



HAL
open science

Sindarin: A Versatile Scripting API for the Pharo Debugger

Thomas Dupriez, Guillermo Polito, Steven Costiou, Vincent Aranega,
Stéphane Ducasse

► **To cite this version:**

Thomas Dupriez, Guillermo Polito, Steven Costiou, Vincent Aranega, Stéphane Ducasse. Sindarin: A Versatile Scripting API for the Pharo Debugger. Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages, Dec 2019, Athens, Greece. 10.1145/3359619.3359745 . hal-02280915

HAL Id: hal-02280915

<https://hal.science/hal-02280915v1>

Submitted on 6 Sep 2019

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

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

Sindarin: A Versatile Scripting API for the Pharo Debugger

Thomas Dupriez
Univ. Lille, CNRS, Centrale Lille,
Inria, UMR 9189 - CRISTAL
France
thomas.dupriez@univ-lille.fr

Guillermo Polito
CNRS - UMR 9189 - CRISTAL, Univ.
Lille, Centrale Lille, Inria
France
guillermo.polito@inria.fr

Steven Costiou
Inria, Univ. Lille, CNRS, Centrale
Lille, UMR 9189 - CRISTAL
France
steven.costiou@inria.fr

Vincent Aranega
Univ. Lille, CNRS, Centrale Lille,
Inria, UMR 9189 - CRISTAL
France
vincent.aranega@univ-lille.fr

Stéphane Ducasse
Inria, Univ. Lille, CNRS, Centrale
Lille, UMR 9189 - CRISTAL
France
stephane.ducasse@inria.fr

Abstract

Debugging is one of the most important and time consuming activities in software maintenance, yet mainstream debuggers are not well-adapted to several debugging scenarios. This has led to the research of new techniques covering specific families of complex bugs. Notably, recent research proposes to empower developers with scripting DSLs, plugin-based and moldable debuggers. However, these solutions are tailored to specific use-cases, or too costly for one-time-use scenarios.

In this paper we argue that exposing a debugging scripting interface in mainstream debuggers helps in solving many challenging debugging scenarios. For this purpose, we present Sindarin, a scripting API that eases the expression and automation of different strategies developers pursue during their debugging sessions. Sindarin provides a GDB-like API, augmented with AST-bytecode-source code mappings and object-centric capabilities. To demonstrate the versatility of Sindarin, we reproduce several advanced breakpoints and non-trivial debugging mechanisms from the literature.

Keywords debugging, object centrics, scripting, Pharo

1 Introduction

Debugging is an important part of software development. Literature describes it as a difficult task, on which developers spend sometimes more than 50% of their development time [37–39, 41, 43]. Traditional debuggers, often called breakpoint-based or online debuggers, support interactive debug sessions via a graphical user interface. In such debuggers, the developer makes a hypothesis about the cause of a bug and manually places breakpoints in the relevant pieces of the code. When the debugger stops in a breakpoint, the developer explores the subsequent execution by commanding a *step-by-step* execution on it.

Studies and work on debugging acknowledge that mainstream debuggers are not well adapted to several debugging scenarios [3, 27, 31]. This has led to the appearance of new debugging techniques proposing to augment traditional interactive debuggers with, *e.g.*, stateful breakpoints [4], control-flow aware breakpoints [5], object-centric breakpoints [10, 34], the automatic insertion of breakpoints based on dynamic executions [45], or declarative statements from the developer [21]. A line of research has also started to study scripting APIs to empower developers to implement debugging scripts adapted to their needs. These scripting APIs help developers to *e.g.*, insert breakpoints based on cross-cutting concerns [42] or declaratively explore post-mortem executions [32]. Even further, the moldable debugger framework [6] allows developers to create domain-specific debuggers with domain-specific views and operations. With the exception of the moldable debugger framework, all these solutions are dedicated tools and the developer is left alone to benefit from them in an integrated way.

Research questions. Our research question is:

- What is a debugger API that is powerful and versatile enough to allow developers to perform many debugging tasks that would normally require specific debugging tools or tedious manual operations?

In this paper we present Sindarin¹, a scriptable on-line debugger API for Pharo² [13], and we use it to reproduce several debugging scenarios and advanced breakpoints from the literature. To do this, Sindarin exposes stepping and introspection operations not always appearing in mainstream debuggers such as VisualStudio or GDB. In addition, it simplifies the

¹ This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020. The work is supported by I-Site ERC-Generator Multi project 2018-2022. We gratefully acknowledge the financial support of the Métropole Européenne de Lille.

²Pharo is a pure object-oriented dynamically typed programming language inspired by Smalltalk - <http://www.pharo.org>.

creation of personalized debugging scripts by providing AST mappings, thus also proposing different stepping granularity over the debugging session. Finally Sindarin facilitates object reachability to use object-centric debugging [12, 34].

Contributions. The contributions of this paper are:

- The identification of three main requirements that a debugger API must satisfy to allow developers to solve many debugging scenarios from the literature: step granularity at the expression level, full access to execution contexts (reading and modifying) and a bytecode-to-AST mapping.
- The design of a debugger API through which developers can express several debugging scenarios and breakpoints from the literature, that would normally require specific tools.

Outline. The paper is structured as follows: Section 2 presents an overview of the Sindarin API and a usage example. Section 3 presents debugging scenarios from the literature and how they are solved with compact Sindarin scripts. Section 4 shows how to use Sindarin to express advanced breakpoints from the literature. Section 5 shows how Sindarin facilitates object-centric debugging. Section 6 analyses the requirements of debugging scenarios and compares them to mainstream debugger APIs. Section 7 gives a glance at the implementation of Sindarin in the Pharo debugger and its integration in the Integrated Development Environment. It also explains how Sindarin could be integrated in mainstream software development and discusses its limitations. Section 8 exposes the related work. Finally, section 9 touches upon other aspects around Sindarin, like self debugging or the generation of persistent artifacts from Sindarin scripts.

2 Sindarin's Overview

Sindarin is a scripting API for the Pharo debugger, implemented as an internal DSL. In this section we introduce Sindarin through an example making use of several of its features. We then document an extract of Sindarin's API, as it is used in the rest of the paper to illustrate more advanced debugging scenarios.

2.1 Sindarin by Example

To illustrate Sindarin, let's consider a program that operates on several files inside a directory. After some seconds running, the program tries to re-open an already open file and fails. The developer would like the program to stop when a file is opened for the second time and know for *this specific file* how and when it was opened the first time. The Sindarin script shown in Listing 1 solves this problem³. It (a) first detects

when a file is opened and records the stack when this happens, then (b) it steps until an already open file is opened again.

```

1 | result := nil
2 | finished := false.
3 | stackDictionary := Dictionary new.
4 | [ finished ] whileFalse: [
5 |   (dbg currentNode isMessageNode
6 |   and: [(dbg messageReceiver isKindOf: File)
7 |   and: [ dbg messageSelector = #open ])
8 |   ifTrue: [
9 |     result := stackDictionary
10 |    at: dbg messageReceiver
11 |    ifPresent: [ finished := true ]
12 |    ifAbsentPut: [ dbg stack copy ]].
13 |   dbg step ]

```

Listing 1. Sindarin illustrated. Step until a file is open twice and return the stack that was saved when it was first opened.

