queries offer a more practical program exploration approach by listing the relevant states.

Seeker proved to be useful for this debugging case. We developed Seeker as an extension of the standard Pharo debugger, providing new debugging capabilities based on time-traveling (Chapter 4) and TTQs (Chapter 6) mechanisms. This experience also shows that by enhancing the standard Pharo debugger with TTQs, we were able to create a new specialized tool that assists developers in finding answers to their debugging questions.

8.3 Identifying False Positives in String-Symbol Comparisons

In Smalltalk, symbols are unique objects. Two symbols with the same representation are the same objects and share the same identity. For example, the code snippet `#aSymbol == #aSymbol` will always return `true`. Conversely, strings are normal objects, and two strings with the same representation do not share the same identity. The code snippet `'aSymbol' == 'aSymbol'` will always return `false`. In Pharo and Squeak, contrary to proprietary Smalltalk implementations, strings are equal to symbols. A string and a symbol with the same representation share the same identity. For example, `'aSymbol' ==`

`#aSymbol` or `'aSymbol' = #aSymbol` will always return `true`. There is an important consequence when migrating proprietary Smalltalk code to Pharo: Any equality check between strings and symbols with the same representation will answer `true` while some programs' semantics wrongly expect that it would return `false`.

MicroDown [Ducasse 2020] is a text parser and formatter whose implementation compares tables of symbols with strings parsed from user inputs. The parser makes decisions depending on equality checks between these symbols and strings. When a string matches a symbol, that string is then used by the parser as equal to that symbol. However, symbols and strings do not answer to the same protocol. This can be a source of bugs because the parser assumes it uses a symbol while manipulating a string. To avoid this problem, the MicroDown developers systematically convert symbols to strings. This can be dangerous for identity-based data structures where the type of the object matters (*i.e.*, `Symbol` or `String`). This is why MicroDown developers needed to identify code locations where strings and symbol equality are mixed and in which data structures.

To solve this problem, we built a query that finds all comparison instructions that compare a string with a symbol. This query uses the selection predicate described by Listing 8.1. We select all program states corresponding to message-sends (Line 4) with the selector #= and one argument (Line 6). This means we compare two objects with the = method. We then consider two cases: the receiver is a symbol and the argument it is compared to is a string, or vice versa (Lines 9 and 10). We select all program states that satisfy one of these two cases (Line 11).

```
1  SelectionForQueryAllStringSymbolEquality>> value: state
2    | selector args rcvClass argClass case1 case2 |
3    (state isMessageSend) ifFalse:[ ^ false].
4    selector := state messageSelector.
5    args := state messageArguments.
6    (selector = #= and: [ args size = 1 ]) ifFalse: [ ^false].
7    rcvClass := state messageReceiver class.
8    argClass := arguments first class.
9    case1 := rcvClass = ByteSymbol and: [ argClass = ByteString ].
10   case2 := rcvClass = ByteString and: [ argClass = ByteSymbol ].
11   ^ case1 or: [case2]
```

Listing 8.1: Selection predicate to find all strings and symbols equality checks.

We executed our query automatically on the MicroDown test suite, composed of 472 unit tests and with code coverage of 66%. The query returned 7 unique cases of string-symbol comparisons in the scope of the MicroDown code covered by tests. Because we executed the query programmatically on a test suite, the query results are not shown in a debugger. We, therefore, programmatically collect and then inspect the query results. We built a projection function to define the information collected in the results. For each one of the equalities detected, the query then creates a result item consisting of a dictionary with the collected

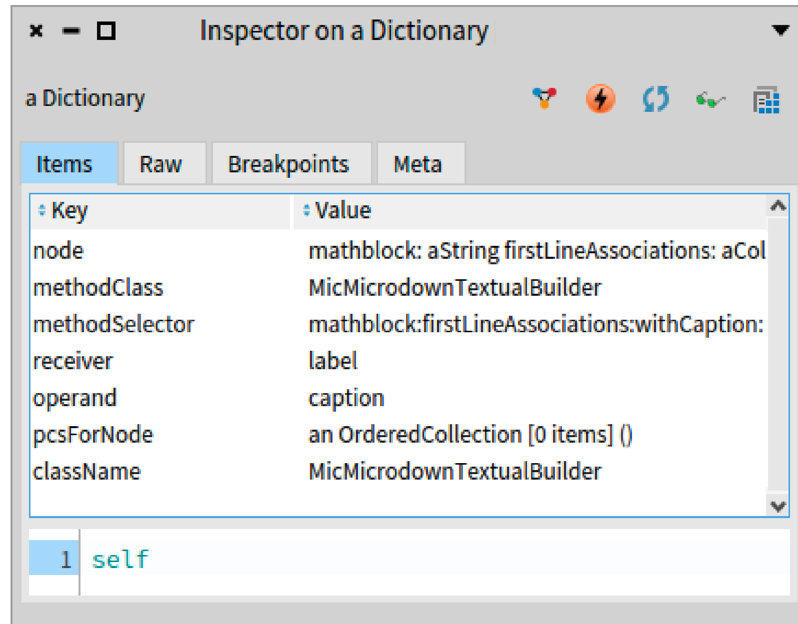


Figure 8.1: Example showing the inspection of a result item of the query that finds all strings and symbols equality.

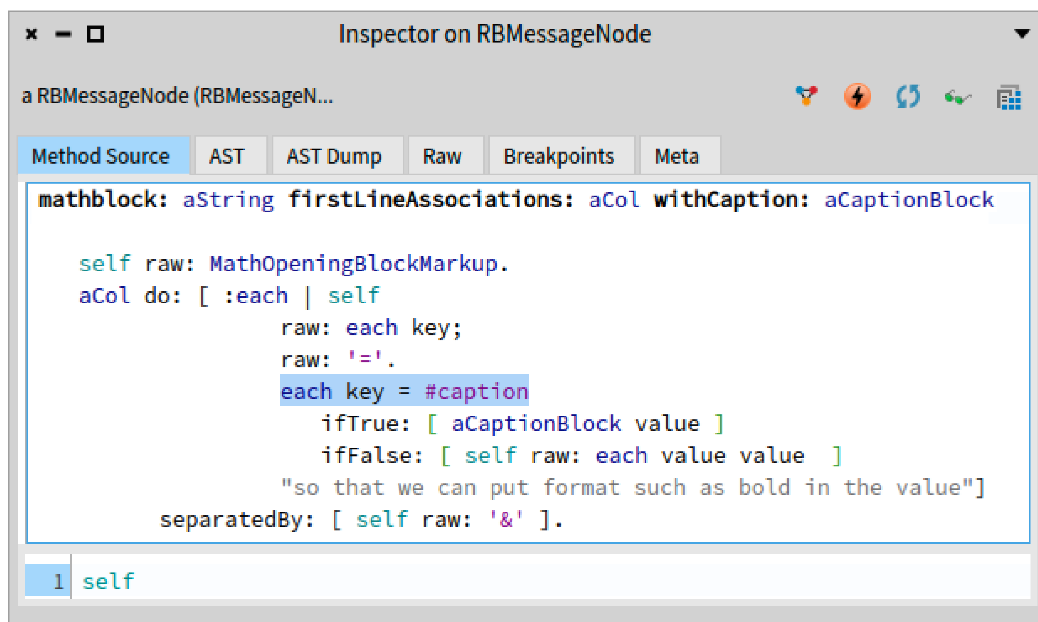


Figure 8.2: Visualization showing the method of a detected string/symbol equality check instruction. The visualization highlights the equality instruction and is accessed by inspecting the *node* value of the query result item (Figure 8.1).

