First Infrastructure and Experimentation in Echo-debugging

Thomas Dupriez, Steven Costiou, Stéphane Ducasse

First Infrastructure and Experimentation in Echo-debugging

Thomas Dupriez
tdupriez@ens-paris-saclay.fr
Univ. Lille, CNRS, Centrale Lille, Inria
UMR 9189 – CRIStAL
Lille, France

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

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

Abstract

As applications get developed, bugs inevitably get introduced. Often, it is unclear why a given code change introduced a given bug. To find this causal relation and more effectively debug, developers can leverage the existence of a previous version of the code, without the bug. But traditional debugging tools are not designed for this type of work, making this operation tedious. In this article, we propose as exploratory work the echo-debugger, a tool to debug two different executions in parallel, and the *Convergence Divergence Mapping* (CDM) algorithm to locate all the control-flow divergences and convergences of these executions. In this exploratory work, we present the architecture of the tool and a scenario to solve a non trivial bug.

ACM Reference Format:
Thomas Dupriez, Steven Costiou, and Stéphane Ducasse. 2020. First Infrastructure and Experimentation in Echo-debugging. In *IWST20: International Workshop on Smalltalk Technologies, September 29th and 30th, 2020, Novi Sad, Serbia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

Nowadays, debugging is still a challenge [19, 24] and sources of hard bugs are numerous [15]. In addition, the distance between a source code change and the emergence (identification) of a bug can be large, which makes it difficult to understand why a given code change caused a given bug [22]. However, in some instances, developers have access to an interesting source of information to help them: a previous version of the software not exhibiting the bug [21].

However, having a reference, working, version of the program is not a panacea. Without dedicated support, developers have to run the two versions in separate debuggers, manually step them in parallel, and visually compare the executions.

Techniques that compare two similar executions to produce various results already exist [21]. In general, these techniques try to isolate the code fragments that are (suspected to be) responsible of an error. *Delta debugging* [22, 23] takes two versions of a program and finds the smallest subset of code change that turned a given test from green to red. *Algorithmic debugging* [17, 18] tries to isolate faulty code based on how developers assert the outputs of faulty and successful executions.

However, these approaches show limits in two scenarios. First, we might know exactly which code change introduced the bug and still we cannot understand how it did so. Second, when we migrate an application from a version of a library/framework to another, the code changes can be gigantic. Detecting code differences between a working execution (using the old version) and a failing execution (using the new version) might not be useful. The meaning itself of the code might have changed, things might have been added and others removed. For instance, when migrating frameworks from a version of Pharo [3] to another, the base classes and tools of the language regularly evolve.

In this paper we present *Echo-debugging*: a technique to compare the executions of the failing and working version of the program and find the control-flow differences to help developers debug the program. The contributions of this paper are:

- The echo-debugger and its architecture: an interactive debugger to debug two similar executions running in different runtimes.
- *Convergence Divergence Mapping* (CDM), an algorithm that fully runs both executions and compares the AST nodes they are executing to build a map of when they diverge and converge in terms of control-flow. The echo-debugger can then jump the executions to any event of this map the developer wants to inspect.

In this paper, we first state our problem of comparing two similar executions, and list the main challenges it involves (Section 2). We then expose our solution: the echo-debugger, its architecture, and the CDM algorithm (Section 3 and 4). We show a concrete example of how to use the echo-debugger to debug a bug in the Pillar editorial chain code (Section 5).

IWST20, September 29th and 30th, 2020, Novi Sad, Serbia

We finally discuss our solution (Section 6), similar works (Section 7), future works (Section 8) and conclude (Section 9).

2 Comparing Two Similar Executions

Problem statement. We have as inputs:

- Two versions of a program. For example before and after a given commit.
- A statement to execute. The developer is interested in how the execution of this statement differs between the two program versions. This will typically be a test that passes in one program version and fails in the other, but it can be any statement.

From these inputs, we want a tool that allows the developer to debug both executions of the statement in a comparative fashion, and to understand the impact of the source code differences between the program version.

Difficulties. Here, we list the main challenges our solution has to overcome.

- **Challenge 1: Running two versions of the same program in parallel, and controlling them.** Our solution requires the two version of the same program to run in parallel. This is typically not possible in the same runtime. Additionally, our solution needs to control and coordinate the two executions.
- **Challenge 2: Comparing objects across executions.** Although the executions are similar, and they create and manipulate similar objects, the default identity operator (==) is entirely unusable because the same objects from different executions are never going to be the same identity-wise.
- **Challenge 3: Comparing control-flows.** The intuitive idea is to find when the executions are *doing different things* and when they are *doing the same thing*. Our solution needs to define these expressions and use these definitions to compare the control-flows of the two executions.
  - **Challenge 3.1: Finding control-flow divergences.** Since the executions both start on the same statement (the one provided by the developer), their control-flows are the same. Our solution needs to step them until their control-flows diverge.
  - **Challenge 3.2: Finding control-flow convergences.** Challenge 3.1 lets us find the first control-flow divergence, but that may not be good enough to understand the bug. Maybe the control-flow of the executions reconverge on a part of the program, and diverge again later. Our solution needs to recognise if the control-flow of the executions reconverge after a divergence.

  Combining challenge 3.1 and 3.2 means building a map of when the control-flows of the two executions diverge, converge, diverge again, converge again...

3 The Echo-Debugger

In this section, we describe our solution to debug two similar executions side-by-side: the *echo-debugger* and its architecture.

For clarity in this section, we assume that the developer is debugging a test, which passes in a given version of the program, but fails in another. In general the echo-debugger works to comparatively debug any statement.

3.1 Echo-Debugging Architecture

Figure 1 shows the overall architecture of an echo-debugging setup.

