Analysis and exploration for new generation debuggers

Thomas Dupriez
ENS Paris-Saclay - RMoD, Inria Lille-Nord Europe
thomas.dupriez@ens-paris-saclay.fr

Guillermo Polito
RMoD - Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
guillermo.polito@univ-lille1.fr

Stéphane Ducasse
RMoD, Inria Lille-Nord Europe
stephane.ducasse@inria.fr

Abstract

Locating and fixing bugs is a well-known time consuming task. Advanced approaches such as object-centric or back-in-time debuggers have been proposed in the literature, still in many scenarios developers are left alone with primitive tools such as manual breakpoints and execution stepping. In this position paper we explore several advanced on-line debugging techniques such as advanced breakpoints and on-line execution comparison, that could help developers solve complex debugging scenarios. We analyse the challenges and underlying mechanisms required by these techniques. We present some early but promising prototypes we built on the Pharo programming language. We finally identify future research paths by analysing existing research and connecting it to the techniques we presented before.

Keywords Debugger, Tool, Stack, Breakpoint, Watchpoint

1. Introduction

Identifying and fixing bugs is an important task in software development. It is also well-known that this is a time-consuming activities [Som01, Zel05]. Several works have been proposed to help developers with such complicated task. Automatic validation of conditions [LHS99] and watchpoints [ADRC17] help developers to spot divergences from the expected program execution. Back-in-time debuggers offer the possibility to navigate the program execution history with several promising research reults [LD03, Hof06, PTP07, LGN08]. More recently, object-centric debuggers presented advanced stepping mechanisms targetting individual objects [RBN12]. Moldable debuggers [CGN14] offer a different perspective to this problem: it should be possible to adapt a debugger to a given domain or task.

In this position paper we motivate the need to mature and develop more advanced techniques by showing a complex debugging scenario obtained from a real use case. We then explore several promising advanced debugging techniques and analyse the key challenges they pose:

Advanced Breakpoints. What if a developer had the potential to create new smart breakpoints? What would be such breakpoints? What kind of runtime information should they have access to to be both useful and efficient?

Execution Comparison. What if a developer could, after modifying his codebase and breaking some tests, compare the executions of the program before the change and after the change to find the cause of the bug? What would be the infrastructure required for this technique? What are the tools we could provide to a developer to perform this task?

Accessing Execution History. Back-in-time debuggers are nowadays the referent debuggers to navigate history. However, many questions remain still open. What would be a both a practical and efficient solution? What are the alternatives to store both the execution and the objects in the program's state? How can we navigate the program history to find a bug?

During our analysis, we also present some promising ideas like the possibility of scripting the stack navigation. Finally we present some prototypes we implemented showing the feasibility of some of them.

2. Debugging Terminology

Debugging is the process of finding and repairing defects in software. Informally, we refer to software defects as **bugs**, due to the difficulty they may present to be spotted and removed. The main challenge of debugging is to spot bugs *i.e.,* to examine a program and understand what is the exact cause of the misbehaviour or error.

In general, a developer performs such debugging process with the aid of a debugger tool, or simply a **debugger**. Debuggers allow developers to suspend a program execution to inspect and explore the execution. This includes both observing the execution path and the state of objects existing in the application.

Mainstream debuggers offer the possibility to suspend the program execution by placing **breakpoints**. When the program execution arrives to a breakpoint, the execution is suspended and the debugger shows to the developer the current state of the execution. This execution is generally shown as a **stack trace**. A stack trace is a sequence of methods executions (usually called contexts or stack frames). In such a sequence, the context that precedes another context is called its **caller** or **sender**. This sequence represents then a program execution path from a first context until a context with a breakpoint. We call the first context of the program its entry point. In most programming languages, a program entry point is the first execution of the main function of a thread. In Pharo, a program entry point is the first context of a Process, usually executing the Block>>#newProcess method.

The utility of the stack trace is twofold. On the one hand, a developer can use it to navigate the execution path and control flow that led to a problem. On the other hand, he can also use it to inspect previous states of the execution, by looking at the execution contexts .

3. A Real Complex Debugging Scenario

To illustrate why debugging is a complex task and motivate the need of new debugging techniques, we isolated a real bug that appeared while refactoring the latest versions of the Pillar markup language[ADCD16]. In this section we present the issue found and the way to reproduce it. We setup a repository explaining in details the steps to reproduce the bug in https://github.com/guillep/pillar-bug,

Pillar uses an object called PRPillarConfiguration to manage all project settings *e.g.,* what is the output format, whether it needs to numerate sections or not, print a contents table or not. PRPillarConfiguration semantics are complicated. First, it overrides doesNotUnderstand: [Duc99] to dynamically interpret some message sends as configuration setters or getters. Second, it uses the Magritte [Ren06] meta-description framework to control how a configuration values should behave by default, be serialized, deserialized, and validated. Finally, a PRConfiguration is organised in a hierarchy of configurations, and some operations may lookup settings in the hierarchy of the configuration while others do not.

