# 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:**
Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. 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 Companion), March 13–17, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3594671.3594678

## 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[1], 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.

[1] https://github.com/hpi-swa-lab/squeak-tracedebugger



<Programming>'23 Companion, March 13–17, 2023, Tokyo, Japan

ACM ISBN 979-8-4007-0755-1/23/03.
https://doi.org/10.1145/3594671.3594678

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*[2] 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.

[2] The 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.

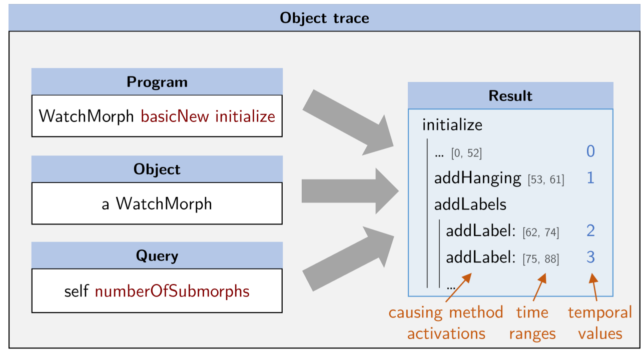



**Figure 1: An object trace for the construction of the WatchMorph. After tracing the program WatchMorph basicNew initialize, the programmer has expressed the query self numberOfSubmorphs for the WatchMorph instance. The output is a hierarchical list of different time-bounded results for the query, each of which is associated with the method activation that caused the change in the composition of the morph.**

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.

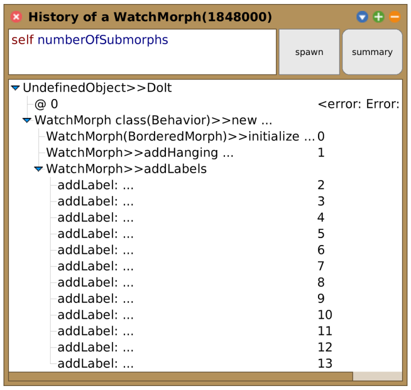
### 3.1 Approach

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:

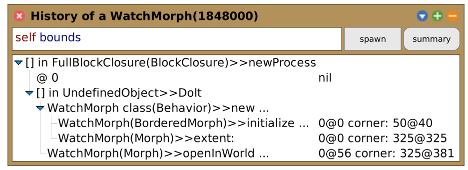
- **object:** An object trace refers to an object that was involved in the executed program and possibly modified by it, and whose prior states have been recorded in the program trace.
- **view:** An object trace is based on a *query* for the object. The query is an expression that communicates with 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.

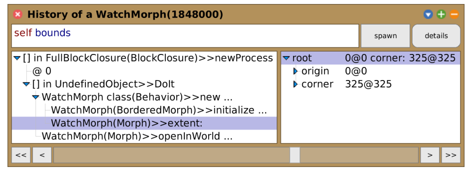
By using the full protocol of messages that an object provides, programmers can ask questions of varying granularity, ranging



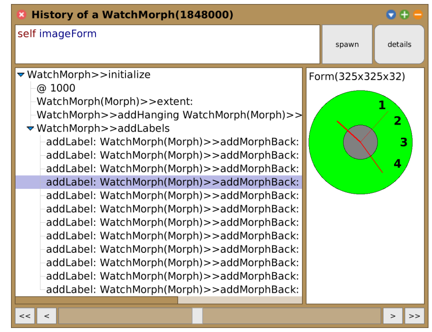

**(a) Viewing the entire history for the number of submorphs.**




**(b) Viewing the morph's geometry after opening it.**




**(c) Inspecting the morph's geometry after opening it, evolved after the third change.**




**(d) Inspecting a rendered screenshot of the morph after constructing the fourth label. All render errors (due to uninitialized variables) have been excluded through a filter from the context menu of the tree.**

**Figure 2: Exploring different object traces for the construction of the WatchMorph in the history explorer. In the upper part of the window, users can enter a query against the morph (self) and edit it to update the object trace. In the *summary mode* (a and b), all results are displayed in a compact hierarchical list. In the *details mode* (c and d), users can select a method activation from the tree and inspect the associated value using a custom representation.**

from simple accessor messages for a single variable value to complex messages that retrieve aggregated or composed information from an object. Queries can also send messages that create *custom representations* of the object such as a printString (a textual representation), a list of properties, or custom visualizations. Thereby, object traces constitute a *moldable tool* [3, 4] whose utility is substantially improved through the offer of domain-specific tools for the object being explored. Queries describe the amount of detail in the resulting object trace: by extending or reducing the expression, programmers can control the level of granularity of the resulting context tree. For example, they can decide the changes of which variables they want to see.

