Scopeo: an Object-Centric Debugging Approach for Exploring Object-Oriented Programs

Valentin Bourcier, Steven Costiou

Scopeo: an Object-Centric Debugging Approach for Exploring Object-Oriented Programs

1st Valentin Bourcier
Univ. Lille, Inria, CNRS, Centrale Lille
UMR 9189 CRIStAL, F-59000 Lille, France
valentin.bourcier@inria.fr

2nd Steven Costiou
Univ. Lille, Inria, CNRS, Centrale Lille
UMR 9189 CRIStAL, F-59000 Lille, France
steven.costiou@inria.fr

***Abstract*—Debugging object-oriented programs requires understanding how objects interact with each other and how their state evolves during execution. It is an arduous activity, as there are no methods to support finding objects to debug and explore their interactions.**

In this paper we present Scopeo, a novel object-centric approach to explore objects and their interactions in object-oriented programs. We evaluate the performance of Scopeo and conclude its applicability to real debugging scenarios. We then open a discussion on the interest of this approach for future research into debugging and program comprehension.

***Index Terms*—Object-centric debugging, Query debugging, Back-in-Time debugging**

I. Introduction

To debug object-oriented programs, developers have to understand programs' behavior at run time. Developers must identify objects and understand chains of interactions between objects. For that, they rely on standard tools such as breakpoints and static code analysis [1], [2]. Therefore finding starting points for debugging remains a manual, tedious task, and looking for a cause-effect chain [3] is difficult.

We focus on *Object-Centric Debugging* [2], a debugging approach that scopes traditional debugging tools (such as breakpoints) to singular objects. We hypothesize that object-centric perspectives and tools ease the debugging of object-oriented programs.

In this paper, we identify the challenges of debugging object-oriented programs and study how object-centric debuggers answer these challenges (section II). Then we present our preliminary work (section III) for a generic method to improve object-centric debugging given the challenges involved. We present Scopeo, a new omniscient object-centric debugger implementing our generic method. It provides developers with tools to build exploration scopes and an API to express queries over objects and their interactions. Exploration scopes help developers to refine their queries by focusing on specific object interactions. Through a performance evaluation (section IV), we conclude that our implementation of this method is acceptable for real-world scenarios. Finally, we discuss our solution and future work (section V).

II. Background and motivation

In this section, we present our motivation in the context of object-centric debugging. We illustrate the challenges of debugging object-oriented programs, *i.e.*, finding objects to debug and understanding their interactions. We detail the related literature and the limitations of object-centric debugging.

A. Challenges to object-oriented program debugging

Debugging object-oriented programs, *i.e.*, finding *Cause-Effect* relations [3] between objects, is challenging.