Given this situation, the pillar developers decided to refactor PRPillarConfiguration to depend less on doesNotUnderstand: semantics. They so decided to introduce in PRPillarConfiguration a disabledPhases instance variable with its respective accessors. Before doing such a change, all 3182 Pillar tests were running ok.

```
$ ./pharo Pharo.image test "Pillar.*"
[...]
3182 run, 3182 passes, 0 failures, 0 errors.
```

However, as soon as we introduce the new accessors, 16 new errors appeared:

```
$ ./pharo Pharo.image test "Pillar.*"
[...]
3182 run, 3166 passes, 0 failures, 16 errors
```

Checking the tests, we observe that the bug happens in an apparently unrelated piece of code, the PREPubMenuJustHeaderTransformer>>actionOn: method. The symptom of the bug is that outputType is nil. However, this piece of failing code plus the fact that 3166 tests are still working, give us no clue about the relation with the change and the bug.

```
PREPubMenuJustHeaderTransformer>>actionOn: anInput
  ^ (self class writers
    includes: anInput configuration outputType writerName)
    ifTrue: [ maxHeader := self maxHeaderOf: anInput input.
      super actionOn: anInput ]
    ifFalse: [ anInput ]
```

In the following sections of this paper, we explore several advanced debugging techniques that could help us finding the cause of this bug.

4. Technique 1: Advanced Breakpoints

Breakpoints are a very common feature among debuggers. They let developers specify a point in a program where the execution should be suspended. However, debuggers usually limit breakpoints to a limited set of pre-existing ones such as breaking when the program arrives to a particular line. Developers are not usually capable of tailoring breakpoints to some more specific needs. This section presents some ideas that would increase the expressiveness of breakpoints and allow developers to specify more precise trigger conditions.

4.1 Idea 1: Conditional Breakpoints

Scenario. The developer debugging Pillar is interested in looking at what happens to the problematic actionOn: method only when it is called with a specific argument. If he places a normal breakpoint in the method, the breakpoint will trigger no matter the argument and he will have to check himself whether the arguments are those he is interested in.

Idea. The developer could place a *conditional breakpoint* that only triggers if the arguments are those he is interested in. Conditional breakpoints are nowadays available in many existing debuggers such as Pharo's and C's gdb, mostly allowing the execution of simple conditional expressions. We would like to explore their limitations and further possibilities.

4.2 Idea 2: Contextual Breakpoints

Scenario. The Pillar developer would like her breakpoints to only be triggered when her code is launched from a test. A normal breakpoint triggers from any execution, whether a test or not. This may interfere with the normal execution of the program, even in cases where the bug does not reproduce or appear.

Idea. The developer could place a *contextual breakpoint*: a conditional breakpoint that depends on the dynamic execution of the program. For example, a breakpoint could detect whether the current execution was initiated by a test process or not, or by an HTTP request or not. Moreover, this selective breakpointing could be useful to debug core libraries such as collections or compilers. Since core libraries are in usage by the runtime, setting a breakpoint in the may cause the entire runtime to suspend.

Alternative Scenario. A developer wrote a test to validate his implementation. He placed breakpoints in different methods invoked in the test because he is adjusting the behaviour of such methods to make the tests pass. Alternately, he may want that when executing the test, only the breakpoints placed in the test method itself trigger.

4.3 Challenges

These ideas rely on the possibility for the breakpoints to decide whether to trigger based on the context they are executed in. This means that the breakpoints must be able to access their context, which is not trivial in most languages. Moreover, we ask ourselves the following research questions: What is the debugger support required to interactively set conditional breakpoints? What is the runtime information they can and could access? How could we easily define dynamic contexts that would help debugging?

5. Technique 2: Execution Comparison

5.1 Scenarios

In our Pillar scenario, the developer has two versions of the program: a first one that passes the test and another one that does not. Moreover, he knows what code-change made the test fail. This scenario is similar to the one studied by Zeller et al. in delta-debugging [Zel05].

The developer can then try to compare the execution of the broken program with some other running version of it. For example, he could try to compare the broken test with its previous version that was working. Alternatively, he could try to compare the broken test with a working test from the same version of the code. If the working test exercises the same program but with a different input, it could give her enough insight on the origin of the problem.

