JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Time-Traveling Queries: Extensible Tools for Faster Program Comprehension

Valentin Bourcier*, Maximilian Willembrinck*, Adrien Vanègue*, Stéphane Ducasse*, Anne Etien†, and Steven Costiou*
*Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
†Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

ABSTRACT Debugging programs require program comprehension. To acquire this comprehension, developers explore the program execution, a task often performed using interactive debuggers. However, exploring a program execution through standard interactive debuggers is tedious and costly. In addition, standard debuggers are generic tools that are inflexible and difficult to use in domain-specific contexts. In this paper we propose Time-Traveling Queries (TTQs) to ease and customize program exploration. TTQs is an extensible 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. Queries can be created or extended for problem-specific or domain-specific debugging scenarios. TTQ have been successfully used on a real-world example of a bug, which shows that in practice the TTQ system is usable. To evaluate more in depth the impact of TTQs on program comprehension activities, we conducted a user study with 34 participants on program comprehension tasks. Results show that compared to traditional debugging tools, TTQs improve developers' precision (39% more correct answers) while reducing required time (27% faster to finish tasks) and effort (45% less debugging actions) when performing program comprehension tasks.

KEYWORDS Time-Traveling Debugging, Debugging, Debugging Queries, Domain-Specific Debuggers.

1. Introduction

Debugging is a difficult and costly activity (Planning 2002). When a program fails, developers resort to standard debuggers in the first place. Debugging is an iterative process: developers first make an observation and then formulate a 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 (Spinellis 2018; Zeller 2009; O'Dell 2017; E. T. Barr & Marron 2014; Phang et al. 2013).

Formulating hypotheses requires to understand programs. Typically, developers ask themselves questions about the execution of their program (Sillito et al. 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 faced with 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 for taking a step too far and therefore to miss critical information (E. T. Barr & Marron 2014). Furthermore, stepping is a generic operation that does not translate directly questions asked by developers to test a hypothesis to a stepping sequence (*i.e., how many steps should we perform to find that information?*). Developers therefore face the challenge of translating their questions into sequences of debugging actions. This translation process is far from direct and gives rise to an abstraction gap. This gap issue is alleviated by domain-specific debugging tools, which offer debugging actions closely aligned with the application domain (Chiş et al.

JOT reference format:
Valentin Bourcier, Maximilian Willembrinck, Adrien Vanègue, Stéphane Ducasse, Anne Etien, and Steven Costiou. *Time-Traveling Queries: Extensible Tools for Faster Program Comprehension.* Journal of Object Technology. Vol. 23, No. 1, 2024. http://dx.doi.org/10.5381/jot.2024.23.1.a7

2014). However, this specialization comes with a trade-off: it sacrifices generalizability, demanding tailored debugging tools for each unique domain. Developing specialized debugging tools or extending existing ones is a difficult task. This hinders the availability of these tools to address problem-specific scenarios.

To help developers explore their program executions, 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 are queries that developers express to extract specific information from a program execution to answer program comprehension questions. A specialized debugger answers these requests by executing the program instructions, one at a time. Thus, the debugger traverses all states of the program to collect the required data from the execution. Developers then use these results to time-travel to the point in time in the execution where the results data were retrieved. When facing singular or domain-specific problems, developers can write their own queries tailored for their specific problem/domain.

This paper is an extension of (Willembrinck et al. 2021). We first describe the difficulties of exploring program executions with standard debugging tools (Section 2). Then, we present contributions from our previous work on time-traveling queries (Willembrinck et al. 2021) (contributions 1 and 5 with an extended experiment results analysis) extended by 3 new contributions (contributions 2, 3 and 4):

1. We present a definition of time-traveling queries and how they support program exploration, along with their requirements in terms of run-time infrastructure and capabilities (Section 3).

2. We selected from the literature (Sillito et al. 2008) six questions developers ask to understand their programs, from which we defined a library of ready-to-use queries (Section 4). We illustrate how to write and use real queries, we describe their implementation, and how we integrate and execute these queries in the Pharo debugger. This shows that our query system is extendable and can be used to customize debugging tools.

3. We propose an implementation of time-traveling queries in Pharo (Black et al. 2009) based on a rudimentary time-traveling debugger (Section 5). Specifically, we highlight the implementation points that the back end should provide for the implementation of our query system.

4. We report a real-world usage of TTQs by developers working on a meta-compiler (Section 6). This report shows that, even if slow, TTQs are usable in practice, and are applicable to real-world programs.

5. We conducted a user study with 34 participants, asking them to solve a set of program comprehension tasks based on these questions (Section 7). We asked participants to answer these questions with and without time-traveling queries. Compared to (Willembrinck et al. 2021) we provide an extended analysis which confirms our original results. We conclude that compared to traditional debugging tools, time-traveling queries significantly improve developers' precision (39% more correct answers), time (27% faster), and efforts (45% less debugging actions) while performing program comprehension tasks. Our analysis of the observed data supports this conclusion.

Finally, we study related work in Section 11 and conclude.

2. Background and motivation: 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.

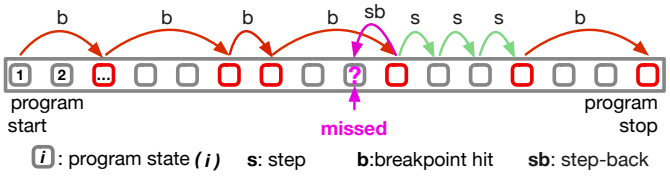

Figure 1 Exploring an execution with breakpoints and manual backward and forward steps.

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 (Zeller 2009). They repeat these operations until they find the information they were looking for, or until the program ends.

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 already understanding parts of the program. Developers, therefore, have to perform preliminary investigations of the program (Ressia et al. 2012), *e.g.,* through source code reading.
- **Missing critical points.** It is common to miss a critical point in the execution (E. T. Barr & Marron 2014), *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.

– **Abstraction gap.** To understand a program, developers have to reason with abstractions among the specific domain of the program and the available debugging actions. Standard debuggers are generic tools, inadequate for specific contexts such as debugging domain-specific problems. Debugging questions cannot be easily translated into sequences of breakpoints and stepping actions. Because debuggers' implementations are inflexible, they are difficult or impossible to extend to cope with new contexts.

With *Time-Traveling Debuggers*, developers travel backward and forward in their program execution. 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 a piece of important 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 remains a manual operation. Without additional means to explore recorded executions, it is as tedious as standard breakpoints and stepping.

With scriptable debuggers, developers build problem-specific debugging tools to explore their executions, and scope debugging to a set of problems. Developers build these tools through highly customizable scripting APIs (Phang et al. 2013; Dupriez et al. 2019), that can be combined with time-traveling debuggers (Phang et al. 2013). Every state of an 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. Developers must also understand and reason within several abstraction domains including the program itself, the scripting API, etc. Sometimes, it also requires deep knowledge of the execution model (Dupriez et al. 2019). On top of that, they must translate their debugging questions into debugging scripts. If developers try to reduce this abstraction gap by making their debugging tools closer to their program domain, they will encounter an additional difficulty since debuggers are not usually designed with a focus on their extensibility and flexibility to create new tools.

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

1. They require a 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 difficulty in translating developer debugging questions to debugging actions,
4. they are generic and are difficult to use in (or to extend for) domain-specific contexts.

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

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

3. 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. This information is presented to developers, who can 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.

We argue that TTQs enable in-depth live program exploration. Developers will directly use pre-existing queries available on the shelves, 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. We then state the properties of a time-traveling back end required to execute TTQs. To be consistent with our implementation (Section 5) and our evaluation (Section 7), we write our examples with the Pharo language[1]. We use without detailing it the API of our time-traveling debugger, *e.g.,* for accessing the program execution state. However, the concepts described in this section are fully independent of Pharo.

3.1. Time-Traveling Queries definition and execution

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). 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 function*, and a *projection function*.

[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.

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. The data source is either the sequence of program states of a program execution or the selected program states resulting from an executed query. In the following Pharo script, we instantiate a query that will iterate over all the program states of a program execution. In this code, the sequence of program states is referred to as `programStates` and should be generated by the underlying debugger. The query we created is just an object and is not executed yet. We will manipulate this query to define a selection function and a projection function before execution.

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

The selection function is a method that implements a condition to decide if a program state is of interest for a given query. The function is evaluated for each item of the data source (in the same way as the OCL selection clause). When evaluated, this condition returns `true` if the program state should be selected and `false` otherwise. In the following script, we configure our query to select program states corresponding to *message sends*:

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

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

The projection function is a method that collects execution data for each selected program state, in a form specified by the developer. The function is evaluated for each item of the data source selected by the selection function. In the following script, for each selected *message send*, we gather in a dictionary the class of the receiver, the selector of the message sent, and its arguments:

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

Listing 2 A projection function that records every *message send* data (receiver class, selector, and arguments).

The query is then executed on-demand by sending the query object the `execute` message.

TTQ execution. When a query is executed (Figure 2), the debugger starts and executes the program, instruction by instruction, advancing from program state to program state. For each state, the debugger tests the query selection function 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.

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 3). After 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.

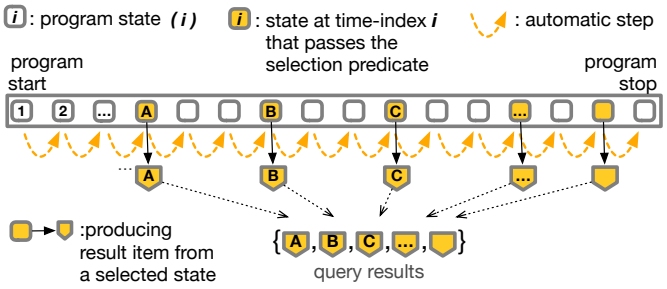

Figure 2 Time-traveling query collecting time-indexed data from the program states of a program execution.

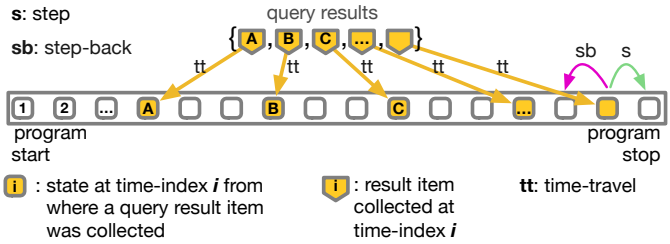

Figure 3 Exploring an execution by time-traveling from the result items of a query. After time travel, developers can perform conventional stepping or another time travel.

3.2. Time-Traveling Queries requirements

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

1. An iterable object that represents the sequence of program states of an execution.
2. A unique time index for every executed instruction (bytecode, opcode, abstract syntax tree...), that the debugger records.
3. A sequence of program states that 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 deterministic re-execution (or replay) order of those instructions.
4. The back end should be able to restore a program execution to any past program state.

In the scope of this paper, we assume we have such a debugger back end, without considering technical details and limitations. We use a virtual machine step-based implementation that executes programs step by step, then entirely restarts and replays step by step the execution to time travel. We provide implementation notes in Section 5.

4. Off-the-Shelf Time-Traveling Queries

In this section, we present a list of key queries that we elaborated on 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 3. Furthermore, we show how to execute queries and how to script custom queries directly from the debugger. By writing queries or by scripting queries in the debugger, developers can customize their debugger to target domain-specific problems.

4.1. Key Time-Traveling Queries

We studied the key program comprehension questions that are important for developers (Sillito et al. 2008). We focus on questions developers ask in object-oriented programming (Kubelka et al. 2014). We selected 6 questions, for which we use the same numbering as in (Sillito et al. 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 exception thrown?