This script continuously steps the execution (lines 4 and 13). After each step, it checks if the execution is about to send a message (line 5) to an instance of the File class (line 6). If the message is #open (line 7), it checks in a dictionary whether the file has already been opened (line 9-10). If the file is being opened for the first time, an entry is added in the dictionary with a copy of the current execution stack as value (line 12). If there is already an entry (line 11) the stack of interest is stored in result and the script terminates.

This example illustrates several aspects of the API:

- Step Operations.** Sindarin commands the execution through a step method performing a step-into (line 13).
- Stack frame access.** The stack frame is queried multiple times during the script to get the receiver of a message before it is executed (lines 6, 10) and to obtain the current stack trace (line 12).
- AST Mapping.** Sindarin's currentNode method maps the current program counter to the corresponding AST node, allowing high-level queries on the execution (lines 5, 7).
- Object-reachability.** All objects created and used during the execution are reachable by the Sindarin script, easing the construction of object-centric debugging scripts (lines 6, 10).

Sindarin scripts are stateful: they can define and use variables.

³For readers unfamiliar with the Pharo/Smalltalk syntax:

- The message-send notation uses spaces instead of dots, and there are no parentheses to specify arguments: `dbg currentNode` is equivalent to `dbg.currentNode()`

- Arguments are specified by colons: e.g., `dbg messageReceiver isKindOf: File` is equivalent to `dbg.messageReceiver().isKindOf(File)`
- Square brackets [] delimit lexical closures.

2.2 Sindarin's API

Sindarin's internal DSL exposes among others:

Expression-level Step Operations. Basic step operations like step into and step over. Unlike in most other debugging APIs, these operations work at the expression level rather than the line level.

Full Context Access. Access to, and manipulation of the stack trace and its stack frames.

AST Mapping. A bytecode-AST mapping, mapping the debugged program's program counter to the AST node currently executed.

Object-Centric Debugging Operations. Scope breakpoints to specific objects.

Breakpoint-related Operations. To set, customize and remove breakpoints.

Sindarin's API is further described in Table 1. The notations used in the table are as follows: dbg stands for scriptable debugger, ctx stands for a context (a.k.a. stack frame), ast stands for an AST node and bp stands for breakpoint.

3 Solving Debugging Scenarios with Sindarin

In this section we present debugging scenarios frequently encountered by developers in the literature and we show how they can be solved with Sindarin.

3.1 Monitoring Assignments to a Variable

In this scenario described originally in [42], a developer spots that the foo instance variable of the class Bar has, at some point of the execution, the unexpected value of 42. To find the cause of the bug, she wants to know which assignments store 42 into the variable, for what she needs to manually search all writes to that variable and set a conditional breakpoint on them.

Fig. 2 shows a Sindarin script solving this problem. This script steps the execution until an assignment with value 42 (line 3) is about to be performed on the #foo variable (line 4) on an instance of the Bar class (line 5). The assignment is detected in line 2 using AST comparison. This condition makes sure the script does not catch assignments to instance variables of the same name but from another class. With this script, the developer can quickly find the unexpected assignment and proceed from there to find the cause of the bug.

```

1 | dbg stepUntil: [
2 |   dbg currentNode isAssignment
3 |   and: [ (dbg assignmentValue == 42)
4 |   and: [ (dbg assignmentVariableName = #foo)
5 |   and: [ dbg receiver isKindOf: Bar ]]]
6 | ]

```

Listing 2. Stopping on specific assignments with Sindarin.

3.2 Stopping Before an Exception

When debugging, developers often want to stop the execution just before a certain situation occurs, for example a specific exception being raised. To raise an exception in Pharo, the message signal must be sent to a subinstance of the class Exception, so this is what the script will look for.

Using Sindarin the developer can write the script shown in Listing 3, that steps the execution until it is about to raise an exception.

```

1 | dbg stepUntil: [
2 |   dbg currentNode isMessage
3 |   and: [ (dbg messageSelector = #signal)
4 |   and: [ dbg messageReceiver isKindOf: Exception ]]]

```

Listing 3. Sindarin script to step until an exception is about to be signalled.

This scenario is very close to the one proposed by Haihan *et al.*, where a developer observes a NullPointerException in a chain of dereferences [42], as illustrated by the following Java expression `total.getObjects().addAll(current.getObjects())`.

In Java, when the exception is raised, the context where the problem happened and its intermediate expressions is lost: only the line number where the error was raised is available. If the developer wants to find the problematic sub-expression, she needs to rewrite the expression with one statement per line, or use a complex chaining of step into operations. However, even if the two problematics are, somehow, equivalent (a specific exception is raised), this time the developer wants to stop before the call is performed, and not before the exception is sent.

A Sindarin script solving this problem is illustrated in Listing 4. This script steps the execution until a message is about to be sent to a receiver with value nil. When the script finishes, the debugger is in the state desired by the developer: the execution is suspended in the problematic sub-expression.

```

1 | dbg stepUntil: [
2 |   dbg currentNode isMessage
3 |   and: [ dbg messageReceiver = nil ]]]

```

Listing 4. Catching message-sends to nil.

3.3 Placing Breakpoints on a Family of Methods

In this scenario, the execution opens the same file multiple times, through different file-opening methods whose names conform to a given regex pattern (e.g., `.*open.*File.*`). The developer wants to set breakpoints in all the methods that call one of these file-opening methods to open the given file (e.g., `myFile.txt`). With Sindarin, the developer writes the script shown in Listing 5. This way of expressing a location in the code using an expression is close to *pointcuts* definitions in *Aspect-Oriented Programming* [20].