information. Figure 8.1 shows one of the result items with this information. Each query result item corresponds to a comparison instruction where a symbol is matched with a string. In the results, we collect the method and class in which the comparison was detected, the instruction for which the string-symbol comparison was detected, and the values of the compared string and symbol (`receiver` and `operand`). For each incriminated instruction, developers can inspect the projected dictionaries and observe the instruction in a source code editor (Figure 8.2).

From these results, MicroDown developers can make design decisions. They can identify data structures using strings or symbols that are compared to parsed inputs and study how they want to improve their code. They could choose to make sure their data structures all use either strings or symbols and make their parser assume it always manipulates objects of the chosen type. This experiment only aims at helping developers obtain dynamic data from their parser that would be difficult to obtain otherwise. Because the test suite does not cover all the MicroDown code, we cannot guarantee that the queried execution will find all possible string/symbol comparisons. Our query has the merit of being simple to implement and detecting that these comparisons exist in the MicroDown code base.

8.4 Domain-Specific Queries for the Moose Platform

Moose is an extensive platform for software and data analysis[2] [Anquetil 2020]. It relies on the Famix metamodel [Ducasse 2011] to represent software systems as models. The main activity in Moose is to navigate models or to write Moose queries on these models, *e.g.*, to identify deprecated methods still in use in a program.

In Moose, once a model representing a software system is built, very few entities are created or modified. Most Moose actions consist of navigating that model. Consequently, existing TTQs over instance creations or assignments are not suitable. TTQs over messages could be used, but Moose domain experts do not think in these terms since there is only model navigation and no behavioral execution.

8.4.1 Navigating a Moose model

To analyze programs, the first step is to abstract them as models. In the context of object-oriented programs, these models consist of entities (such as classes, variables, or methods) and associations between them (*e.g.*, invocation between methods, inheritance between classes, or access between a method and a variable). One common concern for domain experts is to understand which associations have been navigated during the execution of a Moose query. To help Moose experts, we developed a domain-specific TTQ to retrieve all navigated associations.

[2] https://modularmoose.org

The query uses a selection predicate whose code is detailed in Listing 8.2. At the implementation level, associations are reified as interfaces implemented by objects. An association is navigated when certain navigational slots of these association objects are accessed. Therefore, we look for readings of these slots.

```
1  SelectionForQueryAllAssocsMoose >> value: state
2    | msgRcv msgRcvClass selector rcvClass |
3    state isMessageSend ifTrue: [
4      msgRcv := state messageReceiver.
5      msgRcvClass := msgRcv class.
6      (msgRcvClass usesTrait: FamixTAssociation) ifFalse: [^ false].
7      "Reflective accesses to navigational slots"
8      selector := state messageSelector.
9      (selector = 'perform:') ifTrue: [
10       ^ self slotsNames includes: state firstArg ].
11     "...Other reflective access cases..."
12     "Calls to accessors"
13     ^ self slotsNames includes: selector].
14   "Direct slot reading"
15   state node isVariable ifFalse: [ ^ false].
16   rcvClass := state receiver class.
17   (rcvClass usesTrait: FamixTAssociation) ifFalse: [ ^ false].
18  ^ self slotsNames includes: state varName
```

Listing 8.2: Selection predicate that selects all associations navigation of a Moose model.

In the selection predicate, we select all program states that correspond to an association navigation. These program states must contain an object that reifies an association, *i.e.*, whose class implements the association interface (Lines 6 and 17)[3]. Then, we detect accesses to navigational slots. These particular slots are pre-defined by Moose experts in the `slotNames` method. A navigational slot can be accessed in different manners:

- Reflective accesses (Lines 9-10)[4]. The `perform:` message sends to its receiver the message corresponding to the selector passed as a parameter. If that selector corresponds to the name of one of the navigational slots, then a reflective accessor call for that slot is being performed.

- Calls to accessors, whose selectors are identical to slot names (Line 13). If the name of the message being sent (`selector`) corresponds to the name of one of the navigational slots, then the message-send corresponds to a call to a navigational slot accessor.

[3]In Pharo, the `receiver` (Line 16) represents the object executing a method whose instructions are executed, *e.g.*, reading variables or sending messages. The `messageReceiver` (Line 4) is the object receiving the message of a message-send instruction that is executed within the body of a method.

[4]For simplicity, other cases of reflective accesses are not reported in this script.

- Direct slot reading, *i.e.*, instructions reading a variable pointing to the slot (Lines 15-18). If the name of the variable being read corresponds to the name of one of the navigational slots, then the slot is being read directly.

We wrote an object-centric version of this query for finding specific associations navigation. These queries are examples of non-trivial domain-specific queries, for which we needed to sit with domain experts to understand what questions they want to answer to about their Moose execution, and to understand what we needed to extract and compare from a Moose execution to implement the queries.

8.4.2 Querying a Moose model

In the Moose terminology, querying a model consists of navigating this model, selecting some specific elements satisfying the Moose query and putting these elements in a Moose query result. This result can be of two different types depending on the executed Moose query, either a `MooseGroup` or a `MooseQueryResult`. For domain experts, it is a recurrent question to find out when and what elements are added to a Moose query result. We, therefore, created two TTQs: one to find each time an element is added to one of these two types of collections or of their subclasses. These two queries execute the selection predicate described in Listing 8.3. The selection predicate object defines an accessor `modelClass` which returns the type of the Moose result that the query is interested in, *i.e.*, `MooseGroup` to find all modifications of Moose groups and `MooseQueryResult` to find all modifications of Moose query results. The main selection function (Listing 8.3) uses these methods to select message-sends corresponding to modifications of instances of these model classes. First, we only consider message-sends (Line 3). Then, we check if the message selector corresponds to a Moose collection modifier (Lines 4 and 5). Finally, we check if the receiver is one of the Moose model classes we are interested in (Line 6).

```
1 FindAllModificationsSelection >> value: state
2   | selector |
3   (state isMessageSend) ifFalse: [ ^ false ].
4   selector := state messageSelector.
5   (self mooseCollectionModificationSelectors includes: selector) ifFalse: [ ^ false ].
6   ^ state messageReceiver isKindOf: self modelClass
```

Listing 8.3: Selection predicate to find modifications of a Moose collection.