As an example, we use a test method from a real program [4], shown in Listing 1. This program divides students into homogeneous groups depending on their level. The level of each student is represented by a `marker` variable that takes two values: `+` or `−`. The example is written in Pharo[1], but the concepts and challenges enumerated in this section are common to all object-oriented languages.

```
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 1. A unit test of the Ammolite application.

The test calls the `textPrintStudent:on:` (line 6) that prints a student `s` on a stream `str`. The assertion (line 7) expects that the last character printed on the stream is the student's marker.

Getting to the objects: The test fails, indicating that a student named John is displayed with a space as the last character instead of a marker. To understand why, we must first retrieve the student presenting the bug. By putting a breakpoint at line 6 we could retrieve the student. But we would have to manually proceed with the execution until the moment when the student presenting the bug is the one about to be printed. With lots of students, this process would require many efforts. We could use a conditional breakpoint to halt the

[1]For readers unfamiliar with Pharo, a comparison with Java-like syntax:
- Variables are declared between pipes, assignments use `:=`.
- The message-send notation uses spaces: `a foo` is equivalent to `a.foo()`.
- Arguments are specified by colons: `a includes: b` is equivalent to `a.includes(b)`.
- Square brackets `[:x:y|]` delimit lexical closures, `x`,`y` are arguments.
- `#(+ −)` is an array of two elements `+` and `−` which are symbols.

execution whenever the student about to be printed is named John. However, there can be many students named John which again would force us to manually proceed with the execution until we identify the John presenting the bug. This example illustrates how *Getting to the Objects* is a difficult problem [1], [2].

Monitoring Object-Specific Interactions: To understand what causes the bug, we must identify which object has called or modified the student presenting the bug, the data passed to this object, and its state updates. To achieve this, after gaining access to the object we have to put breakpoints in the methods of the students' class. Then we must proceed with the execution until one breakpoint halts the execution, and manually step through the execution to analyze the sequence of statements executed. In the example of Listing 1 this process of *Monitoring Object-Specific Interactions* [2] is difficult because every student receives the same messages.

The process of systematic debugging [5] relies on hypotheses about the program's behavior that may be false. Therefore when debugging, formulating new hypotheses and rerunning the program to explore them is inevitable. When the program uses non-deterministic values, each new execution could differ from the previous one. In that case, when rerunning the program, the bug may have different symptoms or even disappear.

B. Object-centric breakpoints approaches

Object-centric breakpoints are breakpoints halting the execution for particular object events (state update or access and message sends [2]). The approach strongly rely on conventional debuggers in order to find the objects on which to install breakpoints. There is little research on this issue. *Declarative breakpoints* [6] express, for a group of particular objects, the criteria for interrupting the execution. *Instance Pointcuts* [7] express, for a particular object method or attribute, a sequence of conditions to meet to suspend the execution. *Object Miners* [1] track and collect objects produced by (sub) expressions selected from the source code.

We consider these solutions as extended conditional breakpoints. They reduce the amount of manual actions required for *Getting to the Objects* or *Monitoring Object-Specific Interactions*. However, writing the conditions to meet in order to interrupt the execution or to collect an object is a manual process that can be complex [2]. Moreover, it does not help developers avoid the need to restart the program which may result in the bug disappearing.

Therefore, object-centric breakpoint approaches are inconvenient for debugging scenarios where we cannot access the objects to debug or where random values are involved.

C. Object-centric omniscient approaches

We consider an approach to be omniscient if, for a given program execution, it is able to retrieve any state of the program after the first execution. Back-in-time approaches [8]–[10] perform a full record of a program execution and provide ways to explore it afterward. Time-traveling approaches [11]–[13] perform a partial recording of an execution and replay that execution to allow live exploration of the program. With such a record, omniscient debugging approaches capture non-deterministic values and avoid the need for restarting the entire debugging process to inspect any part of the execution. The ability of omniscient approaches to answer the challenges of debugging object-oriented programs depends on what they provide to help developers explore the execution recording.

1) Back-in-time debuggers: Whyline [8] generates and answers *"Why did? and Why did not?"* questions. Developers can ask *"Why did the color of this table change to blue?"*, where the color is an instance variable of the table object. To such a question, Whyline will answer with the sequence of interactions leading to the color being changed to blue. *Unstuck* [9] allows developers to backtrack an object in the program execution. It provides, for a given object, a view of all events related to that object, *e.g.*, the object has been passed in parameters of a method, returned from a method, etc. Unstuck also provides filters to look for specific events in the program execution, such as method calls or variable accesses. *Trace Debugger* [10], [14] supports queries about specific objects using objects' protocol of supported messages, or queries about views of the object state changes.

2) Time-traveling debuggers: Compass [11] tracks and analyzes the flow of objects in a program execution. The flow of an object refers to the events (*e.g.*, an object is created, passed as a parameter, etc.) that the object encounters during its life cycle. It can answer the question *"Where did this object come from?"*, and therefore contributes to solving the challenge of *Monitoring Object-Specific Interactions*. *DrDebug* [12] replays deterministically slices of a program execution. The slices are computed depending on a condition that developers have to determine. While replaying a given slice, the developers can step through it such as with standard tools. *Seeker* [13] supports the execution of *Time-traveling queries (TTQs)*. TTQs allow developers to formulate questions about the execution of a program and to time-travel between the results provided by the debugger. Seeker provides a pre-built list of object-centric queries. The queries are extendable so developers can write their own to customize how they explore the execution.