We analyzed these questions and defined 12 time-traveling queries organized in 4 categories, that aim to support answering those questions. We provide these queries to developers so that they do not need to write them manually to answer 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

4.2. Implementation of Key Time-Traveling Queries

The API to write selection and projection functions (described in Section 3) and for executing queries is implemented in the `Query` class. Developers subclass that `Query` class to implement their own queries, and override a `select:` and `project:` methods to implement the selection and the projection functions. In the following, we describe how we implemented two representative queries of our library from Section 4.1: query *III.3* (Section 4.2.1) and query *I.2* (Section 4.2.2).

4.2.1. Finding all assignments to the instance variables of a class.

This query must find all assignments to instance variables of any instance of a target class. We first subclass the `Query` class with a new class `AssignmentsOfInstVarsOfClassName`. Then, we implement a selection function to filter out all program states corresponding to assignments. Finally, we implement a projection function to build and return a dictionary with information extracted from the filtered assignments.

The selection function. To implement the selection function, we override the `select:` method of the `Query` class (Listing 3). We first use the API of the program state to determine if it corresponds to an instance variable assignment (line 2). If that is the case, we compare the class name of the receiver in which the assignment occurs with a `targetName` variable (line 3). That variable is defined as an instance variable of the query under execution. We use it to store the target class name for which we want to find assignments.

```
1 AssignmentsOfInstVarsOfClassName>>select: pState
2    ^ pState isAssignmentOfInstVar and: [
3        pState receiverClass name == targetName ]
```

Listing 3 Selection function implementation of query *III.3* `AssignmentsOfInstVarsOfClassName`.

Projection functions. To implement the projection function, we override the `project:` method of the `Query` class (Listing 4). We implement a simple projection function that maps specific fields of program states into a dictionary. We use the program state API to collect various execution data about the assignment, such as the method, class, and package where it occurs, the variable name, its current value and new value being assigned, and a technical field named `bytecodeIndex`. This bytecode index is used by the time-traveling back end when replaying execution, to navigate to the execution point where the assignment took place. We collect the execution data in a dictionary, where we associate each piece of information with a human-readable symbol.

Returning dictionaries makes it easy for external tools or users to build and extend tools on top of query results. For instance, these results are easily displayable in tables, and the bytecode index is a single value that can be transmitted to the time-traveling debugger to replay the execution.

```
1 AssignmentsOfInstVarsOfClassName>>project: pState
2    ^ { (#selector -> pState methodSelector).
3        (#class -> pState receiverClass).
4        (#package -> pState receiverPackage).
5        (#varName -> pState variableName).
```

```
        (#currentValue -> pState readVariableValue).
        (#newValue -> pState assignmentValue).
        (#bytecodeIndex -> pState bytecodeIndex).
    } asDictionary
```

Listing 4 Projection function implementation of query *III.3* `AssignmentsOfInstVarsOfClassName`.

4.2.2. *Finding all sendings of a specific message.*

This query must find all occurrences of a specific *message send*, represented by a selector given by the developer. Following the same method as in Section 4.2.1, we implement a query class `AllMessagesSentWithSelector` and its selection and projection functions.

In the selection function (Listing 5), we first check if the current program state corresponds to a message sending (line 2). If that is true, we return the result of the comparison between the message selector with the one given by the developer (stored in the instance variable `selector` of the query).

```
1 AllMessagesSentWithSelector>>select: pState
2   pState isMessageSend ifFalse: [ ^ false ].
3   ^ pState messageSelector == selector
```

Listing 5 Selection function implementation of query *I.2* `AllMessagesSentWithSelector`.

In the projection function (Listing 6), we collect information about the selected *message sends*. Similarly, as in Listing 4, we collect and return this information in the form of a dictionary.

```
1 AllMessagesSentWithSelector>>project: pState
2   ^ {
3       (#selector -> pState messageSelector).
4       (#arguments -> pState messageArguments).
5       (#receiver -> pState messageReceiver).
6       (#bytecodeIndex -> pState bytecodeIndex).
7     } asDictionary
```

Listing 6 Projection function implementation of query *I.2* `AllMessageSends`.

4.3. Executing Queries

To experiment with Time-Traveling Queries, we implemented a debugger prototype in Pharo named Seeker (Figure 4, and Figure 5). Seeker is composed of a time-traveling back end satisfying the properties presented in subsection 3.2, and dedicated query tools integrated into the Pharo debugger (menus, visualizations, time-travel control, etc.). There are two ways to execute queries: by selecting a query from the debugger menu and by writing them directly in the debugger.

Executing queries from the debugger menu. The debugger queries menu is populated from different sources. Off-the-shelf queries (Section 4) are statically added to the menu. Users can also define custom queries using a specific user interface `UserTTQ`. We call these queries *user-defined queries*. User-defined queries are dynamically added to the debugger menu.

To use queries from the debugger menu, the user starts debugging a program with Seeker. The Seeker debugger opens with the standard debugger view on the left and the time-traveling queries view on the right (Figure 4).

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 4 and all user-defined queries.

Some queries require parameters from the execution context. Parameters are obtained from the code pane or the object inspector, *e.g.*, variables, symbols, strings... For example, in Figure 4 we selected the query *All messages sent with a given selector* (query I.2 from Section 4.1) from the contextual menu, and this query requires a selector as a parameter. We selected the `add:` selector in the code pane to provide it as input to the query.

Query results are displayed on the right pane of the debugger (Figure 4). The results are displayed according to the projection function of the executed query, which we defined in Listing 6. The result table lists this projected information, *i.e.*, the bytecode index, the message receiver, the selector, and the message arguments. The bytecode index is displayed in the column labeled *step* and serves as a control for 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 (code, stack, etc.) accordingly. From there, developers can perform any standard debugging actions (such as stepping forward and backward), time-travel again, or execute another query.

Writing Time-Traveling Queries in the scripting presenter. The scripting presenter is available in the right pane of the debugger (Figure 5). There, developers can write and execute queries on the fly directly in the debugger. Developers can therefore ask new questions while exploring an execution.

To write queries in the scripting pane, developers have access to a variable named `programStates`, which represents the collection of all possible program states of the debugged program execution. Scripted queries written in the debugger take the custom selection and projection functions defined in closures.

For example, in Figure 5, we manually stepped the execution until we reached the body of a loop. From there, we defined a query in the scripting pane. This query selects all assignments and projects the name of the variable in which a value is assigned. When we execute this query, the debugger restarts the execution, runs the query over the entire program execution, and displays the results in the query pane (Figure 4). The debugger then sets back the execution to the point where we ran the query, *i.e.*, in the loop line 6 in Figure 5.

5. Implementation

Figure 6 shows a simplified overview of time-traveling-queries and the supporting debugger back end[2]. In the context of this paper, we describe the debugger back end then the query model implementation. We then list limitations of our implementation.

5.1. The debugger back end implementation

In the scope of this paper, we used a naive time-traveling debugger implementation satisfying the required properties enumerated in Section 3. In particular, this implementation did not

[2] The complete implementation of our solution is publicly available at https://github.com/maxwills/SeekerDebugger

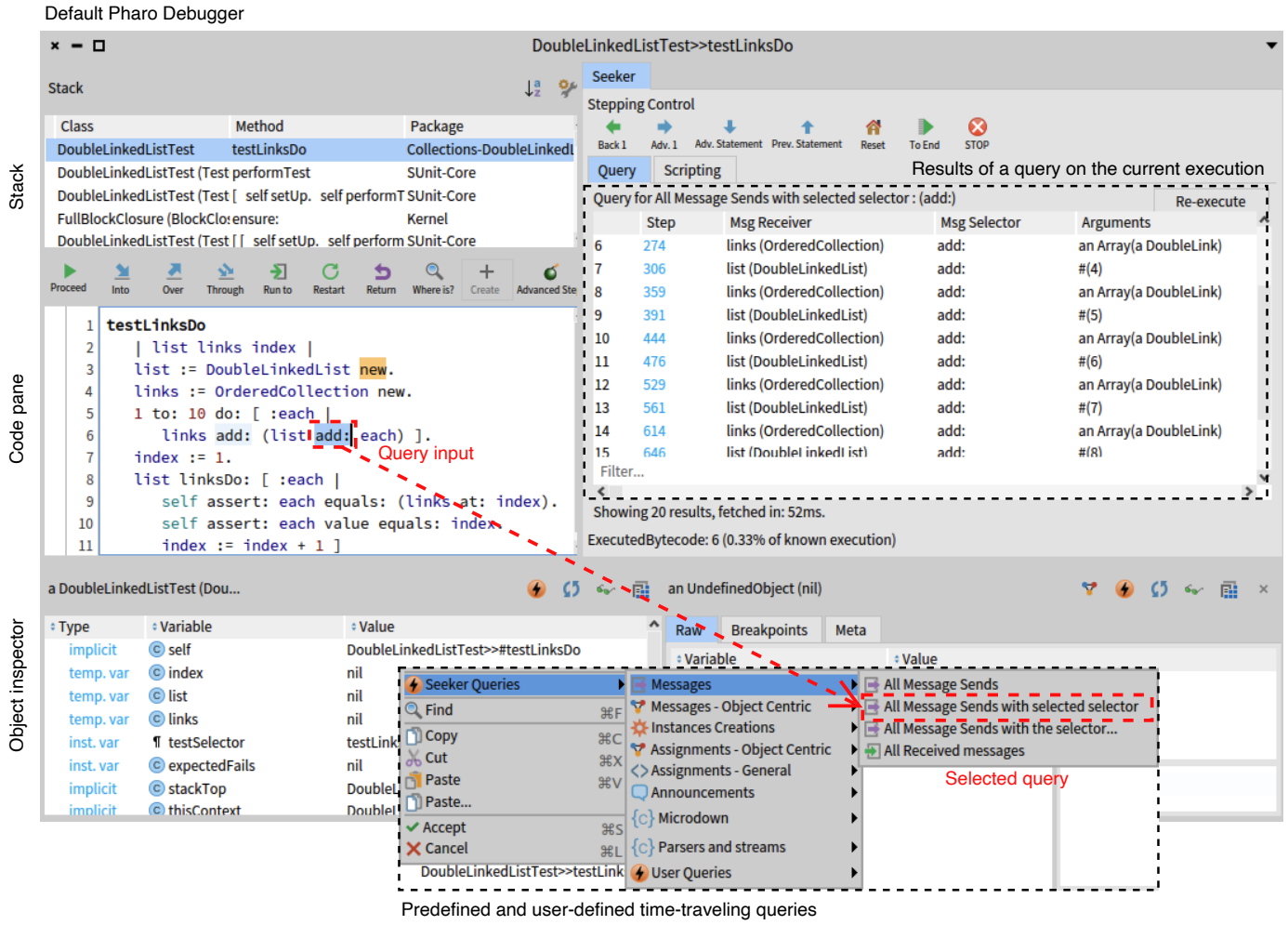

Figure 4 Time-Traveling queries integration in the debugger.