8.5 TTQs for Object-centric Debugging

Object-centric debugging [Ressia 2012] aims at facilitating the debugging of object-oriented programs by focusing debugging operations on specific objects.

In this section, we describe the problems of object-centric debuggers and how our solution improves these problems. Then, we describe the TTQ-powered object-centric debugging features provided by Seeker.

8.5.1 Problems of object-centric debugers

To debug their programs, developers use debuggers, which traditionally provide a set of standard debugging operations. Some of these debugging operations (*e.g.*, breakpoints) are defined for a class (*e.g.*, in a method) and apply to all instances of that class. Meanwhile, *object-centric debugging* is a technique proposed to improve the debugging of object-oriented systems by scoping object-centric debugging operations to specific objects [Ressia 2012, Costiou 2020]. This helps debugging by reducing user interactions to understand bugs [Ressia 2012, Corrodi 2016], and by exposing and hot fixing buggy objects [Costiou 2018].

However, in practice, object-centric debugging is difficult to use. Developers have to manually explore their executions to find objects to debug. This is tedious and error-prone. Developers have to repeat many times the same program exploration process.

Object-centric debugging operations also generate false positives. For example, an object-centric breakpoint might halt the system many times for a single object. Developers then need to go through numerous haltings until they find useful information for debugging. This is as tedious as manually stepping an execution with a standard debugger.

8.5.2 Our proposition: Enhancing object-centric debugging with TTQs

We claim that enhancing object-centric debugging by time-traveling mechanisms would enable new debugging tools that would improve the debugging of object-oriented programs. To investigate this, we explore the application of TTQs in the context of object-centric debugging. We first use our TTQs mechanism to automatically and systematically explore an execution to find objects. Once objects are found, we define and offer time-traveling debugging operations on them.

8.5.3 Identifying particular objects is challenging: A running example

In the following, we illustrate the difficulties developers go through when they try to comprehend objects' behavior during debugging. We explain how object-centric debugging and time-traveling debugging support these challenges and their limitations. As an example, we use a test method from the Pharo 11 code base:

```
1 SplitJoinTest >> testSplitOrderedCollectionOnOrderedCollection
2  self assert: (((1 to: 10) asOrderedCollection) splitOn: ((4 to: 5) asOrderedCollection))
3    equals: {(1 to: 3) asOrderedCollection. (6 to: 10) asOrderedCollection} asOrderedCollection
```

Listing 8.4: The split-join test of collections in Pharo 11.

In this test, an `OrderedCollection` of 10 elements is split by another `OrderedCollection` with two elements. This operation is expected to produce a new `OrderedCollection` containing both left and right sides of the split operation (each side of the split is an `OrderedCollection`). The developer, trying to understand this execution, wants to obtain information about the behavior and state of instances of `OrderedCollection` during the execution of the test. The standard procedure is to step the execution until the desired object is instantiated. This potentially requires numerous and carefully executed steps before the observation of the desired object can take place. Moreover, the execution of this test produces several instances of `OrderedCollection`, and these instantiation calls are not immediately visible in the test code. It is difficult to track each object, as none of them seems to be stored in a variable. This makes such a manual approach even more tedious.

Using an object-centric debugger, developers can improve over such an approach by using specific breakpoints that halt the execution whenever a class is instantiated. With this technique, developers need to halt a certain number of times to reach the `OrderedCollection` of interest. Once the object of interest has been reached, developers can use object-centric breakpoints to observe the object's behavior and the evolution of its state. One downside of this technique is that it might take many halts to reach the object of interest in the first place and then several other object-centric breakpoints to observe a piece of relevant information. If the developer accidentally misses that relevant information, then the procedure has to be restarted from scratch.

By using a time-traveling debugger, we can improve this approach by navigating back and forth in the execution. If developers miss the observation of one of the collections, or if they pass it and want to observe it again, they reverse the execution and look for that collection without the need to redo everything from the beginning.

Object-centric debugging and time-traveling debugging both provide enhancements over standard debugging tools on their own. We argue that they can complement each other to improve debugging further. In the following, we explore how these techniques can benefit from each other. Then, we describe what is required to enable this new joint approach and how we materialize this approach into a tool.

8.5.4 Debugging objects through time with TTQs

Next, we present our debuggers' object-centric capabilities and explain how TTQs are used to implement object-centric operators.

8.5.4.1 Debugging with object-centric time-traveling queries

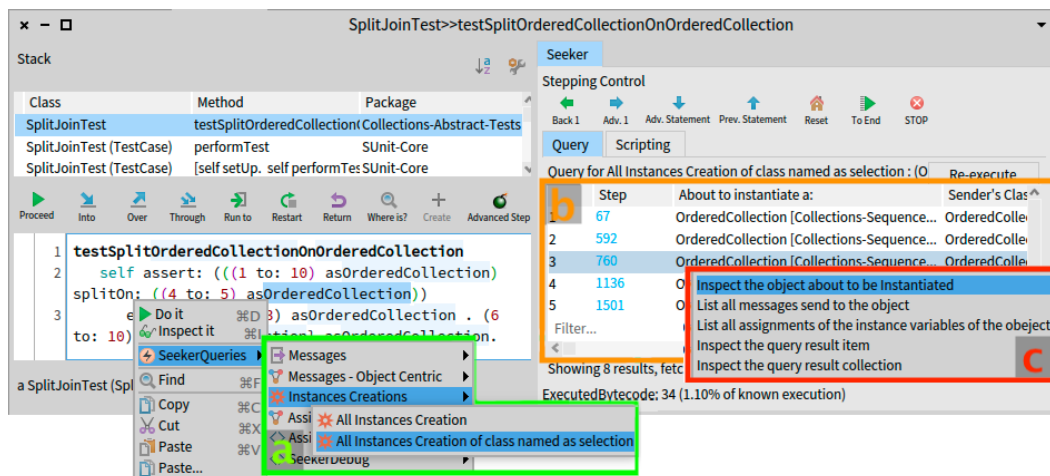


Figure 8.3: *Seeker* interface, showing object-centric operations. To the left, the Time-traveling queries menu(a), available in the code presenter. To the right, the query results table(b) and its context menu that includes object-centric debugging commands(c).

Seeker includes specific queries to ease the task of identifying and tracking objects. These queries are available in the *Seeker Queries* menu (Figure 8.3(a)).

Figure 8.3 shows our debugger opened at the beginning of the execution of the code from Listing 8.4. To find all collections created during the execution of that code, we first select the `OrderedCollection` text in the code presenter. Then we we execute the query *All instances creation of class named as selection* from the context menu (Figure 8.3(a)). Once the query has been executed, the results are displayed in a table (Figure 8.3 `(b)`). These results contain the complete list of all the `OrderedCollection` objects that are instantiated during the execution of the debugged program.

