Object-Centric Time-Travel Debugging
Exploring Traces of Objects

Christoph Thiede
christoph.thiede@student.hpi.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Marcel Taeumel
marcel.taeumel@hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Robert Hirschfeld
robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

ABSTRACT
Traditional behavior-centric debuggers are organized around an extensive call stack, making it hard for programmers to navigate and explore large programs. We present object traces, a novel, object-centric approach to time-travel debugging that enables programmers to directly interact with recorded states of objects and explore their evolution in a simplified call tree. Our approach allows programmers to send messages to the object trace to ask questions of different granularity, from single variable values to custom representations of object graphs. We demonstrate practicability by applying it to the TraceDebugger, a time-travel debugger for Squeak/Smalltalk. We examine the practical opportunities and limitations of object traces and suggest directions for future work.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging. Integrated and visual development environments.

KEYWORDS
object-oriented debugging, time-travel debugging, back-in-time debugging, omniscient debugging, query-based debugging, declarative debugging, program tracing, program comprehension, program exploration, exploratory programming, moldable development, Smalltalk

ACM Reference Format:

1 INTRODUCTION
Debuggers are an important tool in the toolbox of many programmers. They do not only facilitate the namesake activity of fault isolation but are also used by programmers to explore object-oriented software, understand its design and implementation by example, or reason about possible changes and extensions in context. In the last two decades, time-travel debuggers have attracted more attention since they give programmers the freedom to explore a program independently of its original execution order.

While time-travel debuggers facilitate exploring and navigating through a program, these activities remain challenging. Programmers are overwhelmed by complex object choreographies and message flows. For many tasks, programmers are interested in the state and behavior of particular objects. However, traditional debuggers and recent implementations of time-travel debuggers are behavior-centric and mainly provide means for navigation through the hierarchy of method activations, making it hard for programmers to locate and survey the relevant subset of a program trace.

To simplify that activity, we propose object traces which provide a novel object-centric perspective for debugging. Using object traces, programmers are enabled to explore programs through the evolution of specific objects. We further demonstrate the practicability of object traces by applying them to the TraceDebugger, a time-travel debugger for the interactive programming system Squeak/Smalltalk [12, 15, 32]. In this work, we prioritize the demonstration of the new concept over optimizations such as memory efficiency or handling large traces; still, our prototype is fast enough to be used interactively for small to medium-sized programs.

In the remaining sections, we provide some background on our notion of debugging object-oriented systems and related approaches (section 2), describe the concept and implementation of our solution (section 3), discuss the practical opportunities and limitations of object traces (section 4), and conclude with some possible directions for future work (section 5).

2 BACKGROUND
The essence of object-oriented programming is objects and messages. Objects have three defining properties: behavior, state, and identity. Behavior is described by messages that are sent from one object to another and is implemented by methods that process messages. State is described by variables of an object which point to further objects and can change as the side effect of a method execution. Identity is the unique characteristic of an object that distinguishes it from all other objects within the system. For instance, a morph is a graphical object in Squeak whose state describes its geometry, color, and composition, and whose behavior describes its ability to get or change its composition, render itself to the screen, or react to user events. An example is a WatchMorph (for displaying the current time) that responds to the message initialize by constructing itself with a composition of twelve StringMorphs for the labels of the clock. So, we can represent the simple domain of clocks as objects.

1https://github.com/hpi-swa-lab/squeak-tracedebugger
Programmers can debug sending a message to an object to explore the resulting behavior and the caused side effects. For example, if we wonder why the labels of our WatchMorph are constructed in the wrong direction, or if we wish to change the color of the labels, we can debug sending initialize to the morph. However, identifying the methods that are relevant to a particular task or question is often not straightforward, as behavior can be complex and a single message send can result in thousands or millions of other message sends. Traditional debuggers address this complexity by allowing the user to execute a program step-by-step and displaying a context stack of the currently active methods. Time-travel debuggers, also referred to as back-in-time debuggers or omniscient debuggers, enhance this workflow: by recording a program trace that consists of all previous message sends and, optionally, all prior object states, they allow programmers to discretionarily navigate through the context tree, i.e., the recorded hierarchy of method activations, and observe the respective historical states [14, 19, 27].