Table 1. The Sindarin debugging API

Stepping	
dbg step	Executes the next instruction. If the instruction is a message-send, step inside it.
dbg stepOver	Executes over the next instruction. If the instruction is a message-send, step it until it returns.
dbg stepUntil: aPredicate	Steps the execution until the predicate is true.
dbg skipWith: obj	Skips the execution of the current instruction, and puts the object obj on the execution stack.
dbg skip	Skips the execution of the current instruction, and puts nil on the execution stack.
dbg continue	Steps the execution until a breakpoint is hit. Returns a reification of the breakpoint hit.
Stack Access	
dbg isExecutionFinished	Returns whether the debugged execution is finished.
dbg context	Returns a reification of the current stack-frame.
dbg stack	Returns a list of context objects representing the current call stack.
ctx pc	Returns the current program counter of the given context.
ctx sender	Returns the sender context of the given context.
ctx receiver	Returns the receiver of the given context.
ctx selector	Returns the selector of the given context.
ctx method	Returns the method of the given context.
ctx arguments	Returns the arguments of the given context.
ctx temporaries	Returns the temporary variable of the given context.
Stack Modification	
ctx push: aValue	Pushes aValue into the stack-frame's value stack.
ctx pop	Pops a value from the stack-frame's value stack and returns it.
AST and AST Mapping	
dbg currentNode	Returns the node that corresponds to the current program counter.
ast accept: visitor	Visits the current node using a visitor pattern.
ast is*Node	Returns true if the receiver is a node of the specified kind (for example: ast isMessageNode).
Object-Centric Debugging	
dbg haltOnCall: obj	Breaks next time the object obj receives any message.
dbg haltOnCall: obj for: m	Breaks next time the object obj receives the message m.
dbg haltOnWrite: obj	Breaks next time any instance variable of the object obj is written.
dbg haltOnWrite: obj field: iv	Breaks next time the instance variable iv of the object obj is written.
Breakpoints	
dbg setBreakpoint	Sets a breakpoint on the current node, returns an object reifying the breakpoint.
dbg setBreakpointOn: T	Sets a breakpoint on T (a node or a compiled method), returns an object reifying the breakpoint.
bp whenHit: aBlock	Defines a sequence of debugging operations to perform when a breakpoint bp is hit.
bp remove	Removes the breakpoint bp.
bp once	Configures the Breakpoint bp to remove itself the next time it is hit. Returns bp.
Stack Access Helpers	
dbg receiver	Returns the receiver of the current stack-frame.
dbg selector	Returns the selector of the current stack-frame.
dbg method	Returns the method of the current stack-frame.
dbg arguments	Returns the arguments of the current stack-frame.
dbg temporaries	Returns the temporary variables of the current stack-frame.
dbg messageReceiver	Returns the receiver of the message about to be sent, if the current node is a message node.
dbg messageSelector	Returns the selector of the message about to be sent, if the current node is a message node.
dbg messageArguments	Returns the arguments of the message about to be sent, if the current node is a message node.
dbg assignmentValue	Returns the value about to be assigned, if the current node is an assignment node.
dbg assignmentVariableName	Returns the variable name about to be assigned to, if the current node is an assignment node.

```

1 | [ dbg isExecutionFinished ] whileFalse: [
2 | (( '*.open.*File.*' match: dbg selector ) and:
3 | [ (dbg context arguments at: 1) = 'myFile.txt' ])
4 | ifTrue: [ dbg setBreakpointOn: dbg context sender method ].
5 | dbg step ]

```

Listing 5. Setting breakpoints on a family of methods.

This script steps through the whole execution (line 1). When the selector of the current method matches the pattern (line 2) and the argument of the current method is the file the developer is interested in (line 3), the script places a breakpoint in the method that called the current method (line 4). Once all the breakpoints are set, the developer can launch again his program and start the debugging session knowing she will stop each time the file `myFile.txt` is opened.

4 Expressing Advanced Breakpoints

In this section, we show how to use Sindarin to express more advanced breakpoints.

4.1 Control-Flow Breakpoints

In [5], Chern and De Volder present a breakpoint-definition language to define breakpoints based on control-flow aspects of the execution. Their motivational example is that a method `ProjectBrowser>>#trySaveAs:` that they want to debug is called by multiple other methods during the execution. They want the execution to stop in this method, but not if it is called by `ActionSaveProjectAs>>#actionPerformed:` because they identified that situation to be uninteresting.

To express this, a developer using Sindarin writes the script shown in Listing 6. This script steps the execution until the current method is the desired one (line 2) and the method of the sender context is not the undesired one (lines 3-4).

```

1 | dbg stepUntil: [
2 | (dbg method = ProjectBrowser>>#trySaveAs:) and:
3 | [ (dbg context sender method =
4 | ActionSaveProjectAs>>#actionPerformed:) not ]]

```

Listing 6. Sindarin script for a control-flow breakpoint

This scenario is slightly more complicated if the `ActionSaveProjectAs>>#actionPerformed:` method is not the direct sender but is lower in the stack. The developer wants to stop the execution in the `ProjectBrowser>>#trySaveAs:` method, but not if the `ActionSaveProjectAs>>#actionPerformed:` method is present *anywhere in the stack*. To solve this extended scenario, the developer replaces line 3 of the original script to look up the entire stack, as shown in Listing 7.

```

1 | dbg stepUntil: [
2 | (dbg method = ProjectBrowser>>#trySaveAs:) and:

```

```

3 | [ (dbg stack anySatisfy: [:ctx | ctx method =
   | ActionSaveProjectAs>>#actionPerformed:] not ]]

```

Listing 7. Sindarin script for a control-flow breakpoint (extended scenario).

4.2 Chaining Pitons

Some bugs are only reproducible deep inside an execution. Long loops and recursive structures are examples of common programming patterns that harm debugging. Debugging in these scenarios is not only time consuming, but also requires patience and discipline from the developer. On the one hand, manual stepping is error prone: an extra step can go over the cause of the bug, invalidating the debug session and forcing the developer to restart the debug session. On the other hand, a breakpoint inside a complex computation could be triggered hundreds of times before the bug actually appears.

Pitons⁴, also called stateful breakpoints [4], are a sequence of breakpoints that trigger only if they are activated in the right order. That is, given a sequence of breakpoints ($b_1, b_2 \dots b_n$), b_n will only trigger if b_{n-1} already triggered, and b_{n-1} will trigger only if b_{n-2} already triggered, and so on.

Building such a breakpoint with Sindarin boils down to a sequence of instructions to step the execution until the pitons are reached. Listing 8 shows a Sindarin script passing through pitons `method1`, `method2`,... up to `methodN`.

```

1 | dbg stepUntil: [ dbg method = ClassA>>#method1 ].
2 | dbg stepUntil: [ dbg method = ClassB>>#method2 ].
3 | ...
4 | dbg stepUntil: [ dbg method = ClassC>>#methodN ]

```

Listing 8. Sindarin script stepping through pitons.

This script can be further enhanced by extracting common behaviour into a new stepping function that receives as argument a collection of piton methods (Listing 9). The stepping function iterates over the collection of pitons and steps until it has found all of them during the execution.

```

1 | self stepThroughPitons: {ClassA>>#method1. ClassB>>
   | #method2. ... ClassC>>#methodN}.
2 |
3 | MyScript >> stepThroughPitons: anArray
4 | anArray do: [:aMethod |
5 | dbg stepUntil: [ dbg method = aMethod ]]

```

Listing 9. Refactored Sindarin script stepping through pitons.

⁴Pitons in alpinism are sticks placed one by one to secure the path of an alpinist. An alpinist carefully moves its secure rope from one piton to the next one.

4.3 Divergence Breakpoints

Introducing a minor change in the source code can cause a program to behave very differently. When facing this situation, developers are left to manually step multiple executions with and without the changes, and compare them manually [44].

The Pillar bug Scenario. Pillar⁵ is an open-source document generator. It takes a source document written in a markup syntax, and generates an output document (*e.g.*, a pdf). The Pillar test suite is made up of more than 3000 unit tests. One of the tests failed after developers introduced an instance variable and its accessor in a Pillar class named Configuration [11, 14]. This bug was difficult to solve because the symptoms of the bug (a failing test) had no relation with the modification (introducing an accessor) and its supposed impact.

