HAL
open science

Time-Traveling Debugging Queries: Faster Program Exploration

Maximilian Willembrinck, Steven Costiou, Anne Etien, Stéphane Ducasse

To cite this version:

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

HAL Id: hal-03463047

https://inria.hal.science/hal-03463047v1

Submitted on 2 Dec 2021

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.

Time-Traveling Debugging Queries: Faster Program Exploration

Maximilian Willembrinck
Univ. Lille, Inria,
CNRS, Centrale Lille, UMR 9189 CRIStAL
F-59000 Lille, France
maximilian-ignacio.willembrinck-santander@inria.fr

Steven Costiou
Univ. Lille, Inria,
CNRS, Centrale Lille, UMR 9189 CRIStAL
F-59000 Lille, France
steven.costiou@inria.fr

Anne Etien
Univ. Lille, CNRS,
Inria, Centrale Lille, UMR 9189 CRIStAL
F-59000 Lille, France
anne.etien@inria.fr

Stéphane Ducasse
Univ. Lille, Inria,
CNRS, Centrale Lille, UMR 9189 CRIStAL
F-59000 Lille, France
stephane.ducasse@inria.fr

***Abstract*—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 execution through standard interactive debuggers is a tedious and costly task. In this paper, we propose Time-Traveling Queries (TTQs) to ease program exploration. TTQs is a mechanism that automatically explores program executions to collect execution data. This 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 required time and efforts when performing program comprehension tasks.**

***Index Terms*—debugging; program comprehension; time-traveling queries**

I. Introduction

Debugging is a difficult and costly activity [Pla02]. When a program fails, developers resort to standard debuggers in the first place. Debugging is an iterative process: developers first make an observation then formulate an hypothesis about the cause of the failure. 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 [Spi18], [Zel09], [O'D17], [BM14], [PFH13].

Formulating hypotheses requires to understand programs. Typically, developers ask themselves questions about the execution of their program [SMDV08], *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'D17], 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 to step too far and therefore to miss a critical information [BM14]. In addition, stepping is a generic operation that does not translate directly questions asked by developers to test an hypothesis to a stepping sequence (*i.e., how much 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 direct action that interrogates the execution of a program.

To address this problem, we propose *Time-Traveling Queries* (time-traveling queries). Time-traveling queries express requests of specific information about a program execution. A specialized debugger answers the request by executing the program instruction, one at a time, traversing all its states while retrieving the required execution data, collected to produce a *query result set*.

The produced result items are then used by developers to time-travel to the point in time in the execution where the result items data were retrieved.

In this paper, we present a definition of time-traveling queries along with their requirements in terms of run-time infrastructure and capabilities. We propose an implementation of time-traveling queries in Pharo [BDN$^+$09] based on a rudimentary time-traveling debugger.

To evaluate the impact of time-traveling queries on program comprehension activities, we selected from the literature [SMDV08] six questions developers ask to understand their programs. We defined a set of key time-traveling queries that specifically targets these questions, and made them available through the Pharo debugger. We then conducted a user study with 34 participants, asking them to solve a set of program comprehension tasks based on these questions. We asked participants to answer these questions with and without time-traveling queries. Our results show that, compared to

traditional debugging tools, time-traveling queries significantly improve developers' precision (39% more correct answers), time (28% faster), and efforts (38% less debugging actions) while performing program comprehension tasks.

The outline of the article is the following: Section II describes the difficulties of exploring a program execution using standard debugging methods. Section 3 presents time-traveling queries, their definition, and how they support program exploration. Section 4 describes our evaluation design for TTQs for program comprehension tasks, and results are listed in Section 5. Section 6 presents a discussion on the obtained results. Relevant implementation details are listed in Section 7. Then, we study related work in Section 8 and conclude.

II. Background and Motivation: Live Exploration of Program Executions

The *simplified scientific method* [Zel09], [Spi18] 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 to 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). 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 [Zel09]. They repeat these operations until they find the information they were looking for, or until the program ends.

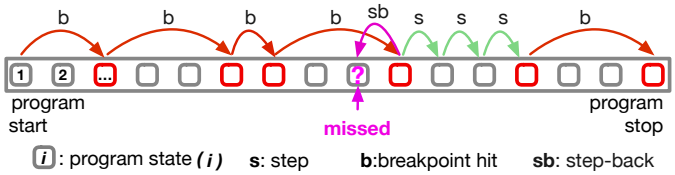

Fig. 1. Exploring an execution with breakpoints and manual backward and forward stepping.

These tools have three main problems:

- **Manual/Tedious.** Developers have to manually choose where to put breakpoints and how to step the execution when it breaks. Choosing efficiently where to put breakpoints requires to already understand parts of the program. Developers therefore have to perform preliminary investigations of the program [RBN12], *e.g.,* through source code reading.
- **Missing critical points.** It is common to miss a critical point in the execution [BM14], *e.g.,* the *missed* program state in Figure 1. Developers have to restart and explore again the execution to look for the information they missed.
- **Question translation difficulty.** Developers' debugging questions cannot be easily translated into sequences of breakpoints and stepping actions.

