Time-traveling object-centric breakpoints

Valentin Bourcier, Steven Costiou, Maximilian Ignacio Willembrinck Santander, Adrien Vanègue, Anne Etien

▶ **To cite this version:**

Valentin Bourcier, Steven Costiou, Maximilian Ignacio Willembrinck Santander, Adrien Vanègue, Anne Etien. Time-traveling object-centric breakpoints. Journal of Computer Languages, 2024, pp.101285. 10.1016/j.cola.2024.101285 . hal-04629161

HAL Id: hal-04629161

https://inria.hal.science/hal-04629161

Submitted on 28 Jun 2024

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

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

Distributed under a Creative Commons Attribution 4.0 International License

Time-Traveling Object-Centric Breakpoints

Valentin Bourcier[a,*], Steven Costiou[a], Maximilian Ignacio Willembrinck Santander[a], Adrien Vanègue[a], Anne Etien[b]

[a] *Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL,*
[b] *Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL,*

Abstract

Object-centric breakpoints aim to facilitate the debugging of object-oriented programs by focusing on specific objects. However, their practical application faces limitations. They often produce false positives and require developers to identify objects to debug in a running program, which is sometimes not possible due to non-determinism. Additionally, object-centric breakpoints are difficult to build because, to the best of our knowledge, their implementations have never been abstracted from low-level concerns. The literature describes complex reflective architectures necessary for implementing these breakpoints, and their rare available implementations are language-specific.

In this paper, we introduce *Time-Traveling Object-Centric Breakpoints (TTOCBs)*, a new definition and implementation of object-centric breakpoints based on *Time-Traveling Queries (TTQs)*. TTQs are an extensible time-traveling debugging system that allows developers to explore their program executions back and forth by executing debugging queries. We argue that our query-based implementation helps to overcome the limitations of traditional object-centric breakpoints. We describe how TTOCBs assist developers in searching for objects to debug within their program executions, even in the presence of non-determinism. We illustrate how existing object-centric breakpoints from the literature can be implemented and how new ones can be created in a few steps using the TTQ abstractions and scripting API. To build breakpoints, developers need to familiarize themselves with a short API instead of learning language reflection techniques and libraries. This makes our TTOCBs independent of the underlying TTQs and debugger implementations.

To evaluate our solution, we conducted an initial anecdotal user study on four example scenarios, providing evidence that debugging with TTOCBs requires fewer actions than with traditional object-centric breakpoints. We then discuss the comparison between object-centric breakpoints and TTOCBs in terms of applicability and performance.

Keywords: Object-Centric Debugging, Time-Travel Debugging, Program Comprehension

1. Introduction

To debug their programs, developers use debuggers which traditionally provide a set of standard debugging tools (*e.g.*, breakpoints, step-by-step execution, etc.). Standard breakpoints are defined for a class (*e.g.*, in a method) and applied to all instances of that class. *Object-centric debugging* [1] is an approach proposed to improve the debugging of object-oriented systems by scoping breakpoints to specific instances of a given class, *i.e.*, a specific object. This helps debugging by allowing developers to avoid writing complex conditional breakpoints or breaking the execution for every instance of a class. Therefore, it reduces the number of user interactions required to understand bugs [1, 2]. These object-centric breakpoints are implemented in the production version of the Pharo system [3] since 2020[1].

Challenges for adopting object-centric breakpoints. In practice, object-centric breakpoints are difficult to use (Section 2). Developers have to manually explore their executions to find objects to debug [1]. This is a tedious, repetitive, and error-prone task. Developers have to place breakpoints and step their program execution to identify and reach objects of interest for debugging [1, 2]. If they miss a point of interest (*e.g.*, by stepping too far the execution [4]), they have to restart all their exploration. If the debugged execution has non-deterministic aspects, there are no guarantees that developers will be able to reproduce their first exploration [5]. Object-centric debugging operations also generate false positives. An object-centric breakpoint might break an execution many times for a single object, *e.g.*, in a graphical rendering loop. Developers then need to go through numerous breakpoint hits and analyze each one of these hits until finding useful information for debugging. This is as tedious as manually stepping an execution with a standard debugger.

*Corresponding authors

Email addresses: valentin.bourcier@inria.fr (Valentin Bourcier), steven.costiou@inria.fr (Steven Costiou), maximilian-ignacio.willembrinck-santander@inria.fr (Maximilian Ignacio Willembrinck Santander), adrien.vanegue@inria.fr (Adrien Vanègue), anne.etien@univ-lille.fr (Anne Etien)

[1] https://thepharo.dev/2020/07/16/object-centric-breakpoints-a-tutorial/

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

Listing 1: The split-join test of collections in Pharo 11.

Furthermore, to the best of our knowledge, the only implementation of object-centric breakpoints is in Pharo [3] (a Smalltalk-based language and IDE). The current set of object-centric breakpoints in Pharo is partial and inflexible. Only a few of the original object-centric breakpoints [1] are implemented. Adding new object-centric breakpoints is difficult because it requires the use of complex reflective tools and system APIs. If developers need to extend object-centric breakpoints to tackle their domain-specific problems, they have to learn the reflective tools and system APIs to build them from scratch. Therefore, object-centric breakpoints are not easily extensible.

Time-traveling debugging to the rescue. Using time-traveling debuggers developers deterministically explore a program execution back and forth in time [6, 7]. When they miss a critical point, developers can rewind an execution and deterministically replay it from a few steps back. This prevents developers from the hassle of having to restart the entire execution and to repeat the same debugging investigation without the guarantee to reproduce the bug when non-deterministic values are present.

In this paper, we leverage time-traveling debugging techniques to overcome the limitations of object-centric breakpoints. For that we use *Time-Traveling Queries* [8, 9, 10] (*TTQs*), a time-traveling mechanism to automatically explore program executions. Using *TTQs*, developers write and execute queries over the program execution. Queries collect execution data and then allow developers to time-travel back and forth in the execution from that collected data. *TTQs* therefore make it possible to express debugging questions that developers can ask directly when the program is executed from a time-traveling debugger.

In a previous paper [11] we presented exploratory work combining *TTQs* with an object-centric perspective. We push our investigation further by expressing (Section 3) and implementing (Section 4) every object-centric breakpoints [1] as *TTQs*. We named this set of *TTQs* the Time Traveling Object-Centric Breakpoints (TTOCBs). On the one hand, TTOCBs enhance object-centric breakpoints with time-traveling capabilities. TTOCBs automatically traverse all program executions and return all potential breakpoint hits, allowing developers to navigate them by time-travelling between the breakpoints. Hence, developers never overlook any breakpoints. They can effortlessly navigate between breakpoint hits to avoid false positives, and can always deterministically reproduce their execution. On the other hand, the *TTQs* act as an abstraction layer to the reflective tools required to design and implement new object-centric breakpoints. We easily implement all the original object-centric breakpoints [1], and we can effortlessly extend them with new variations and additional support. For instance, we introduce new queries to identify objects of interest in a program execution, thus making it easier for developers to find objects by avoiding manual exploration. We describe the integration of time-traveling object-centric breakpoints in Pharo Smalltalk and illustrate their usage on an example (Section 5). As a back end for *TTQs*, we use Seeker [10], a time-traveling debugger available for Pharo and satisfying the *TTQs* requirements. We then discuss and compare the Pharo object-centric breakpoints and their time-traveling counterpart in terms of debugging capabilities and effect, and in terms of performance (Section 6). Finally, we discuss related work (Section 7) and conclude.

2. Challenges in Debugging Objects

In the following, we illustrate the difficulties developers go through when they try to comprehend objects behavior during debugging. We explain how object-centric debugging and time-traveling debugging support these challenges, and their limitations. As an example, we use a test method from the Pharo 11 code base (Listing 1).

In this test, an `OrderedCollection` of ten elements is split by another `OrderedCollection` with two elements. This operation is expected to produce a new `OrderedCollection` containing both the left and right sides of the split operation (each side of the split is an `OrderedCollection`). To understand this execution we want to obtain information about the behavior and state of instances of `OrderedCollection` during the execution of the test. Because there are many collection objects of the same kind involved, we can use object-centric breakpoints to focus the debugging on specific collections.

Challenge 1 [finding objects]: manual exploration to find objects to debug. Developers need to do a preliminary analysis of their program to find the appropriate place to put breakpoints that will allow them to identify and capture the object of interest from a debugger. The standard procedure is to step the execution until the desired object is instantiated. This potentially requires numerous and carefully executed steps before the observation of the desired object can take place. Moreover, the execution of this test produces several instances of `OrderedCollection` in certain method calls that are not immediately visible in the test. It is difficult to track each object, as none of them is seemly stored in a variable. This makes such a manual approach even more tedious.

Challenge 2 [precision]: breakpoints precision. Developers can improve over the aforementioned approach by using specific breakpoints that halt the execution whenever a class is instantiated. With this technique, developers need to halt a certain number of times to reach the `OrderedCollection` of interest. Once the object of interest has been reached, developers can use object-centric breakpoints to observe the object's behavior and the evolution of its state. It might take many halts to reach the object of interest in the first place, and then several other object-centric breakpoints to observe relevant information.