3) Limitations: Object-centric omniscient debuggers do not address the challenge of *Getting to the Objects*. Whyline is mainly applicable to UI problems and assumes that the object is already accessible from the UI. With Unstuck, developers are limited to a search in method trace view which is also manual and tedious. Finally, with time-traveling debuggers, the developer must rely on the back-and-forth navigation feature which remains manual and tedious [13].

The debuggers presented in this section contribute to answering the challenge of *Monitoring Object-Specific Interactions* in object-oriented programs. They provide developers with views or questions that can be used to identify object-related interactions. However, these tools are limited to the predefined set of questions or views which prevents addressing new or unanticipated problems. Moreover, questions are

answered regarding the entire program execution. Developers cannot ask questions about subparts of the program execution, *i.e.*, about sets of objects and interactions. Developers have to monitor interactions specific to one object after the other, until reaching the cause of the bug or invalidating the hypothesis of debugging (*cf. The process of systematic debugging* [5]).

Seeker allows for asking questions to fetch potential objects of interest instead of stepping back and forth through the execution to find those objects. Its set of questions is extensible, but it cannot formulate and answer questions about sets of objects and interactions defined by users.

Therefore, narrowing the debugging exploration to identify *Cause-Effect* relations is still difficult.

D. Summary and research question

We study the potential of object-centric approaches for debugging. We investigate the following question: *How to explore the execution history of an object-oriented program?* Existing object-centric approaches do not cope with the challenges of *Getting to the Objects* to debug and *Monitoring Object-Specific Interactions*, and suffer from the following problems:

1) *Non-determinism*: breakpoint approaches are disrupted by non-deterministic executions.
2) *Inability to explore subparts of a program execution*: they do not offer the possibility to express debugging questions about groups of objects and their interactions, but only about the entire execution history.
3) *Limited expressivity*: the possible questions we can ask are limited to a predefined set. We cannot ask questions for specific domains or problems.

III. Scopeo: an Object-centric Omniscient Debugger

We present Scopeo, our object-centric approach to exploring execution histories. To cope with non-determinism, we base Scopeo on an omniscient debugger. For running examples illustrating Scopeo, we refer the reader to [15].

A. Scopeo in a nutshell

Scopeo records program executions to avoid restarting programs and losing non-deterministic values. On top of that, Scopeo provides a storage area that we call the *exploration scope* in which we persist entities that we collect from an execution history. Scopeo offers a query-based exploration interface with which we search in the execution history. We combine the exploration scope, the query-based exploration interface, and time-traveling debugging features to explore objects and their interactions throughout the execution history.

B. Object-centric representation of the execution history

When recording executions, Scopeo only reifies interactions between objects (*i.e.*, method calls) and changes to the state of objects (*i.e.*, the assignment to variable instances). We derived this definition from [2] to enable the exploration of the execution history with an object-centric perspective. From the original definition, we removed the internal accesses to the state of an object to minimize the memory footprint of Scopeo. Every element of the recording has an *identifier* that corresponds to the *index* of the element in the sequence of instructions executed by the program.

C. Exploration scope

To address the problem of scoping, *i.e.*, to narrow down the exploration of the execution to subgroups of objects and interactions, we provide developers with a space to persist both objects and interactions. We call this storage space the *exploration scope*. Its purpose is to store elements of the execution which are involved in a potential chain of interactions that developers want to analyze while debugging.

D. Query-based exploration interface

To search in the execution history, we propose an exploration API [15] for writing queries based on Boolean predicates. When Scopeo executes a query, it selects every object event from the exploration scope that satisfies the query's predicates. Predicates are atomic and take into account only one condition with one argument. The argument of the condition can be an object, a symbol, or a collection. Fig. 1 shows a predicate verifying if an object event is associated with an object of type `Student`. Fig. 2 shows a composed predicate that we use to write multi-condition queries. To combine predicates, Scopeo uses Boolean operators such as *and*, *or*, *not*, etc.

Thanks to composition, predicates can express precise queries without the need to write code manipulating low-level APIs of the language. Developers can also build new predicates by writing classes defining a special method that checks for the wanted condition. Listing 2 shows the implementation of the `Selector` predicate from Fig. 2. It takes as a parameter an object event `anEvent` (provided by Scopeo at run time), the list of known events `anEventsHistory`, and the current exploration scope built by the developer `aScope`. Developers then use the API of the object event to build the condition. In this simple example, `anEvent` is selected if the method returns (↑) `true`, *i.e.*, the event is an interaction (line 2) and the selector of the interaction corresponds to the symbol passed as a parameter to the predicate (line 3). We can express

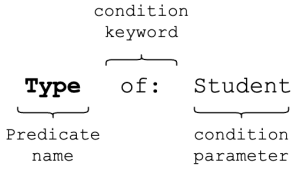

Fig. 1. Syntactic representation of a Scopeo's predicate.

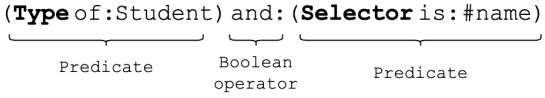

Fig. 2. Syntactic representation of a composed predicate.

more complex conditions using objects and interactions stored in the exploration scope (`aScope`) from previous exploration steps. The ability to extend and combine predicates addresses the *Limited expressivity* problem.