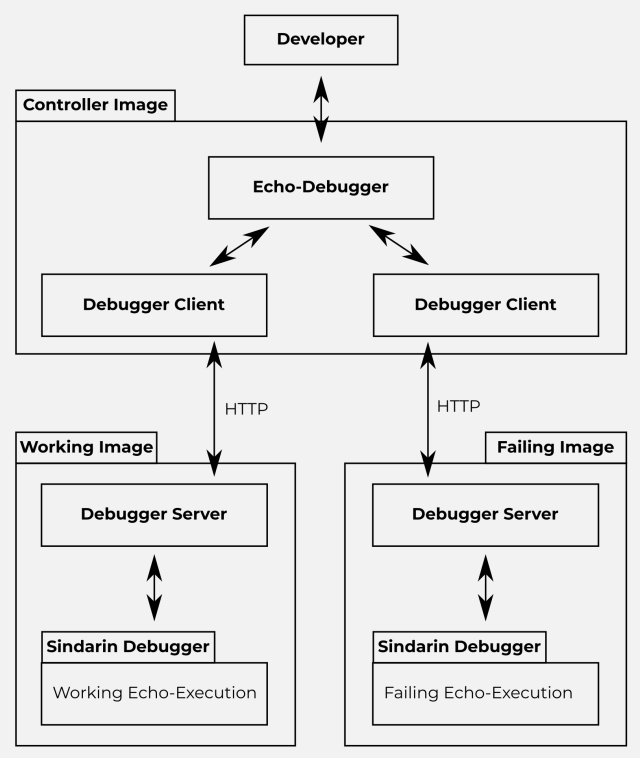


Figure 1. Echo-Debugging Architecture: One controller runtime (image) controls the execution of a failing and working one.

Three Different Runtimes. Because a runtime cannot contain and execute multiple versions of the same code at the same time (challenge 1), the Echo-Debugging architecture is made of three runtimes each one running separately. Each of such runtime runs a different configuration:

- **Working runtime.** This runtime contains the version of the code that works as expected by the developer. For concision, we will call it the *W runtime*.
- **Failing runtime.** This runtime contains the version of the code that *does not* work as expected by the developer. For concision, we will call it the *F runtime*.

- **Controller runtime.** This runtime connects to the other two runtimes to control the executions and collect data. The developer interacts primarily with this runtime during the echo-debugging session. For concision, we will call it the *C runtime*.

We refer to the working and failing runtime as *echo-runtimes*, because they are like echoes of each others: similar, but not exactly the same.

Sindarin Debugger. In each echo-runtime, we use a Sindarin debugger [7] to control the execution of the test. Sindarin is a scriptable, UI-less debugger for Pharo. It can be instantiated on an execution, and its API used to inspect and manipulate the execution.

Debugger Client/Server. For the communications between the echo-runtimes and the controller runtime, the echo-debugger has a companion package with an HTTP-based client/server communication layer. This layer transmits the Sindarin commands coming from the echo-debugger to the Sindarin debuggers in the echo-runtimes, and transmits back the answers. Some objects returned by the Sindarin API cannot be serialized/materialized, such as Contexts and Exceptions, because they reference objects that cannot be serialized. We built custom serializations for them, where we instead serialize a dictionary containing the relevant fields of these objects, excluding the unserializable ones.

Echo-Debugger. The echo-debugger is what the developer interacts with. It communicates with the Sindarin debuggers in the echo-runtimes via the client/server communication layer. For a more detailed description of the echo-debugger, see Section 3.2.

Setup process for an echo-debugging session. Finally, here is the list of steps required to setup an echo-debugging session.

1. Create three runtimes: *Working*, *Failing* and *Controller*.
2. Load the working version of the code in the working runtime.
3. Load the failing version of the code in the failing runtime.
4. Load the echo-debugger and its communication package[1] in all three runtimes.
5. In both echo-runtimes, instantiate a Sindarin debugger [7] on the execution of the test.
6. In both echo-runtimes, run a debugger server for the Sindarin debugger.
7. In the controller image, run a debugger client, connect it to both debugger servers over HTTP, and open its UI.

3.2 The Echo-Debugger

The echo-debugger is responsible for controlling and analyzing both echo-executions. Once the echo-debugging session is setup, the developer only interacts with the echo-debugger, and not directly with the echo-runtimes.

Figure 2 shows the UI of the echo-debugger. It contains three main zones (from left to right):

- A debugger on the *working* execution of the test.
- A debugger on the *failing* execution of the test.
- The *control zone* containing information and commands specific to the echo-debugger.

The control zone is separated in three areas:

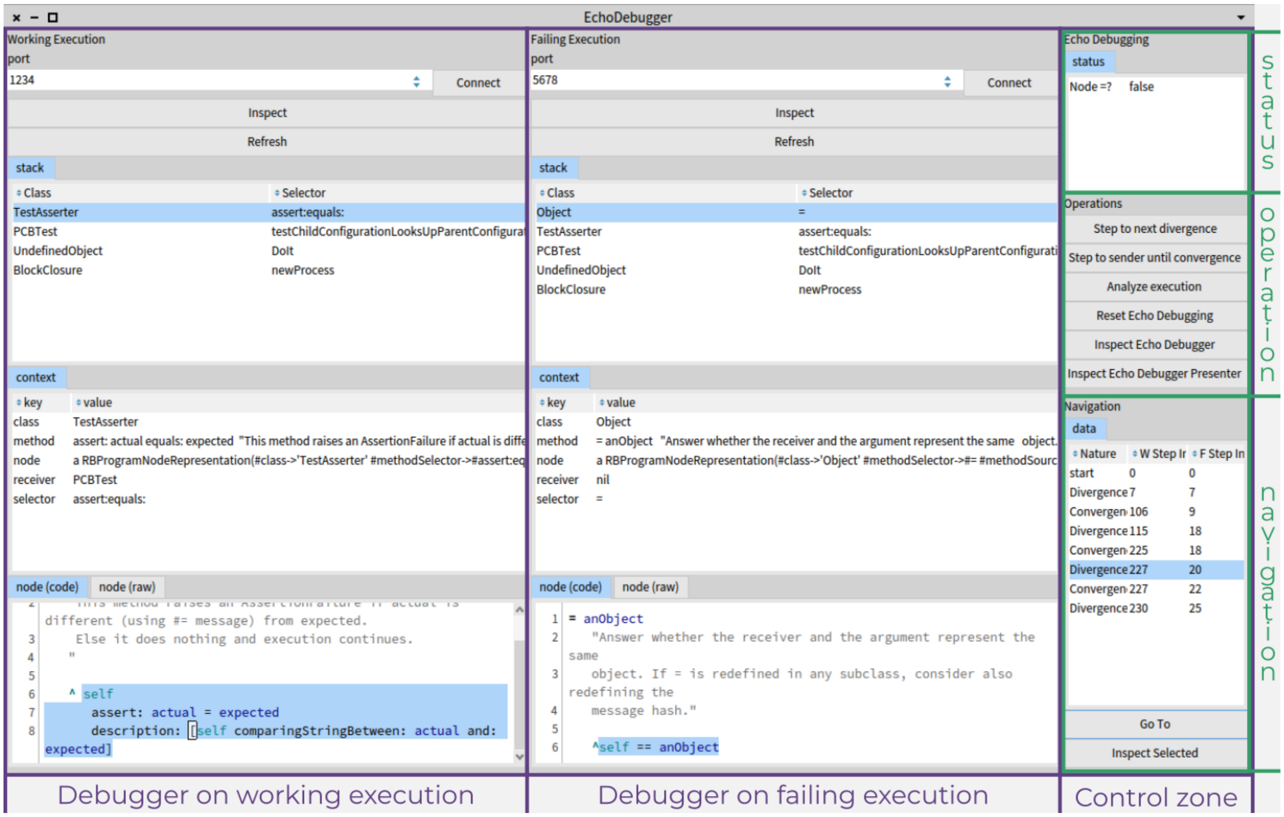
- The *status area* takes the current AST node of the two contexts selected in the debuggers and shows whether they are equal or not.
- The *operations area* lists the echo-debugging operations the developer can perform.
- The *navigation map* lists the convergence and divergence events between the echo-executions, and allows the developer to step both debuggers to when these events happened in the echo-executions.

Remote debuggers. The echo-debugger features a remote debugger for each echo-execution. These debuggers display information on the echo-executions, such as the call stack and the current piece of code being executed. The developer can use these debuggers to debug the echo-executions as he would normally debug in a standard debugger, with the added benefit of having both executions side-by-side in the same image.

Echo-debugging operations. The echo-debugger provides operations to control both echo-executions at the same time and step them to potential places of interest:

- *Step both.* Step both echo-executions once.
- *Step to next divergence.* To be used when the echo-executions are currently convergent. Step both echo-executions until their next divergence. See Section 4 about the CDM algorithm for more details.
- *Step to next convergence.* To be used when the echo-executions are currently divergent. Step both echo-executions until their next convergence. See Section 4 about the CDM algorithm for more details.
- *Analyze executions.* Applies the CDM algorithm described in Section 4 to populate the navigation map with all the convergence and divergence events between the echo-executions.
- *Restart.* Restarts both echo-executions, to start over.
- *Go to.* This operation requires that the navigation map has been populated by analyzing the echo-executions with the CDM algorithm (Section 4). This operation restarts both echo-executions and steps them until they reach the convergence/divergence event that is currently selected in the navigation map. This operation assumes the execution is deterministic

[1] https://github.com/dupriezt/DebuggerCommunication



Class	Selector
TestAsserter	assert:equals:
PCBTest	testChildConfigurationLooksUpParentConfigurat
UndefinedObject	DoIt
BlockClosure	newProcess

key	value
class	TestAsserter
method	assert: actual equals: expected "This method raises an AssertionFailure if actual is diffe
node	a RBProgramNodeRepresentation(#class->'TestAsserter' #methodSelector->#assert:eq
receiver	PCBTest
selector	assert:equals:

```
    This method raises an AssertionFailure if actual is
different (using #= message) from expected.
    Else it does nothing and execution continues.
   "

   ^ self
      assert: actual = expected
      description: [self comparingStringBetween: actual and:
expected]
```

Class	Selector
Object	=
TestAsserter	assert:equals:
PCBTest	testChildConfigurationLooksUpParentConfigurati
UndefinedObject	DoIt
BlockClosure	newProcess

key	value
class	Object
method	= anObject "Answer whether the receiver and the argument represent the same object.
node	a RBProgramNodeRepresentation(#class->'Object' #methodSelector->#= #methodSourc
receiver	nil
selector	=

```
= anObject
   "Answer whether the receiver and the argument represent the
same
   object. If = is redefined in any subclass, consider also
redefining the
   message hash."

   ^self == anObject
```

Node =?	false

Nature	W Step Ir	F Step In
start	0	0
Divergence	7	7
Convergen	106	9
Divergence	115	18
Convergen	225	18
Divergence	227	20
Convergen	227	22
Divergence	230	25

Figure 2. UI of the echo-debugger, after setting up and connecting to the echo-runtimes. The UI is separated into three columns showing, from left to right: the *working* execution, the *failing* execution, and the *control zone*. The control zone is itself separated into three areas: (from top to bottom) the *status area*, the *operations area*, and the *navigation map*.

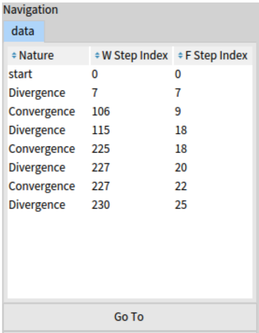
The *Step to next divergence* and *Step to next convergence* operations directly address challenges 3.1 and 3.2. *Analyze executions* is a convenience method to automatically repeat these two steps on the entire execution. *Go to* lets the developer inspect each divergence/convergence event. *Restart* and *Step both* give manual control of the parallel executions to the developer for closer inspection.