From the results table, we effortlessly have access to each one of these objects. We access these objects by right-clicking the result row and choosing the command *inspect object about to be instantiated* (Figure 8.3 `(c)`). This command opens an inspector on the object corresponding to the selected result row. The list of objects in the results includes all the objects that were instantiated during the previous steps of the execution. However, the results also list the objects that will be instantiated during the remaining steps of the execution. As these latter objects are not yet instantiated, if we choose to inspect them, the debugger automatically advances the execution until the object is instantiated.

Every result entry of a TTQ includes the step number of every registered event. Clicking in the `step` column of any result item will advance or reverse the execution, time-traveling to the registered step number. This feature makes query results act as *bookmarks* of an execution, providing a practical and precise alternative to using breakpoints and tedious stepping operations for program exploration.

8.5.4.2 The Enhanced *Inspector*: A Promising Object-centric Querying Hub

Pharo traditionally offers a graphical utility for object inspection — a view that displays information about the state of an object, its API, and other useful information. During a Seeker debugging session, inspecting any object of the debugged execution will open an enhanced version of the Pharo *Inspector*.

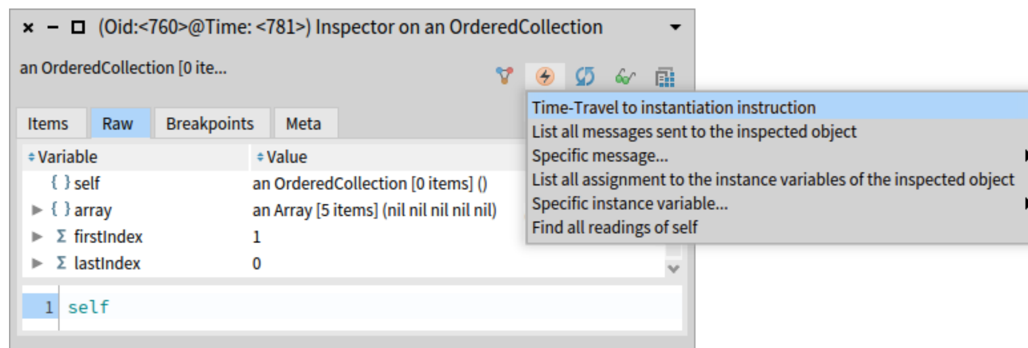


Figure 8.4: Object-centric time-traveling queries menu of the enhanced inspector, offering practical access to relevant object-centric TTQs.

This specialized inspector synchronizes with the time-traveling debugger to keep an updated view of the inspected object. Stepping the execution forward or backward triggers the update of the displayed information. This inspector contains a menu for object-centric operations and queries available for the inspected object (Figure 8.4):

- *Time travel to instantiation instruction*.
- *List all messages sent to the inspected object*.
- Submenu *Specific message...*: displays every message selector that the object responds to. Clicking any particular listed selector launches a query of the type: *List all messages sent with that specific selector to the inspected object*.
- *List all assignments to the instance variables of the inspected object*.
- Submenu *Specific instance variable...*: displays every instance variable of the object. Clicking any particular listed variable name launches a query of the

type: *List all assignments to that specific instance variable of the inspected object.*

- *Find all readings of self.*

8.5.5 Related Work

Object-centric debugging and time-traveling debugging have been the subject of research in recent years. However, and to the best of our knowledge, they have never been combined. Here, we discuss how some of these works relate to ours.

Object Miners [Costiou 2020] is a set of object-centric tools to acquire, capture and replay objects from specific expressions of a program. Developers select (sub)expressions from the program from which objects are automatically captured and debugged during the execution. Specific objects' state can be recorded, then replayed and traversed to observe the evolution of that state. While the tool allows for exploring the past state of an object, it does not support time-traveling operations.

Other works [Lienhard 2006, Lienhard 2008] keep track of changes in objects during the execution of programs. Developers can visualize past states of objects and their behavior. This is a post-mortem approach without time-traveling capabilities.

Expositor [Phang 2013] combines scripting and time-travel debugging to allow programmers to automate complex debugging tasks. From an execution, *Expositor* generates traces that developers manipulate as lists with operations such as map and filter. Our query model provides a similar behavior, where execution traces can be created, operated, and evaluated to generate results. However, *Expositor* does not provide a tool to visualize and exploit results. Furthermore, Expositor does not support object-centric operations.

Whyline [Ko 2004, Ko 2008] offers contextual queries for program comprehension. It offers certain object-centric queries, but they are difficult to extend. In contrast, TTQs are extensible through user-defined TTQs, giving developers the means to create new specialized queries.

8.6 Reproducing the Moldable Debugger Experiments

The Moldable Debugger [Chiş 2014] is a framework to build debuggers that adapt their views and actions to the developers' domain. Therefore, when developers debug their programs, they use a debugger that provides them domain-specific views and actions. The intention is to close the gap between generic debuggers and the domain abstractions that developers use to reason about their program execution, *e.g.*, to ask program comprehension questions.

Each domain-specific debugger, in the sense of moldable debugging, is a composition of:

- A set of predicates that detects particular events in the execution (*e.g.*, the reception of a particular event),
- domain-specific actions associated with predicates (*e.g.*, stepping until the next occurrence of a particular event),
- a domain-specific view, *i.e.*, a specialization of the debugger views adapted for the domain,
- a set of activation predicate, *i.e.*, conditions to detect the presence of specific domain code and activate the corresponding domain-specific debugger,
- and a debugger model specialization whose implementation supports the aforementioned points.

The Seeker debugger can be extended to reproduce Moldable Debuggers. Predicates of moldable debuggers become Seeker selection predicates[5] of domain-specific queries over program states. Domain-specific actions are simulated by time-traveling through queries' results. Views are customizable exactly as the Moldable Debugger ones because we built Seeker with the same Moldable UI [Chis 2016] framework. Activation predicates are equivalent to running queries, and the debugger model specialization is unnecessary as the TTQ implementation is all that is necessary to execute queries.

In the following, we show how we reproduce two Moldable Debugger experiments [Chiş 2014], namely the *Announcement Debugger* and an adaptation of the *PetitParser Debugger* by reusing the predicates described in [Chiş 2014] and their implementation.

8.6.1 Reimplementing the Announcement Debugger with TTQs

In Pharo, *Announcement* is an implementation of the publisher-subscriber pattern. A *subscriber* is an object that registers to an *announcer* (*i.e.*, a publisher) for a type of announcement (*i.e.*, an event). For each subscriber, the announcer stores a *subscription*, which defines an action to perform. When an event occurs, each announcer finds all their subscriptions corresponding to that event and executes the action of these subscriptions. Executing a subscription's action is called *delivering* the subscription.

Code with announcements is notoriously hard to debug, as standard stack-based debugging tools are not adapted to debug event-based programs [Chiş 2013]. The Announcement Debugger is a domain-specific debugger that centers the debugging perspectives on announcements rather than on call stacks.