In both scenarios, the developer would like the debugger to provide her with information about how the two test executions differ so that he can understand how his change made the test fail. For the sake of presentation, we will focus in the following subsections on the first scenario.

5.2 Idea 1: Comparing Execution Paths

The developer would like to spot differences between the execution paths of the programs. A naive idea to achieve this would be to run the two executions in parallel, compare the sequences of messages sent and stop the executions when they diverge. This would effectively pinpoint the first path difference between the two executions.

Unfortunately, this is most likely not enough to entirely cover the needs of the developer. For example, the cause of the bug may not be related to the first found difference, but to a difference occurring later in the execution. A possible way to address this concern would be to offer the developer a way to *smart step* from difference to difference, which would step both executions until the next difference.

Challenge: Defining Execution Path Differences. Notice we referred to the concept of *differences* between execution paths without precisely defining it. Indeed, this is to us an open research question: What is a definition of difference that is useful for debugging? Indeed, a too strict definition of difference (*e.g.,* comparing the two sequences of messages and flagging the positions where different messages were sent) would drown the developer in differences instead of showing a bigger picture, delivering a bad debugging experience. On the other hand, a too relaxed definition of difference could miss some genuine differences.

Besides a definition of execution path difference, we also need a definition of similarity. As the execution paths advance and diverge, they may converge later if the two executions are similar enough, allowing us to find the next divergence. This problem is particularly challenging when debugging complex applications using big libraries or frameworks because they may have complex execution flows.

5.3 Idea 2: Comparing Object Interaction History

The developer would also like to spot differences between the interactions with some particular object(s) in both executions. For example, we would like to compare the times an object was sent a particular message in both executions. Moreover, this idea can also be applied to track the evolutions of the state of an object.Similarly to the Execution Path Differences idea, we could offer the developer a *smart step* that would allow him to navigate the executions according to what happens to a given object.

Challenge: Object Equivalence. Comparing object interactions in both executions requires defining an *object equivalence* criteria. Depending on the debugging scenario, we estimate different criterion could be applied. For example, using strict equality can be useful in the scenario where we debug two versions of the program with the same input. On the other hand, a more relaxed equality, *e.g.,* a manually se-

lection done by the developer, could be useful in the scenario comparing the same program with different input.

Moreover, we identify other open questions related to this topic. Can we automatically detect and propose the most suitable object equivalence strategy given a debugging scenario? Given an object equivalence strategy, can we automatically identify pairs of objects representing conceptually the same instance?

5.4 Technical Challenge

Running and Isolating two Simultaneous Executions. Both of the ideas we expressed in this section relies on the ability of the debugger to run and control two executions at the same time in an isolated fashion. The isolation is important to prevent the two processes from interfering with each other, for example by accessing the same global variable

6. Technique 3: Accessing Execution History

6.1 Scenario

The bug in Pillar is difficult to debug because the stack trace does not contain all the information relevant to the bug. Indeed, even if we know that the bug's symptom is that the expression anInput configuration outputType evaluates to nil, we do not know how that value arrived there nor why.

The general problem is that a stack trace shows only a single path in the program's execution. That is, if we think a program's execution as a call graph [GDDC97], the stack trace shown in the debugger represents a single path from that call graph. In this terminology, a stack trace only shows the current execution path from the program entry point until the break point. Previous execution paths and their execution contexts are discarded and the state they held is not available for debugging anymore.

This problem also happens with the program state. A stack trace only provides access to temporary variables in the stack. Moreover, these values are stored as object references, so if the related objects were modified during in the execution, then the stack trace shows only their last version. Thus, when using a traditional stack trace, we only have a view at the moment the execution was suspended, and not as they were when these execution contexts were captured.

6.2 Idea 1: Storing the History of Executions

In order to have access to more information about the execution, without having to constantly place new breakpoints and re-running it, we could store more information as the execution progresses.

Storing the Result of Expressions Evaluations. A first way the debugger could provide more information to developers is to store the results of the evaluation of the expressions (and their sub-expressions) in the execution as it progresses. These results could then be displayed to developers to improve the debugging session.

Storing the Entire Execution History. Even more information about the execution could be stored by using techniques from the *back-in-time debugging* field [LD03] [PTP07][Hof06] [Fie09] [LGN08].

Challenges 1: Temporality of the Stored Objects. A standard issue with storing objects during an execution is their "temporality", as side effects in the following execution may alter them, hence defeating the point of storing them for later review. A simple solution would be to copy the objects and storing the copies instead of the originals, however, this ties into a second challenge.