With *Time-Traveling Debuggers*, developers travel backward and forward in their program execution. For example, in Figure 1 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 information, they can immediately step back and observe that information. However, looking for an 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 2), developers program sequences of steps to automatically explore an execution and build problem-specific debugging tools [DPC+19]. 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.

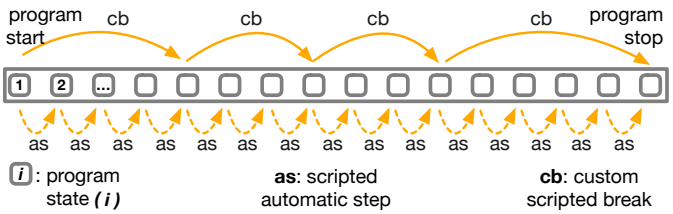

Fig. 2. Breakpoints scripted as automatic sequences of steps.

Problem summary and research question. To explore program executions, standard interactive debugging tools and techniques have the following limitations:

1) They require prior understanding of the program to efficiently explore an execution,
2) they are imprecise and miss critical information while exploring an execution (Figure 1),
3) there is a difficulty in translating developer debugging questions to debugging actions.

Therefore, in the scope of this paper, we investigate the following research question:

> *Can we express general program comprehension questions as queries over programs executions, and does that improve program exploration regarding developers' efforts, time spent and precision, compared to standard debugging tools?*

III. 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 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. 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 backwards steps.

We argue that TTQs will enable in-depth live program exploration. Developers will directly use pre-existing queries available on the shelves, or express their own 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, and we show how to declare a query to answer program comprehension questions extracted from the literature.

A. Key Time-Traveling Queries

In this section we present the list of key queries that we elaborated from the literature survey and that we propose to developers to explore their program execution.

We studied the key program comprehension questions that are important for developers [KBR14]. These questions have been reported in the literature [SMDV08], and we use the same numbering as in [SMDV08]:

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 exception thrown?

We analyzed these questions and defined 12 time-traveling queries organized in 4 categories, that aim to support answering those questions (Table I). Such queries take as input a program execution. Developers directly use these queries instead of writing them manually, to find answers to their program comprehension questions.

B. Time-Traveling Queries in a nutshell

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*.

TABLE I
KEY TIME-TRAVELING QUERIES OVER PROGRAM EXECUTION.

I	**Messages.**
I.1	Find all messages sent during the execution.
I.2	Find all messages, with a given selector, sent during the execution.
I.3	Find all received messages.
II	**Instances Creation.**
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	**Assignments - Object Centric.**
III.1	Find all assignments of instance variables for the receiver of the currently executed method.
III.2	Find all assignments of instance variables for a particular instance.
III.3	Find all assignments of a given instance variable for the receiver of the currently executed method.
IV	**Assignments - General.**
IV.1	Find all assignments of variables with a given name.
IV.2	Find all assignments of any variable.
IV.3	Find all assignments of instance variables for instances of a given class.

The data source is an iterable object that represents a collection of items, from where to select (*i.e.,* filter) and collect (*i.e.,* transform) data. To query data from an *execution*, the data source is either the list of states a program goes through during an execution or the selected program states from another *execution query*. For example, in the following Pharo[1]script we instantiate a query that will iterate over all the program states of a program execution. This just creates the query object that we will manipulate to define the query selection predicate.

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

The selection predicate is a function evaluated for each item of the data source (in similar fashion that OCL select clause). In the following script, we configure our query to select the program states corresponding to *message-sends*:

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

Listing 1. A query that finds all states corresponding to message-sends.

Finally, for each selected program state defined by the *messagesStates* query, we collect specific execution data in a dictionary according to a projection function. In the following script, for each message send, we gather in a dedicated object the class of the receiver, the selector of the message sent and the arguments as defined in the collect block.

```
messages := messagesStates
```

[1]For readers unfamiliar with Pharo, a comparison with Java-like syntax:
- Assignments use `:=`.
- The message-send notation uses spaces: `state isMessageSend` is equivalent to `state.isMessageSend()`.
- Arguments are specified by colons instead of parentheses: `Query from: states` is equivalent to `Query.from(states)`.
- Square brackets `[:x:y|]` delimit lexical closures, `x`,`y` are arguments.

```
  collect: [ :state |
    {(#receiverClass -> state receiverClass).
     (#selector -> state msgSelector).
     (#args -> state arguments)} asDictionary ].
```

Listing 2. A query that records every message-send data (receiver class, selector, and arguments) that take place during an execution.

In our examples, we decomposed the query definition into several queries and steps for illustrative purposes. The messages select: and collect: do not trigger the production of results. Instead, they return a new Query object. Complex queries can be composed from simpler ones, as in our examples, but they can also be defined in a single declaration.

TTQ execution. Queries are executed on-demand. When a query is executed (Figure 3), the time-travel debugger starts and executes the program, instruction by instruction, advancing from state to state.

For each state, the debugger tests the query selection predicate over that state. If the predicate is satisfied, the debugger collects the data as a *result item*.