Challenge 3 [reproducible exploration]: deterministic exploration reproduction. If developers accidentally miss relevant information due to the difficulties of *Challenges 1* and *2*, they have to restart their program exploration from scratch. They have to remember and reproduce manually the procedure they used to find the object they wanted to debug, *i.e.*, where they put their breakpoints, and how many execution steps they performed. In the case of non-deterministic executions, there is no guarantee that the missed information can be found. For example, if the example from Listing 1 converted the collections to sets to remove duplicates, and then converted them back to collections, developers could not rely on the order of the elements to formulate a hypothesis about where to find a particular object each time they restart the execution.

Challenge 4: extensibility. The Pharo object-centric breakpoints implementation is complex and limited. All the original object-centric breakpoints were not implemented. For example, the *halt on object interaction* [1] breaks the execution when two particular objects interact together, *e.g.*, when one is passed as a parameter of a message sent to the other. It is difficult to implement this breakpoint with the tools available in Pharo. We have to combine different reflective techniques [12] to build it, at the cost of great efforts. As a corollary, the implementation of new object-centric breakpoints is also extremely difficult. Developers need the deep expertise of Pharo reflective layers to understand how to build object-centric breakpoints, and as for the case of the *halt on object interaction*, it is not always easily possible. If developers are unable to extend a debugging tool for specific problems or domains, they have to rely on existing tools that may not be suitable.

Taming our challenges with time-traveling debuggers. In the context of these challenges, time-traveling debuggers bring an interesting perspective. Some time-traveling debuggers provide support for finding objects and feed them to object-centric breakpoints [5] and thus partially cope with *Challenge 1 [finding objects]*. They can provide navigational information and travel back and forth in time between this information [8] which helps with *Challenge 2 [precision]*, but with most time-traveling debuggers, developers are left looking for a needle in a haystack. Time-traveling debuggers are not affected by *Challenge 3 [reproducible exploration]* as by definition they guarantee deterministic replay of executions. Some of them are extensible [6, 8] and thus can circumvent *Challenge 4 [extensibility]*. While time-traveling debuggers help with the challenges of debugging objects, they lack the tools to focus on specific objects. To the best of our knowledge, the combination of object-centric breakpoints and time-traveling debugging has never been explored.

Proposition and research question. We propose to build object-centric breakpoints on top of time-traveling debugging, thus benefiting from the advantages of both worlds for debugging object-oriented programs. We explore the following research questions:

- What would be a breakpoint-based object-centric time-traveling debugger that would assist in finding objects to debug, while being precise, deterministic, and extendable, and how to build such a debugger?

- What are the differences when debugging with traditional object-centric breakpoints, time-traveling debugging, and time-traveling object-centric breakpoints, and how should one choose between these tools for object-oriented programs?

Required Properties. Such a solution must first exhibit the properties of an object-centric debugger and of a time-traveling debugger. In their original definition [1], the object-centric breakpoints complement the standard debugging tools. Debugging takes place in the debugger of the running system and the execution is live, *i.e.*, we always observe the running program. In the scope of this paper, we adopt this definition and therefore require the following properties:

1. *Live*: the debugged execution is live.
2. *Object-Centric*: breakpoints are scoped to specific objects of interest.
3. *Reversible and replayable*: breakpoints can be navigated back and forth in time in the live execution.

Then to face our four challenges, such a solution requires the following properties:

4. *Explorable*: the breakpoint system provides means to explore the execution to find objects to debug.
5. *Precise*: a breakpoint of interest cannot be missed.
6. *Deterministic*: an execution that generated objects and hit breakpoints will always generate the same objects and hit the same breakpoints when reversed and replayed.
7. *Extensible*: developers can customize or create breakpoints for their specific problems.

3. Time-Traveling Object-Centric Breakpoints

To answer the challenges from Section 2, we combine object-centric debugging and time-traveling debugging. We redefine object-centric breakpoints as *Time-Traveling Queries* [8, 9, 10], a mechanism of queries on top of a time-traveling debugger. We call these new breakpoints *Time-Traveling Object-Centric Breakpoints* (TTOCB). In this section, we outline the background requirements for TTOCB. Subsequently, we summarize our proposition in light of this background, and we explain how our proposition meets the necessary criteria to address our challenges.

3.1. Background: Time-Traveling Queries

Time-Traveling Queries is a mechanism that automatically explores program executions to collect execution data. Developers use this data to time-travel back and forth in the execution, to each point where the data was collected from. Time-traveling queries require a time-traveling debugger to reify execution data while executing a query and to time-travel when exploring the query results. Figure 1 illustrates the interaction between the time-traveling debugger back end and the time-traveling queries. The time-traveling debugger controls the execution and generates program states each time an event of interest occurs until the execution arrives at its end. Each program state is passed to the executing query and that query uses a *selection function* to decide if the program state should be selected or not. When a program state is selected it is added to the list of the query results.

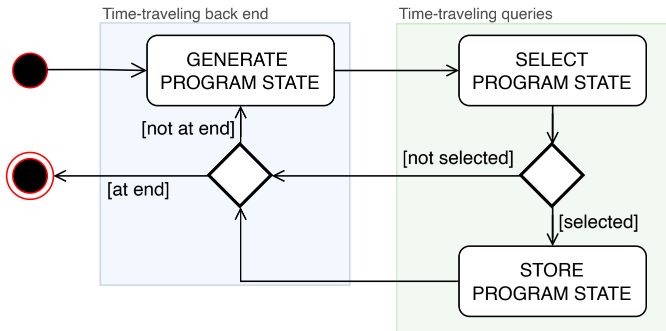

Figure 1: A simplified view of the *Time-Traveling Query* system and its time-traveling back end.

The program state concept is described in Figure 2 for an object-oriented language. This model simplifies common object-oriented language concepts. We consider that a class is a self-defined first-class object (no metaclass), that an object has a single class, we do not model inheritance and we group literals and objects under the *Object* reification.

A program state is a reification of a program instruction of interest about to be executed by the time-traveling debugger. Instructions of interest are assignments, variable reads, message sends, and class instantiations. A program state has an active context that reifies the current stack frame from which the program state is generated. A program state has a time index, used by the time-traveling debugger to identify and time-travel to the execution point from which the program state was generated. Finally, the time-traveling debugger assigns to each object an *oid*, a unique object identifier (*e.g.*, at the implementation level it could be the object's hash). The *oid* is not used by the queries but is a piece of internal information exposed by the underlying time-traveling debugger that we require for object-centric debugging in Subsection 3.2.

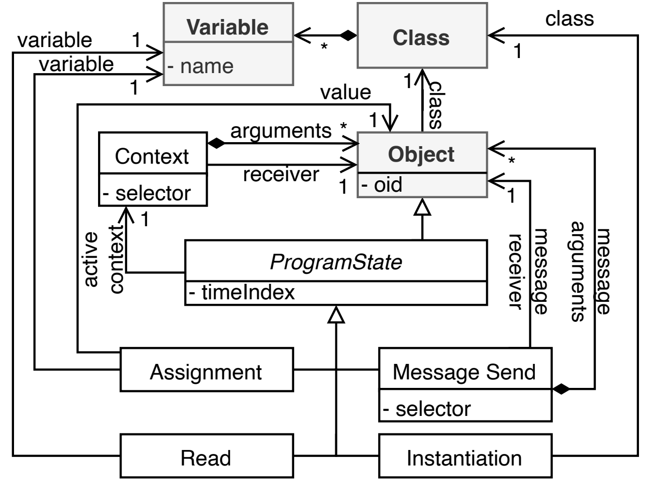

Figure 2: The program state model for an object-oriented language. Simplified concepts from the host language are in gray.

Program state objects expose an API described in Table 1, which covers the model described in Figure 2. To define and write time-traveling queries we reason over that model and use this API. Any time-traveling debugger model and implementation capable of generating program states (Figure 2) and exposing their API (Table 1) can serve as a basis for the implementation of time-traveling queries.

Example. We run a program that executes instructions among which are message sends and assignments. To build a query that filters out and captures all messages sends, we have to write a query selection function, *i.e.*, the selection step of Figure 1. In the following script, we write the `value:` method of that selection function, which takes a program state object as a parameter and returns `true` if that program state represents a message send:

```
1 SelectMessagesSends>>#value: pState
2   ↑pState isMessageSend
```

We can extend this selection function by also filtering specific message sends. For example, in the following script, we filter message sends with the `#helloWorld` selector:

```
1 SelectMessagesSends>>#value: pState
2   ↑pState isMessageSend
3     and:[pState messageSelector = #helloWorld]
```

When we run this query, the time-traveling debugger executes the program and generates program states of interest such as defined in Figure 2. The debugger sends each program state object to the query which executes its selection function. In our example, the query will select all program states corresponding to message sends with the `#helloWorld` selector, and return that subset of program states in an ordered list. This list follows the same order as the sequence of program states of the entire program. From that list, developers can pick up a particular program state and time-travel to the moment in the execution when that program state was generated, thus restoring the execution state at this moment.

Interface	**Returns:**
`arguments`	arguments of the current activated method.
`isAssignment`	`true` if the current bytecode instruction is an assignment, `false` otherwise.
`isInstanceVariable`	`true` if the variable about to be assigned is an instance variable, `false` otherwise.
`isMessageSend`	`true` if the current bytecode instruction is a message send, `false` otherwise.
`receiver`	the receiver object of the current activated method in the context top of the stack.
`selector`	the selector of the activated method of the suspended context.
`messageReceiver`	the object about to receive a message in the body of the activated method.
`messageSelector`	the selector of the message about to be sent in the body of the activated method.
`isInstantiation`	`true` if the message about to be sent in the body of the activated method is going to instantiate an object.
`classInstanciated`	the class about to be instanciated in the body of the activated method.
`oidOf:` *(added by TTOCBs)*	the unique and constant identifier of the object passed as parameter to the interface.
`isVariable`	`true` if a variable is about to be read

Table 1: The program state API.