Challenge 2: Memory Consumption. A limitation of any storing solution is the memory limit. The evolution of the program state and execution path throughout an entire execution represents a lot of data. To circumvent this limitation, one can restrict the scope of the information stored (for example only storing the evolution of the state of a few objects). Another solution consists in taking advantage of the deterministic nature of executions and only storing some of the states of the execution. Then any state of the execution can be reached by simulating the execution starting from one of these stored states [ADRC17]. These two solutions highlight two dimensions of execution information storage techniques: the *granularity* of the storage *i.e.,* how much of the execution state do we store, and the *frequency* of the storage *i.e.,* at which intervals do we store the state of the execution.

6.3 Idea 2: Navigating the History of Executions

Another important aspect of helping developers to debug by showing them more information is providing them with tools allowing them to quickly find the information they need.

Expression Watchpoint. Sometimes developers know the cause of a bug (for example a collection containing corrupted values is sent as argument to a function, that cannot process it and raises an exception) but cannot locate the the point in the execution where this cause appeared (in this example, when did the collection become corrupted). To solve this issue, they could write an expression that would evaluate to true as long as the execution is in a non-bugged state (in this example, the expression would return whether the collection is not corrupted) and have the debugger run the execution and suspend it when the expression evaluates to a different value.

DSL for Custom Debugger Steps. Mainstream debuggers, to control executions, offer generic step commands usable in most circumstances. However, developers working on specific applications may want to leverage the specificities of these applications to improve their debugging experience. For example, if an application processes data via a succession of operation (a pipeline), developers debugging it regularly may greatly benefit from their debugger integrating specific step commands stepping from an operation to the

next. In this regard, debuggers could include a DSL allowing developers to script new step commands specific to their applications.

Challenge: Expressiveness. A common point we glossed over in the previous two paragraphs is how to write the expressions or custom debugger steps. The objects contained in these expressions need to be reachable from the context in which they are evaluated, so these expressions can be evaluated either in the scope of execution context (for example, having access to temporary variables) or in the global scope (only having access to global variables). Both solutions have drawbacks. The first one may create unwanted behaviours if multiple unrelated temporary variables across multiple methods have the same names, and requires a semantic for when the expression is evaluated in a context that for example does not contain temporary variables the expression refers to. The second one requires making relevant objects accessible from the global scope, which poses a constraint on the source code.

7. Prototypes for New Generations Debuggers

This section describes the prototypes we developed based on the ideas we previously exposed in this paper. These prototypes were developed in the Pharo programming language [BDN+09], an implementation of the Smalltalk language.

7.1 Advanced Breakpoints

Conditional breakpoints already exist in Pharo. Conditional breakpoints combine breakpoints with block closures enclosing a boolean condition. When the execution arrives to the breakpoint the condition is evaluated. If the block closure evaluates to true, the breakpoint is activated.

The existing conditional breakpoints can be further combined with contextual information to create *contextual breakpoints*. An example of adding contextual information is to reify the execution environment to say *e.g.,* whether we are in the context of a test or not. Figure 1 shows how we could use such reification to create a conditional breakpoint that will only trigger while tests are run.

```
self haltIf: [ CurrentExecutionEnvironment value
    isTestEnvironment ].
```

Figure 1: Breakpoint that only triggers while tests are run.

7.2 Expression Evaluation Recording

We implemented a mechanism to record the results of expression evaluations inside a given method. This contextual information is useful during debugging because a developer can quickly see the values of previously evaluated expressions without having to re-evaluate them. This is particularly handy when previous expressions depend on side-effects such as reading or writing to a file.

We implemented this prototype by using the code instrumentation features offered by the *metalinks* library [Den08]. Our instrumentation consists in replacing the AST nodes of the method with nodes that perform the same computation but also store the result of it in a dictionary. In addition, return nodes are altered to set a breakpoint and open an inspector on the dictionary the results are stored in.

We consider further enhancing our prototype in two main ways:

More generic handling of the types of nodes. Currently, a specific replacement node must be written for each different type of AST node (*e.g.,* message-send, variable assignment...). For this, we consider using *metalinks* in a more general fashion. Instead of replacing nodes, we will insert *after* metalinks after the nodes, that will access the result values and store them (instead of computing the values themselves).

Improved usability. Our prototype opens an inspector as soon as a return node of an instrumented method is evaluated, immediately stopping the execution. This prevents the program from executing further than this point which is a real limitation.