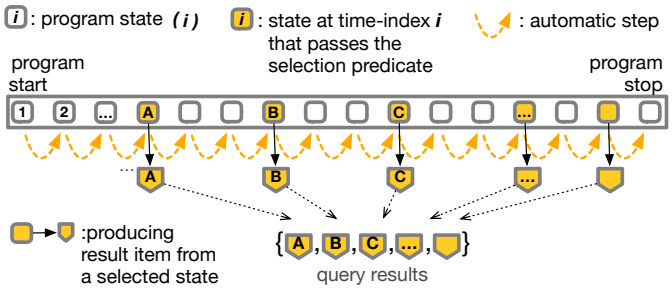

Fig. 3. Time-traveling query collecting time-indexed program execution data from its states.

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 4). After a time travel, they can continue navigating the execution with conventional tools and techniques (*e.g.*, stepping, breakpoints, etc.).

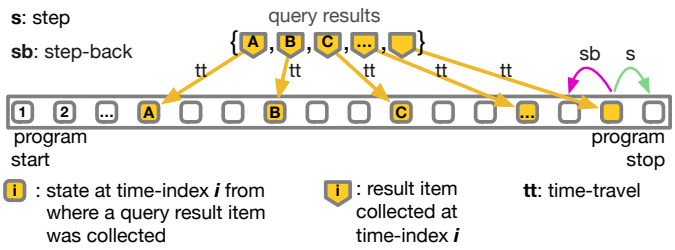

Fig. 4. Exploring an execution by time-traveling from the result items of a query. After a time-travel, developers can perform conventional stepping, or another time-travel.

C. Time-Traveling Queries requirements

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

- An iterable object that represents the program states of an execution,
- for every executed instruction (bytecode, opcode, abstract syntax tree...), that debugger records a unique time index,
- that debugger is able to time-travel by restoring a program execution state for any given time-index.

In the scope of this paper, we assume we have such debugger without considering technical details and limitations. To be consistent with our implementation (section VII) and our evaluation (section IV), we write our examples with the Pharo language. However, the concepts described in this section are fully independent from Pharo. For instance, we use without detailing it the API of our own time-travel debugger, *e.g.,* for accessing the program execution state.

IV. Evaluation

Our goal is to investigate our research question:

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

To investigate the RQ, we ran a quantitative evaluation [EY15], following a repeated measures design [Sel15] 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 Section III-A. 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 between using TTQs and standard debugging tools. We discuss other advanced techniques and how they might compare in Section VIII.

A. 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 RQ along three dimensions (time, precision, and debugging actions), we derived RQ into 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?

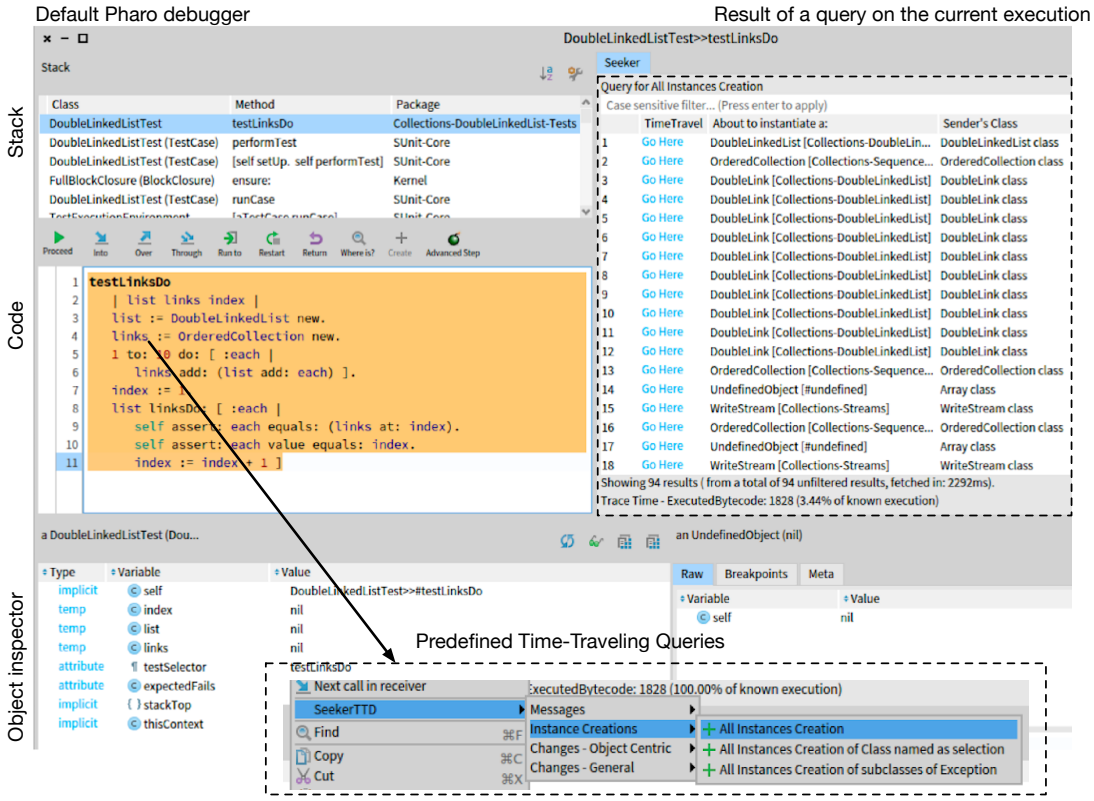

Fig. 5. Integration of the time-traveling queries and their result in the default debugger.