[5] To avoid ambiguity, we use the complete term *selection predicate* when referring to Seeker's.

Hereafter, we reimplement the *Announcement Debugger [Chiş 2014]* using Time-Traveling Queries. Since we have access to the original implementation[6], we can directly reuse the code of the original predicates. The announcement debugger defines 2 predicates:

- *Detect when any subscription is (about to be) delivered.* Listing 8.5 shows the original implementation. The authors filter the execution context by checking if the receiver is a subscription (Line 2) and if the message selector corresponds to the `#deliver:` method selector (Line 3). This method is used by subscriptions to deliver their announcements to their subscribers. The debugger then checks this predicate against all message-sends of the debugged execution. Listing 8.6 shows our selection predicate that implements the same predicate. First, we check if our program state corresponds to a message-send (Line 2). Then, we check if we are using a delivering selector (Line 3), and finally, we check if the receiver is an instance of a class susceptible to delivering announcements (Lines 4 to 7).

```
1 self session createPredicateForBlock: [ :aContext |
2   aContext receiver isSubscription and: [
3     aContext selector = #deliver: and: [
4       aContext closure notNil ] ] ]
```

Listing 8.5: Original predicate to detect when a subscription is delivered.

```
1 SelectionForQueryAllSubscriptionsDeliveries >> value: state
2 ^ state isMessageSend and: [
3   (state messageSelector asString = #deliver:) and: [
4     {SubscriptionRegistry.
5     AnnouncementSubscription.
6     WeakAnnouncementSubscription} includes:
7       state messageReceiver class ] ]
```

Listing 8.6: Selection predicate to select all subscription deliveries.

- *Detect when a particular subscription is (about to be) delivered to its subscriber.* This predicate requests an action from the developer, who has to select a particular subscription and then ask to step until the delivery of that subscription. Listing 8.7 shows the original implementation. First, the subscription object is obtained from the context (Line 3). Then, the predicate checks if the current receiver executing the current method on the stack is the subscriber (Line 5), *i.e.*, the object that subscribed to the target subscription. Finally, the predicate checks if the selector of the currently executing method on the stack corresponds to the selector defined by the subscription's action (Line 6).

[6] http://smalltalkhub.com/AndreiChis/AnnouncerCentricDebugger/

Listing 8.8 shows our selection predicate that implements the same predicate. First, we check if our program state corresponds to a message-send (Line 2). Then, as in the original implementation, we check if the unique identifier of the receiver is the same as the subscriber's (Line 3) and if the current context selector is the same as the subscriber's (Line 4). The unique object identifier method (`receiverOid`) and context selector method (`messageSelector`) are provided by the program state's API. They are checked against instance variables of the query's selection predicate, *i.e.*, `messageReceiverOid` and `messageSelector`, that are populated by Seeker when developers select the subscription in the debugger.

```
1 subscriptionDelivery
2   | aSubscription |
3   aSubscription := self interruptedContext receiver.
4   ^ self createPredicateForBlock: [ :aContext |
5     aContext receiver == aSubscription subscriber and: [
6       aContext method selector = aSubscription action selector]
```

Listing 8.7: Original predicate to detect when a particular subscription is delivered to its subscriber.

```
1 SelectionForQueryAllOccurrencesOfDeliverSelection >> value: state
2   ^ state isMessageSend and: [
3     state receiverOid = messageReceiverOid and: [
4       state methodSelector = messageSelector ]
```

Listing 8.8: Selection predicate to find all occurrences of a particular subscription being delivered to its subscriber.

In the original Announcement debugger, domain-specific actions rely on the predicates to step until the next subscription delivery or to step until a specific subscription is delivered to a specific subscriber. The debugger has a domain-specific view to help navigate between the domain-specific steps. With our queries, we reproduce this result but with a different approach. Instead of a domain-specific view, the queries return, *e.g.*, the list of all subscription deliveries. Developers are free to navigate back and forth between them using the time-traveling debugger.

8.6.2 Reproducing the *PetitParser Debugger*

The *PetitParser Debugger* [Chiş 2014] is a domain-specific debugger for a parser named PetitParser [Kurš 2013]. The debugger provides parser-specific actions with their corresponding predicates, such as stepping to the next parsing event. We partially reproduced this debugger by creating generic queries (corresponding to the domain-specific action) and their selection functions (implementing the predicate) that apply to any kind of parser. In Pharo, there is

the STON parser [Caekenberghe 2012], the MicroDown parser [Ducasse 2020], PetitParser [Kurš 2013] libraries. All these libraries have parsing similarities: they consume streams, have methods with very similar signatures (*e.g.*, `parse:`), and they require a grammar. Our queries allow for asking domain-specific questions for all parsers sharing these similarities. As for the Announcement Debugger, the original PetitParser Debugger only allowed for stepping until the next domain-specific event, while our queries allow for exploring the whole execution and time-travel between parser events.

8.7 Conclusion

We compiled experiences and performed experiments to investigate the capabilities of TTQs to address specialized debugging concerns. Our finding shows that TTQs are extensible and customizable to tackle domain and problem-specific debugging scenarios. In each case, TTQs were able to express and collect the execution data relevant to each specific scenario. This indicates that TTQs present a promising approach to improving debugging tools by facilitating answering developers' problem-specific debugging questions.

Thesis Conclusion

CHAPTER 9

Conclusion and Future Work

Contents

9.1 Conclusion 133
9.2 Future Work 135

In this chapter, we conclude by bringing together the results of our research and presenting perspectives on our work.

9.1 Conclusion

In this dissertation, we proposed solutions that aim to increase the applicability of time-traveling debuggers and improve the interactive debugging experience. In the following lines, we summarize the research problems discussed in this thesis and our contributions.

In **Part I**, we addressed our first research problem regarding the lack of time-traveling solutions for shared memory systems.

(Chapter 3)

- We provided a comprehensive analysis of the problems that current time-travel solutions face when applied to shared memory systems.
- We identified important properties of selective time-travel back ends for shared memory.

(Chapter 4)

- We built two time-traveling debuggers with configurable support of the identified properties for Pharo, a shared memory system.
  - We presented Seeker, a time-traveling debugger for single-threaded programs that enhances the standard Pharo debugger with time-traveling capabilities.
  - We introduced Executor, a prototype time-traveling debugger for multithreaded programs.

(Chapter 5)

- We conducted experiments to study how programs' behavior is affected by time-travel operations in debuggers exposing or lacking these properties.
- The results of the experiments showed that implementations not possessing one of the properties will not be able to produce correct time-traveling operations, hindering the capability of reproducing program behaviors.

Concluding Part I, the properties we identified provided us with a basic framework that enabled us to connect the different topics of time-traveling in shared memory. These properties cover reversal operations, memory scoping issues, deterministic replay, and support for single and multithreaded programs. The properties served not only to design our own time-traveling solution but also as a helpful analysis tool for existing solutions and their applicability to shared memory systems.