4 The CDM algorithm

In this section, we explain the CDM algorithm, used by the echo-debugger to spot all the control-flow *divergences* and *convergence* between the echo-executions. We first define what we mean by *convergence* and *divergence*. We then explain the CDM algorithm. We finally detail a special case of the algorithm when looking for a *convergence*, and how we perform the comparison of AST nodes from different runtimes.

The goal of the Convergence Divergence Mapping algorithm (CDM) is to fully run both echo-executions, and build a map of when they diverge and converge in terms of control flow. This map is a list of divergence and convergence events in the order in which they occurred during the comparative execution. Each event stores the number of steps both executions took to reach it. An example of such map is shown in Figure 3. Using this map, the echo-debugger is able to rerun the echo-executions up until any divergence/convergence event the developer wants to inspect.

Convergence and divergence of echo-executions. We define what we mean by *convergence* and *divergence* as follows. The idea is that we have two *similar* executions, and we want to know when they are doing the same thing (such as executing the same methods), and when they are not. At the start, neither echo-execution has executed anything, and they are both about to execute the same statement, provided by the developer. At this stage, they are definitely doing the same thing. We say they are *convergent* at that point. Then, as the echo-executions progress, at some point, they'll stop doing the same thing. We detect this by comparing the AST nodes they are executing. When they start executing different AST nodes, we say they are now *divergent*. But we know that prior to that point, they were doing the same thing, so if we let them fully step the current method call, the echo-executions will go back to the caller of that method call, and if at that point they are about to execute the same AST node, we say they have



Nature	W Step Index	F Step Index
start	0	0
Divergence	7	7
Convergence	106	9
Divergence	115	18
Convergence	225	18
Divergence	227	20
Convergence	227	22
Divergence	230	25

Figure 3. Result of the CDM algorithm on the Pillar configuration bug. This is the list of the convergence/divergence events observed during the echo-execution. The left column indicates the nature of the event (convergence or divergence). The middle column indicates the number of steps it has taken the *working* echo-execution to reach this event. The right column indicates the number of steps it has taken the *failing* echo-execution to reach this event.

converged. Indeed they were doing the same thing, then they entered a method call in which they started doing different thing, but now that method call is over and they are back to the part where they were doing the same thing. Now that they have converged, we let them progress until they diverge again, and converge again, etc, until either execution is over. This definition of *convergence* and *divergence* is the general idea of the CDM algorithm.

The CDM algorithm. Here is how the CDM algorithm builds a map with the divergences and convergences between the echo-executions. It is mostly a direct translation of the definition of *convergence* and *divergence* we gave in the paragraph above, with the exception of step *2.c.i*.

1. The echo-executions start convergent because they have done nothing yet and are about to execute the same statement, provided by the developer
2. Repeat until either execution is over:
   a. Step to next divergence
      - Repeat until the AST nodes the echo-executions are about to execute are *different*:
        i. Step each echo-execution once
        ii. Compare the AST nodes the echo-executions are about to execute
   b. Register a *divergence* event in the map, with the number of steps each echo-execution took to reach that point
   c. Step to next convergence
      - Repeat until the AST nodes the echo-executions are about to execute are *the same*:
        i. If the call-stack of both echo-executions do not have the same size, step the echo-execution with the longer call stack until its call stack has the same size as the call stack of the other echo-execution
        ii. Otherwise, if the call stack of the echo-executions have the same size, step each echo-execution separately until the size of their call stack is 1 less
        iii. Compare the AST nodes the echo-executions are about to execute
   d. Register a *convergence* event in the map, with the number of steps each echo-execution took to reach that point

Special case when stepping to the next convergence. Sometimes, the echo-executions diverge but their call-stack do not have the same size . This can for example happen when the source code change between the two program versions turned a normal method into a Virtual Machine primitive method. When stepping into a primitive method, the VM automatically executes it and returns to the caller. This means that the execution with the normal method is currently one-step-deep into that method, but the other execution is already back in the caller method. To find a convergence in these cases, our algorithm only finishes the current method call of the echo-execution with the longest call stack (See Figure 5). For comparison, the normal case is shown in Figure 4.

Comparing AST nodes. A fundamental part of the CDM algorithm is comparing the AST nodes the echo-executions are executing to determine whether their control-flows have diverged. Since the goal of the CDM algorithm is to find control-flow divergences, it also has to take into account the method and class the AST nodes belong to. For example, two *1+1* AST nodes are equal (in the = sense), but if they are from different methods/classes, we consider them different for the purpose of control-flow. Therefore, the CDM algorithm requires some form of identity (==) operator to compare the AST nodes. However, the standard identity operator cannot be used because the AST nodes to compare are coming from different runtimes. To get them into the controller runtime for the comparison, they would have to go through serialisation and materialisation. These materialised object are always different with regards to identity. Our solution is to design a new equality operator on remote AST nodes. This operator considers the four properties listed below, and compares them with the equality operator (=). Two remote AST nodes sharing these properties means that they come from the same method of the same class, are of the same type and correspond to the same part of the source code. This fits the need of the CDM algorithm for an AST node identity operator checking whether the control-flow of the echo-executions have diverged.

- **methodSelector**: the name of the method this AST node is from.
- **class**: the name of the class the method containing this AST node is from.