```
verify: anEvent among: anEventsHistory using: aScope
    ↑ anEvent isInteraction
        and: [ <a symbol> = anEvent selector ]
```

Listing 2. Implementation of the `Selector is: <a symbol>` predicate.

E. Exploring a program execution with Scopeo

To explore program executions we propose a method combining the exploration scope and the query-based exploration API (Fig. 3). *The process of systematic debugging* [5] explains how hypotheses are refined during debugging. We extend this definition by explaining how to scope the exploration around specific objects and their interactions (in blue in Fig. 3). We add objects to the exploration scope, refer to them in new queries to *Monitor Object-Specific Interactions* and narrow down our debugging questions to identify *Cause-Effect* relations. Fig. 3 illustrates how Scopeo guides developers depending on their questions:

- Q1 Is the question general to the whole exploration space or specific to a set of objects?
- Q2 If the question is specific, are the objects already stored in the scope of exploration?
- Q3 Is the formulation of another question required to understand the observed phenomenon? If so, is the new question about specific objects' behavior or not?

Step *S1* in Fig. 3 implies that the developer has direct access to the object (*e.g.*, in the context of a debugged method), or that the object was previously added thanks to a first question leading to steps *S0* and *S4*.

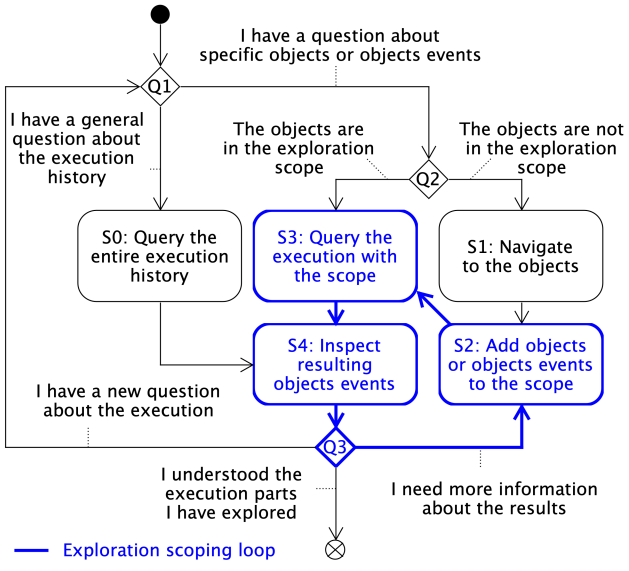

Fig. 3. Program exploration with Scopeo.

IV. Performance evaluation

In this section, we evaluate the cost of collecting the execution data required by Scopeo.

Implementation: We implemented Scopeo with Pharo [16] because its standard debugger has had an object-centric breakpoint implementation in its production version for several years now. Seeker [17] is a time-traveling debugger prototype with object-centric features that integrates with the standard Pharo debugger. These make strong baselines to compare with Scopeo in future empirical evaluations.

Benchmarks: To collect execution data, Scopeo instruments the source code packages of the debugged program to intercept all message sends and assignments, which adds computational overhead.

We evaluated this overhead on three programs. First, a *Control* program that we built, in which two objects run in a recursive loop of 1000000 calls. For each call, each object performs assignments to its instance variable and calls the other object. With this control loop, we measure the speed of simple instrumented instructions, *i.e.*, each executed instruction is instrumented. Second, we ran the test suite of a small application named *Ammolite* [4], which consists of 26 unit tests. The code of this application mostly contains a user interface and simple object manipulation. Because the test suite executes too fast to measure anything, we ran the suite 1000 times in a row and measured the total execution time. Third, we ran a 134 test suite of Roassal [18], the graphs library of Pharo. Roassal has complex and heterogeneous code that covers many syntactic cases of the Pharo language. We ran 100 executions of each benchmark using with Pharo 12 (build 1060) on an Apple M1 Max, 32GB RAM, running on Darwin Kernel Version 22.5.0. Table I shows our results.