In **Part II**, we addressed our second research problem regarding the difficulties that developers face when using standard interactive debuggers.

(Chapter 6)

- We introduced a new debugging mechanism, Time-traveling Queries (TTQs), that leverages time-travel debugging features to support program exploration and comprehension.
- We proposed a new debugging approach based on TTQs to enhance the interactive debugging experience.
- We described our implementation of TTQs and how we used Seeker to integrate TTQs into the Pharo interactive debugging workflow.

(Chapter 7)

- We evaluated how this new mechanism helps improve program comprehension by conducting a controlled experiment.
- Our results show that answering debugging questions using TTQs is faster, more accurate, and requires less effort compared to using standard debugging tools.

(Chapter 8)

- We explored TTQs versatility by showing how TTQs were used to address problem-specific debugging scenarios.
- We reported experiences showing that TTQs are suitable for addressing problematic debugging scenarios that are difficult to approach using standard debugging tools.

- We conducted experiments replicating problem-specific debugging solutions in the literature, showing that TTQs can be used to produce specialized debugging tools.

Concluding Part II, our proposed Time-traveling Queries mechanism enabled a new interactive debugging approach. TTQs showed promising results in improving debugging efficiency by facilitating program comprehension and exploration tasks. Moreover, they showed to be versatile, capable of addressing problematic debugging scenarios, and able to match specialized operations of problem-specific debugging tools.

9.2 Future Work

In this section, we list prospective subjects pushing forward our work.

9.2.1 Enhancing performance in our time-travel solution

Faster selective execution reversal

Improving the efficiency of selective execution reversal should be explored further. To the best of our knowledge, snapshotting is known as the most efficient approach to perform execution reversals. However, as we discussed, snapshotting is not a viable option for shared memory systems. Conversely, our prototype implementation is slow because it observes each step of the debugged program, controlling its execution and tracking state changes. We speculate that this kind of solution is generally avoided in favor of snapshotting techniques for performance reasons.

In practice, our Pharo implementation is fast enough to debug unit tests. This already enables the full potential of time-travel debugging for understanding complex pieces of code involved in unit tests. However, it does not scale for longer executions involving many calculations. Studying how to improve the performance aspect of our solution would enable the production of a prototype usable in the general case. Furthermore, exploring other approaches that could be used as an alternative to snapshotting for time-traveling in shared memory systems could produce interesting and potentially faster solutions.

Instrumentation-based implementations to improve performance

Our time-traveling debuggers follow an interpreter-based approach to advance debugged executions and to produce the time-traveling recorded program data. This approach is arguably slower than instrumentation-based ones, such as [O'Callahan 2017, UDB 2023]. Changing our implementations to follow an instrumentation approach would bring a faster solution. However, we have not

explored how to do this yet. Studying how a solution with equivalent features can be implemented using instrumentation would produce an implementation with improved performance.

9.2.2 Leveraging the TTQs-based debugging approach

Exploring new TTQs applications for program comprehension

Program comprehension is gained by performing static analysis of a program code and dynamic analysis of its execution [Richner 2002, Richner 1999, Cornelissen 2007]. Several tools and techniques offer support for these activities. Nowadays, popular IDEs are shipped with interactive debuggers, and developers use them to perform program comprehension tasks. Our contribution seeks to support the interactive debugging workflow by enabling dynamic analysis capabilities. TTQs can be used to produce trace information to feed dynamic analysis techniques and visualizations, incorporating their advantages within an interactive debugging workflow. We already explored interesting applications in Chapter 8. However, we are certain that there are many new and exciting applications to discover.

Enabling time-traveling queries for echo debugging

Echo-debugging [Dupriez 2020] supports the detection of regression bugs by comparing the executions of two different versions of a program. In the echo-debugging approach, the two versions of the program (the failing version and a previous working one) are loaded and executed simultaneously in separate environments. Developers explore the executions, which are run in parallel, following an interactive approach. During this, the debugger detects and reports the differences in the programs' control flow. As we have described in Chapter 3, system calls can introduce non-deterministic program behavior, which would affect the comparison of the programs' executions. This problem is discussed in their work as a limitation of their solution. To solve such a problem, and as their prototype is built in Pharo, our time-travel back end presents a promising solution. In our work, our implementation enforces the deterministic replay of the execution of a program. However, if a new different time-travel session is started, *i.e.,* a new execution of the same program, the program will not necessarily follow the expected control flow during its first execution of the session, as there are no logs yet in the back end to enforce program behavior. Therefore, investigating how to extend our time-travel solution to support enforcing behavior among different executions would solve the determinism problems in echo debugging. Ultimately, this would allow TTQs to be used in the context of echo debugging, generating opportunities to investigate new specialized queries.

Evaluating new time-traveling queries

In our TTQs evaluation experiment, the proposed tasks were designed based on a subset of common questions developers ask while debugging a program. They do not cover the complete set of problems developers face during their debugging sessions. Even though the experiment result validates the TTQs approach, the specific measures improvements are significant only in the context of these questions. In Chapter 8, we showed that TTQs can be applied to a variety of problem-specific debugging scenarios. However, it is interesting to know precisely how much TTQs help in those scenarios. New focused evaluations are required to investigate if TTQs improve the debugging experience in such problem-specific debugging scenarios. Moreover, new evaluations could be performed on debugging scenarios related to a different set of debugging questions or requiring more complex debugging actions.

Bibliography

[Alice 2004] Alice. *The Alice Project*. http://www.alice.org, 2004.

[Aman 2020] Bogdan Aman, Gabriel Ciobanu, Robert Glück, Robin Kaarsgaard, Jarkko Kari, Martin Kutrib, Ivan Lanese, Claudio Antares Mezzina, Łukasz Mikulski, Rajagopal Nagarajan*et al.* *Foundations of reversible computation*. In International Conference on Reversible Computation, pages 1–40. Springer, Cham, 2020.

[Anquetil 2020] Nicolas Anquetil, Anne Etien, Mahugnon Honoré Houekpetodji, Benoît Verhaeghe, Stéphane Ducasse, Clotilde Toullec, Fatija Djareddir, Jèrôme Sudich and Mustapha Derras. *Modular Moose: A new generation of software reengineering platform*. In International Conference on Software and Systems Reuse (ICSR'20), numéro 12541 de LNCS, December 2020.

[Arya 2017] KAPIl Arya, Tyler Denniston, Ariel Rabkin and Gene Cooperman. *Transition Watchpoints: Teaching Old Debuggers New Tricks*. The Art, Science, and Engineering of Programming, vol. 1, no. 2, July 2017.

[Barr 2014] Earl T. Barr and Mark Marron. *TARDIS: Affordable Time-Travel Debugging in Managed Runtimes*. In Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'14), volume 49, pages 67–82. ACM, oct 2014.