Finding Execution Divergence with Sindarin. To solve this scenario, we command two executions side-by-side and step them until they diverge *i.e.*, until both programs stack-traces differ. First, we copied the Configuration class and the failing test using this class. One test (TestOriginalClass) executes the original behavior with the original class. This test passes. The second test (TestCopyClass) uses the copy class, in which we added an instance variable and its accessor method named #disabledPhases. This test fails. Listing 10 shows a script that commands the executions of the two tests until they step in different methods. Lines 1-2, we create the two debug sessions for the two tests. Line 3-4, we start a stepping loop, in which the two sessions are stepped side-by-side until the current method of both executions are different.

```

1 | dbg1 := ScriptableDebugger debug: [ TestOriginalClass run
  |   ].
2 | dbg2 := ScriptableDebugger debug: [ TestCopyClass run ].
3 | [ dbg1 method = dbg2 method ]
4 | whileFalse:[ dbg1 step. dbg2 step ]

```

Listing 10. Script to debug the two executions side-by-side.

Analysis. At first, one execution uses an instance of the original Configuration class, and the second execution uses an instance of its modified counterpart, with the initialized new instance variable and its accessor. At some point of the execution, those instances are composed with new uninitialized instances of their own class, named *sub-configuration*. When a sub-configuration, instance of the modified Configuration class, receives the #disabledPhases message, it answers an uninitialized value. When a sub-configuration, instance of the original Configuration class, receives the same message, it raises a #doesNotUnderstand: exception, which starts a hidden developer-defined lookup to retrieve the value from the original Configuration instance. We are immediately able to see

⁵<https://github.com/pillar-markup/pillar>

where the two executions diverge, and notice that adding an accessor prevents the hidden lookup to start, thus producing our bug.

4.4 Domain Specific Breakpoints

Generic debugging operations such as step or stepOver are useful because they are applicable to executions of many different domains. However, when using generic stepping operations, debugging high-level libraries and internal DSLs such as Roassal [2] or Glamour becomes a repetitive and error prone task. Indeed, the steps are too small and go through a lot of places that the developer knows to be uninteresting. Let's take for example the internal iterators of the Pharo Collections library such as #collect:⁶, whose code is shown in Listing 11. This iterator receives a block closure as argument, executes it over each element of the collection and returns a new collection with the collected elements. This iterator is defined using the #do: iterator, which applies a block closure to every element of a collection. The #do: iterator is implemented differently for each collection. When encountering a call to #collect:, the developer usually does not want to step inside the implementation of #collect: (and even less inside the one of #do:). Instead, she would like to jump through the Collection library's code to when her block closure is executed on the elements of the collection.

```

1 | "Usage example"
2 | self bigCollection collect: [ :each | each double ].
3 |
4 | "Collect code"
5 | Collection >> collect: aBlock
6 | | newCollection |
7 | newCollection := self species new.
8 | self do: [:each | newCollection add: (aBlock value: each)].
9 | ^newCollection

```

Listing 11. The #collect: internal iterator from Pharo collections.

Listing 12 illustrates how the developer writes a script that steps until the 173th iteration of a #collect: on a large collection. Line 1 steps inside the #collect:. Line 2 captures the first argument of the invoked method: the block closure. We then call repeatedly a custom stepping function #stepToNextIteration that will step until the next invocation of the block. This stepping detects the invocation of the block when the block is used as method. Also, it recognizes a *new invocation* as a context it has not seen before.

```

1 | dbg stepUntil: [ dbg method = (Collection >> #collect:) ].
2 | blockClosure := dbg arguments first.
3 |

```

⁶In Pharo, the #collect: method implements the well-known map function from functional programming.

```

4 | "step to the third iteration"
5 | 173 timesRepeat: [ self stepToNextIteration ]
6 |
7 | MyScript >> stepToNextIteration
8 | dbg stepUntil: [ :ctx | lastCtx ~~ ctx and: [ ctx method =
   |   blockClosure ] ].
9 | lastCtx := dbg currentContext

```

Listing 12. The #collect: internal iterator from Pharo collections.

5 Easing Object-Centric Debugging

Object-centric debugging focuses on debugging operations at the level of objects rather than static method representations and execution stacks [34]. For example, this includes stopping the execution when a *specific* object receives a message, or when the state of this particular object is accessed or modified.

The first step to debug an object is to find that object but object-centric debuggers do not provide dedicated means to find objects. We first give an example of application for object-centric debugging, then, we show how scripting the debugger with Sindarin eases acquiring objects for object-centric debugging.

5.1 Why Object-centric Debugging: an Illustration

Imagine *AtomViewer*, an incarnation of the legendary Self [18] graphical application, displaying lots of random small shapes called *atoms*. Such atoms can be of different shapes (squares, dots, circles, stars, etc.). Listing 13 shows the method called by the display loop for each atom:

```

1 | AtomViewer >> displayAtom: anAtom
2 | anAtom renderWith: self randomAtomDrawer
3 | CircleAtom >> renderWith: anAtomDrawer
4 | anAtomDrawer renderCircle: self
5 | TorusAtom >> renderWith: anAtomDrawer
6 | anAtomDrawer renderTorus: self
7 | SphereAtom >> renderWith: anAtomDrawer
8 | anAtomDrawer renderSphere: self

```

Listing 13. How to debug atom drawers?

Each atom is instance of the *Atom* class. Every time the display is refreshed, each atom is redrawn through the *displayAtom:* method. It takes the atom instance to display, and passes a *random* drawer object to it (line 2). This drawer knows how to display atoms, as atoms have different shapes. Each drawer uses specific display options to draw the atoms' shapes (like motion blur, glow, blend...). To that end, drawers implement different display methods, and know which method to call depending on the display options.

Imagine a bug that sometimes occurs when circular atoms that have a radius higher than 50 pixels are rendered. It is difficult to debug since we do not know which drawer is

erratic, and drawers are randomly chosen at display time. In addition, circular atoms (spheres, torus, circles...) all call a different rendering behavior from drawers (lines 4, 6, 8). To avoid putting conditional breakpoints everywhere, we want to put an object-centric breakpoint on the drawer object. We also want to set this breakpoint before the drawer is passed to an atom with a problematic radius, so that the drawer halts each time it receives a rendering message. However, we cannot easily access the drawer object. It is returned by a call to the *randomAtomDrawer* method, used immediately as a parameter to the *acceptVisitor:* message, then discarded and garbage collected.

5.2 Easing Object-centric Debugging with Sindarin

With Sindarin, we control the execution when breakpoints are hit: we navigate the execution to reach the right context, from which we extract and capture objects of interest. We apply object-centric debugging operations to those objects, then continue the execution. It gives us a systematic and automatic means to capture objects for object-centric debugging.