Figure 2 shows some screenshots of the *history explorer*, our prototypical UI for exploring object traces within the TraceDebugger. 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 TraceDebugger in Squeak/Smalltalk.

*Tracing.* To create the program trace, we record both the program's behavior and its state during the execution.[3] 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

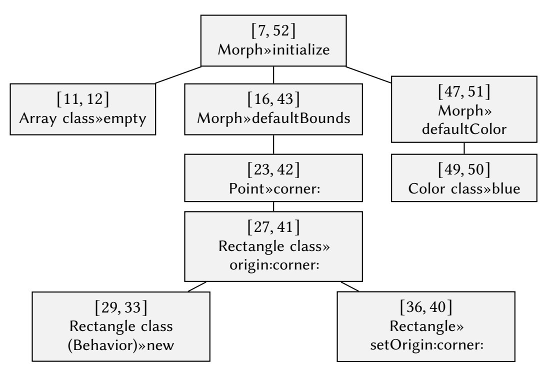



**Figure 3: A context tree for tracing the behavior of sending initialize to a new Morph instance. Each context is associated with a time interval describing its lifespan.**

[3]In our metamodel of programs, information about the current execution such as the program counter or the variable stack is treated as state of the respective context objects.

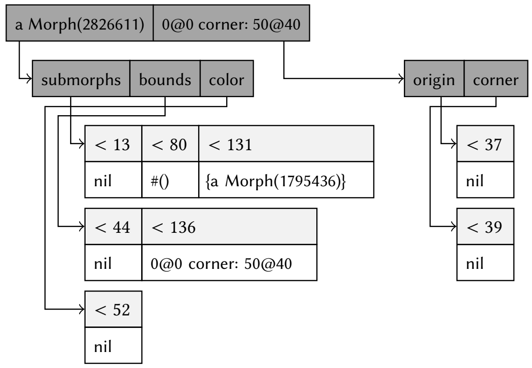



**Figure 4: Example of the incremental historic memory structure for tracing the state of sending initialize to a new Morph instance. The dark boxes represent associative arrays of pointers. For each involved object slot, all former displaced states are preserved in a sparse array. For example, the morph's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

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 SimulationStudio[5] 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.

[4]In 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.
[5]https://github.com/LinqLover/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 accessors[6]). 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.

[6]For 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.

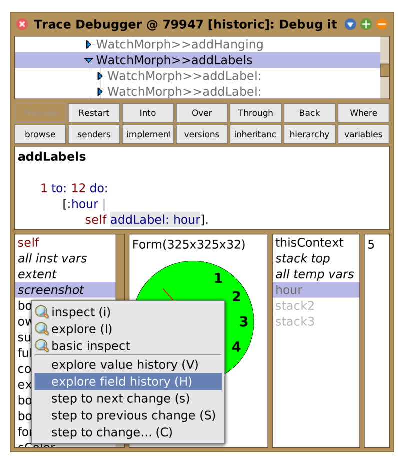



**Figure 5: Integration of the history explorer into the behavior-centric interface of the TraceDebugger. Programmers can select an object from the program trace, choose a predefined query, and explore its evolution in a history explorer. From the history explorer, they can switch back to the behavior-centric view on a selected context.**

*Performance.* Our current prototype is a proof of concept and, like the entire TraceDebugger project, favors a simple and explorable 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].[7]

## 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 application by asking for the session state or for screenshots of a graphical user interface [22].

Two limitations of object traces are that programmers need to ask the right questions which can be a cognitive overhead but helps to efficiently navigate large program traces, and that the system being explored should model changes through side effects but not through a functional programming style with copies of immutable objects.

In the future, we plan to improve the performance of our prototype by employing efficient program instrumentation and possibly concolic execution for query evaluation. Qualitatively, we envision a filtering mechanism to reduce the complexity of large object traces, and we are considering designs to reduce the semantic distance [33] between state-centric and behavior-centric views on the same program trace. Finally, we look forward to exploring ways that enhance the interactive exploration of object traces. Programmers should be able to express queries and navigate results through domain-specific representations to save debugging time and improve the overall quality of the program.

[7]Hypothetically, for a maximally efficient implementation of program tracing and retracing within the VM, we could also trade in our own programming experience.