7.3 Transition Watchpoints

We wrote a prototype of Transition Watchpoints, inspired by the original paper [ADRC17]. Transition watchpoints are watchpoints that evaluate the value of a given expression in several points in the execution, to find when such evaluation transitions from one expected value to a different one. When the evaluation differs we activate a breakpoint, suspending the execution and opening the debugger. Well chosen expressions can find the precise line of code breaking an invariant and introducing a bug.

Our prototype takes as input an expression that can only reference elements reachable from the global scope (like global variables). This is to always be able to evaluate the expression regardless of the precise execution context.

Moreover, we figured out it was not necessary to restrain ourselves to expressions with boolean value and to the true-to-false transition as in the original implementation. Our expression is evaluated before the execution starts and this initial value is stored and serves as the reference value. The obvious limitation of not using snapshots as described in the paper is the potentially very high performance cost if the expression is complex to evaluate, as it is evaluated after each execution step.

8. Related Works

Delta-Debugging. *Delta-Debugging* [Zel05] encompasses multiple techniques whose common idea is to perform automated debugging by finding failure-inducing circumstances.

One of these techniques takes as in put a buggy source code, and attempt to find the part of it that makes the code buggy, by selectively removing pieces of it and checking whether the same bug is present.

Stateful Breakpoints. *Stateful Breakpoints* [Bod11] are about building runtime monitors out of normal breakpoints: a developer defines some normal breakpoints and gives them a name and an expression (built out of the variables available in the scope the breakpoint is in). Then he gives a pattern (a regex on the labels given to the breakpoints). The Eclipse plugin creates a runtime parameterized monitor out of these information, that triggers when the breakpoints are encountered in an order that matches the regex AND the expressions associated to the breakpoints evaluate to the same value/objects.

Back in Time Debugging. The basic idea behind *Back in Time debugging* [LD03] [PTP07][Hof06] [Fie09] [LGN08] is to store information about an execution to be able to go back in time in it and analyse it as it was.

Domain Specific Debugger. *Domain Specific Debuggers* [CGN14] are about providing debuggers dedicated to particular user's needs.

Watchpoints. *Watchpoints* [LHS99] [ADRC17] [Cor16] are tools to specify conditions upon which an execution should be suspended for analyse purposes.

9. Conclusion

References

[ADCD16] Thibault Arloing, Yann Dubois, Damien Cassou, and Stéphane Ducasse. Pillar: A Versatile and Extensible Lightweight Markup Language. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, August 2016.

[ADRC17] Kapil Arya, Tyler Denniston, Ariel Rabkin, and Gene Cooperman. Transition watchpoints: Teaching old debuggers new tricks. *The Art, Science, and Engineering of Programming*, 1(2), July 2017.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[Bod11] Eric Bodden. Stateful breakpoints: A practical approach to defining parameterized runtime monitors. In *ESEC/FSE'11*, 2011.

[CGN14] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. The Moldable Debugger: A framework for developing domain-specific debuggers. In Benoit Combemale, DavidJ. Pearce, Olivier Barais, and JurgenJ. Vinju, editors, *Software Language Engineering*, volume 8706 of *Lecture Notes in Computer Science*, pages 102–121. Springer International Publishing, 2014.

[Cor16] Claudio Corrodi. Towards efficient object-centric debugging with declarative breakpoints. In *SATToSE 2016*, 2016.

[Den08] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[Fie09] Julien Fierz. Compass: Flow-centric back-in-time debugging. Master's thesis, University of Bern, January 2009.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, pages 108–124, New York, NY, USA, 1997. ACM.

[Hof06] Christoph Hofer. Implementing a backward-in-time debugger. Master's thesis, University of Bern, September 2006.

[LD03] Bill Lewis and Mireille Ducassé. Using events to debug Java programs backwards in time. In *OOPSLA Companion 2003*, pages 96–97, 2003.

[LGN08] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[LHS99] Raimondas Lencevicius, Urs Hölzle, and Ambuj Kumar Singh. Dynamic query-based debugging. In R. Guerraoui, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *LNCS*, pages 135–160, Lisbon, Portugal, June 1999. Springer-Verlag.

[PLW09] Frédéric Pluquet, Stefan Langerman, and Roel Wuyts. Executing code in the past: Efficient in-memory object graph versioning. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09)*, pages 391–408. ACM, 2009.

[PTP07] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, 42(10):535–552, 2007.

[RBN12] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012.

[Ren06] Lukas Renggli. Magritte — meta-described web application development. Master's thesis, University of Bern, June 2006.

[Som01] Ian Sommerville. *Software Engineering (6th ed.)*. Addison-Wesley, 2001.

[Zel05] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, October 2005.