In Listing 14, we capture the object returned by the *self randomAtomDrawer* message in Listing 13, before it is used as parameter to the *renderWith:* method. To achieve this, we set a breakpoint on the *displayAtom:* method (line 1). When the breakpoint is hit, we first do a *stepOver*. It executes the *self randomAtomDrawer* expression and puts the result on the value stack. At this point, the debugger is about to send the *renderWith:* message, with an *Atom* as receiver, and the object that has been put on the stack as argument (the atom drawer). We use the execution introspection API to recover both objects (lines 4-5) and check conditions (line 6) before applying an object-centric breakpoint to the drawer (line 8) and removing the original breakpoint (line 9).

```

1 | bpoint := dbg setBreakpointOn: AtomViewer>>
   |   #displayAtom:.
2 | bpoint whenHit: [
3 |   dbg stepOver.
4 |   atom := dbg messageReceiver.
5 |   drawer := dbg messageArguments first.
6 |   (atom shape isCircular and: [atom shape radius > 50])
7 |   ifTrue: [
8 |     dbg haltOnCallTo: drawer.
9 |     bpoint remove].
10 | dbg continue ]

```

Listing 14. Capture example: capturing an atom drawer.

5.3 Replaying Objects

Bugs can be hard to reproduce when programs contain non-deterministic aspects. For instance, in our example from Listing 13, the drawer object is chosen randomly. To replay the

bug, we re-inject the problematic drawer in the control flow using the `skipWith:` operation (illustrated in Figure 1).

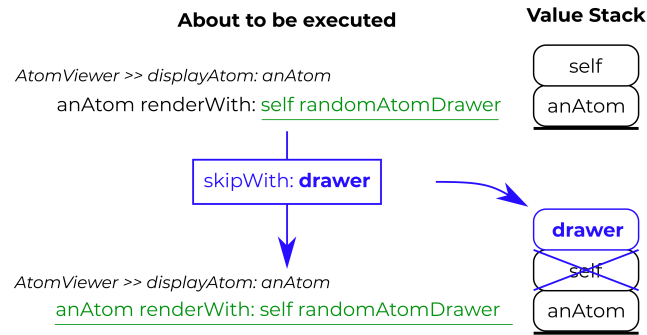


Figure 1. Removing non-determinism: `skipWith: anObject` steps over a node about to be executed (underlined) but without executing it, and pushes anObject on the stack instead.

We show in Listing 15 how to achieve this with Sindarin. Each time we break, we capture the node whose execution creates a random drawer (line 3), and the drawer it returns (line 4-5). If conditions for replay are satisfied (line 6), we remove the original breakpoint and configure a replay breakpoint on the drawer node (line 8). When this replay breakpoint is hit (line 9), we instruct the debugger to skip over the current node (the drawer creation node) and use the captured drawer object instead (line 10).

```

1 | breakpoint := dbg setBreakpointOn: AtomViewer>>
   | #displayAtom:.
2 | breakpoint whenHit: [
3 | drawerNode := dbg currentNode.
4 | dbg stepOver.
5 | drawer := dbg messageArguments first.
6 | "condition for replay" ifTrue: [
7 | breakpoint remove.
8 | replayPoint := dbg setBreakpointOn: drawerNode.
9 | replayPoint whenHit: [
10 | dbg skipWith: drawer.
11 | dbg continue]].
12 | dbg continue]

```

Listing 15. Replaying an atom drawer after an exception

6 Evaluation

We have already shown throughout the paper that Sindarin is a versatile scripting DSL that covers different debugging scenarios from the literature. In this section we Sindarin with respect to mainstream debugger APIs such as GDB or JDI. We first identify the different features required by Sindarin to implement each debugging scenario described in this paper. We then evaluate how those features are supported in several mainstream debuggers with debugger APIs.

6.1 Sindarin Debugger Requirements

Each of the debugging scenarios used throughout this paper requires a different set of features from Sindarin. We analyzed and categorized those requirements into three different axes:

Step Granularity. The granularity required for the step operation *i.e.*, the minimal amount of code that is executed between two step operations. In our debugging scenarios, we observed the need for two different granularities: a fine-grained *expression* granularity and a coarse-grained *method* granularity.

Context Access. The access provided to the execution stack. We have categorized context access in two main operations. *Basic access* refers to the access to the list of stack frames and their methods, receiver and argument objects, and their temporary variables. *Full access* refers to the access to the intermediate values of expressions during the execution (*Stack access*), and the ability to modify the current stack to *e.g.*, pop or push a value into it (*Stack modification*).

AST Mappings. Whether the scenario requires AST mappings or not to identify code of interest dynamically.

Table 2 summarizes our analysis for each of our debugging scenarios. We observed that several scenarios require expression step granularity to operate at the level of expressions instead of methods. All scenarios requiring expression step granularity also require AST mappings to identify the interesting expressions, and stack access to access intermediate results of those expressions stored in the stack. For instance, although it is possible, to some extent, to implement object-centric breakpoints with those debuggers, the object capture scenario is mandatory for their application. This scenario is one example requiring stack access to access intermediate results. Finally, replaying objects is the only scenario in our scenario collection to require stack modification. Replaying an object instead of an expression requires to skip the expression and push the object to replay in the stack.

6.2 Comparison with Debugger APIs

Several debugger implementations provide an API or protocol to connect to it and drive the debugging of a program. Examples of these are GDB [35] and Java's debugger interface through JDI and JVMTI [19, 30]. In this section we take several mainstream debuggers and compare their APIs to Sindarin's, to analyze which debugging scenarios are applicable to them.

The summary of our analysis is reflected in Table 3. We identify five families of debugger APIs: (1) Unmanaged-code debuggers (UCD) covers debuggers for non-managed runtimes, for which we analyzed GDB, LLDB, WinDBG, (2) Python's basic debugger functions (BDB) [8, 16], (3) NodeJS debugging API, (4) Java's debugger interface through JDI and JVMTI [19, 30] and (5) Microsoft's CLR EnvDTE debugger interface.

Table 2. Feature Requirements of the Debugging Scenarios

Scenario	Step Granularity	Context Access	AST Mapping
Monitoring Assignments to a Variable	Expression	Stack Access	✓
Stopping before an Exception	Expression	Stack Access	✓
Placing Breakpoints on a Family of Methods	Method	Basic Access	✗
Chaining Pitons	Method	Basic Access	✗
Control-Flow Breakpoints	Method	Basic Access	✗
Divergence Breakpoints	Method	Basic Access	✗
DSL Stepping	Method	Basic Access	✗
Capturing Objects	Expression	Stack Access	✓
Replaying Objects	Expression	Stack Modification	✓

Table 3. Mainstream Debugging APIs.

Feature \ API	UCD ¹	BDB	NodeJS	JDI	CLR ⁴	Sindarin
Step Granularity	line ²	expression	line	line	line	expression
Context Access	full ³	basic	basic	basic	basic	full
AST Mapping	✗	✗	✗	✗	✗	✓

(1) Unmanaged Code Debuggers: GDB, WINDBG, LLDB.

(2) A line stepping operation at the machine instruction level is available.

It matches the source code, but developers must know when to start and stop stepping to achieve an expression level granularity, as many assembly instructions/bytecodes can be related to a single instruction in source code.

(3) Possible by reading and writing directly into the memory of the program.

(4) Common Language Runtime debuggers (C#, J#, .NET).