B. Experimental design

Our experiment is two-fold: first a tasks-solving part, following a repeated measures design, immediately followed by 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 actions they performed to resolve a task or their employed time. We suggested participants to 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 5 shows the integration of time-traveling queries and their results in the default Pharo debugger. Participants did not have to leave the debugger to perform 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 to answer 1 or 2 program comprehension questions.

The two sets of tasks are:

- The control set, composed of 5 tasks. We asked participants to provide an answer using exclusively standard Pharo debugging tools.
- The TTQ set, 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. It seemed to the pilot that performing the control set first helped understanding 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 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 II) based on the questions described in Section III-A.

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 many 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 to 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 had not 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 started 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.

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

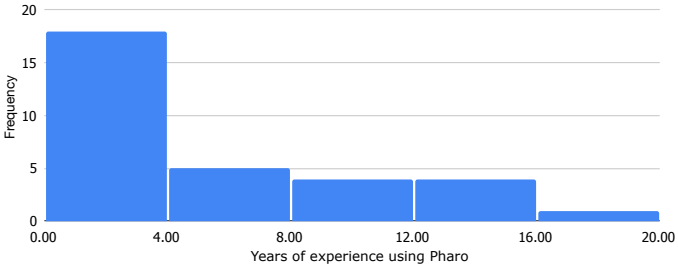

Fig. 6. Histogram of participants' years of experience in Pharo.

V. Results

In this section, we analyze the data collected from the experiment[2] and its statistical significance. We then analyze the data collected from the post-study survey.

A. 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 make any comparison unreliable. The following analysis is therefore based on results from 32 participants out of the 34 who performed the experiment.

Figures 7, 8, and 9 show the differences for each participant respectively for the score, time, and debugging action metrics. For example in Figure 7, 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 less debugging actions.

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

TABLE II
TASKS IN THE CONTROLLED EXPERIMENT

T	Method	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

T is the task id, *SQ* refers to the question types of [SMDV08] selected in Section III-A.
*: the task question is a variation of the original SQ.

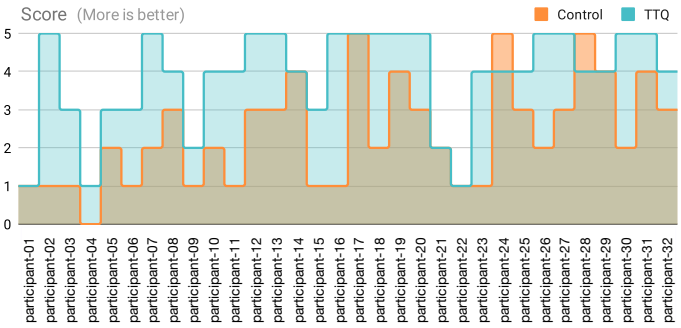

Fig. 7. Participants scores. In the horizontal axis, participants are sorted in ascending order by their years of experience in Pharo.

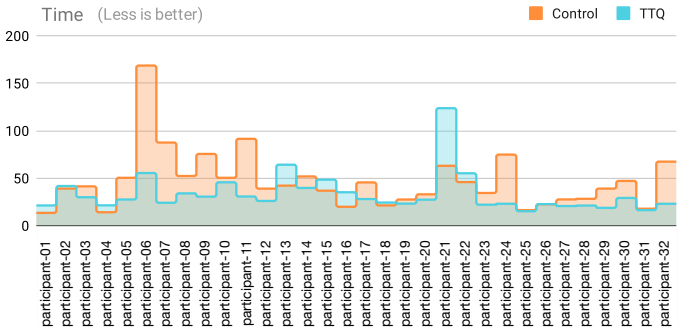

Fig. 8. Participants total time per sequence, in minutes. In the horizontal axis, participants are sorted in ascending order by their years of experience in Pharo.

Figure 10 shows the averages over all participants for each one of these metrics. In 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.

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

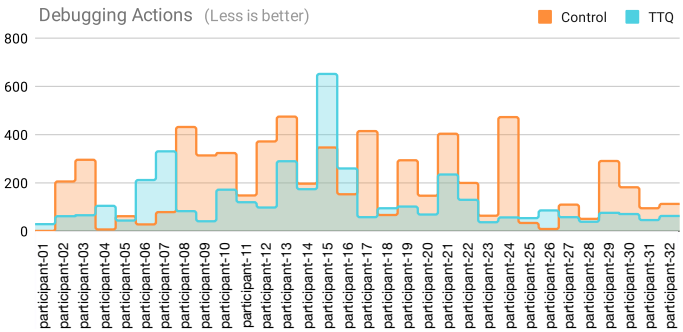

Fig. 9. Participants total debugging actions. In the horizontal axis, participants are sorted in ascending order by their years of experience in Pharo.