3.2. Our solution in a nutshell

We define *Time-Traveling Object-Centric Breakpoints* as specialized time-traveling queries that reproduce the object-centric breakpoints effects and behavior [1]. We use objects' *oid*s in the selection function of object-centric breakpoints queries to scope those queries to particular objects. Instead of executing and breaking the execution each time an object-centric breakpoint hits, these queries select all program states corresponding to possible breakpoint hits for a given object. We can then navigate back and forth in the live execution throughout these breakpoints.

This new definition of object-centric breakpoints benefits from the queries' live time-traveling features and satisfies most of our required properties. The properties *1.Live* and *6.Deterministic* are automatically satisfied through the time-traveling back end, as time-traveling debuggers can deterministically restore live executions to any point in time (as long as their implementation scale). Note that in the scope of this work, we do not enforce any constraint on the time-traveling debugger, as long as it provides the model and API from Figure 2 and Table 1.

Time-traveling queries satisfy property *7.Extensible*, as it is designed to let developers write their own queries and combine them to form more complex queries [8, 9, 10]. The combination of time-traveling queries and object-centric debugging satisfies the following properties:

2. *Object-Centric*: object-centric breakpoints are defined and implemented in the form of time-traveling queries scoped to specific *oid*s,
3. *Reversible and Replayable*: we can navigate back and forth between breakpoint hits,
5. *Precise*: breakpoint queries provide all possible breakpoints hit, *i.e.*, we cannot miss a breakpoint.

To satisfy property *4.Explorable* we wrote a new kind of query dedicated to the finding of all object instantiations in an execution. At run time, we can execute these queries to search for all instance creations or for instantiations of a specific class. The queries return these instantiations and we can select objects to debug from there. To complement these queries, we built tools (context menu options, inspectors) dedicated to expose and extract objects from the live environment or to facilitate the application of breakpoint queries to objects (Section 4).

4. Implementation

In this section, we describe the implementation of time-traveling object-centric breakpoints, more specifically the *TTQs* we wrote to implement them.

4.1. Seeker: our time-traveling debugger back end

As a back end, we use Seeker [10], an interactive, live time-traveling debugger. Seeker is an interpreter-based debugger for Pharo that executes programs step by step, each step executing a single bytecode. At each step, Seeker generates a program state object such as defined in Section 3. Seeker actually generates more program states than the

ones used by the queries and defined in Figure 2. However, this is transparent for the *TTQs* that just discard unknown program states. Seeker takes care that sufficient execution data is recorded to deterministically reverse and replay an execution, thus enabling live time-traveling[2].

Figure 3 illustrates the concrete interactions between Seeker and *TTQs*. For each interpreter step, Seeker passes the generated program state object to the executing *TTQ*, which selects the program state or not. All selected program states are stored in a collection, and exposed to the developer. Seeker implements the program state API from Table 1 by using Sindarin [13], a scriptable debugger API. Sindarin allows developers to interact with the interpreter and to access its state and the state of the program, which facilitates the implementation of the Table 1 API. The program states implementation also gives control over Seeker, which allows one to time-travel in the execution between the selected program states.

We implemented all the original object-centric breakpoints defined by Ressia et al. [1] on top of Seeker as *TTQs* extensions (Figure 3). In the following, we describe the implementation of these new queries. However, to be integrated into the Seeker debugger, queries require glue code to answer to the API of the debugger (*e.g.*, to be called from menus, extract parameters from the debugger, etc.). We do not show that generic aspect of the implementation, and we focus on the core of the query that implements time-traveling object-centric breakpoints. In the following, we refer the reader to Figure 3 and Section 3 for the Seeker and time-traveling queries vocabulary. For the sake of clarity, we also refer to Seeker and to the base *TTQs* implementation as simply Seeker.

4.2. Queries to find objects to debug

For our time-traveling queries to become object-centric, we need to scope them to specific objects. Therefore, we need means to find objects and to use those objects in our queries. To find objects to scope the breakpoints to, developers look for execution contexts in the debugger from which they can identify and select objects of interest for debugging. We provide three ways to find such execution contexts.

First, as we are in a live execution, developers can explore their execution by putting normal breakpoints or by performing steps in the debugger. This is the method proposed by the object-centric debugging original work [1] and in place in the Pharo debugger.

Second, developers can use time-traveling queries from the Seeker debugger, which provides general-purpose queries. Developers then explore the execution by time-traveling through the query results. Executing a particular query allows the developer to reduce the search scope of objects as it provides selective views of the execution. If we are looking for an object receiving a particular message, we can execute a query that finds all the program states where that particular message was sent, and try to identify an interesting object among the receivers of that message. This is the method proposed by Seeker [10].

Third, we add a new method to find objects based on two new time-traveling queries for finding all objects instantiated during an execution. These queries provide a selective view of all object instantations, that we can filter by class or by property based on the objects' state. To discriminate objects between them, we use the *oid* information of the back end. We modified the program state API to expose this information (Table 1).

The first query finds all instantiations and is named `AllInstantiations`. We write it as follows, using the program state API to find if an instantiation message is about to be sent:

```
1  AllInstantiations>>value: pState
2    ↑ pState isInstantiation
```

Capturing all instantiations provides many results, and we have to filter these results to find objects of interest. We do that by applying standard Pharo selection operations over the collection of results. This is compatible with Seeker filtering tools directly from the debugger GUI (Section 5). This filtering can still be tedious if there are many results.

To scope instantiations results further, we built another query that takes as a parameter a class name and selects only the instantiations of that class. We define this query as in the following script. First, we find out if the message about to be sent is an instantiation message (line 2). Then we find the name of the class that will be instantiated (lines 3-8). Finally, we check the name of that class (`aboutToBeInstantiated`) against the class name (`targetClassName`) whose instances we want to find, and that was passed as parameter to the query (line 9).

```
1  AllInstantiationsOfClass>>value: pState
2   ↑ pState isInstantiation and: [
3       | receiver |
4       receiver := self messageReceiver.
5       aboutToBeInstantiated :=
6          receiver isClass
7              ifTrue: [ receiver name ]
8              ifFalse: [ receiver class name ].
9       aboutToBeInstantiated = targetClassName ]
```

4.3. Writing time-traveling object-centric queries

Once we find objects to debug, we need a reliable way to refer to them in a time-traveling query. One difficulty is that, for each of the three available methods to find objects, we have to at least partially execute the program a first time and select an object from that first execution. We then pass this object to a time-traveling object-centric query and execute it. However, *TTQs* execution reverses

[2]The complete Seeker implementation is described elsewhere [10].

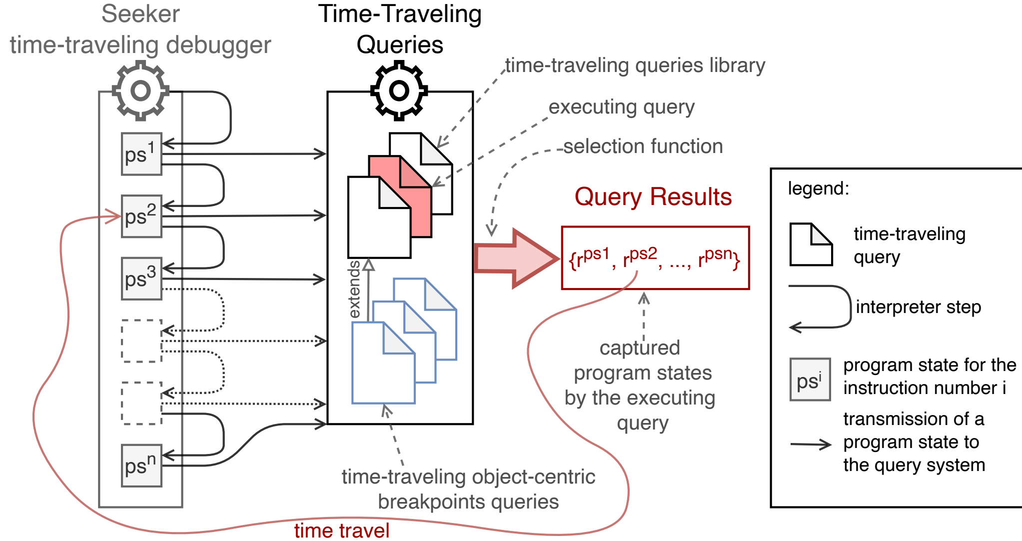

Figure 3: A simplified view of the *Time-Traveling Query*, highlighting the integration of new queries implementing *Time-Traveling Object-Centric Breakpoints*.

and replays entirely a program execution to cover all possible program states of that execution. The problem is that the object to which we scoped the query might not exist in program states before the one the object was captured from. Therefore, the query cannot compare objects from program states with an object that does not exist yet, until the target object is finally created when its instantiation program state is reached by the query.

To solve that, we use a technical feature of the Seeker time-traveling back end named the `object OID`. The `OID` is a unique identifier deterministically attributed to each object created during a program debugged execution and persists across execution replays. We are therefore able to deterministically compare objects across different replays.