Our benchmarks showed a noticeable difference between the control and the Ammolite programs which both show a $\times 2$. overhead with the Roassal test suite that shows a $\times 3.4$ overhead. One possible explanation is the diversity and complexity of the Roassal code involved in the benchmark, compared to the two other programs. We instrumented the entire Roassal code potentially touched by the test suite, *i.e.*, 5094 methods among which 18389 messages were sent and 1321 assignments performed. There are 13 packages involved. The test suite covers them at different levels. The less covered package has a coverage of 5%, the most covered package has a coverage of 87% and the median is 33% coverage. It would be interesting to repeat this type of benchmark by varying the applications studied. This will allow us to highlight the criteria an application must meet to be debuggable with Scopeo.

The measured overhead seems acceptable for experimenting with Scopeo. The Ammolite application has a GUI that

TABLE I
Overhead of Scopeo's instrumentation measured on three programs.

	Pharo (in ms)	Scopeo (in ms)	Overhead
Control	504 ±5.16	1089 ±9.98	×2.16
Ammolite	580 ±0.196	1280 ±0.476	×2.2
Roassal	469 ±3.91	1595 ±7.06	×3.4

remains usable without noticing any slowdown. The 134 test cases of Roassal ran under an acceptable time. Solutions like Seeker [17] can slow down test cases by a factor of 500 and are still applicable to the debugging of Pharo unit tests.

V. Discussion

Scopeo's strength lies in the ability to build customized exploration scopes tailored for daily debugging problems that developers encounter. We hypothesize that successfully finding objects in the execution history depends on the ability of the querying exploration interface to express object properties (attributes and protocols). For our future research, we wonder what queries are interesting to support and what are the requirements for such support. To address these research questions, we plan to study the questions developers ask while debugging. Previous experiments on questions asked during software evolution tasks [19], [20] will be interesting starting points. After, we intend to build sets of pre-defined predicates so that Scopeo can express and answer these questions. We then plan to evaluate the cost of answering these questions using Scopeo.

We evaluated the overhead of Scopeo's instrumentation for execution recording, but not for storing execution data and memory consumption. Although we will evaluate these aspects in the future, this is less concerning as work on scalable omniscient debugging already exists [21].

Finally, empirical evaluations are necessary to evaluate the impact of Scopeo on debugging and program understanding. That is challenging, as it will require teaching participants to use Scopeo's API and tools which are different from traditional debugging perspectives and tools. However, the comparison with baselines will be facilitated as Pharo already possesses object-centric tools in production.

VI. Conclusion

We presented Scopeo, our object-centric omniscient debugger prototype. Scopeo's objectives are to help find objects to debug in an execution and scope program explorations around a specific chain of interactions. We illustrated how Scopeo combines an exploration scope, a querying API, and time-traveling facilities to address the challenges of debugging object-oriented programs. We evaluated Scopeo's performance overhead on three programs, which showed a $\times 2.16$ to $\times 3.4$ overhead. Finally, we discussed our plans for future work to complete and evaluate our approach.

Acknowledgment

This work was funded by the ANR JCJC OCRE Project (https://anr.fr/Project-ANR-21-CE25-0004).

References

[1] S. Costiou, M. Kerboeuf, C. Toullec, A. Plantec, and S. Ducasse, "Object miners: Acquire, capture and replay objects to track elusive bugs." *The Journal of Object Technology*, vol. 19, pp. 1:1–32, Jul. 2020.

[2] J. Ressia, A. Bergel, and O. Nierstrasz, "Object-centric debugging," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 485–495.

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