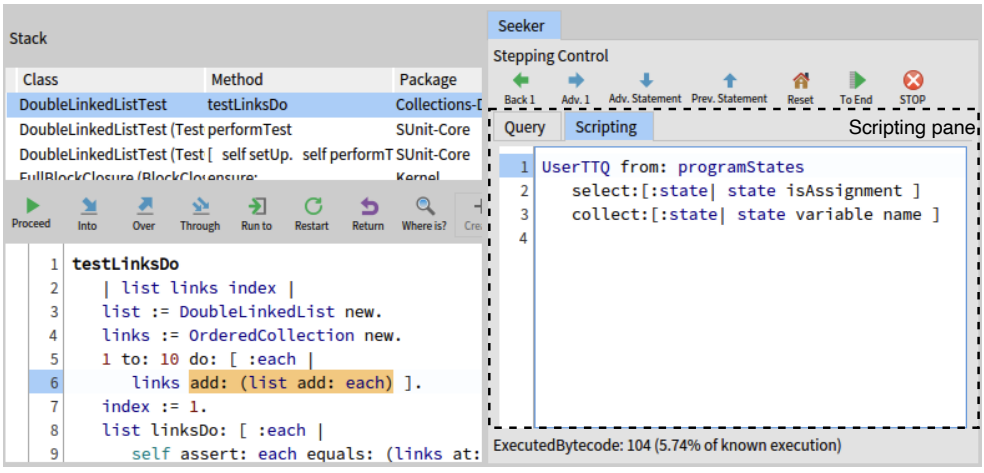

Figure 5 Scripting presenter in Seeker to write time-traveling queries on the fly.

account for concurrency, primitive execution (*i.e.*, i/o calls), or non-determinism. This did not impact our experiment, since we carefully chose tasks not involving these aspects. An updated implementation takes care of these aspects (Willembrinck Santander 2023). In the following, we describe the implementation used in the context of this paper.

We use a bytecode interpreter that executes each program bytecode step by step. That debugger exposes the `Debugger` interface from Figure 6.

The counter. A unique time index corresponding to the total number of bytecode steps performed since the beginning of the program execution. In our implementation, it is an integer. It starts at zero.

The `programStates()` interface. An interface to the debugger internal state that generates `ProgramState` objects. The current counter is generated for each generated instance of program state. Clients of the debugger interact with this interface.

The `currentState()` accessor. The accessor to the last generated program state, *i.e.*, the program state representing the current execution state after the last bytecode execution.

The `restoreProgramInitialState()` method. This method restarts the entire execution to its original state, flushing out all existing state. It sets the current state as the original execution state and sets its counter to 0. If the execution starts for the first time, this method just configures the current state and the counter.

The `timeTravelTo(timeIndex)` method. This method moves the execution forward or backward to attain a time index corresponding to a program state. Clients of the debugger use this method to time-travel to a specific program state.

The `step()` method. Interprets a new bytecode. Each interpreter step increments the counter by one and generates a new program state.

The program state generation procedure. Clients obtain program states from the debugger back end by using the `do(aBlock)` method of the `programStates()` interface. The executed procedure is shown in Listing 7. The debugger first reloads the initial state of the execution. Then the states of the program are advanced one at a time by stepping the debugged execution. Each step advancement is composed of two steps:

1. The debugger applies the iteration block over each program state represented by the ProgramState object and is obtained by calling the currentState() method on the debugger. In the context of this paper, the block is actually a query object.
2. The debugger calls its `step()` method.

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

Listing 7 Iterable `ProgramStates` object. Iteration routine.

This same API can be implemented in other back ends, especially *replay-based* back ends. Replay-based back ends reconstruct a past state by executing forward from an older saved state (Engblom 2012) (*i.e.*, a snapshot). Depending on the implementation, debuggers choose to sparsely capture snapshots on specific program events (*e.g.,* method calls), to repeatedly snapshot after a certain amount of time has passed, or to do both (Engblom 2012; UDB 2023).

To reverse an execution to a past state, instead of restarting the entire execution and reconstructing the target state by stepping all instructions until that state is reached (Listing 7), these debuggers reconstruct the closest snapshot to the target past state then they advance the execution deterministically from that snapshot until the target state is reached, thus reconstructing the target execution state.

In practice, most replay-based back end implementations follow this approach (Arya et al. 2017; E. T. Barr & Marron 2014; E. Barr et al. 2016; King et al. 2005; Montesinos et al. 2008; O'Callahan et al. 2017; Phang et al. 2013; Pothier & Tanter 2011; UDB 2023; Vilk et al. 2018) and are therefore potential candidates to serve as alternative debugger back ends for TTQ.

5.2. The query model implementation

Queries are declared by specifying a data source, a selection predicate, and a projection function, which are stored in the dataSource, selectionBlock, and projectionBlock fields, 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 is basically an API over the call stack. Each state of each frame in the call stack is reified as a ProgramState, whose API we use to answer queries. Because we use a step-based debugger back end, it is easy to capture all possible frame states by creating a new program state after each debugger step.

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.

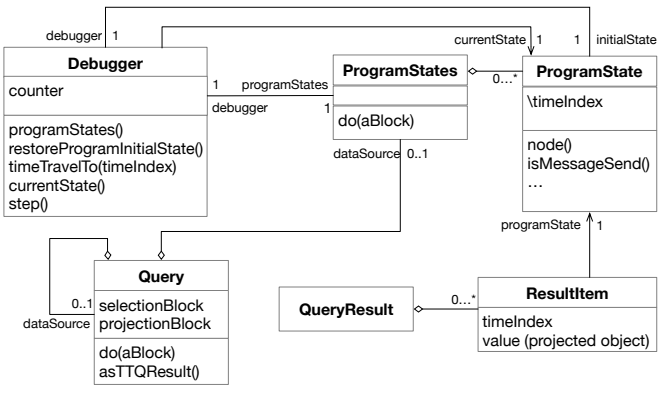

Figure 6 TTQ and supporting debugger model.

The evaluation of a Query is performed by calling its asTQResult() 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 on the current state of an execution. Program states provide accessors that internally use the debugger to obtain the requested data of the current execution state, such as the time index of the state, the currently executed AST node, etc.

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.

5.3. Discussion

Our rudimentary time-traveling debugger poses important limitations that prevent taking our current time-traveling queries implementation to production debugging environments (Lienhard et al. 2008). Nonetheless, time-traveling queries are agnostic of the debugger implementation as long as (1) the back end provides the API specified in Section 5.1 and (2) the back end provides sufficient means to reify program states and provide their API. Basically, the queries we implemented in Section 4.1 require from program states to know the current frame's current receiver and its class (all queries), if the current frame is about a message send (queries $I.1$, $I.2$) with its receiver and its arguments, the current frame's method (query $I.3$) with its arguments, if the current frame is about a variable access (queries III and IV) with the name of the variable and its value and if the current frame is instantiating an object (queries II). All query variations are conditionals over this information and do not require a specific implementation or back end. Time-traveling debuggers usually easily obtain this information as they need to store similar execution data to restore executions later. Scriptable debuggers are also able to provide such information, and in our implementation, we directly use the Sindarin scriptable debugger API (Dupriez et al. 2019) to build our program states. To scale this solution, we could therefore use production-level time-traveling back-ends instead of Seeker.

6. A Real World Scenario: Debugging a meta compiler

Seeker has been used by the developers of Druid[3], a meta-compiler for *Just In Time* compiler code generation. When compiling, Druid uses an intermediate representation (IR) in the form of a Control Flow Graph (CFG). During the compilation process, Druid runs combinations of multiple optimizations called optimization passes. Sometimes, an optimization pass introduces wrong changes in the CFG which then produces a compilation error.

This scenario has been encountered by Druid developers when executing one of Druid's unit tests. This test takes as input the *Abstract Syntax Tree* AST of the primitive method named `primitiveSize`, generates the IR, optimizes it, generates the target code, and finally runs the code to verify the behavior. However, the test assertion detects an anomaly in the behaviour of the code at run time. After a first investigation using the Pharo debugger, CFG visualizations, and Druid's machine code debugger, developers have been able to determine that this error comes from the optimization phase. The optimization phase introduces a `NULL` value in the intermediate representation (Figure 15, Appendix A). From there Druid developers had to find which optimization pass among all the optimization passes introduces the `NULL` value.

The challenge of identifying faulty compiler optimizations. Most of the time, we cannot easily determine which optimization pass breaks the CFG. Indeed, the optimization passes are destructive, *i.e.*, they apply several transformations, modifying or deleting certain parts of the CFG. Because they are destructive, these passes cannot be reverted and reapplied several times on the same CFG. Therefore, to find the faulty pass using conventional tools developers have to put a breakpoint before and after each optimization to observe the transformed CFG. If developers miss the exact point where that faulty 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. Moreover, to seek information from the CFG, Druid developers rely on a visualization (Figure 15, Appendix A). Using this visualization, identifying the problematic `NULL` value in the CFG is demanding to the developer because the value can be in any of the statements of the intermediate representation (represented by colored squares in the CFG). In our scenario, the test applies 7 optimizations passes and in total 48 optimizations to the CFG. Using conventional tools developers could end up inspecting manually the CFG 48 times or more in case of errors.

Identifying faulty compiler optimizations using Seeker. To reduce the effort required to find the faulty optimization of our scenario, Druid's developers run the generation of the intermediate representation and optimization using Seeker. To identify the optimization pass causing the bug they performed the query *I.2 - Find all messages sent with a given selector* from our Time-Traveling Queries library 4.1. Druid's developers chose this query because they know that applying an optimization pass to a CFG requires executing the method `ControlFlowGraph » #applyOptimisation:`.

Figure 7 shows the results of the query. The results are presented in a table, with some details about each call to the method `ControlFlowGraph » #applyOptimisation:`. The interesting information for Druid developers is displayed in the first column named *Step*. This column contains the *time travel index*, an identifier for the moment in the execution where each call to the method `ControlFlowGraph » #applyOptimisation:` occurred. By clicking on a time travel

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

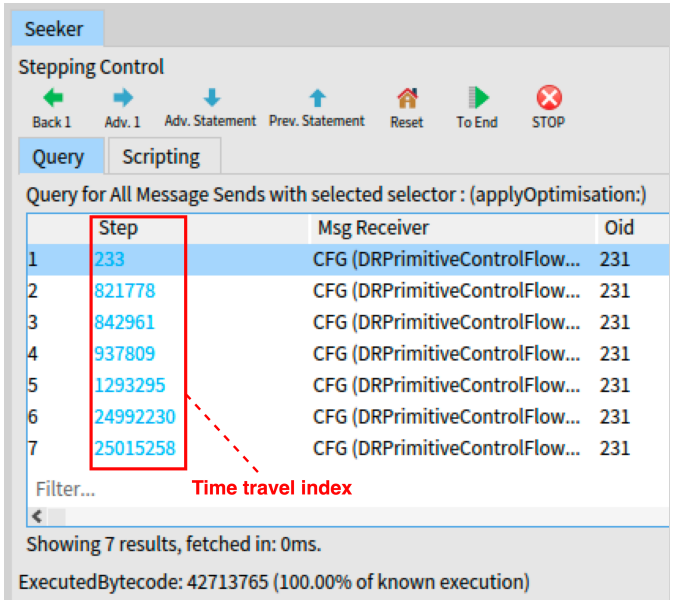

Figure 7 Seeker results to the query *I.2 - Find all messages sent with selector* `#applyOptimisation:`

index in the table, one can navigate to the moment where the optimization was performed. This is the main reason why Druid developers chose Seeker to debug their unit test. This functionality allows developers to visualize the state of the intermediate representation in the form of CFG after any optimization pass without requiring the use of breakpoints.