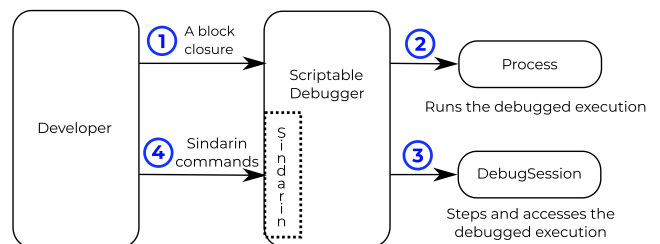
From our analysis we observe that most debuggers provide a rather coarse grained granularity for stepping of an entire line of code, which prevents a straight forward implementation of the scenarios discussed in this paper. Indeed, this granularity does not allow developers to differentiate between different expressions (and sub-expressions) within a single line. Unmanaged code debuggers, on the other hand, provide by default a step operation that operates at the line level while a second stepping operation does it at the machine instruction level.

Another point of divergence is the fact that not all debugger APIs allow full stack manipulation, preventing advanced scenarios such as object-replay. We consider that unmanaged code debuggers support full stack manipulation since they can modify the execution stack and the objects pointed by it as any other memory region.

Finally, no mainstream debugger provides high level mappings between instructions and AST such as Sindarin's AST mappings.

7 Implementation

We implemented Sindarin for the Pharo programming language. In this section we briefly describe the infrastructure of our implementation, how it integrates within the Integrated Development Environment, how it could be integrated into mainstream software development, as well as its limitations.

**Figure 2.** Creation process of a scriptable debugger instance

7.1 Infrastructure

Starting a Sindarin session. Sindarin drives a scriptable debugger instance and configures it for a dedicated piece of code

that will be executed. Figure 2 shows the creation process of a scriptable debugger instance. The developer first provides a block closure of the code she wants to debug (1). The scriptable debugger then creates a process to run the closure (2), and a DebugSession to step and access the execution being debugged (3). Finally, the developer sends commands written using the Sindarin API to the scriptable debugger to control and/or inspect the debugged execution (4).

Contexts. The contexts exposed by our Sindarin implementation are of the same class as the ones returned by the Smalltalk thisContext pseudo-variable.

Step Granularity. Our implementation of Sindarin runs in Pharo. Pharo code is compiled into bytecode, that is interpreted by a virtual machine. Through reflection, Pharo programs can access the bytecode interpreter of the virtual machine and step it to progress a frozen execution. The dbg step operation of Sindarin is implemented using this feature to provide a stepping granularity at the expression level.

Context Access. Sindarin provides full access to the contexts of the debugged execution through instances of the Context class. Context objects are automatically reified by the Pharo virtual machine when needed for reflection purposes.

Mapping AST to bytecode. The Pharo compiler keeps a mapping between the AST and the bytecode, as well as a mapping between the AST and the source code. By default, the granularity of each step is at the level of expressions. Sindarin makes use of this infrastructure to provide fine grained control over stepping, without losing the bytecode abstraction. This granularity eases the ability to set breakpoints inside statements. The direct access and handling of the AST also enables to express complex conditions for breakpoints.

Breakpoints. The imperative design of Sindarin means that users will often set breakpoints on the AST node the debugged execution is currently at. Typical implementation of breakpoints modify the bytecode of the method to insert the breakpoint behaviour. This approach does not fit the use-case of Sindarin, because modifying methods that are already on the call stack is hard [40]. To go around this limitation, the breakpoints set by Sindarin are "virtual" in the sense that no code modification is performed. Instead, Sindarin remembers the AST nodes on which breakpoints were set, and stops debugged execution that would step through these nodes.

7.2 IDE Integration

We modified the Pharo debugger to make it scriptable through the Sindarin API. The developer writes Sindarin scripts in an editor or directly in the debugger (Figure 3). When scripts are run in the debugger, the dbg variable is automatically bound to the current debugger.

7.3 Integration into Mainstream Software Development

We highlighted a few existing debugger APIs from mainstream languages in section 6.2. These could be used as a basis to implement the Sindarin API in these languages. The three main challenges of such implementations are those outlined in section 6.2: step granularity, context access and AST mapping.

Implementing expression-level step granularity in the languages that do not natively support it can be achieved with automatic program rewriting [22], to rewrite the expressions over multiple lines, or AOP weaving [42].

From Table 3, UCD already provides full context access through direct memory access. For Python, Python code can interact in an indirect way with the Python virtual machine to access/modify the context stack of the execution. An alternative method is probably achievable for Java. Of note is that these solutions require significant engineering effort.

AST mapping is not present in any of the debugging APIs we considered. It could maybe be achieved by modifying the compiler of the host language to keep the AST that is generated during the compilation, and build a binding between the bytecode generated by the compiler and this AST.

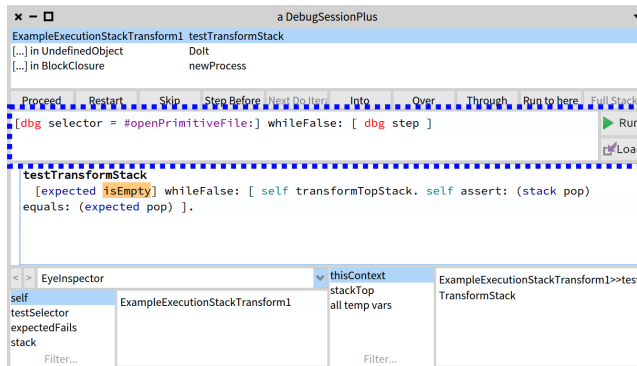


Figure 3. Command line implementing Sindarin integrated into the Pharo debugger

7.4 Limitations

A limitation of Sindarin is its low-level aspect. It offers a lot of possibilities, but lacks a higher-level layer that would offer a quick and expressive way for developers to express higher-level debugging operations. This limitation is mitigated by the ability to write and re-use debugging scripts. Another limitation is that effectively using Sindarin to debug requires understanding and manipulating the concepts of contexts, AST nodes and stack.

8 Related work

We present in the following related work on debugger customization and means to control debugging sessions.

8.1 Domain Specific Debugging Operations

Kompos [29] is a concurrency-agnostic debugger protocol, which decouples the debugger from the concurrency models employed by the target application. As a result, the underlying language run-time can define custom breakpoints, stepping operations, and execution events for each concurrency model it supports, and a debugger can expose them without having to be specifically adapted. With Sindarin, it is possible to place standard breakpoints and to step in concurrent code. We plan to introduce additional support for concurrent debugging, including specific breakpoints like Kompos'.

The Moldable Debugger framework [6] allows developers to create domain-specific debuggers, by defining and combining domain-specific debugging operations and views. The moldable debugger adapts itself to a domain by selecting at run-time the appropriate appropriate debugging operations and views. Debugging operations executes the program until a debugging predicate is matched, or performs an action every time a debugging predicate is matched. Debugging predicates are either primitive predicates like attribute reads or method calls, or combinations of these. A key difference with Sindarin is the "immediacy". With Sindarin, developers write a script on the fly to help with the particular bug under investigation. By contrast, the moldable debugger framework is geared towards investing time into creating a debugger able to assist the debugging of a specific domain, rather than a specific bug. Both approaches are not incompatible, we believe that Sindarin can be used to build moldable debuggers and to script their specialized parts.