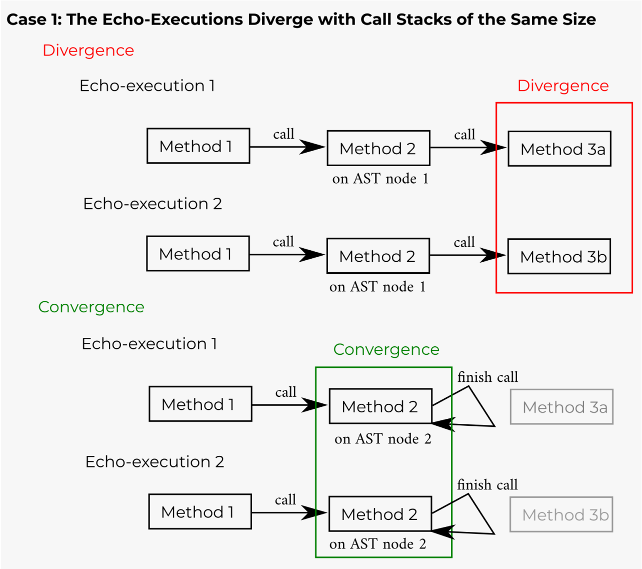


Figure 4. When the two echo-executions diverge and have call stacks of the same size, our algorithm steps the echo-executions to finish the current method call and go back to the last method call in which the execution were convergent. It then compares the current AST nodes of the two echo-executions. If it is the same (as is the case in this figure where AST node 2 = AST node 2), the echo-executions have converged. Otherwise, the algorithm repeats the process by finishing the call to method 2, comparing the current AST nodes.

- **source**: the source code covered by this AST node. For example, Point new for the message node representing the send of the new message to the Point class.
- **nodeType**: whether this AST node is a message node, a literal node...

5 Example: The Pillar Configuration Bug

Pillar[2] is a markup syntax and a tool-suite to generate documentation, books, websites and slides [2, 6]. In this section, we use the echo-debugger on a simplified version of a bug encountered in pillar: the pillar configuration bug [5].

5.1 Starting Knowledge about the Pillar Configuration Bug

Pillar uses nested configurations to store properties such as the authors, title, default folder and options for the generation given by the users. In addition, each file may override new properties (such as authors in a collection of articles). Each configuration is an environment *i.e.*, a dictionary of properties, and has a parent configuration. Asking a configuration for a given key *key1* is done by sending the message key1 to the configuration. This message is meant not to be understood by the configuration, to call its doesNotUnderstand: method [3]. The doesNotUnderstand: method calls the lookupProperty: method of the configuration. The lookupProperty: methods performs the lookup in the property dictionary of the configuration. If this dictionary does not contain *key1*, then the lookupProperty: method of the parent configuration is called...

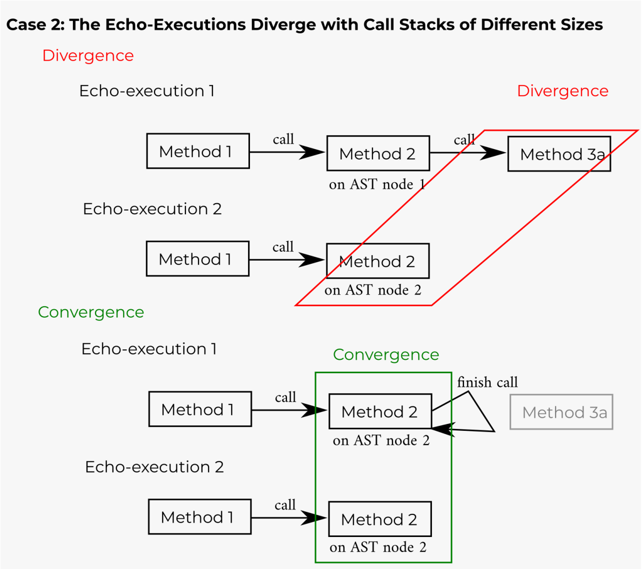


Figure 5. When the two echo-executions diverge and have call stacks of different size, our algorithm only finishes the current method call of the echo-execution with the longest call stack.

5.2 The test and the source code change

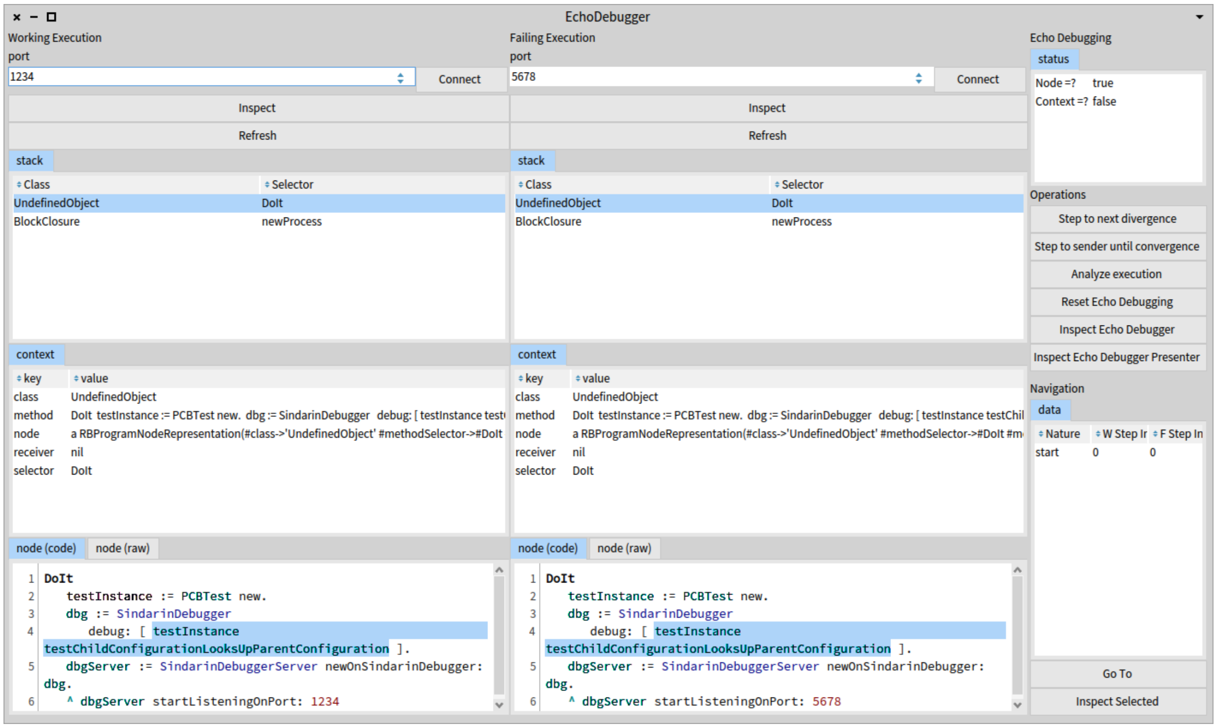
The test we are interested in is shown in listing 1. In this test, we create a first configuration c1 (line 3) and set the value of its mySetting key to 0 (line 4). We then create a second configuration c2 (line 5) and declare c1 as its parent configuration (line 6). Finally, we assert that the value of configuration c2 for the mySetting key should be 0, because it should inherit this value from c1.

```
1 PCBTest>>
      #testChildConfigurationLooksUpParentConfiguration
2 | c1 c2 |
3 c1 := PCBConfig new.
4 c1 mySetting: 0.
5 c2 := PCBConfig new.
6 c2 parentConfig: c1.
7 self assert: c2 mySetting equals: 0
```