For example, we extended the query `SelectMessagesSends` (Section 3) by subclassing its selection function with an object-centric selection function. We show the implementation of this new selection function in the following script. We first check if the program state corresponds to a message send (line 2) by calling the method from the super class. Then, we use the program state API method `oidOf:` which returns the unique identifier of the object in argument, in our case, the object that the message is being sent to. We compare that `OID` with the `targetOID` (line 3), which is the unique identifier of the object that we found during our exploration and to which we scoped the query.

```
1 SelectMessagesSendsToObject>>#value: pState
2   ↑(super value: pState)
3     and: [ (pState oidOf: pState messageReceiver) ==
        targetOID ]
```

4.4. Extending and Reusing Time-Traveling queries

We now illustrate via an example how we specialize existing queries in Seeker to become object-centric, and how we combine them to create new ones. We first extend Seeker's queries to find all readings and writings in a program execution with two queries respectively finding all readings of a particular object's state and all writings to a particular object's state. We then combine these two queries into a query to find all readings and writings to a particular object's state. In the following, we suppose that we found an object and that we stored its `OID` in the instance variable `targetOID` of the query's selection function.

We first extend Seeker's query to find all variable readings of the execution. This query defines the following selection function, which just checks if the passed program state is about to read a variable. Seeker defines this selection function as:

```
1 SelectAllReadings>>#value: pState
2   ↑ pState isVariable
```

As in the previous section, we have to write a subclass of that selection function and use a conditional checking for the receiver `OID` against the `targetOID`:

```
1 SelectAllReadingsInObject>>#value: pState
2   ↑ (super value: pState) and:[
3     (pState oidOf: pState receiver) = targetOID]
4
```

In doing so, we extended Seeker's original query to find all variable readings in an execution to find all variable readings of a single object. This new query is integrated into Seeker's menu by implementing Seeker's GUI API methods, and if executed behaves similarly to an object-

centric breakpoint that would break every time the target object's state is read. Instead, the query will produce a list of results corresponding to all possible breakpoint hits.

The selection function `SelectAllWritings` for finding all writings of the state of an object is written following the exact same procedure. We extend the Seeker selection function to find all writings of variables in all the executions (which tests for `isAssignment` instead of `isVariable`) and add a conditional on the receiver `OID`.

We now implement a query to find all readings and writings accesses to a specific object. However, we already wrote two queries to find readings and writings separately. We therefore build a new selection function that combines both of these queries. We write a specific instantiation method for our selection function, that takes the `OID` of the object of interest as a parameter. The implementation is shown in the following listing. We first store the target oid in an instance variable for comparison with each program state's receiver `OID` (line 2). Then, we instantiate our reading and writing selection functions and store them into instance variables (lines 3-4).

```
1  SelectAllReadingsAndWritingsInObject>>#targetOid: oid
2    targetOid := oid.
3    readingSelection := SelectAllReadings objectOid: oid.
4    writingSelection := SelectAllWritings objectOid: oid
```

We then implement our new selection function by combining the selection functions of the two existing queries, which will select program states that would be selected by either of these queries:

```
1  SelectAllReadingsAndWritingsInObject>>#value: pState
2    ↑ (readingSelection value: pState) or: [
3        writingSelection value: pState ]
```

By composing existing selection functions, we built a new time-traveling object-centric query, in a simple and extensible way. We implemented all time-traveling object-centric breakpoints following a similar procedure. Table 2 describes, for all original object-centric breakpoints, their equivalent time-traveling object-centric breakpoint implementation.

Special case: the halt on interaction breakpoint. Some object-centric breakpoints are difficult to implement in Pharo. For example, implementing the base *Halt on interaction* that breaks the execution each time an object sends or receives a message to/from another object requires to intercept every message sent to both objects. This cannot be done simply, and we have to combine different reflection techniques [12] to implement that breakpoint.

Listing 2 shows our implementation of the *Halt on interaction* breakpoint with time-traveling queries. This implementation uses the target object OIDs (line 5) obtained from the objects we want to instrument in the debugger and the current receiver and sender OIDs (line 6). We perform three checks:

1. Line 3: we check that the current program state corresponds to a message send (*i.e.*, an interaction between objects),
2. line 9: to reject self messages sends, we check that the message sender (*i.e.*, the receiver of the activated method *cf.* table 1) is different from the object that receives the message (*i.e.*, the message receiver),
3. line 10: we check that the sender and receiver OIDs correspond to the expected objects OIDs.

```
1  SelectAllMessagesBetweenObjects>>#value: pState
2    | expectedOids messageOids |
3    pState isMessageSend ifFalse: [ ↑ false ].
4
5    expectedOids := {firstObjectOid. secondObjectOid}.
6    messageOids :=
7        {pState receiverOid. pState messageReceiverOid}.
8
9    ↑ pState receiverOid ~~ pState messageReceiverOid
10       and: [ expectedOids includesAll: messageOids ]
```

Listing 2: The implementation of the *Halt on interaction* time-traveling object-centric breakpoint. We simplified the object OID recovery API usage for the sake of clarity.

An additional problem with such a breakpoint is, provided we can implement it with the tools available in Pharo, developers would face twice *Challenge 1 [finding objects]* because they now have to find two different objects to debug their interactions. With time-traveling object-centric breakpoints, we improve that situation by providing additional queries to find objects (Section 4.2).

5. Debugging Objects Through Time

In this section, we present our object-centric extension of the *Seeker* debugger. We illustrate how we use it on the example from Section 2.

5.1. Debugging with Time-Traveling Object-Centric Breakpoints

The *Seeker* debugger is integrated into the Pharo debugger, where it offers a list of general-purpose queries that can be executed from the context menu of the debugger. We extended this menu with new queries implementing object-centric breakpoints. In the following, we illustrate how we use these new queries to explore the program example from Listing 1.

Figure 4 shows our debugger opened at the beginning of the execution of the code from Listing 1. To find all collections created during that execution, we select the `OrderedCollection` string and we execute the query *All instances creation of class named as selection* from the context menu (Figure 4(a)). Once the query has been executed, the results are displayed in a table (Figure 4(b)). These results contain the complete list of all the `OrderedCollection` objects that are instantiated during the execution of the debugged program.

Object-Centric Breakpoints [1]	**Time-Traveling Object-Centric Breakpoints implementation**
Halt on read	Selects the program states corresponding to an **instance variable read**, where the oid of the **receiver object** matches with the oid of the selected object.
Halt on write	Selects the program states corresponding to an **instance variable assignment**, where the oid of the **receiver object** matches with the oid of the selected object.
Halt on call (the observed object receives a message)	Selects the program states corresponding to a **message send**, where the oid of the **receiver object** matches with the oid of the selected object.
Halt on invoke (the observed object sends a message)	Selects the program states corresponding to a **message send**, where the oid of the **sender object** matches with the oid of the selected object.
Halt on creation (a given class is instantiated)	Selects the program states corresponding to an **instanciation message**, where the **name of the class** about to be instantiated matches with the one selected.
Halt on object in call or invoke (the observed object is passed as a parameter to a message send)	Selects the program states corresponding to a **message send**, where the oid of the selected object matches with one of the **arguments oids**.
Halt on interaction (two particular objects invoke each other)	Selects the program states corresponding to a **message send**, where the oid of the **sender or the receiver** matches with the oid of two selected objects.

Table 2: Mapping between object-centric breakpoints [1] and their Time-Traveling Object-Centric Breakpoints counterpart implementation.

From the results table, we have access to each of these objects. We can filter the list of results using a text field at the bottom of the pane. By right-clicking the corresponding result item, we can execute the command *inspect object about to be instantiated* (Figure 4(c)) to observe the object in more detail.

This command opens an inspector on the object corresponding to the selected result row and shows the object state at the current moment of the execution from which we executed the query. For example, let us say that we stepped the execution up to a program state p, in which there is an initialized object O^p. We then execute the query to find all instances from that point in time. Seeker executes the whole program to find all instance creations and display the results in the list. Then, Seeker places back the program execution to the state p so that we have the impression that the query found information about the future and the past of the current view of the execution displayed by the debugger. If we find O in the list and inspect it, we will see that object in the current state O^p. If O has not yet been instantiated, the execution is advanced to such an event so that the object can be inspected.

Every result entry of *TTQs* includes a timestamp (*i.e.*, the `timeIndex` from the model described in Figure 2) of every registered event, that can be seen in the `step` column of Figure 4. Clicking on the `step` column of any result item will advance or reverse the execution, time-traveling to the registered timestamp. This feature makes query results act as *bookmarks* of an execution, providing a practical and precise alternative to breakpoints and stepping operations for program exploration. Since every instantiated object is accessible from the results list, it avoids the tedious process of stepping and halting to access them.

5.2. Seeker Inspector: an Object-Centric Querying Hub

Pharo traditionally offers the *inspector*, a graphical utility for object inspection — a view that displays information about the state of an object, its API, and other useful information. We extended *Seeker* with an enhanced object-centric version of the Pharo *Inspector*.