Expositor [32], inspired by the work on MzTake [28], combines scripting and time-travel debugging. The fundamental abstraction provided by Expositor is the execution trace, which is a time-indexed sequence of program state snapshots. Developers can manipulate traces as if they were simple lists with operations such as map and filter. From that perspective Expositor has a conceptual interface close to the Pharo `thisContext` pseudo-variable that reifies stack on demand.

Barr *et al.*, describe a time-traveling debugger for JavaScript/Node.js [1]. In particular, they describe three time-travel operations: *reverse-step* (rs) to step to the previously executed line in the *current* stack frame, *reverse-step dynamic* (rsd) to step back in time to the previously executed statement in the *any* frame, and *reverse to callback origin* (rcbo): a more specific operation to step back from the currently executing callback to the point in time when the callback was registered. These three operations are good candidates to extend the Sindarin API with time-traveling capabilities.

CBD is a Control-flow Breakpoint Debugger [5]. CBD uses a dynamic pointcut language to characterise control-flow breakpoints. These breakpoints are conditions on the control-flow, through which they were reached. As such, CBD leverages *aspect-oriented programming* to assist debugging operations. The difference between CBD and Sindarin is

the versatility. CBD is specialised in expressing control-flow breakpoints, and as a result expresses them in a concise fashion. Sindarin allows developers to express the same breakpoints as CBD, and to combine these breakpoints with other features such as object-centric debugging.

Bugloo [7] is a source level debugger for Scheme programs, that are compiled into JVM bytecode by the Bigloo compiler. Bugloo traces methods that are entered, and the source code location where a given variable is read/written. The debugging commands are recorded, which allows developers to replay their debugging sessions. A command line is available for interactive debugger control.

Other solutions focus on exposing bugs through execution traces. PTQL [17] is a query language, through which developers express what part of their program they want to trace. Queries are automatically translated to instrumentation. ange *et al.*, [23] combines Prolog-like queries on program traces and visualization to understand program execution. Working on traces implies a post-mortem approach, such works do not support controlling step by step execution of the debugged program, the precise queries of an execution moment nor object-centric facilities.

8.2 Querying Objects

Finding objects of interest for debugging is difficult. Using traditional debugging tools, developers have to put breakpoints to halt the execution and manually select objects [10, 34]. This approach requires much interaction from developers and forbids any automation, *e.g.* breakpoints in loops break the execution at each loop iteration.

Developers need systematic ways to obtain objects, for object-centric debugging or to provide more details about the debugged execution. Fox [33] is a language to perform queries over the object graph of an execution snapshot. Bugloo [7] provides a heap inspector allows some access to the object graph of the execution. *Query-based debugging* [25, 26] refers to debuggers which build collections of objects from user-defined queries, written in a dedicated query language. *Reactive object queries* [24] are user-defined requests on a program, which results create and maintain dynamic collections of objects. Both solutions automatically search the object space to update their collections of objects. However they cannot express conditions regarding the context from which an object is obtained [12]. These solutions are good candidates for extending Sindarin. Sindarin allows the developer to bring executions to points of interest, from where developers could perform queries over the object space.

8.3 Aspect-Oriented Programming

Aspect-oriented programming (AOP) [9, 15, 20] is a programming paradigm where the developer can add cross-cutting behavior (an "advice") to existing code without modifying the latter, by specifying which code is modified via a "pointcut" specification. AOP can be used for debugging purposes [42] and is able to solve some of the scenarios we described in

this paper. AOP cannot directly tackle the Divergence Breakpoints scenario, as it requires two executions to be ran side-by-side. The Capturing Objects and Replaying Objects scenarios could be implemented with AOP. This would however require additional work to define a model for storing and re-using objects across executions. AOP differs from Sindarin in the way debugging operations are expressed. Sindarin is imperative while AOP is declarative. The imperative paradigm is closer to the interactions developers have with mainstream debuggers: for example repeatedly clicking on the "step" button until the desired state is reached. We could use AOP to implement the Sindarin API in other languages. We leave as future work a comparison between the imperative (Sindarin) and declarative (AOP) styles in terms of learning curve and convenience for developers.

9 Discussion

In this section, we discuss a few points of note about Sindarin, namely the advantages of implementing it as an internal DSL, the ability and limits of debugging a Sindarin implementation with itself, and ideas about how Sindarin scripts written for debugging purposes could be leveraged as persistent artifacts.

9.1 Internal DSL

Our implementation of Sindarin is an internal DSL of Pharo. This brings advantages in terms of usability, as a developer debugging Pharo code will already be familiar with the general syntax of Sindarin scripts (e.g. how to define a variable, block closures...) and have access to the standard libraries. In addition, this makes Sindarin directly usable within the Pharo IDE, with no additional integration work.

9.2 Self-debugging

Since Sindarin is an internal DSL, a debugging session using Sindarin is a program execution in the same host language. As such, Sindarin can be used to debug itself. This is in contrast to other debugging tools that cannot debug themselves. For example, Aspect-oriented programming is unable to debug itself in this fashion, as aspects cannot be added to other aspects [42]. A limit to the self-debugging ability of Sindarin is when a core feature is impacted, for example the step operation itself. Indeed, if step itself does not work, the buggy execution of step cannot be stepped to be debugged. This limitations echoes that of Kansas [36]: a reflective system where developers interact with objects in a world. When a Kansas world is broken, another one is created from which the first world can be repaired. However, when a core feature like the ability to create new worlds is broken, Kansas cannot be debugged from within itself.

9.3 Artifact Generation

By debugging with Sindarin, a developer generates debugging scripts. These scripts constitute valuable artifacts that can be harvested and used for other purposes. For example, a script written to reach a certain point in the execution and check a property on a variable can be turned into a test. A script

defining a particular domain-specific stepping operation (like in our DSL Stepping scenario) can be kept to assist future developers in debugging the same section of the code. A script written to reach a point in the execution that is relevant for a given bug can be shared on the issue tracker. This allows other developers to immediately reach the relevant point without requiring a tedious textual description of which buttons to click in the debugger. These alternative, long-term uses for Sindarin scripts give additional value to the time spent by developers to write them during debugging sessions.

Additionally, debugging scripts could be automatically generated from a developer's interaction with a mainstream debugger. For example the sequence of buttons she clicked.

10 Conclusion

In this paper we presented Sindarin, a versatile API for scripting an online debugger. Sindarin supports the definition of advanced stepping operations: It offers different interfaces such as AST or runtime stack introspection letting the developer use the abstractions she needs. Sindarin supports object-reachability in the sense that all the objects accessed during execution can be manipulated. This eases the construction of object-centric debugging scripts. Sindarin will be shipped with Pharo 80 as a central part of the new debugging solution.