Listing 1. Test highlighting the Pillar Configuration Bug

[2] https://github.com/pillar-markup/pillar

[3] This implementation was changed and is not available anymore in recent Pillar distributions because it was a bad idea according to Pillar maintainers.



Class	Selector
UndefinedObject	DoIt
BlockClosure	newProcess

key	value
class	UndefinedObject
method	DoIt testInstance := PCBTest new. dbg := SindarinDebugger debug: [testInstance test
node	a RBProgramNodeRepresentation(#class->'UndefinedObject' #methodSelector->#DoIt
receiver	nil
selector	DoIt

```
DoIt
   testInstance := PCBTest new.
   dbg := SindarinDebugger
      debug: [ testInstance
testChildConfigurationLooksUpParentConfiguration ].
   dbgServer := SindarinDebuggerServer newOnSindarinDebugger:
dbg.
   ^ dbgServer startListeningOnPort: 1234
```

Class	Selector
UndefinedObject	DoIt
BlockClosure	newProcess

key	value
class	UndefinedObject
method	DoIt testInstance := PCBTest new. dbg := SindarinDebugger debug: [testInstance testChil
node	a RBProgramNodeRepresentation(#class->'UndefinedObject' #methodSelector->#DoIt #m
receiver	nil
selector	DoIt

```
DoIt
   testInstance := PCBTest new.
   dbg := SindarinDebugger
      debug: [ testInstance
testChildConfigurationLooksUpParentConfiguration ].
   dbgServer := SindarinDebuggerServer newOnSindarinDebugger:
dbg.
   ^ dbgServer startListeningOnPort: 5678
```

Node =?	true
Context =?	false

Nature	W Step Ir	F Step In
start	0	0

Figure 6. Echo-debugger opened on the Pillar Configuration Bug.

This test originally passes, but fails after the following source code change: the developer adds an instance variable to the PCBConfig class, with a getter and a setter method. Without knowing that the name was already used for a property, the developer names this variable mySetting. After this change, the test fails, with the message that the property mySetting of c2 is nil instead of 0. The test fails because the lookup of mySetting on c2 now returns the value of the mySetting variable (nil) instead of calling the doesNotUnderstand: method as it used to.

5.3 Echo-debugging the Pillar Configuration Bug

Setup. We run three Pharo runtimes in which we loaded the Pillar program [4] and the echo-debugger with its companion communication packages[5]. We then have:

Working runtime. A *working runtime* in which no other code is loaded.
Failing runtime. A *failing runtime* in which in addition we loaded the breaking changes[6].
Controller runtime. A *controller runtime*. This is from this controller runtime that we will drive the echo-debugging session.