To search for the faulty optimization pass, they manually performed a binary search among the query results. At first, developers clicked on the fourth time travel index. It brought them to the point in the execution of the test where the fourth optimization was triggered.

```
1 self previousOptimizations do: [ :opt |
2   cfg applyOptimisation: opt
3 ]
```

Listing 8 Code displayed by Seeker after a click on a time travel index in the result table 7.

Listing 8 shows the code displayed in Seeker whenever a developer clicks on a time travel index from Figure 7. After clicking on the fourth time travel index, Druid developers inspected the content of the variable `cfg` at line 2, which opened a CFG visualization (such as in Figure 15, Appendix A). The visualization did not show any unexpected `NULL` value. It means that the optimization passes up to the third one were not faulty. When inspecting the CFG after clicking on the sixth time travel index from the result table in Figure 7, Druid developers observed the `NULL` value. It meant that the faulty optimization pass was either the fourth or the fifth one. With this approach, Seeker allowed Druid developers to narrow down their research for the faulty optimization pass without the need for using breakpoints or inspecting the state of the intermediate representation after every optimization.

Once Druid developers found the optimization pass causing the bug (which turned out to be the fourth one) they wanted to precisely identify where in the source code the `NULL` value was introduced. To do so, they used the query *II.2 - Find all instance creations of a class with a given name* from our Time-Traveling Queries library 4.1 with the class of the `NULL` value as argument. The query provided the list of moments in the execution where a `NULL` value has been created. It returned two results from which Druid developers were able to time travel (see Figure 8).

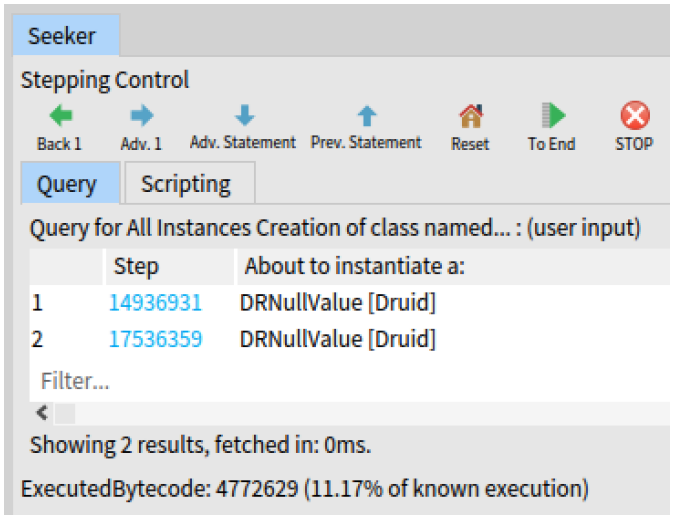

Figure 8 Seeker results to the query *II.2 - Find all instance creations of class NULL.*

On a click on either of the two results of the query, Druid developers have been able to identify the faulty code. As we can observe in Listing 9, the `NULL` value was introduced at line 15 of the method `phiWithVariables:`.

```
1  phiWithVariables: vars
2
3  | finalVars allPossibleVars phi |
4  controlFlowGraph buildDominatorTree.
5
6  allPossibleVars := vars asOrderedCollection copy.
7  self predecessors size > vars size ifTrue: [
8     self predecessors size - vars size
9    timesRepeat: [allPossibleVars :=
       allPossibleVars , vars]].
10
11 finalVars := self predecessors collect: [ :b |
12   allPossibleVars
13       detect: [ :i | b checkLivenessOf: i ]
14       ifFound: [ :i | allPossibleVars remove: i ]
15       ifNone: [ DRNULLValue new ] ].
16
17 phi := self instructionFactory
18   phiWithVariables: finalVars.
19 ^ self addInstruction: phi
```

Listing 9 Code displayed by Seeker after a click on a time travel index in the result table 8

Once they understood the problem and causes of the bug, Druid developers used the standard debugger to fix it. Therefore in this scenario, the TTQs came as a complementary tool to the standard debugger to obtain important knowledge about a bug. While other time-traveling debuggers could be used the same way and be as effective, this experience shows that our

off-the-shelf queries are usable in practice.

In this example, each query execution and each time travel between a query result took about ten minutes. However, the time-traveling queries remained attractive to the Druid's developers. The queries avoided them having to manually go through the entire optimization sequences to find a problem in a method optimization.

7. Empirical Evaluation

For the investigation of the second part of *RQ*, we formulate the following hypothesis:

H: Compared to standard debugging tools, program comprehension questions expressed as queries over program executions improve program exploration.

We ran a quantitative evaluation (Elmqvist & Yi 2015) following repeated measures design (Seltman 2012) 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 4. 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 those using standard debugging tools. We discuss other advanced techniques and how they might compare in Section 11.

7.1. 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). We investigate *H* along three dimensions of program exploration (time, precision, and debugging actions). We therfore derive *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 spent 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?

7.2. Experimental design

Our experiment is two-fold: first a tasks-solving part, following a *repeated measures design* (Christensen et al. 2011), immediately followed by a survey.

Experimental setup. A pilot participant performed the same sets of tasks prior to the participants to ensure there was no technical problem with the experimental and technical setups. We then 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.

We informed participants that the Pharo images they received were instrumented to log their actions. However, they were not informed about what was going to be measured, such as the number of actions they performed to resolve a task or the time spent debugging. 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 all queries developed in Section 4. Figure 4 shows the integration of time-traveling queries and their results in the default Pharo debugger[4]. Participants perform all their debugging tasks within this debugger.

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 one or two program comprehension questions.

The two sets of tasks are:

- The control set is composed of five tasks. We asked participants to provide an answer using exclusively standard Pharo debugging tools.
- The TTQ set is composed of five 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 them 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 one pilot. Most of them are Pharo developers with experience ranging from a few months to 20 years (Figure 9). 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 3) based on the questions described in Section 4. The number of 14 tasks is motivated by covering each possible TTQ defined in the TTQ library we built 4.1. There are at least 2 tasks for each question covered by a set of possible TTQ, except tasks 7 and 8 which are variations of 5 and 6, for which there are possible TTQ variations in the library.

When defining tasks, 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.,* hundreds of values.

We distributed the 14 tasks in different task groups, each group containing 5 tasks. We can see these groups in Table 1. We chose to not distribute tasks in groups of 7 to save participants' time and avoid them to spend too much effort in the

[4] During the experiment, participants used a prototype version of the Seeker debugger (Willembrinck Santander 2023).

control phase and being fatigued while starting the TTQ sequence. Therefore, we tried as much as possible to pair each control task with an equivalent treatment task, both targeting the same program comprehension question. We also tried, as much as possible, to make every task 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}).

As we can see in Table 1, this led us to define eight groups of ten tasks, five as control and five as TTQ tasks. Some tasks are not equally distributed nor equally paired between control and TTQ, such as 13 and 14. This is to limit the number of groups and combinations of tasks. Because groups are randomly assigned to participants, depending on the number of participants there could be groups that are never or almost never assigned. Because of the choice of limiting groups, there can be some conflicts when distributing paired tasks among the groups. 13 and 14 are, in our opinion, the most difficult tasks as they ask to recover specific values in a program execution. We therefore chose to avoid conflicts by moving these tasks together when they cannot be distributed equally between control and TTQ. For example in groups g1a, 14 appears in TTQ but in g1b appears in control. In group g4b, 13 and 14 are grouped together in the TTQ group to avoid biasing the control in favor of our hypotheses.

Table 1 and 2 respectively show how many times each group has been assigned to a participant and how many times each equivalent tasks were assigned in control and TTQ.

Table 1 Groups of tasks: each participant is randomly assigned one of the groups. The *Nb* column shows how many times each group has been assigned to a participant.

Group	Control	TTQ	Nb
g1a	02, 04, 08, 10, 12	01, 03, 05, 09, 14	6
g1b	01, 03, 05, 09, 14	02, 04, 08, 10, 12	6
g2a	02, 03, 07, 10, 11	01, 04, 06, 09, 13	5
g2b	01, 04, 06, 09, 13	02, 03, 07, 10, 11	4
g3a	03, 04, 05, 06, 07	08, 11, 12, 13, 14	4
g3b	03, 05, 07, 11, 13	04, 06, 08, 12, 14	2
g4a	01, 05, 07, 11, 13	03, 06, 08, 12, 14	1
g4b	01, 03, 05, 06, 07	08, 11, 12, 13, 14	4

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 two times for each participant: for control and TTQ tasks. Participants did not know 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

Table 2 Distribution of each task over the task sequences performed by participants.

Task	Control	TTQ
1, 2	26	21
3, 4	35	24
5, 6, 7, 8	51	35
9, 10	21	21
11, 12	14	29
13, 14	13	30

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 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 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 so we included it. 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. For a given participant, the control time is the sum of all control task times and the TTQ time is the sum of all TTQ task times.

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 TTQ results. For a given participant, the control actions value is the sum of all control task actions and the TTQ actions value is the sum of all TTQ task actions.

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

Table 3 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?	S19
		What are the values of the arguments in each message?	
10	RSNormalizerTest>>#testBasic	What are the classes of every object receiving the #color: message?	S19
		What are the values of the arguments in each message?	
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 (Sillito et al. 2008) selected in Section 4.
*: the task question is a variation of the original SQ.

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 contrast how participants perceived and trusted TTQs regarding their measured efficiency during the experiment.

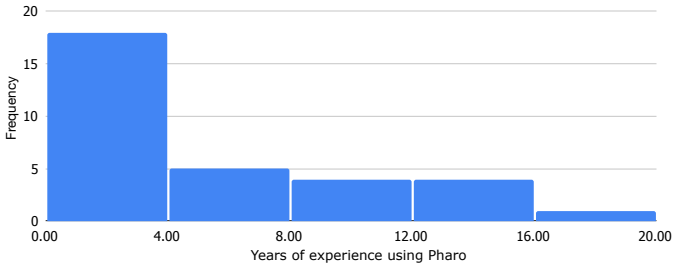

Figure 9 Histogram of participants' years of experience in Pharo.

8. Results

In this section, we analyze the data collected from the experiment[5]. We then analyze the data collected from the post-study survey.

8.1. 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 10, 11, and 12 show the differences for each participant respectively for the score, time, and debugging action metrics. For example, in Figure 10, 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 13 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 27% less time, and performed 45% less debugging actions.

[5] The data and their reproduction package are publicly available at https://github.com/StevenCostiou/2024-TTQ-Extended-Analysis.

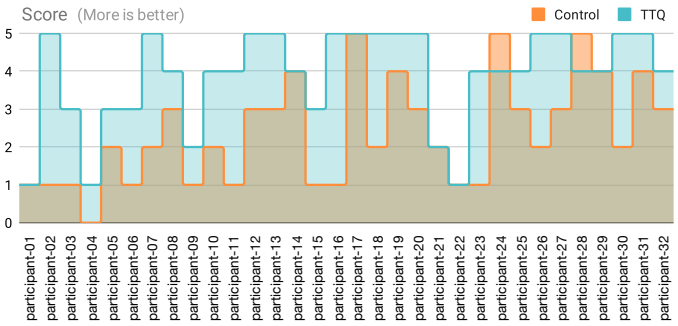

Figure 10 Participants scores. In the horizontal axis, participants are sorted in ascending order by their years of experience in Pharo.