[Barr 2016] Earl Barr, Mark Marron, Ed Maurer, Dan Moseley and Gaurav Seth. *Time-travel debugging for JavaScript/Node.js*. In Proceedings of the International Symposium on Foundations of Software Engineering, pages 1003–1007, nov 2016.

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Caekenberghe 2012] Sven Van Caekenberghe. *Smalltalk Object Notation (STON)*, 2012.

[Chiş 2013] Andrei Chiş, Oscar Nierstrasz and Tudor Gîrba. *Towards a Moldable Debugger*. In Proceedings of the 7th Workshop on Dynamic Languages and Applications, 2013.

[Chiş 2014] Andrei Chiş, Tudor Gîrba and Oscar Nierstrasz. *The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers*. In Software Language Engineering, pages 102–121. Springer, 2014.

[Chis 2016] Andrei Chis. *Moldable tools*. PhD thesis, University of Bern, 2016.

[Cornelissen 2007] Bas Cornelissen. *Dynamic Analysis Techniques for the Reconstruction of Architectural Views*. In Proceeding of Working Conference on Reverse Engineering (WCRE). IEEE, 2007.

[Corrodi 2016] Claudio Corrodi. *Towards Efficient Object-Centric Debugging with Declarative Breakpoints*. In SATToSE 2016, 2016.

[Costiou 2018] Steven Costiou. *Unanticipated behavior adaptation : application to the debugging of running programs*. PhD thesis, Université de Bretagne occidentale - Brest, November 2018.

[Costiou 2020] Steven Costiou, Mickaël Kerboeuf, Clotilde Toullec, Alain Plantec and Stéphane Ducasse. *Object Miners: Acquire, Capture and Replay Objects to Track Elusive Bugs*. Journal of Object Technology, vol. 19, no. 1, pages 1:1–32, July 2020.

[Devietti 2009] Joseph Devietti, Brandon Lucia, Luis Ceze and Mark Oskin. *DMP: Deterministic shared memory multiprocessing*. In Proceedings of the 14th international conference on Architectural support for programming languages and operating systems, pages 85–96, 2009.

[Ducasse 2011] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval and Tudor Girba. *MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family*. Rapport technique, RMod – INRIA Lille-Nord Europe, 2011.

[Ducasse 2020] Stéphane Ducasse, Laurine Dargaud and Guillermo Polito. *Microdown: a Clean and extensible markup language to support Pharo documentation*. In Proceedings of the 2020 International Workshop on Smalltalk Technologies, 2020.

[Dupriez 2019] Thomas Dupriez, Guillermo Polito, Steven Costiou, Vincent Aranega and Stéphane Ducasse. *Sindarin: A Versatile Scripting API for the Pharo Debugger*. In International Symposium on Dynamic Languages (DSL'19), pages 67–79. ACM, 2019.

[Dupriez 2020] Thomas Dupriez, Steven Costiou and Stéphane Ducasse. *First Infrastructure and Experimentation in Echo-debugging*. In Proceedings of the 2020 International Workshop on Smalltalk Technologies, 2020.

[Elmqvist 2015] Niklas Elmqvist and Ji Soo Yi. *Patterns for visualization evaluation*. Information Visualization, vol. 14, no. 3, pages 250–269, 2015.

[Engblom 2012] Jakob Engblom. *A review of reverse debugging*. In System, Software, SoC and Silicon Debug Conference (S4D), 2012, 2012.

[Goldberg 1983] Adele Goldberg and David Robson. Smalltalk 80: the language and its implementation. Addison Wesley, Reading, Mass., May 1983.

[Goldberg 1984] Adele Goldberg. Smalltalk 80: the interactive programming environment. Addison Wesley, Reading, Mass., 1984.

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

[Google 2023] Google. *Chrome Javascript Debugger*. https://www.google.com/chrome/, 2023.

[Hammond 2004] Lance Hammond, Vicky Wong, Mike Chen, Brian D Carlstrom, John D Davis, Ben Hertzberg, Manohar K Prabhu, Honggo Wijaya, Christos Kozyrakis and Kunle Olukotun. *Transactional memory coherence and consistency*. ACM SIGARCH Computer Architecture News, vol. 32, no. 2, page 102, 2004.

[Herlihy 1993] Maurice P. Herlihy and J. Eliot B. Moss. *Transactional Memory: Architectural Support For Lock-Free Data Structures*. In Proceedings of the 20. Annual International Symposium on Computer Architecture, pages 289–300, 1993.

[Hofer 2006] Christoph Hofer, Marcus Denker and Stéphane Ducasse. *Design and Implementation of a Backward-In-Time Debugger*. In Proceedings of NODE'06, volume P-88 of *Lecture Notes in Informatics*, pages 17–32. Gesellschaft für Informatik (GI), September 2006.

[Huang 2013] Jeff Huang, Charles Zhang and Julian Dolby. *Clap: Recording local executions to reproduce concurrency failures*. Acm Sigplan Notices, vol. 48, no. 6, pages 141–152, 2013.

[Ingalls 2008] Daniel Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari and Tommi Mikkonen. *The lively kernel a self-supporting system on a web page*. In Workshop on Self-sustaining Systems, pages 31–50. Springer, 2008.

[King 2005] Samuel T King, George W Dunlap and Peter M Chen. *Debugging operating systems with time-traveling virtual machines*. In Proceedings of the 2005 USENIX Technical Conference, pages 1–15, 2005.

[Ko 2004] Andrew J. Ko and Brad A. Myers. *Designing the whyline: a debugging interface for asking questions about program behavior*. In Proceedings of the 2004 conference on Human factors in computing systems, pages 151–158. ACM Press, 2004.

[Ko 2008] Andrew J. Ko and Brad A. Myers. *Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior*. In In Proceedings of the 30th International Conference on Software Engineering, ICSE 08, 2008.

[Kubelka 2014] Juraj Kubelka, Alexandre Bergel and Romain Robbes. *Asking and Answering Questions During a Programming Change Task in the Pharo Language*. In Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU '14, pages 1–11, New York, NY, USA, 2014. ACM.

[Kurš 2013] Jan Kurš, Guillaume Larcheveque, Lukas Renggli, Alexandre Bergel, Damien Cassou, Stéphane Ducasse and Jannik Laval. *PetitParser: Building Modular Parsers*. In Deep Into Pharo, page 36. Square Bracket Associates, September 2013.

[Lanese 2018] Ivan Lanese. *From reversible semantics to reversible debugging*. In Reversible Computation: 10th International Conference, RC 2018, Leicester, UK, September 12-14, 2018, Proceedings 10, pages 34–46. Springer, 2018.

[Lauwaerts 2023] Tom Lauwaerts, Carlos Rojas Castillo, Elisa Gonzalez Boix and Christophe Scholliers. *Out-of-Place Debugging on Constraint Devices with the EDWARD Debugger*. In Proceedings of the 1st ACM International Workshop on Future Debugging Techniques, pages 3–4, 2023.