## ACKNOWLEDGMENTS

We thank Tom Beckmann and Stephan Lutz for their feedback on earlier versions of the TraceDebugger. We are grateful to Eliot Miranda for suggesting the use of code simulation for program instrumentation and for his support in working with the code simulator. We sincerely thank the anonymous reviewers and the workshop participants for their extensive and valuable feedback. Finally, we thankfully acknowledge the financial support of the HPI Research School on Service-oriented Systems Engineering.[8]

## 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).

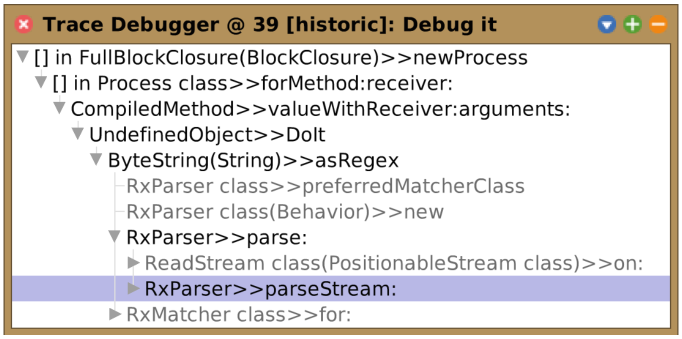
In the TraceDebugger, we explore the execution of the program (fig. 6a) and find that the RxParser, near the root of the context tree below the message parse:, indirectly accesses the source string through an intermediate ReadStream instance (an iterator object for a collection). We inspect this stream 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

[8]https://www.hpi.de/en/research/research-schools

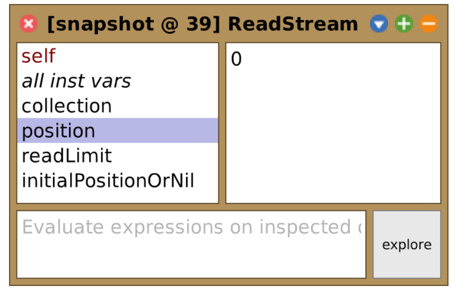
**Table 1: Time and memory consumption for recording (tracing) and exploring (retracing) program traces for different workloads.**

| Domain | Tracing: Program | Tracing: Time [s][a] | Tracing: Memory [kB] | Retracing: Query | Retracing: Time [s][a] | Retracing: Memory [kB] |
|---|---|---|---|---|---|---|
| Regular expression parser | '\w+' asRegex | 0.174 | 1802 | collection first: position - 2 "evaluate for ReadStream" | 0.253 | 374 |
| UI widget construction (13 elements) | WatchMorph basicNew initialize | 0.797 | 15 299 | self imageForm | 4.558 | 523 804 |
| UI rendering (89 elements, 650 px × 425 px) | aSystemBrowserWindow imageForm | 8.905 | 2 567 832 | self copy: self relativeRectangle | 153.857 | 6 307 677 |

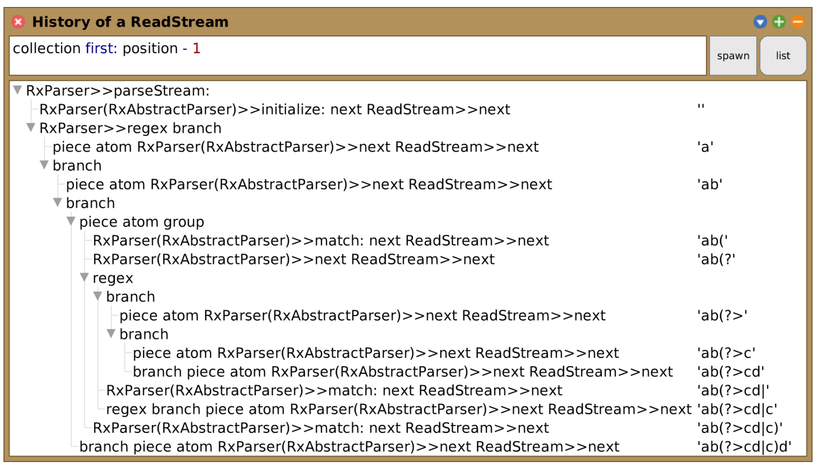
[a] Test machine: Intel i7-8550U CPU @ 1.80 GHz. Environment: Open Smalltalk Cog/Spur VM of version 202206021410.