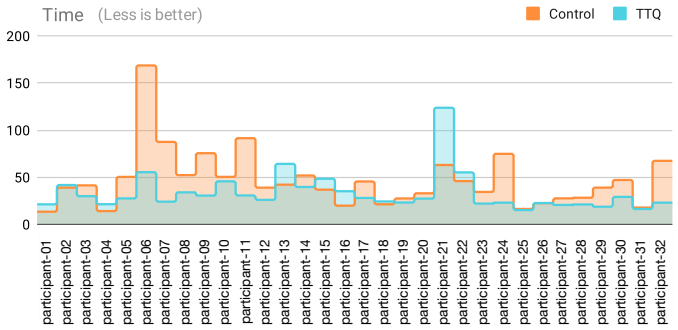

Figure 11 Participants total time per sequence, in minutes. In the horizontal axis, participants are sorted in ascending order by their years of experience in Pharo.

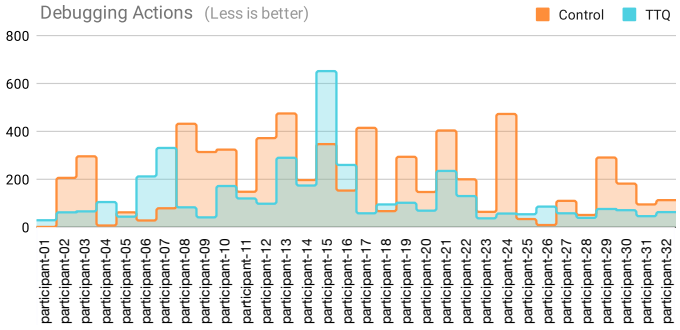

Figure 12 Participants total debugging actions. In the horizontal axis, participants are sorted in ascending order by their years of experience in Pharo.

8.2. Results analysis

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 non-parametric Wilcoxon signed-rank[6] test to compare the paired differences between 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 4). All p-values are < 0.05, the data therefore seem to support the rejection of all null hypotheses. We measure the effect size using the *Rank-Biserial Correlation* (Jané et al. 2024). All reported values show a large effect ($> |0.3|$ (Funder & Ozer 2019)), suggesting that TTQ significantly enhances program comprehension.

Table 4 Wilcoxon signed-rank test results for H_0.

	N	p-value	effect size
H_0 $EQ1$ - Score	26	<0.001	-0.937
H_0 $EQ2$ - Time	32	0.014	0.496
H_0 $EQ3$ - Debug. Actions	32	0.018	0.602

The data therefore seem to support our hypotheses that to answer program comprehension questions, TTQs improve program exploration regarding developers' efforts, time spent, and precision compared to standard debugging tools.

8.3. TTQ usage analysis

To assess if and how participants used TTQs, we first computed the number of TTQ actions that they performed during their TTQ tasks. For each participant's TTQ task, we extracted from the logs all actions related to TTQs. We divided these actions in three types: queries, filters over queries' results, and time-travel actions from a query result. We counted how many actions of each type each participant did over all their TTQ tasks, then computed a per-type average over all participants. We excluded three outliers, one from the time-travel type and two from the queries type.

Results are shown in Figure 14. On average (rounded numbers), participants performed 22 queries, 29 time-travel operations, and 3.7 filter operations over queries' results when doing TTQ tasks.

[6] We used JASP (JASP Team 2024) to perform statistical tests. JASP files are available at https://github.com/StevenCostiou/2024-TTQ-Extended-Analysis.

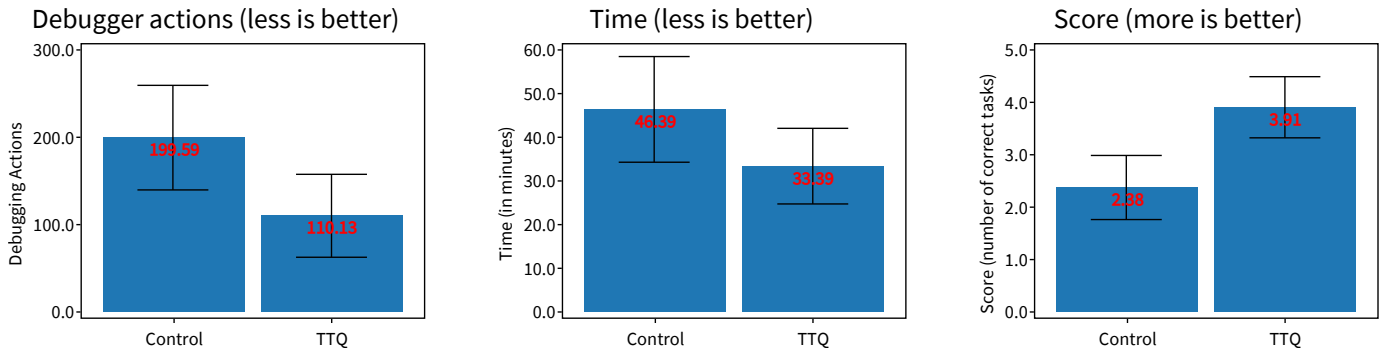

Figure 13 Average experiment results for each metric, with 95% confidence intervals.

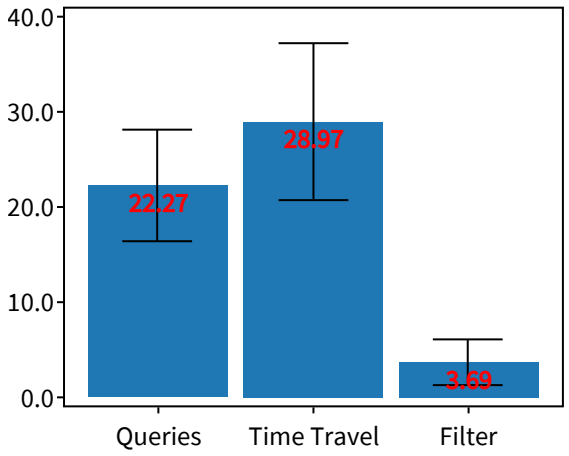

Figure 14 Average number of TTQ actions during TTQ tasks, with 95% confidence intervals and separated by action type.

These results show that participants did use TTQs, but only provide a general overview of how they used TTQ tools to answer the program comprehension questions of their tasks. In particular, we observe that on average, multiple TTQs seem to be necessary to answer a single program comprehension question (*i.e.*, 22 queries on average for 5 program comprehension questions).

Appendix B shows which queries were performed by participants for each group of tasks. Each diagram gathers the used TTQs for two to four different tasks, regrouped by task groups (Table 3). For each task in a diagram, we see the total number of the queries used by all participants. Each radius corresponds to a TTQ as presented in section 4.1. The point corresponding to the most used TTQ is always on the external disk, regardless of the diagram's tasks. The points corresponding to the other TTQs used for the tasks of the diagram are then proportionally placed, from 0 in the middle to the maximum value in the external disk. For example, for task 1, considering all participants, 9 TTQs have been used. The most used TTQ for this task is II.3 (*i.e., II.3 Find all instance creations of exceptions*), then followed by TTQ I.I (*i.e., Find all messages sent during the execution*) and so on. For Task 2, 8 TTQs have been used. The most used one is I.I and the second most used TTQ is I.3 (*i.e., Find all received messages by any object*). The larger the area covered by the dots, the greater the number of different TTQs. We observe the following:

- Several TTQs have been used for each task.
- For each task, there is always a TTQ that has been more used than the others, and that TTQ belongs to the expected TTQ family with regard to the questions asked in the tasks. In other words, when we designed the tasks with a specific question we expected participants to use a specific query family, and each time they did. We can explain this by the fact that each TTQ has a specific purpose that was adapted to the question asked in the task. This suggests that for each task participants used specific TTQs because they believed these TTQs were adapted for the task, and not randomly because they were advised to do so.
- For each pair of equivalent tasks (except tasks 1 and 2) the diagrams are similar, or at least the most used TTQ is the same (*e.g.,* TTQ II.2 for Tasks 5 and 6, TTQ II.1 for tasks 7 and 8). Since equivalent tasks have been performed by different participants, this reinforces the idea that TTQs have specific purpose and have been used relevantly.
- The usage frequency of the TTQs of the same family (I to IV) per task varies a lot according to the families. For example for task 1, the most frequent TTQs are in the order, I.1, I.3, II.1, IV.1, I.2, and II.3 (with values relatively close for II.3, I.2, and IV.1). For task 4, the most used TTQ is I.2 and just after IV.2 and then I.1. In these examples, each TTQ inside families I and II are very distinctly used by the participants. For tasks 11 to 14 relative to variable assignments, it is always TTQ IV.2 that is the most used but the only other used TTQs belong to families III and IV.
- Some TTQs have not been largely used in the experiment. It is mostly the case for the TTQs over assignments in general or on a specific object (family III and IV). The proposed tasks certainly do not cover all the TTQ. Indeed, over all the TTQs relative to assignment, only TTQ IV.2 (*Find all assignments of instance variables for a particular object*) has been largely used.

These observations suggest the equivalence of the tasks, *i.e.,* for each task group the same TTQ family was used and that

Table 5 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%

the TTQs have been used relevantly, *i.e.,* the TTQ usage is not due to chance or to a trial-and-error behavior from participants. However, we must consider that in four of the task groups (namely *g1a*, *g1b*, *g2a* and *g2b*) tasks 1 and 2 are always performed first in the TTQ treatment. This is where in diagrams the areas covered by TTQs are the largest, which could suggest a learning effect where participants poked around in the available TTQs before understanding what to do and how to use TTQs. On the one hand, this could influence the results in favor of the control groups as participants would lose time trying TTQs. On the other hand, the rest of the diagrams seem to show that participants learned which TTQ to use which would actually benefit to the TTQ group (which is the researched effect).

8.4. Post-study survey

Table 5 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.

9. Threats to validity

In this section, we discuss and mitigate threats to validity. In particular we analyse alternative hypotheses that could explain our results, and perform robustness tests to see how our hypothesis resist to variations in the data.

9.1. General design threats

Answers correctness. We produced the list of expected correct values to decide if a task answer was correct 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 considered them as correct. However, it is not possible to prove the completeness of this list.

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 and we removed two inactivity periods from the data, one of 15 minutes and one of several hours (the participant interrupted the experiment and continued it later). However, we did not monitor participants for small interruptions and distractions that might affect the results.

9.2. Carryover effect on the experiment order

We balanced the order of the experiment in two sequences ({control, TTQ} or {TTQ, control}) to avoid a learning effect between the control and the TTQ tasks such as the one reported by the pilot. However, at first glance, the data suggest a learning effect in favor of the control tasks. Participants performed better on all metrics in their control tasks with the {TTQ, control} sequence. 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 would have learned while doing the TTQ tasks first because of the questions and therefore they were more efficient during the following control tasks. This could be explained by the fact that TTQs remove part of the burden of finding out what to look for in the program execution, such as understanding that one should look for state changes in a specific object. Furthermore, multiple questions (or TTQ) can be asked with no additional cognitive cost by using the debugger menu until a TTQ yields interesting results. Therefore, when performing a similar control task participants might be informed about what they should look for, although they have to find the same information with the standard debugger.

Another way to look at this would be that for the {control, TTQ} sequence, participants have two learning phases (one per experiment). Participants were not familiar neither with the comprehension tasks nor with the TTQs. Starting with the TTQ experiment they learned both during the first part of the

experiment.

Both theories could explain the apparent difference in results between the two sequences, and both would disturb the observation of accurate data in favor of our hypothesis (*i.e.*, because it is in favor of the control, it could mask the true effect of the TTQ). We therefore formulate the following alternative hypothesis:

Ha1: There is a learning effect in favor of control tasks when the experiment is done with the {TTQ, control} sequence.