Still, existing time-travel debuggers are behavior-centric, making it hard for programmers to follow specific objects and observe the changes that have been made to their (composed) state as a result of different message sends. If we wish to identify the methods that are relevant to constructing the labels on our WatchMorph by using a traditional or time-travel debugger, our best chance is to continually try, fail, and repeat: step into or over single message sends during the initialization of the morph depending on whether they seem relevant, observe the current state of the morph until a relevant change has occurred, and finally restart or revert the program execution to descend into the last skipped message send.

Object-centric breakpoints attempt to address this problem by providing programmers with a set of commands for advancing the execution of a program until a selected portion of state in particular objects changes [5, 29]. On the contrary, the WHYLINE approach looks back into the past of a program to explain the origin of selected objects or states by using a combination of program slicing and tracing [16]. Scriptable debugging or declarative debugging addresses the same problem on a more holistic level by defining a query language for retrieving various events from a program trace [8, 11, 13, 17, 23, 25, 26]. In addition to method activations, returns, or read accesses to state, some event types cover side effects such as assignments to variables. Some approaches visualize query results using interactive object diagrams or sequence diagrams to show the connection between events and their location in the program [18]. Still, queries are expressed from an indirect metaperspective and can only refer to atomic states (e.g., the size of the array in the submorphs variable of our WatchMorph) instead of communicating directly with the objects in question (e.g., by sending our WatchMorph the message numberOfSubmorphs).

Some other approaches to behavior-centric program exploration trace the flow of objects [20], the side effects between them [21], or their intercommunication [7, 30], and visualize the results in an interactive graph. It is furthermore possible to take a source-code-centric perspective by employing manual or automatic logging techniques to trace all evaluations of a selected expression in the program [6, 28]; however, this approach places the burden of identifying relevant locations in the source code on programmers.

We try to tackle the limitations of existing object-centric debugging tools by proposing object traces through that programmers interact directly with the history of relevant objects to explore a program from the perspective of these objects.

### 3 SOLUTION

In the following, we introduce object traces and sketch an implementation of them for the TRACEDEBUGGER.

#### 3.1 Approach

An object trace is a dynamic view on an object in the course of program execution over time (fig. 1). Object traces are defined by the following four properties:

- **object**: An object trace refers to an object that was involved in the executed program and possibly modified by it, and whose prior states have been recorded in the program trace.
- **view**: An object trace is based on a query for the object. The query is an expression that communicates with the object by sending it messages.
- **time**: A view’s query divides the program trace into one or multiple time ranges for each of which the expression evaluates to a different value. The resulting time ranges and values are arranged in a filtered version of the context tree that contains only the method activations that caused a change in the query result.
- **dynamic structure**: By revising the query, the level of detail of the object trace can be adjusted.

Like any other object in the live system, an object trace is a tangible entity that programmers can interact with directly: they can ask questions about an object from a traced program by sending it messages to explore its state and evolution over time.

By using the full protocol of messages that an object provides, programmers can ask questions of varying granularity, ranging

---

2The term “declarative debugging” is overloaded. In this paper, we refer to the concept of declarative queries on program traces but not to semi-automated bug detection which employs a human oracle for declaring correct program behavior.
variables they want to see. objects.

Tracing. To create the program trace, we record both the program’s behavior and its state during the execution. For the behavior, we log all method activations in a context tree together with their child contexts and lifespans, which we represent by pairs of time indices that refer to a global instruction counter (fig. 3).

For the state, we maintain an incremental historic memory structure that contains a sparse array of former values for each changed slot of any object (fig. 4). The sparse arrays are time-indexed using the global instruction counter. Whenever an object slot is assigned a new value, we append the previous value to the corresponding sparse array with the current time index. Each array contains only such displaced values since the current values can be retrieved from the present object space. Given a typically large object space with a small number of changes, incremental snapshots scale better than full snapshots, usually fit into the main memory (section 4), and allow for efficient random access to particular states at different points in time.

Accessing historic state. To evaluate an expression for a historic version of an object at a single point in time (point retracing), we

Figure 2 shows some screenshots of the history explorer, our prototypical UI for exploring object traces within the TRACEDEBucker. In the summary mode, programmers can get an overview of all the different method activations and values in a hierarchical list. In the details mode, they can select an entry from the tree and inspect the associated value in a custom representation. By asking different questions about an object, programmers can explore its creation or evolution and identify relevant methods for particular state changes. For example, we can change the query in the object trace on our WatchMorph to retrieve different information about its composition, geometry, or appearance.