As a future work we will add support for concurrent programming debugging, back-in-time debugging, and provide ways to support the debugging based on multiple execution stacks. In addition, we would like to perform a user-study with experimented developers to assess multiple characteristics of Sindarin like its learning-curve and how much it helps with debugging scenarios that are impractical to tackle without the support of specialized debugging tools. This study could also compare Sindarin to aspect-oriented-programming techniques on these two points to evaluate the difference between the imperative and declarative styles of debugging.

References

- [1] E. Barr, M. Marron, E. Maurer, D. Moseley, and G. Seth. Time-travel debugging for javascript/node.js. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium*, pages 1003–1007, nov 2016.
- [2] A. Bergel. *Agile Visualization*. LULU Press, 2016.
- [3] A. Bergel, F. Banados, R. Robbes, and D. Röthlisberger. Spy: A flexible code profiling framework. *Journal of Computer Languages, Systems and Structures*, 38(1), Dec. 2011.
- [4] E. Bodden. Stateful breakpoints: A practical approach to defining parameterized runtime monitors. In *ESEC/FSE'11*, 2011.
- [5] R. Chern and K. De Volder. Debugging with control-flow breakpoints. In *Proceedings of the 6th International Conference on Aspect-oriented Software Development, AOSD '07*. ACM, 2007.
- [6] A. Chis, M. Denker, T. Girba, and O. Nierstrasz. Practical domain-specific debuggers using the moldable debugger framework. *Journal of Computer Languages, Systems and Structures*, 44:89–113, 2015.
- [7] D. Ciabrini and M. Serrano. Bugloo: A source level debugger for scheme programs compiled into jvm bytecode. In *Proceedings of the International Lisp Conference 2003*, oct 2003.

- [8] A. L. Coetzee. *Combining reverse debugging and live programming towards visual thinking in computer programming*. PhD thesis, Stellenbosch University, 2015.
- [9] A. Colyer and C. A. Aspect-oriented programming with aspectj. *IBM Systems Journal*, 44(2):301–308, 2005.
- [10] C. Corrodi. Towards efficient object-centric debugging with declarative breakpoints. In *SATToSE 2016*, 2016.
- [11] S. Costiou. *Unanticipated behavior adaptation : application to the debugging of running programs*. Theses, Université de Bretagne occidentale - Brest, Nov. 2018.
- [12] S. Costiou, M. Kerboeuf, A. Plantec, and M. Denker. Collectors. In *PX'18 - Programming Experience 2018*, Companion of the 2nd International Conference on Art, Science, and Engineering of Programming, page 9, Nice, France, Apr. 2018. ACM Press.
- [13] S. Ducasse, D. Zagidulin, N. Hess, D. C. O. written by A. Black, S. Ducasse, O. Nierstrasz, D. P. with D. Cassou, and M. Denker. *Pharo by Example 5*. Square Bracket Associates, 2017.
- [14] T. Dupriez, G. Polito, and S. Ducasse. Analysis and exploration for new generation debuggers. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies, IWST '17*, pages 5:1–5:6, New York, NY, USA, 2017. ACM.
- [15] J. Fabry and D. Galdames. Phantom: a modern aspect language for pharo smalltalk. *Software: Practice and Experience*, 2012.
- [16] P. S. Foundation. bdb - debugger framework. <https://docs.python.org/3/library/bdb.html>, 2019.
- [17] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 385–402, New York, NY, USA, 2005. ACM Press.
- [18] U. Hölzle, B.-W. Chang, C. Chambers, and D. Ungar. *The SELF Manual*. Computer Systems Laboratory of Stanford University, 1991.
- [19] Sun microsystems, inc. JVM tool interface (JVMTI). <http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/>.
- [20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP '97*, volume 1241 of *LNCS*, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.
- [21] A. J. Ko and B. A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the International Conference on Software Engineering, ICSE 08*, 2008.
- [22] I. Kume, E. Shibayama, M. Nakamura, and N. Nitta. Cutting java expressions into lines for detecting their evaluation at runtime. In *Proceedings of the International Conference on Geoinformatics and Data Analysis, ICGDA 2019*, pages 37–46, New York, NY, USA, 2019. ACM.
- [23] D. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 342–357, New York NY, 1995. ACM Press.
- [24] S. Lehmann, T. Felgentreff, J. Lincke, P. Rein, and R. Hirschfeld. Reactive object queries. In *Constrained and Reactive Objects Workshop (CROW)*, 2016.
- [25] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *OOPSLA'97*, pages 304–317, 1997.
- [26] R. Lencevicius, U. Hölzle, and A. K. 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.
- [27] H. Lieberman. Introduction. *Commun. ACM*, 40(4):26–29, Apr. 1997.
- [28] G. Marceau, G. H. Cooper, S. Krishnamurthi, and S. P. Reiss. A dataflow language for scriptable debugging. In *IEEE International Conference on Automated Software Engineering (ASE'04)*, 2004.
- [29] S. Marr, C. Lopez, D. Aumayr, E. Gonzalez Boix, and H. Mossenbock. Kompos: A platform for debugging complex concurrent applications. In *Programming'17*, pages 1–2, apr 2017.
- [30] Oracle. Java debug interface (jdi). <http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html>, 2013.
- [31] M. Perscheid, B. Siegmund, M. Taeumel, and R. Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25(1):83–110, 2017.
- [32] K. Y. Phang, J. S. Foster, and M. Hicks. Expositor: Scriptable time-travel debugging with first-class traces. In *International Conference on Software Engineering (ICSE)*, pages 352–361, may 2013.
- [33] A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. In *Proceedings of the Australian Software Engineering Conference (ASWEC'04)*, page 251. IEEE Computer Society, 2004.
- [34] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceeding of the International Conference on Software Engineering, 2012*.
- [35] S. S. Richard Stallman, Roland Pesch. *Debugging with GDB*. Gnu Press, 2003.
- [36] R. B. Smith, M. Wolczko, and D. Ungar. From kansas to oz: collaborative debugging when a shared world breaks. *Commun. ACM*, 40(4):72–78, Apr. 1997.
- [37] D. Spinellis. Modern debugging: The art of finding a needle in a haystack. *Commun. ACM*, 61(11):124–134, Oct. 2018.
- [38] G. Tassej. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 2002.
- [39] M. Telles and Y. Hsieh. *The science of debugging*. Coriolis Group Books, 2001.
- [40] P. Tesone, G. Polito, L. Fabresse, N. Bouraqadi, and S. Ducasse. Dynamic software update from development to production. *Journal of Object Technology*, 2018.
- [41] I. Vessey. Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Transactions on Systems, Man, and Cybernetics*, 16, 1986.
- [42] H. Yin. *Defusing the Debugging Scandal - Dedicated Debugging Technologies for Advanced Dispatching Languages*. PhD thesis, University of Twente, Dec. 2013.
- [43] A. Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.
- [44] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, SE-28(2):183–200, Feb. 2002.
- [45] C. Zhang, D. Yan, J. Zhao, C. Yuting, and S. Yang. Bpgen: An automated breakpoint generator for debugging. In *ICSE '10*, 2010.