This specialized inspector synchronizes with the time-traveling debugger to keep a live updated view of the inspected object. Stepping the execution forward or backward triggers updates of the displayed information. The title now includes the inspected object OID along with the current execution time index (*Time* in Figure 5), expressing that the inspector is displaying a specific object at a particular time of the execution. This inspector contains a menu for object-centric queries available for the inspected object (Figure 5):

- *Time travel to instantiation instruction.*
- *List all messages sent to the inspected object.* Finds all occurrences of any message sent to the inspected object during execution. Equivalent to the *Halt on call* breakpoint.
- *Specific message....* Opens a submenu from which we can select a message from the inspected object's protocol. Finds all occurrences of that message sent

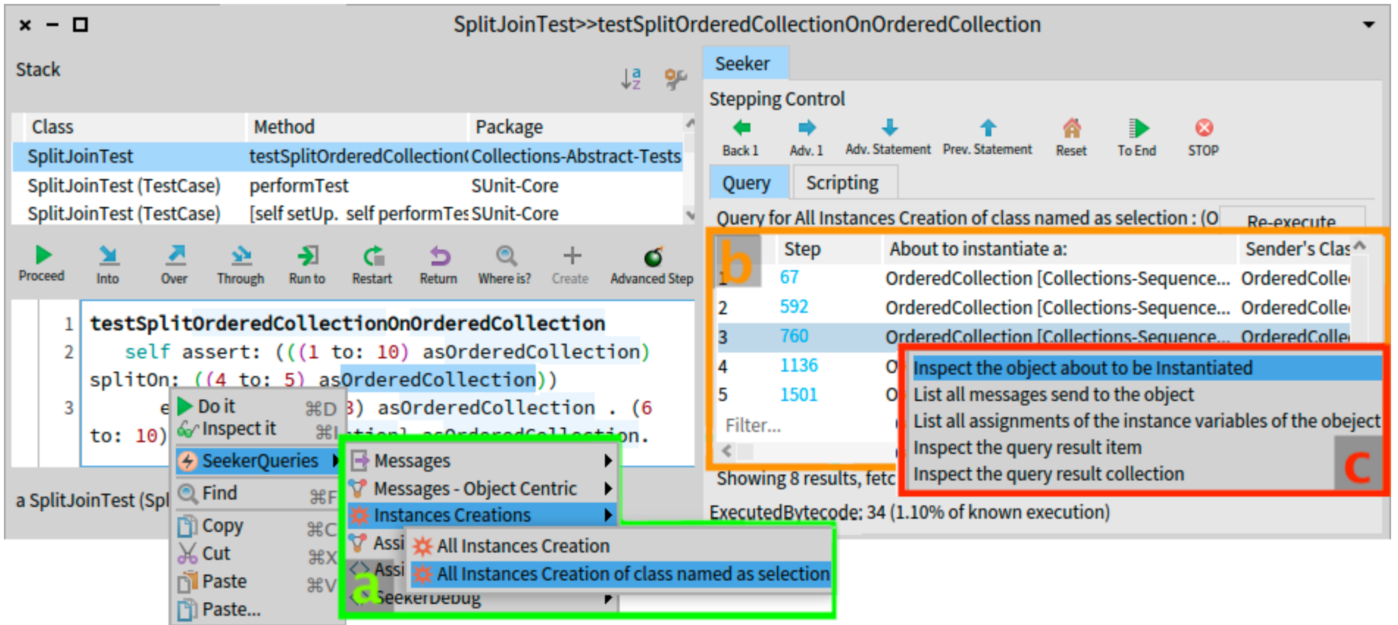

Figure 4: *Seeker* interface within the Pharo debugger. To the left, the *Time-Traveling Queries* menu(a) is available in the code presenter. To the right, the Query Results Table(b) and its context menu(c).

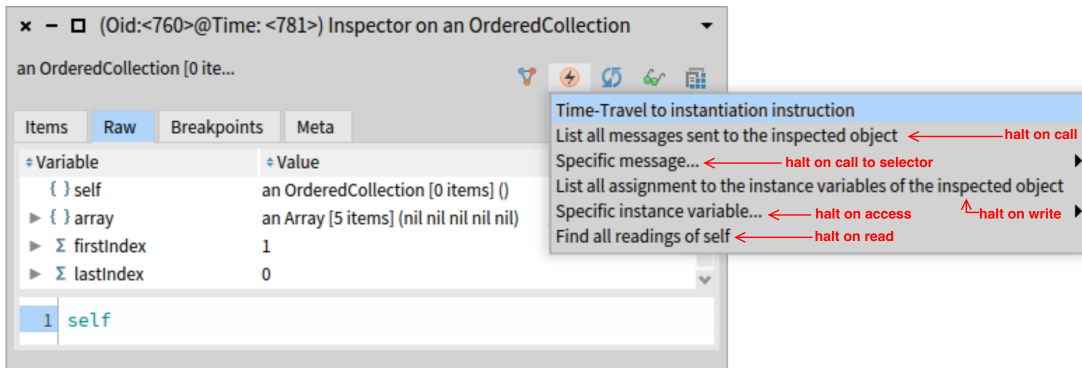

Figure 5: Menu in the *Seeker* inspector: a practical access to time-traveling object-centric breakpoints. Annotated in red in the menu, we see the correspondence between our queries and the object-centric breakpoints they implement.

to the inspected object during execution. Equivalent to the *Halt on call* breakpoint for a given selector.

- *List all assignments to the instance variables of the inspected object.* Finds all writings of instance variables of the inspected object. Equivalent to the *Halt on write* breakpoint.
- *Specific instance variable....* Opens a Submenu from which we can select an instance variable of the object. Finds all accesses to that specific instance variable of the inspected object during execution. Equivalent to the *Halt on read/write* breakpoint.
- *Find all readings of self.* Finds all readings of the instance variables of the inspected object during execution. Equivalent to the *Halt on read* breakpoint.

Using Time-Traveling Object-Centric Breakpoints developers can automatically capture all points in execution where an object received a message, was updated, and so on. Developers no longer need to repeatedly step back and forth in execution to retrieve these points. The results provided by a Time-Traveling Object-Centric Breakpoints can be used to find new points of interest in the execution from which to continue exploration. The new phase of exploration can be performed using either the standard tools or a new TTOCB.

The execution is *live* and provides *object-centric* views and operations (*i.e.*, the breakpoints). The debugger and the query to find object instantiations help finding objects to debug (*explorable*). After executing a breakpoint query, they access all information from the *Query Results Table* and developers cannot miss a breakpoint hit (*precise*). They can navigate back and forth in time through the breakpoints (*deterministic, reversible, and replayable*). Finally, as demonstrated in Section 3 and 4, developers can extend the breakpoints by writing new queries (*extensible*).

6. Evaluation

In this section, we highlight the practical differences between object-centric breakpoints (OCB) and time-traveling object-centric breakpoints (TTOCB). We show the trade-offs between the two solutions in terms of usability, results, and performance.

6.1. Running examples comparison

To highlight the differences between the use of OCB and TTOCB, we performed an anecdotical experiment in which a single developer performs a set of tasks twice, once with OCB and once with TTOCB. Our objective is to compare informally the practical differences between the two kind of breakpoints. We selected four tasks from an empirical study [8] on program comprehension, in which the developer has to answer four questions (one per task). These questions require the developer to adopt an object-centric perspective. We asked a student in software engineering to first perform the tasks (and answer the questions) with OCB, then to try answering the same questions again but with the help of TTOCB. At the time of the experiment, the student was currently at the end of a two years long internship on software debugging tool development and used Pharo daily. We observed the process and reported our observations in detail in Appendix A.

Both OCB and TTOCB allowed us to find the elements required to answer the program comprehension questions. The ways we reached these elements are however different. We counted the number of debugging actions we used to obtain the correct answers for these four questions. We consider as debugging action any execution action (putting a breakpoint, stepping the execution, resuming the execution, inspecting an object, selecting a variable) or observation action (*i.e.*, deducing information that helps answering the question from the debugger). For example in the table describing the debugging procedures for the *Question 3* (Appendix A), the third step of the procedure with OCB reports the following action: *Click Proceed. we observe that* `pc` *is assigned* `29`. We count two actions: first, the *Proceed* action resuming the execution, and second the observation of the assignment of the value `29` to the variable `pc`. When counting the number of actions, we consider that putting an object-centric breakpoint is equivalent to executing its time-traveling query counterpart, *i.e.*, its TTOCB equivalent. We apply this procedure for each line of the report tables and add up the numbers.

Results are shown in Table 6.1. We used more breakpoints with TTOCB than with OCB but we performed half the debugging actions. That is, it seems that the time spent to reach the same execution information is different. With OCB, the developer used a lot of actions (such as stepping) to get to the point where a breakpoint could be put. With TTOCB, the developers put more breakpoints to reach the same information but did 50% fewer actions.

TTOCB is based on Time-Traveling Queries, for which an empirical experiment [8] has shown that developers using queries did 38% less debugging actions than developers using the standard Pharo debugger. Therefore these results seem not inconsistent, but they are anecdotical since they are based on examples. These results should be investigated more in-depth empirically to see if they can be confirmed, and also if that higher difference compared to *TTQs* is due to OCB that would be less effective than the Pharo standard debugger, or to TTOCB that are more effective than OCB.

	OCB	TTOCB
Question 1		
Actions	10	9
Breakpoints	1	1
Question 2		
Actions	13	6
Breakpoints	1	1
Question 3		
Actions	16	7
Breakpoints	1	3
Question 4		
Actions	18	4
Breakpoints	1	1
Total		
Actions	57	26
Breakpoints	4	6

Table 3: Count of installed breakpoints and performed debugging actions when answering 4 program comprehension questions with OCB and TTOCB (see Appendix A).