3.2 Implementation

In the remaining section, we describe our implementation of object traces for the TRACEDEBucker in Squeak/Smalltalk.

Figure 3: A context tree for tracing the behavior of sending initialize to a new Morph instance. Each context is associated with a time interval describing its lifespan.
instrument the execution of the expression and forward read accesses to any state to the historic memory. If the requested state is not present in the historic memory for the requested time index, we fall back to the present object space. We further run the expression inside a sandbox to isolate any side effects (e.g., a cache update caused by a message send to an object) from the present object space by managing them in a separate memory structure.

Evaluating range queries. Naively, we could evaluate a range query by point-retracing it separately for each time index in the requested range and combining the results. However, to ensure a practical performance, we instead evaluate the expression once only for the entire time range but instrument its execution with vectorization (range retracing).

For each read access to state, we pass back a sparse vector to the query that contains all different values and time ranges of the requested state from the historic memory. We extend all operators of the query (e.g., the arithmetic addition) with optional vector semantics. If the control flow diverges according to the values in a vector (e.g., at a conditional branch), we fork the execution into independent threads for subintervals of the original time range. Thus, range retracing compares to the hardware concept of SIMD semantics where multiple data are handled by a single processor instruction [10]. It also compares to a form of online symbolic execution [1, 2] where the symbolic state has time-based semantics.⁴

Instrumented execution in Squeak/SmallTalk. In Squeak, we customize the code simulator (an interpreter that resides in the object space) using SIMULATIONSTUDIO⁵ to instrument the execution of programs for tracing and queries for retracing. We override the bytecode instructions and primitives relevant to activating methods, reading or writing state, or working with possibly vectorized state.

⁴In fact, we believe that by applying concolic execution or veritesting (two optimized styles of symbolic execution), the performance of range retracing could be improved further.

⁵https://github.com/LinQLower/SimulationStudio

4 DISCUSSION

Experience report. In the past few months, we have successfully used our prototypical implementation of object traces to explore several components of Squeak. For example, we have discovered possible extension points for a new feature in the regular expression parser (appendix A), pinpointed the origin of a rendering bug in the Morphic UI system, and created an animation of a Morphic layout computation that served as an artifact for refining and discussing our understanding of the layout engine.

We do not view object-centric debugging as a replacement but as a complement to behavior-centric debugging. While we see great potential in object traces for surveying large program traces on a high level and for aiding navigation, traditional, source-centric views remain a key component of fine-grained program exploration. Thus, we tightly integrated our prototype with the existing behavior-driven interfaces of the TraceDebugger and the default forward debugger in Squeak to combine the best of all worlds (fig. 5).

Object traces entail an alternative navigation workflow for debugging: instead of performing a non-specific search (i.e., “browsing”) in the context tree, programmers select a specific object and express a query about it. While this allows for a more targeted and efficient search, it requires programmers to make a greater initial cognitive investment, which can reduce the experience of immediacy. The benefit of object traces depends on the size and complexity of the program trace and on the design of the system under exploration. Suitable systems have an intuitive and concise state model, or they offer tools for exploration (e.g., meaningful string representations or convenience accessors⁶). However, systems with a very simple behavioral model and primarily functional architectures with few side effects may be easier to explore with other types of debuggers.

⁶For example, each morph in Squeak contributes a screenshot field to the general-purpose inspector tool (fig. 5), removing the need from programmers to manually find the correct message for rendering the morph.

Figure 4: Example of the incremental historic memory structure for tracing the state of sending initialize to a new Morph instance. The dark boxes represent associative arrays of pointers. For each involved object slot, all former displaced states are preserved in a sparse array. For example, the morph’s submorphs variable was assigned the empty collection #() during the time interval [13, 80].

Figure 5: Integration of the history explorer into the behavior-centric interface of the TraceDebugger. Programmers can select an object from the program trace, choose a predefined query, and explore its evolution in a history explorer. From the history explorer, they can switch back to the behavior-centric view on a selected context.
Performance. Our current prototype is a proof of concept and, like the entire TraceDebugger project, favors a simple and exploratory architecture over maximum performance. Compared to regular execution in the virtual machine (VM), tracing introduces a runtime overhead between 100 000 % and 1 000 000 % and retracing introduces an overhead between 2000 % and 100 000 %. Nevertheless, the history explorer offers practical performance and interactive response times [31, p. 473] of less than 1 second for typical small-sized workloads and less than 5 seconds for typical medium-sized workloads (table 1). To enable interactive exploration of data- or compute-intensive programs in the future, we see great potential in replacing our current implementation strategy of code simulation with a program-instrumentation-based approach [9, 14, 24].