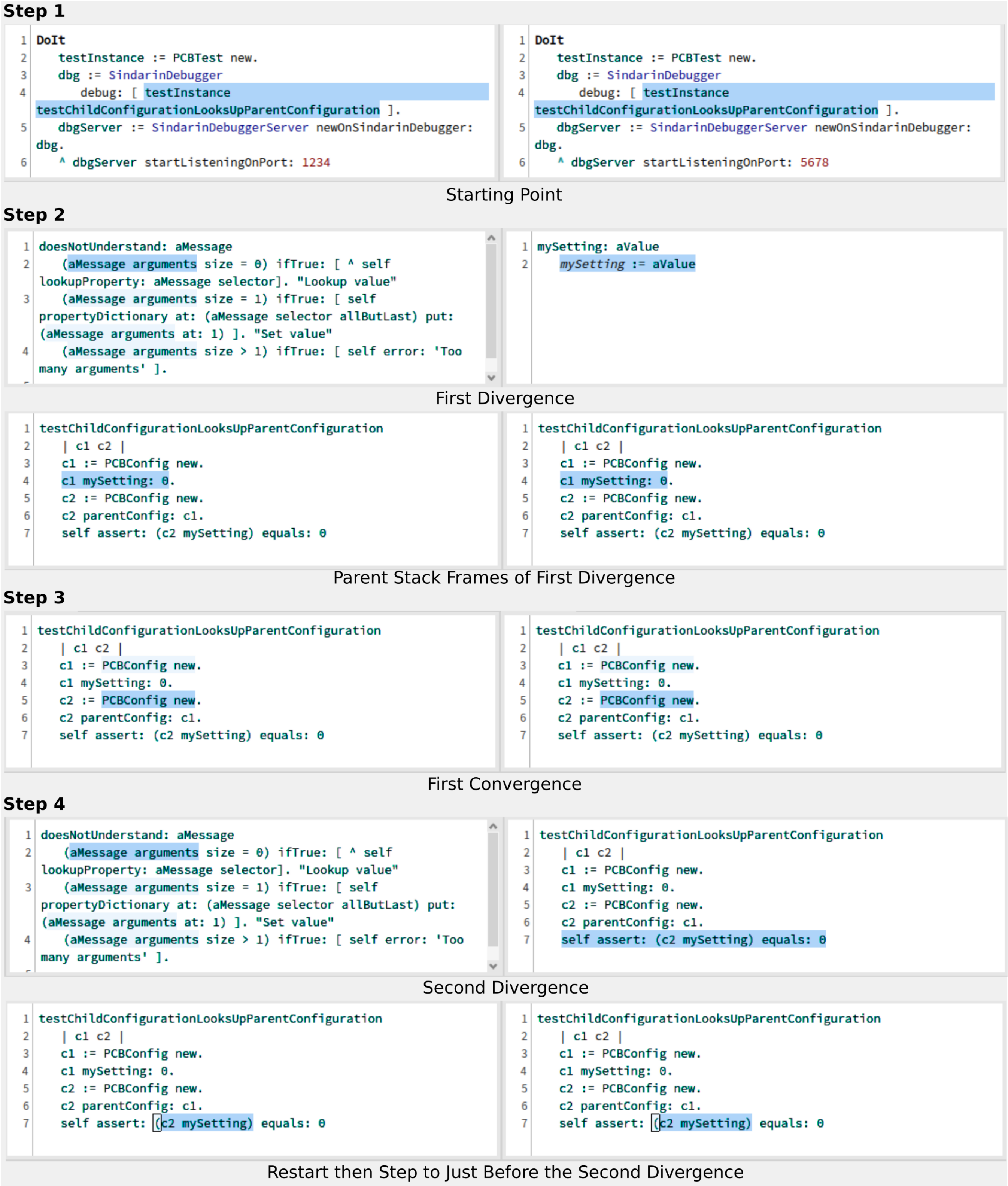
After connecting the runtimes and launching the echo-debugger, as described in the setup process detailed in Section 3.1, we see the echo-debugger UI shown in Figure 6.

Running the CDM algorithm. In the control zone, clicking the *analyze execution* button triggers the CDM algorithm described in Section 4. The result of the CDM algorithm is shown in Figure 3.

Investigating the echo-executions. Now we explain step by step how the echo-executions help us find the root cause of the problem. Figure 7 contains the relevant screenshots for the steps listed below, marked in bold in the text.

1. **Starting Point.** This is the setup code that has been executed to instantiate a Sindarin debugger on the test execution. The highlighted statement, about to be executed, is the test execution itself. In this figure and all the similar ones, the working execution is shown on the left, while the failing execution is shown on the right.
2. We step both echo-executions to the **First Divergence**. The W execution is in a doesNotUnderstand: method, while the F execution is in the mySetting: setter method that was added by the source code change. On the **Parent of these Stack Frames**, we see that the test executions are setting the value of the mySetting property in configuration c1. We deduce that the configuration did not understand the mySetting: message in the W execution, but it did in the F execution. The developer

[4] https://github.com/dupriezt/PillarConfigBug_Working
[5] https://github.com/dupriezt/DebuggerCommunication
[6] https://github.com/dupriezt/PillarConfigBug_Failing



Step 1

```
DoIt
   testInstance := PCBTest new.
   dbg := SindarinDebugger
      debug: [ testInstance
testChildConfigurationLooksUpParentConfiguration ].
   dbgServer := SindarinDebuggerServer newOnSindarinDebugger:
dbg.
   ^ dbgServer startListeningOnPort: 1234
```

```
DoIt
   testInstance := PCBTest new.
   dbg := SindarinDebugger
      debug: [ testInstance
testChildConfigurationLooksUpParentConfiguration ].
   dbgServer := SindarinDebuggerServer newOnSindarinDebugger:
dbg.
   ^ dbgServer startListeningOnPort: 5678
```

Starting Point

Step 2

```
doesNotUnderstand: aMessage
   (aMessage arguments size = 0) ifTrue: [ ^ self
lookupProperty: aMessage selector]. "Lookup value"
   (aMessage arguments size = 1) ifTrue: [ self
propertyDictionary at: (aMessage selector allButLast) put:
(aMessage arguments at: 1) ]. "Set value"
   (aMessage arguments size > 1) ifTrue: [ self error: 'Too
many arguments' ].
```

```
mySetting: aValue
   mySetting := aValue
```

First Divergence

```
testChildConfigurationLooksUpParentConfiguration
   | c1 c2 |
   c1 := PCBConfig new.
   c1 mySetting: 0.
   c2 := PCBConfig new.
   c2 parentConfig: c1.
   self assert: (c2 mySetting) equals: 0
```

```
testChildConfigurationLooksUpParentConfiguration
   | c1 c2 |
   c1 := PCBConfig new.
   c1 mySetting: 0.
   c2 := PCBConfig new.
   c2 parentConfig: c1.
   self assert: (c2 mySetting) equals: 0
```

Parent Stack Frames of First Divergence

Step 3

```
testChildConfigurationLooksUpParentConfiguration
   | c1 c2 |
   c1 := PCBConfig new.
   c1 mySetting: 0.
   c2 := PCBConfig new.
   c2 parentConfig: c1.
   self assert: (c2 mySetting) equals: 0
```

```
testChildConfigurationLooksUpParentConfiguration
   | c1 c2 |
   c1 := PCBConfig new.
   c1 mySetting: 0.
   c2 := PCBConfig new.
   c2 parentConfig: c1.
   self assert: (c2 mySetting) equals: 0
```