We discovered more practical insights while running these examples. The manual search for objects made us miss information happening prior to finding an object of interest with OCB whereas TTOCB sometimes provided too much detailed information whose filtering was a new source of difficulty:

Manual search of objects. With OCB, we need to manually step the execution until we find an object of interest. This can be tedious, for example for question 4 we needed to perform multiple *step-throughs* and search the stack of values in the execution context. We missed a lot of instance variables writes, because a lot of them were performed before we got to the object. Notably, we lost the assignments of instance variables made during the initialization of the object of interest. We can also miss messages sent to that object of interest that occurred before we got to the object.

With TTOCB, we used a query to obtain all instance creations during the execution of the program. It does not matter when we get to the object because the TTOCB will find them all and allow us to time-travel to the instantiation of any of those objects. From there we can use TTOCB to find all variables writes of that object. It is then that we observed many writes in that object, that we did not see with OCB.

Finding and filtering resuls. If TTOCB easily provides the complete list of object instantiations, or of messages sent and variable writes to an object, the number of results can be tremendous. Filtering is then an additional burden, and TTOCB provides two ways for that. We can filter the list of results based on a string representation (*e.g.*, by filtering method names). To do this, it is necessary to know what we are looking for, which is not always the case when debugging. We can also time-travel to one of the results and execute another TTOCB based on contextual information. Similarly, we need to know when to time-travel and on what new elements base our TTOCB exploration.

Threats to validity. In this empirical experiment report, we face two main threats to validity. First, this analysis is based on only four examples. Second, we count only the debugging actions that we can see from the report. For example, we do not know how many implicit actions the developer may or may not have performed to decide when to step into a message send or when to step over it, etc. Future work should consider evaluating the scenario in a controlled manner, with a *Think-Aloud* protocol [14] to capture the developers' intents and provide a more accurate comparison.

6.2. Performance evaluation

To highlight the differences between the performances of OCB and TTOCB we used a unit test (detailed in Appendix B). We run 30 systematic executions of each benchmark using *ReBench* [15], with Pharo 11 image (Pharo-11.0.0+build.714) on an Apple M1 Max, 32GB RAM, running on Darwin Kernel Version 22.5.0.

We performed seven benchmarks, one for each type of object-centric breakpoint implemented in Pharo, *i.e.* breakpoints on variable read, writes, and method call (see Table 6.1). For each benchmark we took 30 measures running the unit test, first using Pharo OCB implementation and then with the TTOCB implementation.

Setup for Pharo OCB. A Pharo object-centric breakpoint is supposed to interrupt the execution. To create a runnable benchmark, we modified the breakpoints behavior to not trigger an execution interruption but to return just before breaking. This way, the breakpoint mechanism is executed but the execution break is simulated and the execution continues.

We manually provide the object to debug and we install the breakpoint on it at the beginning of the benchmark. Instrumentation time is included in the measures, but it is negligible over the whole benchmark execution (less than 10 milliseconds).

Setup for TTOCB. To provide the object to debug, we have to run the debugger once over the entire execution so that the object OID is known by the debugger. This step would be performed anyway by developers who would execute the *Find all instantiations* query. Then, we execute the query corresponding to the evaluated breakpoint. We provide the same object to OCB and TTOCB.

Results. We detailed the results of the benchmarks in Table 6.1. We observe that the average time required to execute the test with OCB is between 140 and 240 milliseconds. The average time for the equivalent operation using TTOCB is 97004 milliseconds. Although this performance evaluation shows that on average TTOCB is 602 times slower than OCB, these results must be balanced with previous observations. For instance, the developer performing the examples of Section 6.1 with TTOCB did not report any problematic slowdown compared to the OCB. None of the participants of a previous experiment with Time-Traveling Queries [8] reported any problematic slowdown either. In our own informal experiments, TTOCB seems usable on unit tests, but less usable with much longer executions. On a practical aspect, the reduction in the number of debugging actions with TTOCB, if confirmed through more experiments, could compensate for the slowdown compared to OCB. This could also explain why the persons exposed to the tool did not report any problem about that aspect.

7. Related Work

Object-centric breakpoints and time-traveling debuggers are orthogonal tools, and to the best of our knowledge, they have never been combined to support each other. Since object-centric debugging research is less abundant, we chose to be exhaustive when studying the object-centric debugging literature while selecting only relevant and significant time-traveling debugging results. In particular, to scope our research we selected time-traveling debugging solutions that satisfy completely or partially at least 3 properties. Table 7 presents an analysis of object-centric debugging solutions and time-traveling debuggers with regard to the properties identified in Section 2.

Object-centric debugging. All object-centric debuggers work are live debugging tools [1, 2, 5, 17, 21, 19, 22]. Most of them are extensible using meta-programming, such as behavioral adaptation [16, 17, 18] or meta-objects [21, 20, 19, 22]. In theory, the original object-centric breakpoints [1] are extensible through a meta-object protocol [20]. However, in practice that work is no longer available in the latest Pharo versions and is difficult to reproduce. The current Pharo implementation of the breakpoint is made with Reflectivity [19, 22]. These solutions satisfy none of the other properties.

Object Miners [5] is the only object-centric solution that exhibits some properties of time-traveling debuggers. The technique provides a set of object-centric tools to acquire, capture, and replay objects from specific expressions of a program. Developers manually select (sub)expressions

Breakpoint type	**Execution time with OCB (in ms)**		**Execution time with TTOCB (in ms)**		**TTOCB / OCB overhead**
Halt on state access	170	± 0	97512	± 961	×574
Halt on read	163	± 2	102331	± 692	×629
Halt on write	150	± 0	93131	± 851	×621
Halt on call (1 selector)	240	± 1	91927	± 554	×382
Halt on state access (1 var)	150	± 0	99570	± 1229	×664
Halt on read (1 var)	149	± 1	98413	± 689	×660
Halt on write (1 var)	140	± 0	96146	± 1232	×687

Table 4: Time in milliseconds required to execute a controlled unit test (Appendix B) with the Pharo object-centric breakpoints and the equivalent time-traveling object-centric breakpoints. *(*1 var) means the breakpoint is set on one specific variable, *1 selector* means the breakpoint is set on one specific selector.

	Live	OC	Reverse-Replay	Explorable	Precise	Deterministic	Extensible
TTOCB	■	■	■	□	■	■	■
OC Breakpoints [1, 2]	■	■					□
Seeker [8, 9]	■	□	■	□	■	■	■
Sindarin [13]	■	□					■
Object-Miners [5]	■	■	□			□	
Run-time adaptation [16, 17, 18]	■	■					■
Reflection [19, 20, 21, 22]	■	■					■
Flow Debugger [23, 24, 25]		■	■	□	■	*N/A*	
Trace Debugger [26, 27]		■	■	□	■	*N/A*	
Whyline [28, 29]			■	□	■	*N/A*	
Expositor [6]			■	□	■	*N/A*	■
Unstuck [30]		■	■	□	■	*N/A*	
FReD [31]	■		■	□	■	■	□
DeloreanJS [32]	■		■	□	■	■	□

Table 5: Analysis of object-centric debuggers and time-traveling debuggers with regards to the desired properties to answer the challenge from Section 2. Legend: *OC*: Object-Centric, *TTOCB*: Time-Traveling Object-Centric Breakpoints, *N/A*: Non Applicable, ■ fully supported property, □ partially supported property.

from the program from which objects are automatically captured and debugged during the execution. With the Miners, we can only force a specific object to be replayed deterministically from the point it was captured from, *i.e.*, if a method call returned an object that we capture we can force subsequent executions of that method to deterministically return that same object. This can only work if we manage to find and capture objects of interest, otherwise it does not support time-traveling operations in the general sense.

Scriptable debuggers. Sindarin [13] is an outsider: it is a debugger that controls program executions with user-defined scripts. The debugger gives access to a rich and versatile API from which developers can obtain several reifications such as receivers and senders of messages. One could therefore easily build object-centric debugging scripts, but that is not natively supported by the API. Sindarin is by definition live as it manipulates directly the execution, and fully extensible as it only works through user-defined scripts.

Similarly, Expositor [6] combines scripting and time-travel debugging to allow programmers to automate complex debugging tasks. From an execution, *Expositor* generates traces that developers manipulate as lists with operations such as map and filter. Expositor also satisfies the *extensible* property by definition.

To build time-traveling object-centric breakpoints, we rely on a sub-part of the Sindarin API that is exposed by Seeker [8].

Time-traveling debuggers. All these debuggers can time-travel and are precise, *i.e.*, there is not one aspect of the execution that we cannot access or that we can miss. Live debuggers [8, 9, 31, 32] are fully deterministic, while post-mortem debuggers [23, 24, 25, 26, 27, 28, 29, 6, 30] record data and then navigate through records instead of replaying the execution – this property therefore makes no sense for these debuggers.

Some debuggers show object-centric capabilities in the form of object histories. The Flow debugger [33, 23] keeps track of changes in objects during the execution of programs. Developers can visualize the history of past states of objects and their behavior. The Trace Debugger [26, 27] records a trace from which one can navigate the history of objects and execute queries based on their protocol. Un-

stuck [30] also provides an explorable object history. Object histories satisfy the *object-centric* property but they only provide post-mortem perspectives. As such it differs from the live definition of object-centric debugging that we adopt in our work (Ressia [1]).