5 CONCLUSION

We have proposed object traces as a novel tool for object-centric debugging which allows programmers to explore program traces based on specific object changes. We believe that object traces offer a promising new perspective for understanding the creation or evolution of objects and object-based compositions and for navigating through large programs. We have shown that an implementation of our concept is feasible and provides practical performance even in our unoptimized prototype. The concept is not limited to the Squeak/Smalltalk environment but can be implemented in any environment that offers a program tracer and a symbolic execution engine with customizable memory access. For instance, programmers could use object traces to debug user interactions with a JavaScript or compute-intensive programs in the future, we see great potential in replacing our current implementation strategy of code simulation with a program-instrumentation-based approach [9, 14, 24].

A CASE STUDY: EXTENDING A REGULAR EXPRESSION PARSER

Here, we describe another real-world use case of object traces to identify a possible extension point for a new feature in an existing software system. At the time of writing, the regular expression engine of Squeak (Regex) does not support atomic groups such as ab(?>cd|c)de which restrict computing-intensive backtracking. To implement this capability in the engine, we first need to add support for the new syntactic element to the parser. Provided that we are not yet familiar with the parser’s implementation details, a traditional dynamic approach to this problem is to debug the parser with the new regular expression ‘ab(?>cd|c)de’ asRegex and to find out where the execution starts behaving “wrong”, i.e., to identify the first method in the context tree whose behavior should be changed to respect the extended syntax. Unfortunately, running the parser for our regular expression involves more than 400 method activations with more than 800 source lines of code in the Regex package. To avoid this complexity for our task, we instead take an object-centric perspective on the parsing process (fig. 6).

In the TraceDebugger, we explore the execution of the program (fig. 6a) and find that the RxParser, near the root of the context tree below the message parse, indirectly accesses the source string through an intermediate ReadStream instance (an iterator object for a collection). We inspect this stream object and learn that it holds the current reading state in a position variable (fig. 6b). Given this information, we can construct an object trace for the position of the stream by exploring the history of the field from the inspector. The resulting object trace breaks down the original context tree

Table 1: Time and memory consumption for recording (tracing) and exploring (retracing) program traces for different workloads.

<table>
<thead>
<tr>
<th>Domain</th>
<th>Program</th>
<th>Tracing</th>
<th>Retracing</th>
<th>Query</th>
<th>Retracing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular expression parser</td>
<td>‘\w+’ asRegex</td>
<td>0.174</td>
<td>1802</td>
<td>collection first: position - 2</td>
<td>0.253 374</td>
</tr>
<tr>
<td></td>
<td>WatchMorph basicNew initialize</td>
<td>0.797</td>
<td>15 299</td>
<td>evaluate for ReadStream</td>
<td>4.558 523 804</td>
</tr>
<tr>
<td></td>
<td>aSystemBrowserWindow imageForm</td>
<td>8.905</td>
<td>2 567 832</td>
<td>self copy: self relativeRectangle</td>
<td>153.857 6 307 677</td>
</tr>
</tbody>
</table>

*a Test machine: Intel i7-8550U CPU @ 1.80 GHz. Environment: Open Smalltalk Cog/Spur VM of version 202206021410.
ones (fig. 6d). Thus, by identifying a meaningful object state in groups, and we can finally place our new check next to the existing handles other special cases such as lookarounds or non-capturing By browsing this method, we find out that the method already (?>cd|c) a special instruction. We can also see that the parsing of the whole the parser interprets the of the input string for each step of the parser (fig. 6c).

the query of the object trace to request the already-consumed prefix position, i.e., consume the next character from the input string. To improve the visual intuition of each historic stream state, we change into a reduced version with only 12 leaf contexts that increment the program execution, we were able to more efficiently search the the program trace that relates to our intended perspective on the program trace, we were able to more efficiently search the the program trace that relates to our intended perspective on the program trace, we were able to more efficiently search the

*We add a position offset of -1 because the RxParser maintains an internal lookahead character.

Figure 6: Exploring Squeak’s regular expression parser using the TRACEDEBUGGER and object traces to find a possible extension point for a new syntactic element. The recorded program is ‘ab(?>cd|c)de’ asRegex.

REFERENCES