> $H_0 1$ 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 hypotheses, considering the differences $TTQ - control$ per participant, for each metric (Table III). All p-values are < 0.05, we therefore reject all null hypotheses.

We conclude that, to answer program comprehension questions, our time-traveling queries improves program exploration regarding developers' efforts, time spent, and precision com-

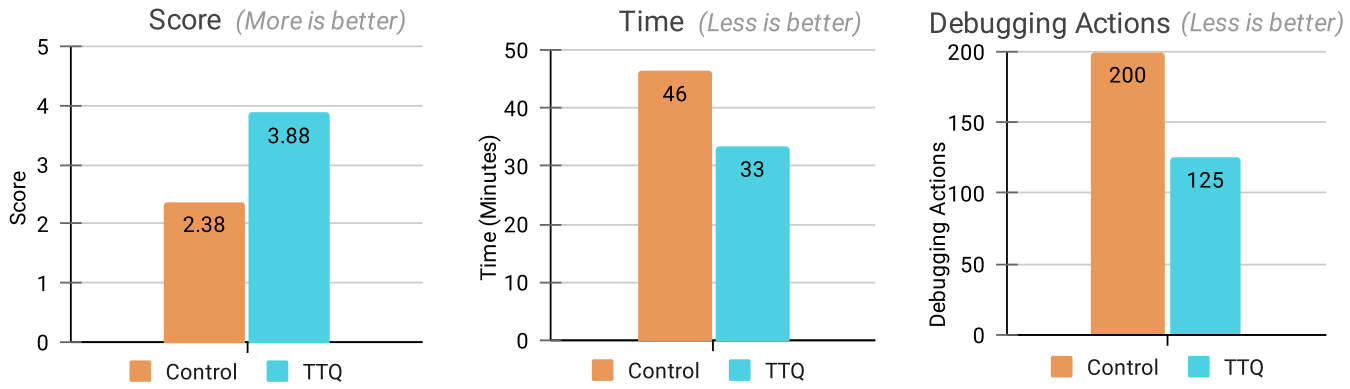

Fig. 10. Experiment results of each research dimension, averaged.

TABLE III
H_0 REJECTION TABLE WITH WILCOXON SIGNED-RANK TEST VALUES.

	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

pared to standard debugging tools.

B. Post-study survey

Table IV summarizes the results of the post-study survey. Most participants found that TTQs were useful and of intuitive usage. Most participants were more confident in the precision of their answers with TTQs than with standard debugging tools. Most participants perceived the tasks as less difficult with TTQs than with standard debugging tools. 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 easier to learn and use than standard debugging tools.

VI. DISCUSSION

A. Answering the research question

Can we express general program comprehension questions as queries over programs executions, and does that improve program exploration regarding developers' efforts, time spent and precision, compared to standard debugging tools? The experiment results positively answers it. Nonetheless, its is important to remark that the experiment tasks were based on a subset of common questions developers ask while debugging a program. They don't cover the complete set of problems developers face during their debugging sessions. Even though the experiment result validates the time-traveling queries approach, the specific measures improvements are significant only in the context of these questions. To conclude if time-traveling queries improves debugging in other areas, new focused evaluations are needed.

B. The proposed TTQs

While the program comprehension questions selected from the literature are simple to understand, answering them presents a difficult and time consuming challenge. In this article, we proposed a solution along with its evaluation. Our contribution is composed by the TTQs mechanism for program exploration, and a set of key queries supporting common debugging questions. We propose these queries not as a final all-purpose debugging solution, but a starting point from which to build more specialized queries and debugging tools, seeking to cover actual debugging needs of developers to improve debugging efficiency.

C. Scaling to real debugging problems

Our rudimentary time-traveling poses important limitations that prevent taking our current time-traveling queries implementation to production debugging environments [LGN08]. Nonetheless, time-traveling queries are agnostic of the debugger implementation. To scale this solution, we could therefore use production-level time-traveling back-ends instead. Additionally, and implementation aside, our experiment task design focused on measuring how time-traveling queries support program exploration in the context of test cases which presented no bugs. The results shows the success of the concept, and evaluating its usefulness on real bugs offers an interesting research subject.

D. Queries to support dynamic analysis

Program comprehension is gained by performing static analysis of a program code and dynamic analysis of its execution [Ric02], [RD99], [Cor07]. 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. Time-traveling queries can be used to produce trace information to feed dynamic analysis techniques and visualizations, incorporating their advantages within an interactive debugging workflow.

TABLE IV
POST STUDY SURVEY. PARTICIPANTS' EVALUATION OF THE TOOL (DEBUGGER WITH TTQS).