[Lencevicius 1997] Raimondas Lencevicius, Urs Hölzle and Ambuj K. Singh. *Query-Based Debugging of Object-Oriented Programs*. In Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97), pages 304–317, New York, NY, USA, 1997. ACM.

[Lencevicius 1999] Raimondas Lencevicius, Urs Hölzle and Ambuj Kumar Singh. *Dynamic Query-Based Debugging*. In R. Guerraoui, editeur, Proceedings of European Conference on Object-Oriented Programming (ECOOP'99), volume 1628 of *LNCS*, pages 135–160, Lisbon, Portugal, June 1999. Springer-Verlag.

[Lencevicius 2003] Raimondas Lencevicius, Urs Hölzle and Ambuj K Singh. *Dynamic query-based debugging of object-oriented programs*. Automated Software Engineering, vol. 10, pages 39–74, 2003.

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

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

[Lienhard 2008] Adrian Lienhard, Tudor Gîrba and Oscar Nierstrasz. *Practical Object-Oriented Back-in-Time Debugging*. In Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08), volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[Lienhard 2009] Adrian Lienhard, Julien Fierz and Oscar Nierstrasz. *Flow-Centric, Back-In-Time Debugging*. In Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009, volume 33 of *LNBIP*, pages 272–288. Springer-Verlag, 2009.

[McCarthy 1960] John McCarthy. *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.* CACM, vol. 3, no. 4, pages 184–195, April 1960.

[McConnell 2004] Steve McConnell. Code complete. Pearson Education, 2004.

[Microsoft 2023] Microsoft. *Visual Studio IDE*. https://visualstudio.microsoft.com/vs/, 2023.

[Montesinos 2008] Pablo Montesinos, Luis Ceze and Josep Torrellas. *Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently*. ACM SIGARCH Computer Architecture News, vol. 36, no. 3, pages 289–300, 2008.

[O'Callahan 2017] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll and Nimrod Partush. *Engineering record and replay for deployability: Extended technical report*. arXiv preprint arXiv:1705.05937, 2017.

[O'Dell 2017] Devon H O'Dell. *The Debugging Mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills.* Queue, vol. 15, no. 1, pages 71–90, 2017.

[Oracle 2023] Oracle. *Java*. https://www.java.com/en/, 2023.

[PDB 2023] Python PDB. *PDB*. https://docs.python.org/3/library/pdb.html, 2023.

[Phang 2013] K. Y. Phang, J. S. Foster and M. Hicks. *Expositor: Scriptable time-travel debugging with first-class traces*. In 2013 35th International Conference on Software Engineering (ICSE), pages 352–361, may 2013.

[Pothier 2007] Guillaume Pothier, Éric Tanter and José Piquer. *Scalable Omniscient Debugging*. Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07), vol. 42, no. 10, pages 535–552, 2007.

[Pothier 2011] Guillaume Pothier and Éric Tanter. *Summarized trace indexing and querying for scalable back-in-time debugging*. In European Conference on Object-Oriented Programming, pages 558–582. Springer, 2011.

[Python 2023] Python. *Python*. http://www.python.org, 2023.

[Ressia 2012] Jorge Ressia, Alexandre Bergel and Oscar Nierstrasz. *Object-Centric Debugging*. In Proceeding of the 34rd international conference on Software engineering, ICSE '12, 2012.

[Richard Stallman 2003] Stan Shebs Richard Stallman Roland Pesch. Debugging with gdb. Gnu Press, 2003.

[Richner 1999] Tamar Richner and Stéphane Ducasse. *Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information*. In Hongji Yang and Lee White, editeurs, Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99), pages 13–22, Los Alamitos CA, September 1999. IEEE Computer Society Press.

[Richner 2002] Tamar Richner. *Recovering Behavioral Design Views: a Query-Based Approach*. PhD thesis, University of Bern, May 2002.

[Savidis 2021] Anthony Savidis and Vangelis Tsiatsianas. *Implementation of Live Reverse Debugging in LLDB*. arXiv preprint arXiv:2105.12819, 2021.

[Seltman 2015] Howard J Seltman. *Experimental design and analysis*. Retrieved from, 2015.

[Sillito 2008] J. Sillito, G.C. Murphy and K. De Volder. *Asking and Answering Questions during a Programming Change Task*. IEEE Transactions on Software Engineering, vol. 34, no. 4, pages 434–451, jul 2008.

[Spinellis 2018] Diomidis Spinellis. *Modern Debugging: The Art of Finding a Needle in a Haystack*. Commun. ACM, vol. 61, no. 11, pages 124–134, October 2018.

[Tassey 2002] Gregory Tassey. *The economic impacts of inadequate infrastructure for software testing*. National Institute of Standards and Technology, 2002.

[Torres Lopez 2021] Carmen Torres Lopez, Louise Van Verre and Elisa Gonzalez Boix. *What's the problem? interrogating actors to identify the root cause of concurrency bugs*. In Proceedings of the 11th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control, pages 24–36, 2021.

[UDB 2023] UDB. *UDB Time Travel Debugger*. http://undo.io/, 2023.

[Ulidowski 2020] Irek Ulidowski, Ivan Lanese, Ulrik Pagh Schultz and Carla Ferreira. Reversible computation: extending horizons of computing: selected results of the cost action ic1405. Springer Nature, 2020.

[Ungar 1987] David Ungar and Randall B. Smith. *Self: The Power of Simplicity*. In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 227–242, December 1987.

[Vilk 2018] John Vilk, Emery D Berger, James Mickens and Mark Marron. *McFly: Time-Travel Debugging for the Web*. arXiv preprint arXiv:1810.11865, 2018.

[Wallaby.js 2023] Wallaby.js. *Wallaby.js Time Travel Debugger for Javascript*. https://wallabyjs.com/docs/intro/time-travel-debugger.html, 2023.

[Willembrinck 2021] Maximilian Willembrinck, Steven Costiou, Anne Etien and Stéphane Ducasse. *Time-Traveling Debugging Queries: Faster Program Exploration*. In International Conference on Software Quality, Reliability, and Security, Hainan Island, China, December 2021.

[Willembrinck 2022a] Maximilian Willembrinck, Steven Costiou, Anne Etien and Stéphane Ducasse. *Time-Traveling Queries for Faster Debugging and Program Comprehension*. Journées Nationales du Génie de la Programmation et du Logiciel 2022, June 2022. Poster, https://inria.hal.science/hal-03738585.

[Willembrinck 2022b] Maximilian Willembrinck, Steven Costiou, Adrien Vanègue and Anne Etien. *Towards Object-Centric Time-Traveling Debuggers*. In International Workshop on Smalltalk Technologies : IWST 22, Novi Sad, Serbia, August 2022. ACM Digital Libraries.

[Zeller 2009] Andreas Zeller. Why programs fail: a guide to systematic debugging. Elsevier, 2009.