Seeker [8, 9, 10] exposes the Sindarin API [13] and therefore developers can write queries where they use this API to scope the results of said queries to specific objects. This only partially satisfies the *object-centric* property as, if in practice we could do it, there was nothing explicitly done until we mapped object-centric breakpoints to new object-centric queries in this paper.

Some debuggers are extensible. We already mentioned Expositor as a scriptable debugger. Seeker is extensible through an API to write new queries and integrate them in the debugging environment. FReD [31] and DeloreanJS [32] are partially extensible, as they allow developers to customize the conditions (*e.g.*, invariants) upon which breakpoints are simulated.

None of the studied time-traveling debuggers fully support the *explorable* property. As defined in Section 2, with this property we aim at easing and tooling ways to find objects to debug. Most debuggers have to query a trace [23, 24, 25, 26, 27, 6, 30] or an execution [8, 9, 31, 32] iteratively until developers can identify an object of interest. *Whyline* [28, 29] provides, by design, access to the objects for debugging (i.e., objects available in a graphical user interface) and allows developers to ask questions to explore the chain of events to which the object belongs. We cannot claim that our solution fully satisfies this property either. We provide a new way of finding and filtering objects systematically in the form of a new query that finds all object instantiations in the execution and allows us to navigate the execution back and forth throughout these instantiations. We combine this new way with the original method from object-centric debugging [1] (*i.e.*, finding objects manually by stepping in the debugger) and with querying executions similarly to all the other debuggers.

Seeker could already explore executions but without explicit support for identifying objects to debug. The Seeker back end maintains an unique object identifier (*oid*) for every object but that information is not exposed by the back end and therefore Seeker (and the TTQs) do not use it. In TTOCBs, we modified the Sindarin API to expose the object *oid* and added specific queries based on the *oid* information to help finding objects.

Time-Traveling Object-Centric Breakpoints. Because we combine and extend object-centric breakpoints [1] and Seeker [8, 9], and as described in Sections 3, 4 and 5, we satisfy all the other properties.

8. Conclusion

In this paper, we address the limitations of traditional object-centric breakpoints by introducing Time-Traveling Object-Centric Breakpoints (TTOCBs), based on Time-Traveling Queries (TTQs). This new approach mitigates issues like false positives and the challenge of identifying objects to debug in non-deterministic program executions. TTOCBs abstract away low-level concerns, saving developers from having to understand and use complex language reflection techniques for implementing and extending object-centric breakpoints. This makes TTOCBs independent of specific TTQs and debugger implementations, broadening their applicability.

The anecdotal user study we conducted across four example scenarios indicates that TTOCBs might reduce the number of actions required for debugging compared to traditional object-centric breakpoints. This suggests a promising improvement in debugging efficiency and effectiveness. Therefore, we plan to perform a controlled experiment to study the impact of TTOCBs on the debugging object-oriented programs.

Finally, we highlighted the object-centric breakpoints and time-traveling object-centric breakpoints tradeoffs in terms of usability, results, and performance.

References

[1] J. Ressia, A. Bergel, O. Nierstrasz, Object-centric debugging, in: Proceeding of the 34rd international conference on Software engineering, ICSE '12, IEEE Computer Society, 2012. doi:10.1109/ICSE.2012.6227167.

[2] C. Corrodi, Towards efficient object-centric debugging with declarative breakpoints, in: Post-proceedings of the 9th Seminar on Advanced Techniques and Tools for Software Evolution, SATToSE 2016, CEUR-WS.org, 2016.

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

[4] M. Beller, N. Spruit, D. Spinellis, A. Zaidman, On the dichotomy of debugging behavior among programmers, in: Proceedings of the: 40th International Conference on Software Engineering, ICSE '18, ACM, 2018. doi:10.1145/3180155.3180175.

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

[6] K. Y. Phang, J. S. Foster, M. Hicks, Expositor: Scriptable time-travel debugging with first-class traces, in: Proceedings of the 35th International Conference on Software Engineering, ICSE '13, IEEE Computer Society, 2013. doi:10.1109/ICSE.2013.6606581.

[7] E. T. Barr, M. Marron, Tardis: Affordable time-travel debugging in managed runtimes, in: Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA '14, ACM, 2014. doi:10.1145/2660193.2660209.

[8] M. Willembrinck, S. Costiou, A. Etien, S. Ducasse, Time-Traveling Debugging Queries: Faster Program Exploration, in: 21st IEEE International Conference on Software Quality, Reliability and Security, QRS '21, IEEE, 2021. doi:10.1109/QRS54544.2021.00074.

[9] M. Willembrinck, S. Costiou, A. Etien, S. Ducasse, Time-Traveling Queries for Faster Debugging and Program Comprehension, Journées Nationales du Génie de la Programmation et du Logiciel 2022, poster (2022).

[10] M. I. Willembrinck Santander, An interactive debugging approach based on time-traveling queries, Ph.D. thesis, Université de Lille (2023). doi:https://tel.archives-ouvertes.fr/tel-04398079.

[11] M. Willembrinck, S. Costiou, A. Vanègue, A. Etien, Towards Object-Centric Time-Traveling Debuggers, in: Proceedings of the International Workshop on Smalltalk Technologies, IWST '22, ACM Digital Libraries, 2022.
[12] S. Costiou, V. Aranega, M. Denker, Reflection as a tool to debug objects, in: Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering, SLE '22, ACM, 2022. doi:10.1145/3567512.3567517.
[13] T. Dupriez, G. Polito, S. Costiou, V. Aranega, S. Ducasse, Sindarin: A versatile scripting api for the pharo debugger, in: Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, DLS '19, ACM, 2019. doi:10.1145/3359619.
[14] K. A. Ericsson, Protocol analysis, A companion to cognitive science, Wiley Online Library (2017).
[15] S. Marr, ReBench: Execute and Document Benchmarks Reproducibly, Zenodo (2023). doi:10.5281/zenodo.8219119.
[16] J. Ressia, T. Gîrba, O. Nierstrasz, F. Perin, L. Renggli, Talents: an environment for dynamically composing units of reuse, Software: Practice and Experience, John Wiley & Sons, Ltd (2012). doi:10.1002/spe.2160.
[17] S. Costiou, M. Kerboeuf, G. Cavarle, A. Plantec, Lub: A pattern for fine grained behavior adaptation at runtime, Science of Computer Programming 161, Elsevier (2018). doi:10.1016/J.SCICO.2017.09.006.
[18] P. Tesone, S. Ducasse, G. Polito, L. Fabresse, N. Bouraqadi, A new modular implementation for stateful traits, Science of Computer Programming 195, Elsevier (2020). doi:10.1016/j.scico.2020.102470.
[19] M. Denker, Sub-method structural and behavioral reflection, PhD thesis, University of Bern (2008). doi:10.7892/BORIS.104452.
[20] J. Ressia, Object-centric reflection, Ph.D. thesis, Institut fur Informatik und angewandte Mathematik (2012). doi:10.7892/BORIS.104720.
[21] N. Papoulias, M. Denker, S. Ducasse, L. Fabresse, End-user abstractions for meta-control: Reifying the reflectogram, Science of Computer Programming 140, Elsevier (2017). doi:10.1016/j.scico.2016.12.002.
[22] S. Costiou, V. Aranega, M. Denker, Sub-method, partial behavioral reflection with reflectivity: Looking back on 10 years of use, The Art, Science, and Engineering of Programming Vol. 4, aosa, Inc. (2020). doi:10.22152/programming-journal.org/2020/4/5.
[23] A. Lienhard, T. Gîrba, O. Nierstrasz, Practical object-oriented back-in-time debugging, in: Proceedings of the 22nd European Conference on Object-Oriented Programming, ECOOP '08, Lecture Notes in Computer Science, Springer, 2008. doi:10.1007/978-3-540-70592-5_25.
[24] A. Lienhard, Dynamic object flow analysis, Phd thesis, University of Bern (2008). doi:10.7892/BORIS.104605.
[25] A. Lienhard, J. Fierz, O. Nierstrasz, Flow-centric, back-in-time debugging, in: Objects, Components, Models and Patterns, Proceedings of TOOLS Europe, Lecture Notes in Business Information Processing, Springer-Verlag, 2009. doi:10.1007/978-3-642-02571-6_16.
[26] C. Thiede, M. Taeumel, R. Hirschfeld, Object-centric time-travel debugging: Exploring traces of objects, in: Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming, Programming '23, Association for Computing Machinery, 2023. doi:10.1145/3594671.3594678.
[27] C. Thiede, M. Taeumel, R. Hirschfeld, Time-Awareness in Object Exploration Tools: Toward In Situ Omniscient Debugging, in: Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2023, Association for Computing Machinery, 2023. doi:10.1145/3622758.3622892.
[28] A. J. Ko, B. A. Myers, Designing the whyline: a debugging interface for asking questions about program behavior, in: Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, ACM Press, 2004. doi:10.1145/985692.985712.
[29] A. J. Ko, B. A. Myers, Debugging reinvented: Asking and answering why and why not questions about program behavior, in: In Proceedings of the 30th International Conference on Software Engineering, ICSE '08, ACM, 2008. doi:10.1145/1368088.1368130.
[30] C. Hofer, Implementing a backward-in-time debugger, Master's thesis, University of Bern (2006).
[31] K. Arya, T. Denniston, A. Rabkin, G. Cooperman, Transition watchpoints: Teaching old debuggers new tricks, aosa, inc., The Art, Science, and Engineering of Programming Vol. 1 (2017). doi:10.22152/programming-journal.org/2017/1/16.
[32] P. Leger, F. Ruiz, H. Fukuda, N. Cardozo, Benefits, challenges, and usability evaluation of deloreanjs: a back-in-time debugger for javascript, PeerJ Computer Science, PeerJ Inc. (2023). doi:10.7717/PEERJ-CS.1238.
[33] A. Lienhard, S. Ducasse, T. Gîrba, O. Nierstrasz, Capturing how objects flow at runtime, in: Proceedings International Workshop on Program Comprehension through Dynamic Analysis, PCODA '06, 2006.