To find out if the data support this hypothesis, we separated the data of both sequences and compared them using a Mann-Whitney U test, suitable when comparing two independent groups. Table 6 presents the means of the score, time, and debugging actions metrics for the two sequences and the results of the statistical tests. Each time, we compare the control and the TTQ groups of each sequence – each group being composed of 16 data points (*i.e.*, 16 for each sequence). Except for the time metric when comparing the control groups, the data does not seem to support *Ha1*, or put another way that the observed data does not seem different between each control-control or TTQ-TTQ groups comparisons. For the time metric, the Mann-Whitney U test yields a $p - value$ of .00782 $<$ .05, which would indicate that the time data does support *Ha1*. We could accept *Ha1* for the time metric, if we refine our theories above by saying that participants learned what they should look for in their program executions using TTQ, but they should still do it with a standard debugger in the control, involving as much actions as if they did the control tasks first. They would perhaps also be more confident with their abilities to answer the task, as they learned during the TTQ tasks. They would therefore be faster, without necessarily getting a better score or performing fewer actions.

However, one must be cautious with the interpretation of these results. We divided the number of participants by two (16 instead of 32) to be able to compare the control and TTQ groups as independent groups. Therefore there is a risk that the statistical tests are underpowered because the number of data points is too small. Instead of accepting *Ha1* we leave it open, meaning that the presence of a learning effect in favor of half of the control tasks could mask the true effect of the TTQ. This does not change our acceptance of our main hypothesis H, but opens interesting research prospects: could TTQ help developers acquire stronger debugging skills through a learning effect?

9.3. 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 (three of the authors have more than 10 years of professional development experience). The pilot did not report anything related to tasks difficulty. Formally proving this equivalence is not possible in practice. Potentially, there might be small differences in task difficulties. We tried to distribute the tasks between control and TTQ to mitigate the potential observation of data influenced by small differences in task difficulty. If an easy task is assigned to a control group and its slightly harder counterpart is assigned to a TTQ group, those would be reversed in another group. However, this would not suffice to mitigate the threat of significant differences in task difficulty. We therefore formulate the following alternative hypothesis that could also explain our results:

Ha2: The tasks are not equivalent, i.e., their difficulty is significantly different, and participants' results are influenced by luck depending on the task order they were assigned to.

Comparing per-task means suggesting this equivalence (score, time, and debugging actions). Still, there are not enough samples for each task individually to tell if the data support *Ha2* or not.

Table 1 shows the different task groups assigned to participants. These task groups represent enough data to perform statistical tests. We performed a *jackknife robustness test* (Neumayer & Plümper 2017) to see how well our original hypothesis resists to changes in the data when removing task groups. We started by excluding data corresponding to the first group of tasks ($g1$) and then performed the same Wilcoxon test as in Section 8.2. We then reinserted the $g1$ data, removed the $g2$ data, performed the statistical test, and so on until all task groups had been excluded once.

If tasks are not equivalent (*Ha2*), removing task groups from the data should impact the results of the statistical tests. The group removal should especially yield strong differences for $g1a$ and $g1b$ and $g2a$ and $g2b$. First, these four groups cumulate 21 participants out of the 32 of the experiment. The data corresponding to these groups should weigh significantly in the results. Second, they have reversed groups of tasks between their control and TTQ tasks (Table 6). For example, The tasks of $g1a$ and $g1b$ are the exact same, but their control and TTQ tasks are reversed. Let us imagine that tasks 01 and 02 are not equivalent while they should be: task 01 is significantly harder than task 02. This means that doing task 01 as control in $g1a$ is harder than doing its counterpart task 02 later with TTQ in $g1a$, which would bias the observed results in favor of our base hypothesis H. This reasoning applies to each group of equivalent tasks reported in Table 3.

Results are shown in Table 7. The score is always higher in TTQ than in control, and time and actions are always smaller in TTQ than in control. All $p - values$ remain $<$ .05, therefore still supporting our original hypothesis *H*. Results are very similar between groups such as $g1a$, $g1b$, suggesting that $g1a$ and $g1b$ have the same impact and therefore that their tasks are equivalent between their control and TTQ groups. We observe the same with $g2a$, $g2b$. We also observe these similarities between $g3a$, $g3b$ and $g4a$, $g4b$ with the exception of a 12% difference in the number of actions between $g3a$ and $g3b$. For these four groups, the tasks are not exactly reversed between the control and TTQ groups (some are, some are not). This suggests that even with different tasks, the potential difficulty difference is not big enough to be measured and have an impact on the overall results. The *jackknife robustness test* results therefore do not seem to support of *Ha2*, and we conclude that the effects measured in the primary analysis are robust with regards to the threats to validity emerging from *Ha2*.

Table 6 Results according to the experiment order.

Metric	Sequence	Sequence Order			
		(A) ***Control→TTQ***	(B) ***TTQ→Control***	***Difference*** $(B-A)$	$p-value$
Score	Control	2.13	2.63	+0.5 (+23%)	.31732
	TTQ	3.59	4.25	+0.66 (+18%)	.08012
Time	Control	59.9	32.9	-27 (-45%)	.00782
	TTQ	30.5	36.3	+5.8 (+19%)	.86502
Actions	Control	206.7	192.7	-14 (-7%)	.83366
	TTQ	80.4	139.8	-59.4 (+74%)	.28914

Table 7 Results of the *jackknife robustness test* based on tasks groups. *Group* column: the removed group before testing. N: the number of data points after removal of the group.

Group	Metric	Results	
		TTQ/Control	$p-value$
g1a (N=26)	Score	+67%	.00012
	Time	-27%	.0466
	Actions	-44%	.01242
g1b (N=26)	Score	+60%	.00014
	Time	-26%	.03
	Actions	-48%	.00236
g2a (N=27)	Score	+74%	<.00001
	Time	-28%	.02382
	Actions	-37%	.01242
g2b (N=28)	Score	+74%	< .00001
	Time	-30%	.01684
	Actions	-43%	.01078
g3a (N=28)	Score	+61%	.0001
	Time	-28%	.03078
	Actions	-42%	.0114
g3b (N=30)	Score	+61%	< .00001
	Time	-25%	.01552
	Actions	-54%	.00034
g4a (N=31)	Score	+64%	< .00001
	Time	-29%	.01684
	Actions	-45%	.0048
g4b (N=28)	Score	+58%	.00012
	Time	-30%	.01242
	Actions	-43%	.00694

9.4. 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 significant 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 as a traditional one. The participant population described in Figure 9 ensures that we do not have such bias. Figure 9 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 seeing that Pharo is not taught at the University around the place where our experiment happened, therefore having already 4 years of 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 have 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 8.1, we discarded one participant's results for this reason. We acknowledge the need to minimize these scenarios for future experiments.

10. Discussion

In the following, we briefly discuss and contextualize our results.

Answering the 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, compared to standard debugging tools?* The experiment results positively answer it. Nonetheless, it is important to remark that the experiment tasks were 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 results support

the time-traveling queries approach, the specific measure improvements are significant only in the context of these questions. To conclude whether TTQ improves debugging in other areas, new focused evaluations are needed.

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 of 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 as 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.

11. Related Work

Many debugging approaches offer the possibility to perform assertions over the state of a program. For example, debugging techniques and tools offer the means to analyze and reason about a program. Depending on the nature of the debugged scenario *(i.e. the programming environment, language, etc.)*, certain techniques present advantages over others. The key differences between such work and ours are:

- We support the access to and assertions over any program state within a program execution,
- we provide an extensible framework to craft domain and problem-specific program comprehension tools.

In the following, we focus in addition on solutions offering tracing, back-in-time, or time-traveling support.

11.1. Queries Over Objects

For object-oriented programming languages, there are several approaches that account for object-specific aspects. While not directly in the debugging area, in (Jorgenson & Erickson 1994) the authors argue that testing object-oriented software should not focus on units but on the message exchange between them in a scenario.

Object-centric debugging. Object Miners (Costiou et al. 2020) is an object-centric approach for acquiring, capturing, and replaying objects to localize bugs. Developers manually select expressions in the source code from which Object Miners gathers objects at run time. Developers can then navigate the history of captured objects using a filter over the objects' state.

The *practical object-oriented back-in-time debugging approach* (Lienhard et al. 2008) builds an object changes history based on how objects are passed through the control flow of the execution. Developers navigate back and forth in the lives of objects throughout history.

Both approaches implicitly allow developers to query objects, by filtering or navigating object histories. These approaches can be expressed as time-traveling queries by selecting objects from specific program states (*e.g.*, object read, write, passed as parameter...) and building histories from the collected results.

However, the known implementations of these approaches require highly reflective language features (Costiou et al. 2020) or virtual machine specialization (Lienhard et al. 2008). The level of expertise required to use such language features increases the cost of adapting the presented approaches to a specific domain or problem different from the one they were designed for. In contrast, time-traveling queries allow developers to write queries to tackle dedicated problems. However, they require a deterministic time-traveling backend exposing the state of the program to the time-traveling query system. Whenever such a deterministic time-traveling backend is available in a system, it is easier to use time-traveling queries to tackle a domain-specific problem rather than relying on the reflective features of that system.

Logic-based event debugging queries. Some debuggers based on logic programming use queries to explore program executions. For example, OPIUM (Ducassé 1999b) and Coca (Ducassé 1999a) base their queries on event traces, and Query-based debugging (Lencevicius et al. 1997, 1999) base their queries on logic expressions to find objects satisfying particular conditions. These solutions do not provide means to query past states of objects for asking program comprehension questions.

Auguston (Auguston 1998, 1995) proposes a language to analyze a program execution trace based on a model of events. The events describe the attributes of procedural statements during execution, such as the code of the statement or its return value. This is similar to the program state API used by the time-traveling queries, which models for example the *message sends* and the assignments that are executed.

Although not focused on debugging, TESTLOG (Ducasse et al. 2006) *reifies execution traces and uses logic programming to express tests on them* (Ducasse et al. 2006). Thereby, it avoids the need to bring the system to a specific state for the test and to gather knowledge about the program behavior before the test execution. However, according to the authors, the language used to express and perform the logic tests on the traces is not well suited for end users. Similarly, Caffeine (Guéhéneuc et al. 2002) uses *the Java debugging API to capture execution events and uses logic programming to express and execute queries on a dynamic trace* (Ducasse et al. 2006). However, Caffeine do not provide state reification such as the ones required by time-traveling queries.

TTQs could be expressed on top of Prolog-based trace reification. However, the fundamental aspect of our work is to expose developers with a library of key time-traveling queries capturing important questions supporting debugging sessions. This library is usable by end users as supported by the results of our user study experiment. In addition, time-traveling queries are extensible and can be seamlessly composed and mixed with traditional debugging tools (breakpoints, step-by-step execution, etc.).

11.2. 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 (E. T. Barr & Marron 2014; E. Barr et al. 2016; Miraglia et al. 2016; Lienhard et al. 2008).

On the other hand, there are other projects that do research on how to exploit time-traveling debuggers for program comprehension and debugging (Ko & Myers 2004, 2008; Phang et al. 2013). This is where we position our work, and compare similarities against related research projects listed in the following paragraphs.