	TTQ Reception			Participants confidence in their answers			Perceived difficulty of Sequence	
Rating (More is Better)	***Usefulness***	***Intuitive Usage***	***Rating (More is Better)***	***Control***	***TTQ***	***Rating (Less is Better)***	***Control***	***TTQ***
Poor: 1	6%	3%	Not sure at all: 1	6%	6%	Easy: 1	0%	38%
Fair: 2	6%	0%	2	34%	3%	2	12%	28%
Satisfactory: 3	25%	18%	3	28%	19%	3	22%	25%
Very good: 4	44%	28%	4	19%	41%	4	41%	9%
Excellent: 5	19%	50%	They are for sure the correct ones: 5	12%	31%	Difficult: 5	25%	0%

E. Unmodified program debugging

Several debugging techniques require instrumenting program instructions. In those cases, the debugged execution is a modified version of the original, obscuring its analysis. Our debugger implementation performs debugging without the need of modifying the program execution. TTQs run without the need to account for execution instrumentation related changes. All debugging actions are performed by the monitoring process, and the code of the debugged program is unaffected. While this presents an advantage for analysis, the drawback is an introduction of additional processing overhead.

F. Experiment design: about habits and trust

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 6 ensures that we do not have such bias. Figure 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). It is worth to see that Pharo is not taught at the University around the place our experience happened, therefore having already 4 years Pharo developing experience exhibits a solid experience with the system including debugging.

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 V-A, we discarded one participant results for this reason. We acknowledge the need to minimize these scenarios for future experiences.

G. Threats to validity

Answers correctness. The list of expected correct values, 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 theses 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 V 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 less 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 experiment they learnt both during the first part of the experiment. In the {control, TTQ} order, they have two learning phases, one per experiment.

TABLE V
RESULTS ACCORDING TO THE EXPERIMENT ORDER.

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

Tasks equivalence. Every control task has an equivalent TTQ task in terms of difficulty. This makes possible to compute per-participant means over the control tasks and over the TTQ tasks, 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 suggest 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.

VII. Implementation

Figure 11 shows the model of time-traveling-queries and the supporting debugger model[3]. Queries are declared by specifying a data source, a selection predicate, and projection function, which are stored in the dataSource, selectionBlock, and projectionBlock field, respectively. The dataSource can be either a ProgramStates object, or another Query. A ProgramStates is a collection of ProgramState, and it is generated by the Debugger by stepping through the execution from the initialState. A ProgramState corresponds to a specific execution state, whose execution data are available through an API. These data are used to answer a query.

The execution of a Query produces a QueryResult, which is a collection of ResultItem. Each ResultItem stores a timeIndex *i.e.,* a timestamp to identify a unique program state of an execution, and a value *i.e.,* an object that is produced by the projection function of a query for each selected program state.

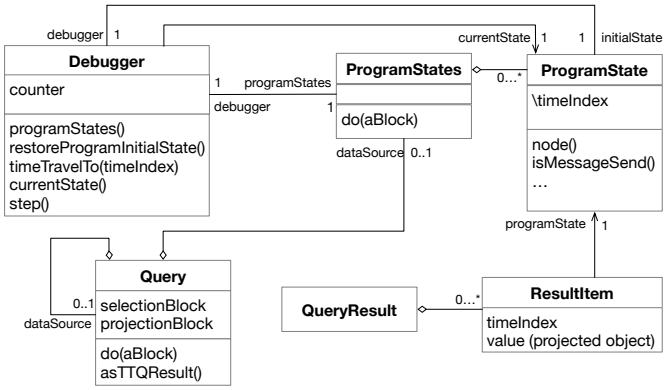

Fig. 11. Time-traveling queries and supporting debugger model.

The evaluation of a Query is performed by calling its asTTQResult() method. First, the query's dataSource items are iterated. If an item satisfies the selection predicate (selectionBlock), a ResultItem is instantiated with a timeIndex corresponding to the current program state. The value of this result is computed by applying the projection function (projectionBlock) to the item. Second, ResultItems are aggregated into a QueryResult, which is returned by the query.

ProgramState objects contain no fields, besides a reference to the debugger, *i.e.,* they store no execution data. Instead, these objects offer an API to access data of the current state of an execution. When iterating over ProgramStates (Listing 3), the debugger first reloads the initial state of the execution and sets its counter to 0. Then, the states of the program are advanced one at a time by stepping the debugged execution. With every step, the debugger increases its counter by one. This counter is used as the timeIndex to identify each state. Finally, the debugger applies the iteration block over each program state represented by the ProgramState object and obtained by calling the currentState() method on the debugger.

```
ProgramStates >> do: aBlock
   debugger restoreProgramInitialState.
   [debugger isFinished] whileFalse:
      [aBlock value: debugger currentState.
       debugger step]
```

Listing 3. Iterable ProgramStates object. Iteration routine.

Nothing about the execution is automatically stored (except the timeIndex). It is the projection function of queries, defined by developers, that determines which information will be recorded in the results.

VIII. Related work

Many works offer the possibility to perform assertions over the state of a program. Testing is one case, debugging another one. The key difference between such work and ours is that the later support the access and assertions of any program state within a program execution. We focus on the work offering trace or time traveling support.

Debugging techniques and tools offer the means to analyze and reason about a program, and depending on the nature of the debugged scenario *(i.e. the programming environment, language, etc.)*, certain techniques present advantages over the others.

A. Object-centric debugging

For object-oriented programming languages, there are several proposed approaches that account for object-specific aspects. While not directly in the debugging area, in [JE94] the authors argue that testing object-oriented software should not focus on units but on the message exchange between them in a scenario.