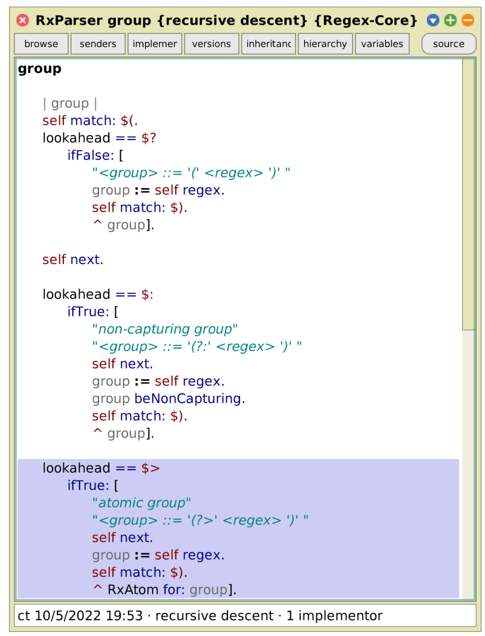

**(a) Exploring the top of the recorded context tree in a TraceDebugger.**




**(b) Inspecting a snapshot of the ReadStream instance taken from the program trace.**




**(c) Exploring the history of the ReadStream instance using an object trace. The query asks for the prefix of the input stream that has already been consumed.[a] All computation errors (due to uninitialized variables) have been excluded through a filter.**




**(d) Adding support for atomic groups in the implementation of RxParser»group.**

[a]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.**

into a reduced version with only 12 leaf contexts that increment the position, i.e., consume the next character from the input string. To improve the visual intuition of each historic stream state, we change the query of the object trace to request the already-consumed prefix of the input string for each step of the parser (fig. 6c).

From the final object trace, we can see that the current version of the parser interprets the > as a normal character (atom) instead of as a special instruction. We can also see that the parsing of the whole construct (?>cd|c) takes place inside and below the method group. By browsing this method, we find out that the method already handles other special cases such as lookarounds or non-capturing groups, and we can finally place our new check next to the existing ones (fig. 6d). Thus, by identifying a meaningful object state in the program trace that relates to our intended perspective on the program execution, we were able to more efficiently search the context tree for the relevant methods.

## REFERENCES


[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *Comput. Surveys* 51, 3, Article 50 (2018), 39 pages. https://doi.org/10.1145/3182657

[2] Cristian Cadar and Koushik Sen. 2013. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM* 56, 2, 82–90. https://doi.org/10.1145/2408776.2408795

[3] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. 2015. The Moldable Inspector. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (Pittsburgh, PA, USA) *(Onward! 2015)*. Association for Computing Machinery, New York, NY, USA, 44–60. https://doi.org/10.1145/2814228.2814234

[4] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. 2014. The Moldable Debugger: A Framework for Developing Domain-specific Debuggers. In *Software Language Engineering*, Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju (Eds.). Springer International Publishing, 102–121. https://doi.org/10.1007/978-3-319-11245-9_6

[5] Claudio Corrodi. 2016. Towards Efficient Object-centric Debugging With Declarative Breakpoints. In *Post-proceedings of the 9th Seminar on Advanced Techniques and Tools for Software Evolution, Bergen, Norway, July 11-13, 2016 (CEUR Workshop Proceedings, Vol. 1791)*, Mircea Lungu, Anya Helene Bagge, and Haidar Osman (Eds.). CEUR-WS.org, 32–39. http://ceur-ws.org/Vol-1791/paper-04.pdf

[6] Steven Costiou, Clotilde Toullec, Mickaël Kerboeuf, and Alain Plantec. 2018. Back-in-Time Inspectors: An Implementation With Collectors. In *International Workshop on Smalltalk Technologies*, Vol. 10. Cagliari, Italy. https://hal.univ-brest.fr/hal-02320434

[7] Ward Cunningham and Kent Beck. 1986. A Diagram for Object-oriented Programs, In Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications. *ACM Sigplan Notices* 21, 11, 361–367. https://doi.org/10.1145/28697.28734

[8] Jeffrey K. Czyz and Bharat Jayaraman. 2007. Declarative and Visual Debugging in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange* (Montreal, Quebec, Canada) *(eclipse '07)*. Association for Computing Machinery, New York, NY, USA, 31–35. https://doi.org/10.1145/1328279.1328286

[9] Marcus Denker, Stéphane Ducasse, and Éric Tanter. 2006. Runtime bytecode transformation for Smalltalk. *Computer Languages, Systems & Structures* 32, 2 (2006), 125–139. https://doi.org/10.1016/j.cl.2005.10.002

[10] Michael J Flynn. 1966. Very High-speed Computing Systems. *Proc. IEEE* 54, 12, 1901–1909. https://doi.org/10.1109/PROC.1966.5273

[11] Hani Z Girgis and Bharat Jayaraman. 2006. *JavaDD: A Declarative Debugger for Java.* Technical Report. Department of Computer Science and Engineering, University at Buffalo. https://cse.buffalo.edu/tech-reports/2006-07.pdf

[12] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., USA. https://dl.acm.org/doi/book/10.5555/273