Time-traveling queries. Our work shares similarities with Expositor (Phang et al. 2013). Like our solution, Expositor combines scripting and time-traveling debugging to allow programmers to automate complex debugging tasks. Expositor uses GDB (UndoDB n.d.) as an execution logging backend, which grants time-traveling capabilities. In contrast, we use an execution-level debugging API (Dupriez et al. 2019; Willembrinck Santander 2023) offering similar capabilities. One of Expositor's main contributions is the 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 Expositor, are lazily evaluated to generate the results. Compared to Expositor, we provide an extensible list of ready-to-use queries that are mapped to common developer questions asked during debugging. Finally, Expositor was only evaluated on examples. Our empirical experiment provides new insights, suggesting that tools like our queries or Expositor have the potential to significantly enhance program comprehension activities.

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* (Ko & Myers 2004, 2008) offers to developers contextual queries to simplify the hypothesis formulation and querying activity. In contrast, our query proposal features a different approach by offering commonly needed general-purpose execution queries. Even though it does not directly support hypothesis formulation, it relieves the burden of writing queries from the developer which translates into increased efficiency as shown in our evaluation. Another point of comparison is the extensibility of the solution. In *Whyline*, only two queries can be performed, *Why did?* and *Why didn't?* and they cannot be extended. 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.

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 (Richner 2002; Richner & Ducasse 1999).

Nowadays, it is not uncommon for debuggers to provide visualization enhancements. They use dynamic information to display traces, providing visualizations of a program behavior. As discussed in Section 11.2 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 (Consens & Mendelzon 1993; Lange & Nakamura 1995). DePauw et al. (De Pauw et al. 1998) and Walker et al. (Walker et al. 1998) use program events traces to visualize program execution patterns and event-based object relationships such as method invocations and object creation.

Queries to support dynamic analysis. Program comprehension is gained by performing static analysis of a program code and dynamic analysis of its execution (Richner 2002; Richner & Ducasse 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 enhancing 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.

12. 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 an extensible query mechanism based on a time-traveling debugger. Time-traveling queries help developers answer program comprehension questions, and craft debugger extensions to domain and problem-specific debugging scenarios.

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 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.

Acknowledgments

We thanks Nahuel Palumbo for the Druid bug investigation using TTQs. This work is funded by Inria and by the ANR JCJC OCRE Project (https://anr.fr/Project-ANR-21-CE25-0004).

References

Arya, K., Denniston, T., Rabkin, A., & Cooperman, G. (2017, July). Transition watchpoints: Teaching old debuggers new tricks. *The Art, Science, and Engineering of Programming*, *1*(2). doi: 10.22152/programming-journal.org/2017/1/16

Auguston, M. (1995, May). 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.*

Auguston, M. (1998, August). Building program behavior models. In *European conference on artificial intelligence ECAI-98, workshop on spatial and temporal reasoning, brighton, england.*

Barr, E., Marron, M., Maurer, E., Moseley, D., & Seth, G. (2016, nov). Time-travel debugging for javascript/node.js. In *Proceedings of the international symposium on foundations of software engineering* (p. 1003-1007). doi: 10.1145/2950290.2983933

Barr, E. T., & Marron, M. (2014, oct). Tardis: Affordable time-travel debugging in managed runtimes. In *Proceedings of international conference on object-oriented programming systems, languages, and applications (OOPSLA'14)* (Vol. 49, p. 67-82). ACM. Retrieved from https://www.microsoft.com/en-us/research/publication/tardis-affordable-time-travel-debugging-in-managed-runtimes-2/

Black, A. P., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., & Denker, M. (2009). *Pharo by example*. Kehrsatz, Switzerland: Square Bracket Associates. Retrieved from http://books.pharo.org

Chiş, A., Gîrba, T., & Nierstrasz, O. (2014). The Moldable Debugger: A framework for developing domain-specific debuggers. In *Software language engineering* (p. 102-121). Springer. Retrieved from http://scg.unibe.ch/archive/papers/Chis14b-MoldableDebugger.pdf doi: 10.1007/978-3-319-11245-9_6

Christensen, L. B., Johnson, B., Turner, L. A., & Christensen, L. B. (2011). *Research methods, design, and analysis*. Allyn & Bacon Boston, MA.

Consens, M. P., & Mendelzon, A. O. (1993). Hy+: A hygraph-based query and visualisation system. In *Proceeding of international conference on management data* (pp. 511–516).

Cornelissen, B. (2007). Dynamic analysis techniques for the reconstruction of architectural views. In *Proceeding of working conference on reverse engineering (WCRE).* IEEE.

Costiou, S., Kerboeuf, M., Toullec, C., Plantec, A., & Ducasse, S. (2020, July). Object miners: Acquire, capture and replay objects to track elusive bugs. *The Journal of Object Technology*, *19*, 1:1-32. doi: 10.5381/jot.2020.19.1.a1

De Pauw, W., Lorenz, D., Vlissides, J., & Wegman, M. (1998). Execution patterns in object-oriented visualization. In *Proceedings of conference on object-oriented technologies and systems (COOTS'98)* (pp. 219–234). USENIX.

Ducassé, M. (1999a). Coca: An automated debugger for C. In *International conference on software engineering* (pp. 154–168).

Ducassé, M. (1999b). Opium: An extendable trace analyser for prolog. *The Journal of Logic programming*. Retrieved from citeseer.nj.nec.com/ducasse97opium.html

Ducasse, S., Gîrba, T., & Wuyts, R. (2006). 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. doi: 10.1109/CSMR.2006.37

Dupriez, T., Polito, G., Costiou, S., Aranega, V., & Ducasse, S. (2019). Sindarin: A versatile scripting api for the pharo debugger. In *International symposium on dynamic languages (DLS'19)* (pp. 67–79). ACM. doi: 10.1145/3359619

Elmqvist, N., & Yi, J. S. (2015). Patterns for visualization evaluation. *Information Visualization*, *14*(3), 250–269.

Engblom, J. (2012). A review of reverse debugging. In *System, software, soc and silicon debug conference (s4d), 2012.*

Funder, D. C., & Ozer, D. J. (2019). Evaluating effect size in psychological research: Sense and nonsense. *Advances in methods and practices in psychological science*, *2*(2), 156–168.

Guéhéneuc, Y.-G., Douence, R., & Jussien, N. (2002). No java without caffeine: A tool for dynamic analysis of java programs. In *Ase* (p. 117). IEEE Computer Society.

Jané, M. B., Xiao, Q., Yeung, S. K., Ben-Shachar, M. S., Caldwell, A. R., Cousineau, D., ... others (2024). Guide to effect sizes and confidence intervals. *DOI: https://doi.org/10.17605/OSF. IO/D8C4G*.

JASP Team. (2024). *JASP (Version 0.19.0)[Computer software].* Retrieved from https://jasp-stats.org/

Jorgenson, P. C., & Erickson, C. (1994, September). Object-oriented integration testing. *CACM*, *37*(9), 30–38.

King, S. T., Dunlap, G. W., & Chen, P. M. (2005). Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 usenix technical conference* (pp. 1–15).

Ko, A. J., & Myers, B. A. (2004). 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. doi: 10.1145/985692.985712

Ko, A. J., & Myers, B. A. (2008). 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.*

Kubelka, J., Bergel, A., & Robbes, R. (2014). 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* (pp. 1–11). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/2688204.2688212 doi: 10.1145/2688204.2688212

Lange, D., & Nakamura, Y. (1995). 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: ACM Press.

Lencevicius, R., Hölzle, U., & Singh, A. K. (1997). 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: ACM. doi: 10.1145/263698.263752

Lencevicius, R., Hölzle, U., & Singh, A. K. (1999, June). Dynamic query-based debugging. In R. Guerraoui (Ed.),

Proceedings of European Conference on Object-Oriented Programming (ECOOP'99) (Vol. 1628, pp. 135–160). Lisbon, Portugal: Springer-Verlag.

Lienhard, A., Gîrba, T., & Nierstrasz, O. (2008). Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)* (Vol. 5142, pp. 592–615). Springer. Retrieved from http://scg.unibe.ch/archive/papers/Lien08bBackInTimeDebugging.pdf (ECOOP distinguished paper award) doi: 10.1007/978-3-540-70592-5_25

Miraglia, A., Vogt, D., Bos, H., Tanenbaum, A., & Giuffrida, C. (2016). Peeking into the past: Efficient checkpoint-assisted time-traveling debugging. In *2016 IEEE 27th international symposium on software reliability engineering (ISSRE)* (pp. 455–466).

Montesinos, P., Ceze, L., & Torrellas, J. (2008). Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. *ACM SIGARCH Computer Architecture News*, *36*(3), 289–300.

Neumayer, E., & Plümper, T. (2017). *Robustness tests for quantitative research*. Cambridge University Press.

O'Callahan, R., Jones, C., Froyd, N., Huey, K., Noll, A., & Partush, N. (2017). Engineering record and replay for deployability: Extended technical report. *arXiv preprint arXiv:1705.05937*.

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

Phang, K. Y., Foster, J. S., & Hicks, M. (2013, may). Expositor: Scriptable time-travel debugging with first-class traces. In *2013 35th international conference on software engineering (ICSE)* (p. 352-361). doi: 10.1109/ICSE.2013.6606581

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

Pothier, G., & Tanter, É. (2011). Summarized trace indexing and querying for scalable back-in-time debugging. In *European conference on object-oriented programming* (pp. 558–582).

Ressia, J., Bergel, A., & Nierstrasz, O. (2012). Object-centric debugging. In *Proceeding of the 34rd international conference on software engineering*. Retrieved from http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf doi: 10.1109/ICSE.2012.6227167

Richner, T. (2002). *Recovering behavioral design views: a query-based approach* (Doctoral dissertation, University of Bern). Retrieved from http://scg.unibe.ch/archive/phd/richner-phd.pdf

Richner, T., & Ducasse, S. (1999, September). Recovering high-level views of object-oriented applications from static and dynamic information. In H. Yang & L. White (Eds.), *Proceedings of 15th IEEE international conference on software maintenance (ICSM'99)* (pp. 13–22). Los Alamitos CA: IEEE Computer Society Press. doi: 10.1109/ICSM.1999.792487

Seltman, H. J. (2012). *Experimental design and analysis*. Carnegie Mellon University Pittsburgh.

Sillito, J., Murphy, G., & De Volder, K. (2008, jul). Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, *34*(4), 434–451. doi: 10.1109/TSE.2008.26

Spinellis, D. (2018, October). Modern debugging: The art of finding a needle in a haystack. *Commun. ACM*, *61*(11), 124–134. doi: 10.1145/3186278

UDB. (2023). *Udb time travel debugger.* http://undo.io/.

UndoDB. (n.d.). *Undodb time travel debugger.* http://undo.io/.

Vilk, J., Berger, E. D., Mickens, J., & Marron, M. (2018). Mcfly: Time-travel debugging for the web. *arXiv preprint arXiv:1810.11865*.

Walker, R. J., Murphy, G. C., Freeman-Benson, B., Wright, D., Swanson, D., & Isaak, J. (1998, October). 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.

Willembrinck, M., Costiou, S., Etien, A., & Ducasse, S. (2021, December). Time-Traveling Debugging Queries: Faster Program Exploration. In *International Conference on Software Quality, Reliability, and Security.* Hainan Island, China. Retrieved from https://inria.hal.science/hal-03463047

Willembrinck Santander, M. I. (2023). *An interactive debugging approach based on time-traveling queries* (Theses, Université de Lille). Retrieved from https://theses.hal.science/tel-04512544

Zeller, A. (2009). *Why programs fail: a guide to systematic debugging*. Elsevier.