In debugging, object miners [CKT+20] proposes a non-intrusive object-centric approach for acquiring, capturing and replaying objects, used to track *elusive bugs*.

The practical object-oriented back-in-time debugging approach [LGN08], proposes to keep object changes history information together with the regular objects in the application memory, in contrast to conventional trace recording approaches.

B. Logic-based event debugging queries

OPIUM [Duc99b] is a tool that allows a user to debug Prolog program using a set of debugging queries on event traces. Prolog is used as a base language and as meta language to reason about events. The main usage scenario of OPIUM is the implementation of a high level debugger for Prolog that allows forward navigation to the next event that satisfies a certain condition. Coca [Duc99a] supports the debugging of C programs based on events. Opium and Coca are mainly used

[3]The complete implementation of our solution is publicly available at https://github.com/willembrinck/2021-TTQs

to show the values of variables. In addition, both Opium and Coca do not support object-oriented programming and object state analysis.

Auguston [Aug98], [Aug95] also uses a trace composed of event models and test programs. However it is based on procedural programming languages and does not take into account the specific behavioral aspects of object-oriented languages such object creation and the state of objects.

Query-based debugging [LHS97], [LHS99] uses logic programming to express complex queries over a large number of objects. Some queries are triggered at run-time while the program is running. However, they cannot express objects temporal relationship or refer to previous states of an object.

Although not focused in debugging, in TESTLOG [DGW06], the approach reifies the execution traces and uses logic programming to express tests on them. Thereby it eliminates the need to programmatically bring the system in a particular state, and handles the test-writer a high-level abstraction mechanism to query the trace.

Caffeine [GDJ02] is a Java-based tool that uses the Java debugging API to capture execution events and uses a Prolog variant to express and execute queries on a dynamic trace. The main difference with TESTLOG is that Caffeine has a linear representation of a trace, and hence it is not possible to reason about nested events. Caffeine is also missing state reification.

The TTQs could be expressed on top of Prolog-based trace reification. The fundamental aspect of our work is to expose developers with key queries capturing important questions supporting debugging sessions. In addition, time-traveling queries can be seamlessly composed and mixed with more traditional debugging actions (step in, step over...).

C. Time-traveling debugging

Through the years, time-traveling debuggers have presented attractive means for improving debugging. An important part of the research in the field focuses on solving different implementation challenges, contributing with highly performant solutions [BM14], [BMM+16], [MVB+16], [LGN08].

On the other hand, there are other projects that do research on *how to exploit them [KM04], [KM08], [PFH13], for program comprehension and debugging*. This is where we position our work, and compare similarities against related research projects listed in the following paragraphs.

D. Time-traveling queries

Our work shares similarities with Expositor [PFH13]. Like our solution, Expositor combines scripting and time-travel debugging to allow programmers to automate complex debugging tasks. Expositor uses GDB [Und] as an execution logging backend, which grants the time-traveling capabilities. In contrast, we implemented a rudimentary time-traveling debugger, *Seeker*, offering similar capabilities, although supporting limited scenarios. One of their main contributions is their abstraction of the execution trace, which is a time-indexed sequence of program state snapshots or projections. Programmers can manipulate traces as if they were simple lists with operations such as map and filter. Our query model follows the same idea: execution traces can be created and operated with time-traveling queries, and like in Expositor, are lazily evaluated to generate the results. Our queries are declared using list comprehension expressions which are common in modern programming languages. Therefore, they don't require specialized APIs or DSL knowledge to write them. Lastly, our contribution contains a list of ready-to-use queries that are mapped to common developer questions asked during debugging.

E. Program-comprehension in debugging

During debugging sessions, developers require knowledge of a program to formulate good hypotheses, and then to write effective queries to answer their questions. *Whyline* [KM04], [KM08] derived work offers to developers contextual queries to simplify the hypothesis formulation and querying activity. In contrast, our queries proposal features a different approach by offering commonly needed general purpose execution queries. Even though it doesn't directly support hypothesis formulation, it relieves the burden of writing queries from the developer which translates in increased efficiency as shown in our evaluation. Another point of comparison is the extensibility of the solution. In *Whyline*, taking the solution to a different context is difficult. In contrast, with time-traveling queries, developers have the means to create new specialized queries, optionally reusing existing ones, to answer their own debugging questions.

F. Querying executions for dynamic analysis

Queries are used in dynamic analysis to obtain program execution information. In the context of *Reverse Engineering and Design Recovery*, static and dynamic program queries are performed over programs to create high-level views of its components, and their connections [Ric02], [RD99].

Nowadays, it is not uncommon that debuggers provide visualization enhancements. They use dynamic information to display traces, providing visualizations of a program behavior. As discussed in Section VI-D we relate our work to visualization because TTQs provide a simple mechanism to generate program traces. Visualizing debuggers can work directly via instrumentation on the program being executed, or are based on post-mortem traces [CM93], [LN95]. DePauw et al. [DPLVW98] and Walker et al. [WMFB+98] use program events traces to visualize program execution patterns and event-based object relationships such as method invocations and object creation.