[13] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. 2005. Relational Queries Over Program Traces. *ACM SIGPLAN Notices* 40, 10 (oct 2005), 385–402. https://doi.org/10.1145/1103845.1094841

[14] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. 2006. Design and implementation of a backward-in-time debugger. In *NODe 2006 – GSEM 2006*, Robert Hirschfeld, Andreas Polze, and Ryszard Kowalczyk (Eds.). Gesellschaft für Informatik e.V., Bonn, 17–32. https://dl.gi.de/20.500.12116/24100

[15] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself, In Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (Atlanta, Georgia, USA). *SIGPLAN Not.* 32, 10, 318–326. https://doi.org/10.1145/263700.263754

[16] Amy J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions About Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 301–310. https://doi.org/10.1145/1368088.1368130

[17] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. 1997. Query-based Debugging of Object-oriented Programs. *ACM SIGPLAN Notices* 32, 10 (1997), 304–317. https://doi.org/10.1145/263698.263752

[18] Demian Lessa, Bharat Jayaraman, and Jeffrey K Czyz. 2010. *Scalable Visualizations and Query-based Debugging.* Technical Report. University at Buffalo, Department of Computer Science and Engineering. https://cse.buffalo.edu/tech-reports/2010-10.pdf

[19] Bil Lewis. 2003. Debugging Backwards in Time, In Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003). *CoRR* cs.SE/0310016. https://doi.org/10.48550/ARXIV.CS/0310016

[20] Adrian Lienhard, Stéphane Ducasse, and Tudor Gîrba. 2009. Taking an Object-centric View on Dynamic Information With Object Flow Analysis, In ESUG 2007 International Conference on Dynamic Languages (ESUG/ICDL 2007). *Computer Languages, Systems & Structures* 35, 1, 63–79. https://doi.org/10.1016/j.cl.2008.05.006

[21] Adrian Lienhard, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. 2008. Test Blueprints – Exposing Side Effects in Execution Traces to Support Writing Unit Tests. In *Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering (CSMR '08)*. IEEE Computer Society, USA, 83–92. https://doi.org/10.1109/CSMR.2008.4493303

[22] Jens Lincke, Robert Krahn, and Robert Hirschfeld. 2011. Implementing Scoped Method Tracing with ContextJS. In *Proceedings of the 3rd ACM International Workshop on Context-Oriented Programming* (Lancaster, United Kingdom) *(COP '11)*. Association for Computing Machinery, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/2068736.2068742

[23] Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding Application Errors and Security Flaws Using PQL: A Program Query Language, In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (San Diego, CA, USA). *ACM SIGPLAN Notices* 40, 10, 365–383. https://doi.org/10.1145/1094811.1094840

[24] Michael Perscheid, Bastian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt. 2010. Immediacy through Interactivity: Online Analysis of Run-time Behavior. In *2010 17th Working Conference on Reverse Engineering*. 77–86. https://doi.org/10.1109/WCRE.2010.17

[25] Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2013. Expositor: Scriptable Time-travel Debugging With First-Class Traces. In *2013 35th International Conference on Software Engineering (ICSE)*. 352–361. https://doi.org/10.1109/ICSE.2013.6606581

[26] A. Potanin, J. Noble, and R. Biddle. 2004. Snapshot Query-based Debugging. In *2004 Australian Software Engineering Conference. Proceedings*. 251–259. https://doi.org/10.1109/ASWEC.2004.1290478

[27] Guillaume Pothier and Éric Tanter. 2009. Back to the Future: Omniscient Debugging. *IEEE Software* 26, 6 (2009), 78–85. https://doi.org/10.1109/MS.2009.169

[28] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3, 9. https://doi.org/10.22152/programming-journal.org/2019/3/9

[29] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. 2012. Object-centric Debugging. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 485–495. https://doi.org/10.1109/ICSE.2012.6227167

[30] Leon Schweizer. 2014. *PathObjects: Revealing Object Interactions to Assist Developers in Program Comprehension.* Master's thesis. Hasso Plattner Institute, University of Potsdam. https://github.com/leoschweizer/PathObjects-Thesis

[31] Ben Shneiderman and Catherine Plaisant. 2005. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (4th ed.). Pearson Education. http://seu1.org/files/level5/IT201/Book%20-%20Ben%20Shneiderman-Designing%20the%20User%20Interface-4th%20Edition.pdf

[32] Christoph Thiede and Patrick Rein. 2021. *Squeak by Example.* Vol. 5.3.1. https://wiki.squeak.org/squeak/6546

[33] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (apr 1997), 38–43. https://doi.org/10.1145/248448.248457