A. Druid control flow graph with an error

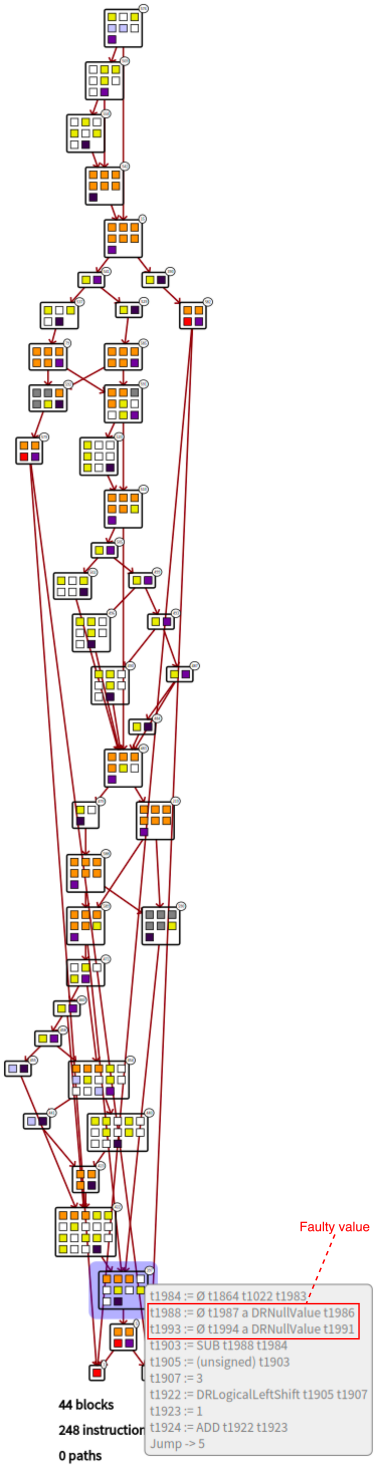

Figure 15 Druid generated control flow graph with an error.

B. Distribution of queries during tasks

The following graphics show which TTQs from our queries were used during each task. Tasks are regrouped by groups targeting a specific question from Silito.

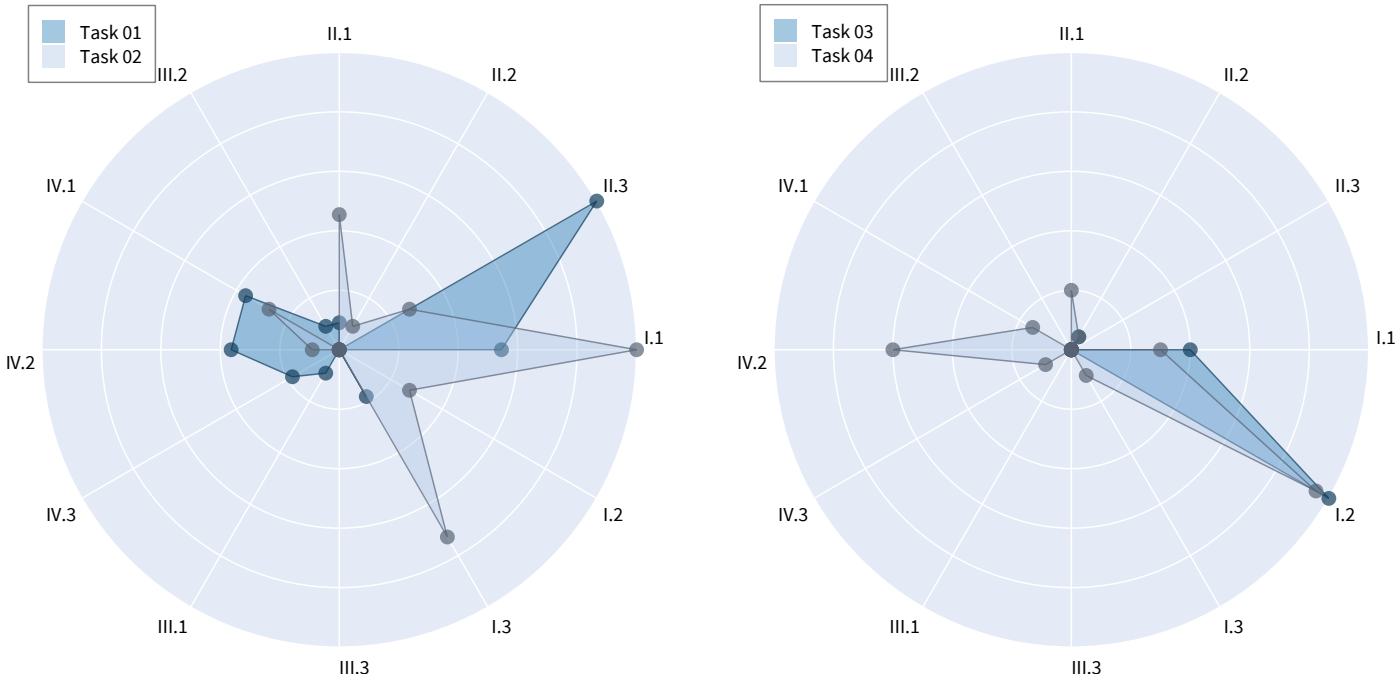

Figure 16 Distribution of queries for tasks 1-4.

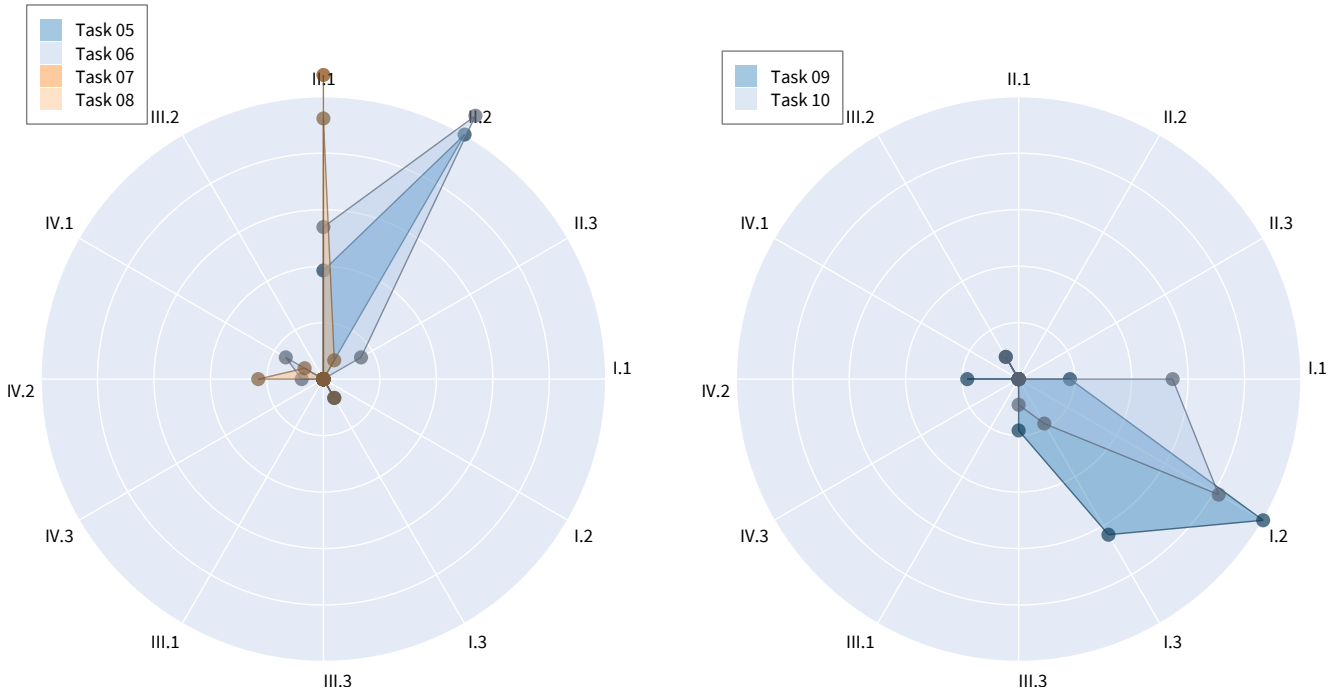

Figure 17 Distribution of queries for tasks 5-10.

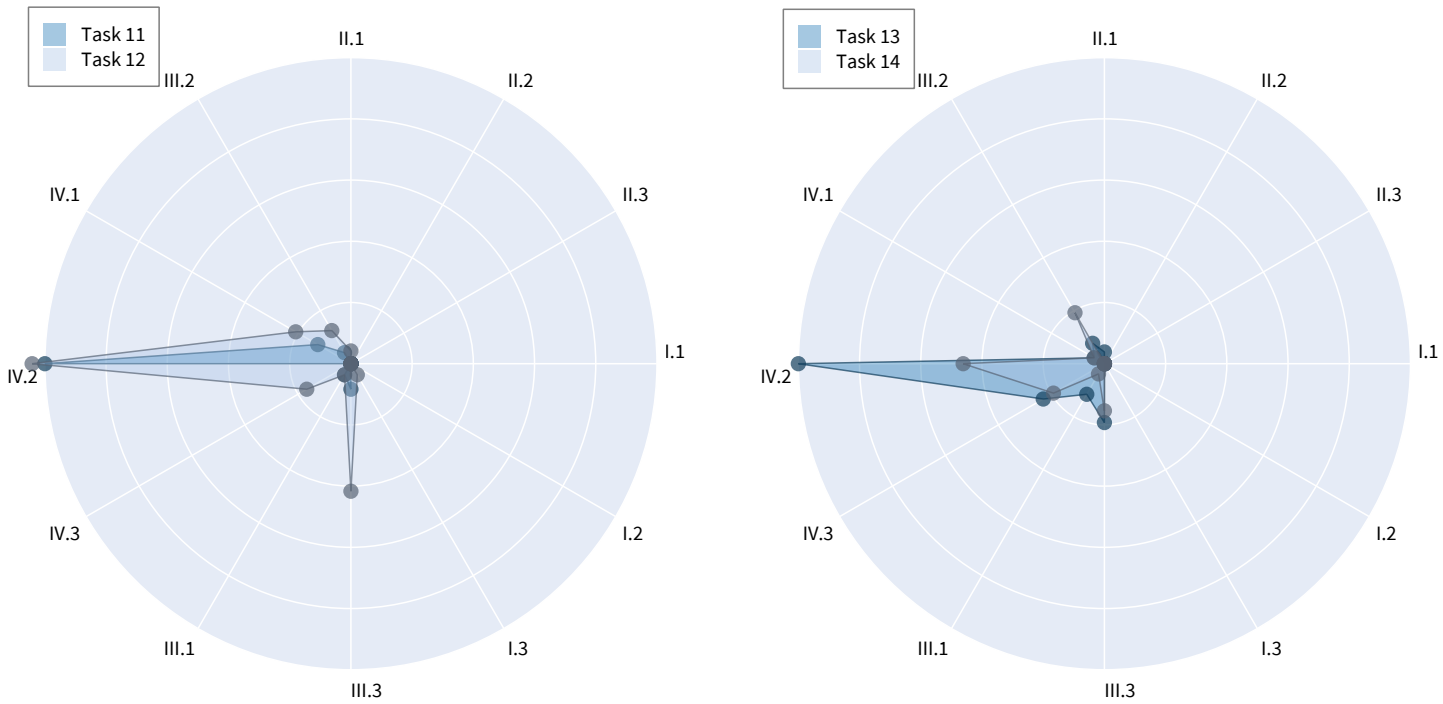

Figure 18 Distribution of queries for tasks 11-14.