Appendix A. Comparison of debugging procedures between Pharo object-centric and time-traveling object-centric breakpoints

Tables A1 to A4 show comparisons of procedures to answer program comprehension questions from an empirical study [8]. We selected these program comprehension questions specifically because to be answered they required participants to take an object-centric perspective. During that experiment, participants did use a preliminary version of our time-traveling object-centric breakpoints.

To highlight practical differences in the use of Pharo object-centric breakpoints (OCB) and our time-traveling object-centric breakpoints (TTOCB), we report the procedures to answer these questions with both tools. For each question, we provide a table in which the first column shows the OCB procedure and the second column shows the TTOCB procedure.

The developer (one of the authors) who performed the procedures knows how to use both tools. The procedure is stopped when the developers find the required answer, *i.e.*, when the developers observe the required information to answer the question. These precise elements are provided with the expected answer in [8].

Appendix B. Unit test used for the performance evaluation

```
1 TestClass >> #testStudentPrinting
2   | group |
3   group := AMGroup new.
4   self students do: [ :s | | str |
5     str := WriteStream on: String new.
6     group textPrintStudent: s on: str.
7     self assert: (#(+ −) includes: str contents last)]
```

Listing 3: A unit test of the Ammolite application.

Question 1

Procedure with OCB	**Procedure with TTOCB**
Debug the following test with Pharo: `RSNormalizerTest>>#testBasic`	Debug the following test with Seeker: `RSNormalizerTest>>#testBasic`
Do 1 *step-over* [**Action 1**], then 3 *step-through* [**Actions 2-4**] then 1 *step-over* [**Action 5**] to execute the method `#new` on `RSBox` in the block closure	Select the class `RSBox` [**Action 1**], right-click and execute the query `All Instances Creation of class named as selection` [**Action 2**]
Select `thisContext` [**Action 6**] in the debugger inspector and select the `RSBox` instance [**Action 7**] at the 3rd position in the stack	Seeker shows all created instances in the execution, in a table. Time-travel [**Action 3**] to the first instance creation
In the inspector, select the object class (`RShape`) [**Action 8**], right-click on the `color:` method and set an object-centric breakpoint on that method [**Breakpoint 1**] .	Do 1 *step-over* [**Action 4**] and 1 *step-into* [**Action 5**] to reach the `RSBox` instance, and execute the query: `All messages sent to self` [**Breakpoint 1**] (`self` in this context represents the `RSBox` instance.)
Click *Proceed* [**Action 9**]. The breakpoint hits. We observe [**Action 10**] that `#color:` is called within the method `RSNormalizer>>#normalize` that is called in the test, with the color green as an argument.	Seeker shows all messages sent to the `RSBox` instance, in a table. We observe [**Action 7**] that `#color:` is called only once. If we time-travel [**Action 8**] to the corresponding step, we observe [**Action 9**] that the method is called from the test within the method `RSNormalizer>>#normalize`, with the color green as an argument.

Table A.6: Procedure to answer the program comprehension question *"When (=from which methods) is the first object in the `shapes` collection receiving the #color: message? What are the values of the arguments in each message?"*. The answer provided by [8] is: *The `color:` message is called only once on the first object of the `shapes` collection, within the method `RSNormalizer>>#normalize` that is called from the test, with the color green as an argument.*

Question 2

Procedure with OCB	**Procedure with TTOCB**
Debug the following test with Pharo: `RSAttachPointTest>>#testVerticalAttachPoint`	Debug the following test with Seeker: `RSAttachPointTest>>#testVerticalAttachPoint`
Do 4 *step-over* [**Actions 1-4**] to do the assignment `b1 := RSBox new size: 20`	Do 4 *step-over* [**Actions 1-4**] to do the assignment `b1 := RSBox new size: 20` to get the object of interest `b1`
Inspect the object of interest `b1` [**Action 5**], do not select any instance variable and set an object-centric breakpoint [**Breakpoint 1**] on writings of all instance variables	Select the variable `b1`[**Action 5**], right-click and execute the query `All the assignments of instance variables, of the object currently pointed by the selected variable` [**Breakpoint 1**]
Click *Proceed* [**Action 6**] and a breakpoint is hit. We observe [**Action 7**] that an instance variables was modified. We repeat these actions (*Proceed* and observe) until the execution ends, which happens 3 more times [**Actions 8-9, 10-11, 12-13**]. The following variables, in this order, were modified: `encompassingRectangle`, `connectedLines`, `encompassingRectangle`, `parent`	Seeker shows all the assignments of all instance variables of the object `b1`, in a table. We observe [**Action 6**] that the instance variables are modified in the following order: `paint`, `isFixed`, `matrix`, `shouldUpdateLines`, `baseRectangle`, `baseRectangle`, `encompassingRectangle`, `path`, `baseRectangle`, `encompassingRectangle`, `path`, `encompassingRectangle`, `connectedLines`, `encompassingRectangle`, `parent`

Table A.7: Procedure to answer the program comprehension question *"What instance variables of `RSBox b1` are modified during this test?"* The answer provided by [8] is: *The instance variables of `RSBox b1` that are modified during this test are `encompassingRectangle` (twice), `connectedLines`, `parent`, `entryIndex`, and `isDirty`.*

Question 3

Procedure with OCB	**Procedure with TTOCB**
Debug the following test with Pharo: `ContextTest>>#testSteppingReturnSelfMethod`	Debug the following test with Seeker: `ContextTest>>#testSteppingReturnSelfMethod`
Do 9 *step-over* [**Actions 1-9**] to do the assignment `newContext := aMethodContext step`	Select the variable `newContext` [**Action 1**] and execute the query `All the assignments of variable with selected name` [**Breakpoint 1**]
Inspect the object of interest `newContext` [**Action 10**], select the instance variable `pc` [**Action 11**], and set an object-centric breakpoint on writings of this instance variable [**Breakpoint 1**] (via right-click on `pc`)	Seeker shows all assignments of the variable `newContext`. Time-travel [**Action 2**] to the first assignment and do 1 *step-over* [**Action 3**] to get the object of interest `newContext`
Click *Proceed* [**Action 12**]. We observe [**Action 13**] that `pc` is assigned 29	Select the variable `newContext` [**Action 4**], right-click and execute the query `All assignments of instance variables of the object currently pointed by the selected variable` [**Breakpoint 2**]
Click *Proceed* [**Action 14**]. We observe [**Action 15**] that `pc` is assigned `nil`	Seeker shows all assignments of all instance variables of the object `newContext`, in a table. To filter the results to get only the assignments of the instance variable `pc`, time-travel [**Action 5**] to an assignment of the instance variable `pc`, select the variable `pc` [**Action 6**], right-click and execute the query `All the assignments of a selected instance variable, of self` [**Breakpoint 3**]. We observe [**Action 7**] that the instance variable `pc` is assigned 4 times, the values: 25, 28, 29, `nil`.
Click *Proceed* [**Action 16**], the test ends its execution.	

Table A.8: Procedure to answer the program comprehension question *"What are the different values of the `pc` instance variable of the `newContext` object during this test?"* The answer provided by [8] is: *The values of the `pc` instance variables of the `newContext` object during this test are 29 and `nil`*

Question 4

Procedure with OCB	**Procedure with TTOCB**
Debug the following test with Pharo: `GeneratorTest>>#testAtEnd`	Debug the following test with Seeker: `GeneratorTest>>#testAtEnd`
Do 2 *step-over* [**Actions 1-2**] to do the assignment `generator := self numbersBetween: 1 and: 3`	Do 2 *step-over* [**Action 1-2**] to do the assignment `generator := self numbersBetween: 1 and: 3` to get the object of interest `generator`
Inspect the object of interest `generator` [**Action 3**], select the object class `Generator` [**Action 4**] and right-click on the method `#atEnd` and set an object-centric breakpoint on that method [**Breakpoint 1**]	Select the variable `generator` [**Action 3**], right-click and execute the query `All messages sent to the selected object` [**Breakpoint 1**]
Click *Proceed* 7 times [**Action 5-11**] until the test ends its execution. 7 breakpoints are hit. We do 7 observations (one at each breakpoint) [**Action 12-18**] of the method `#atEnd` being called: 4 times from the method `GeneratorTest>>#testAtEnd` and 3 times from the method `Generator>>#next`	Seeker shows all the messages sent to the object `generator`, in a table. We observe [**Action 4**] that the object `generator` receives the `atEnd` message 7 times: 4 times from the method `GeneratorTest>>#testAtEnd`, 3 times from the method `Generator>>#next`

Table A.9: Procedure to answer the program comprehension question *"How many times is `generator>>#atEnd` called on the `generator` object and from which methods?"* The answer provided by [8] is: *The method `Generator>>#atEnd` is called 7 times: 4 times from the method `GeneratorTest>>#testAtEnd` and 3 times from the method `Generator>>#next`.*