IX. Conclusion

There are different tools and methodologies through which developers gain program understanding. Program exploration using interactive debuggers remains a common, yet difficult and tedious approach. To improve program exploration, we proposed time-traveling queries, which help developers answer program comprehension questions. We conducted a controlled experiment to evaluate how queries help to answer common

program comprehension questions. Results show that with time-traveling queries, developers perform program comprehension tasks more accurately, faster, and with less effort than with standard debugging tools. The positive reception of the tool suggests that our solution is easier to learn and to use for program comprehension than standard debugging tools. This represents a promising research step, from where we acknowledge the importance of exploring time-traveling queries to ease debugging activities.

REFERENCES

[Aug95] M. Auguston. Program behavior model based on event grammar and its application for debugging automation. In *2nd International Workshop on Automated and Algorithmic Debugging, Saint-Malo, France*, May 1995.

[Aug98] M. Auguston. Building program behavior models. In *European Conference on Artificial Intelligence ECAI-98, Workshop on Spatial and Temporal Reasoning, Brighton, England*, Aug. 1998.

[BDN+09] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BM14] E. T. Barr and M. 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, pp. 67–82. ACM, oct 2014.

[BMM+16] E. Barr, M. Marron, E. Maurer, D. Moseley, and G. Seth. Time-travel debugging for javascript/node.js. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pp. 1003–1007, nov 2016.

[CKT+20] S. Costiou, M. Kerboeuf, C. Toullec, A. Plantec, and S. Ducasse. Object miners: Acquire, capture and replay objects to track elusive bugs. *Journal of Object Technology*, 19(1):1:1–32, July 2020.

[CM93] M. P. Consens and A. O. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding of International Conference on Management Data*, pp. 511–516, 1993.

[Cor07] B. Cornelissen. Dynamic analysis techniques for the reconstruction of architectural views. In *Proceeding of Working Conference on Reverse Engineering (WCRE)*. IEEE, 2007.

[DGW06] S. Ducasse, T. Gîrba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pp. 35–44. IEEE Computer Society Press, 2006.

[DPC+19] T. Dupriez, G. Polito, S. Costiou, V. Aranega, and S. Ducasse. Sindarin: A versatile scripting api for the pharo debugger. In *ACM SIGPLAN International Symposium on Dynamic Languages (DSL'19)*, pp. 67–79. ACM, 2019.

[DPLVW98] W. De Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings of Conference on Object-Oriented Technologies and Systems (COOTS'98)*, pp. 219–234. USENIX, 1998.

[Duc99a] M. Ducassé. Coca: An automated debugger for C. In *International Conference on Software Engineering*, pp. 154–168, 1999.

[Duc99b] M. Ducassé. Opium: An extendable trace analyser for prolog. *The Journal of Logic programming*, 1999.

[EY15] N. Elmqvist and J. S. Yi. Patterns for visualization evaluation. *Information Visualization*, 14(3):250–269, 2015.

[GDJ02] Y.-G. Guéhéneuc, R. Douence, and N. Jussien. No java without caffeine: A tool for dynamic analysis of java programs. In *ASE*, pp. 117. IEEE Computer Society, 2002.

[JE94] P. C. Jorgenson and C. Erickson. Object-oriented integration testing. *CACM*, 37(9):30–38, Sept. 1994.

[KBR14] J. Kubelka, A. Bergel, and R. 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, pp. 1–11, New York, NY, USA, 2014. ACM.

[KM04] A. J. Ko and B. 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*, pp. 151–158. ACM Press, 2004.

[KM08] A. J. Ko and B. 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.

[LGN08] A. Lienhard, T. Gîrba, and O. 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*, pp. 592–615. Springer, 2008. ECOOP distinguished paper award.

[LHS97] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, pp. 304–317, New York, NY, USA, 1997. ACM.

[LHS99] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. In R. Guerraoui, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *LNCS*, pp. 135–160, Lisbon, Portugal, June 1999. Springer-Verlag.

[LN95] D. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pp. 342–357, New York NY, 1995. ACM Press.

[MVB+16] A. Miraglia, D. Vogt, H. Bos, A. Tanenbaum, and C. Giuffrida. Peeking into the past: Efficient checkpoint-assisted time-traveling debugging. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 455–466. IEEE, 2016.

[O'D17] D. H. O'Dell. The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue*, 15(1):71–90, 2017.

[PFH13] 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)*, pp. 352–361, may 2013.

[Pla02] S. Planning. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.

[RBN12] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012.

[RD99] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang and L. White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pp. 13–22, Los Alamitos CA, Sept. 1999. IEEE Computer Society Press.

[Ric02] T. Richner. *Recovering Behavioral Design Views: a Query-Based Approach*. PhD thesis, University of Bern, May 2002.

[Sel15] H. J. Seltman. Experimental design and analysis. *Retrieved from*, 2015.

[SMDV08] J. Sillito, G. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, jul 2008.

[Spi18] D. Spinellis. Modern debugging: The art of finding a needle in a haystack. *Commun. ACM*, 61(11):124–134, Oct. 2018.

[Und] UndoDB. Undodb time travel debugger. http://undo.io/.

[WMFB+98] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, pp. 271–283. ACM, Oct. 1998.

[Zel09] A. Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.