[4] S. Costiou, "Ammolite magenta," https://github.com/StevenCostiou/AmmoliteMagenta, 2020, [Online; accessed 10-November-2023].

[5] D. Spinellis, "Modern debugging: The art of finding a needle in a haystack," *Commun. ACM*, vol. 61, no. 11, pp. 124–134, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3186278

[6] C. Corrodi, "Towards efficient object-centric debugging with declarative breakpoints," in *SATToSE 2016*, 2016.

[7] C. Bockisch and K. Hatun, "Instance pointcuts for program comprehension," in *Proceedings of the 1st workshop on Comprehension of complex systems*, 2013, pp. 7–12.

[8] A. J. Ko and B. A. Myers, "Designing the whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the 2004 conference on Human factors in computing systems*. ACM Press, 2004, pp. 151–158.

[9] C. Hofer, M. Denker, and S. Ducasse, "Design and implementation of a backward-in-time debugger," in *Proceedings of NODE'06*, ser. Lecture Notes in Informatics, vol. P-88. Gesellschaft für Informatik (GI), Sep. 2006, pp. 17–32.

[10] C. Thiede, M. Taeumel, and R. Hirschfeld, "Object-centric time-travel debugging: Exploring traces of objects." Tokyo, Japan: In Proceedings of the Programming Experience 2023 (PX/23) Workshop, companion volume to the International Conference on the Art, Science, and Engineering of Programming (¡Programming¿), co-located with the International Conference on the Art, Science, and Engineering of Programming (¡Programming¿), pages xxx-xxx, The University of Tokyo, Tokyo, Japan, March 14, 2023, ACM DL. (TO APPEAR), 2023.

[11] A. Lienhard, J. Fierz, and O. Nierstrasz, "Flow-centric, back-in-time debugging," in *Objects, Components, Models and Patterns, Proceedings of TOOLS Europe 2009*, ser. LNBIP, vol. 33. Springer-Verlag, 2009, pp. 272–288. [Online]. Available: http://scg.unibe.ch/archive/papers/Lien09aCompass.pdf

[12] Y. Wang, H. Patil, C. Pereira, G. Lueck, R. Gupta, and I. Neamtiu, "Drdebug: Deterministic replay based cyclic debugging with dynamic slicing," in *Proceedings of annual IEEE/ACM international symposium on code generation and optimization*. ACM, 2014, p. 98.

[13] M. Willembrinck, S. Costiou, A. Vanègue, and A. Etien, "Towards Object-Centric Time-Traveling Debuggers," in *International Workshop on Smalltalk Technologies (IWST 22)*. Novi Sad, Serbia: ACM Digital Libraries, Aug. 2022. [Online]. Available: https://inria.hal.science/hal-03825736

[14] C. Thiede, M. Taeumel, and 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*, ser. Onward! 2023. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 89–102. [Online]. Available: https://dl.acm.org/doi/10.1145/3622758.3622892

[15] V. Bourcier, "Scopeo example era," https://github.com/ValentinBourcier/ScopeoExampleERA, 2023, [Online; accessed 20-November-2023].

[16] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker, *Pharo by Example*. Kehrsatz, Switzerland: Square Bracket Associates, 2009. [Online]. Available: http://books.pharo.org

[17] M. Willembrinck, S. Costiou, A. Etien, and S. Ducasse, "Time-Traveling Debugging Queries: Faster Program Exploration," in *International Conference on Software Quality, Reliability, and Security*, Hainan Island, China, Dec. 2021. [Online]. Available: https://inria.hal.science/hal-03463047

[18] A. Bergel and ObjectProfile, "Roassal 3.0," 2020. [Online]. Available: http://agilevisualization.com

[19] J. Sillito, G. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, jul 2008.

[20] J. Kubelka, R. Robbes, and A. Bergel, "Live programming and software evolution: Questions during a programming change task," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 30–41.

[21] G. Pothier, E. Tanter, and J. Piquer, "Scalable omniscient debugging," *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, vol. 42, no. 10, pp. 535–552, 2007.