First Convergence

Step 4

```
doesNotUnderstand: aMessage
   (aMessage arguments size = 0) ifTrue: [ ^ self
lookupProperty: aMessage selector]. "Lookup value"
   (aMessage arguments size = 1) ifTrue: [ self
propertyDictionary at: (aMessage selector allButLast) put:
(aMessage arguments at: 1) ]. "Set value"
   (aMessage arguments size > 1) ifTrue: [ self error: 'Too
many arguments' ].
```

```
testChildConfigurationLooksUpParentConfiguration
   | c1 c2 |
   c1 := PCBConfig new.
   c1 mySetting: 0.
   c2 := PCBConfig new.
   c2 parentConfig: c1.
   self assert: (c2 mySetting) equals: 0
```

Second Divergence

```
testChildConfigurationLooksUpParentConfiguration
   | c1 c2 |
   c1 := PCBConfig new.
   c1 mySetting: 0.
   c2 := PCBConfig new.
   c2 parentConfig: c1.
   self assert: (c2 mySetting) equals: 0
```

```
testChildConfigurationLooksUpParentConfiguration
   | c1 c2 |
   c1 := PCBConfig new.
   c1 mySetting: 0.
   c2 := PCBConfig new.
   c2 parentConfig: c1.
   self assert: (c2 mySetting) equals: 0
```

Restart then Step to Just Before the Second Divergence

Figure 7. Investigating the echo-executions of the Pillar Configuration Bug

already expects this, since he just added the mySetting: setter method on purpose.

3. We step both echo-executions to the **First Convergence**. We see that after the execution of the mySetting: message was different between the echo-executions, they reconverge at the next statement of the test method. Notice that the W execution took 106 steps to reach this convergence, while the F execution only took 9.
4. We step both echo-executions to the **Second Divergence**. Here, the F execution is about to execute the whole assertion of the test, while the W execution is in a doesNotUnderstand method. To have a better look, we can **Restart** both echo-executions and step them until they reach the step just before this divergence event (114 steps for the W execution, 17 for the F execution). We see that both executions were about to execute the c2 mySetting statement of the test assertion. We deduce that this call resulted in a doesNotUnderstand: in the W execution, while it resulted in the mySetting getter method being called in the F execution. Using the debuggers, we separately inspect the two echo-executions from this point. In the W execution, doing a few steps shows the configuration c2 not understanding the message mySetting, looking up its property dictionary, and delegating the lookup to its parent configuration. In the F execution, we inspect the c2 configuration object to find that the value of its mySetting instance variable is nil.
5. We found the cause of the bug: adding a getter for mySetting on the pillar configuration class caused it to understand the mySetting message. This prevented the property lookup from escalading to the parent configuration.

6 Discussion

State differences. A limitation of our solution is that it only considers differences between the echo-executions in terms of control-flow. While such differences are important and helpful, differences in terms of *state* may also be very helpful to the developer. For example, recognizing when the echo-executions have the same control-flow but act on objects with different states.

Back-in-time debugging. After the CDM algorithm presented in Section 4 has fully run both executions to detect when divergence and convergence events occur, the echo-debugger restarts the echo-executions and steps them forward to reach the events the developer wants to inspect. This is a rudimentary form of back-in-time debugging, which assumes that the echo-executions are deterministic. More advanced techniques of back-in-time debugging [8, 11, 13, 16] could be used to remove this assumption.

Optimization of the CDM algorithm. In this paragraph, we discuss an implementation detail that proved critical in terms of performance. While our initial implementation of the CDM algorithm was almost instantaneous for small executions (around 250 steps), it was very slow for larger executions (more than 1 hour for around 5 million steps). The biggest performance bottleneck were the HTTP requests between the controller and echo-runtimes. The naïve implementation of the CDM algorithm sends many small HTTP requests to the echo-runtimes. Among others, one request per step, one request per AST node comparison to get the AST node, and one request each time the size of the call stack is needed.

To reduce the number of HTTP requests necessary, we simplified the data needed by the CDM algorithm running in the controller runtime. With this simplification, we no longer need the echo-executions to run in parallel. Instead, the echo-runtimes fully run their echo-executions locally, collecting the necessary data, and then send this data in big batches to the controller runtime. The controller runtime then performs the CDM algorithm offline on the data.

Data simplification: since the CDM algorithm only compares AST nodes to each other, it does not need the full dictionary representation of these nodes, and can work simply with the hashes of these representations. Also, the CDM algorithm does not need the complete call-stacks of the echo-executions, it only needs their size. With these two simplifications, the echo-runtimes fully run their echo-execution locally with no intervention from the controller runtime. After each execution step, they log a) the hash of the dictionary representation of the current AST node and b) the size of the call-stack.

These optimizations reduced the time necessary to run the CDM algorithm on an execution around 5 million steps long from more than an hour to 2 minutes.

7 Related Works

Test inputs. Palikareva *et al.,* [14] describe a technique called *Shadow symbolic execution*, designed to generate test inputs that cover new program behavior introduced by a patch. This technique symbolically executes a test in both program versions (before and after the patch) and compares these executions to find test inputs that lead to new behavior in the patched program and should be tested. This technique requires the developer to manually annotate the program to merge the old and new versions of the code. The Echo-Debugger does not have this requirement.

Brumley *et al.,* [4] solve logical formulae created from two different implementations of the same protocol (for example HTTP) to find deviations: inputs such that the output of the two implementations are semantically different. This technique produces inputs generating deviations between two implementations. By contrast, the Echo-Debugger is a tool to explore how two programs deviate on a given execution.

Delta Debugging and compared execution. Zeller [22, 23] presents the *Delta debugging* algorithm. This algorithm takes 2 versions of a program, and a test that was passing in the old version, but is failing in the new version. Delta debugging uses a divide-and-conquer approach to try multiple subsets of the code change and find the smallest subset that turns the test from green to red.

Abramson *et al.,* [1] propose *relative debugging*, a paradigm where the developer formulates a set of equality assertions about key data structures in the old and new versions of a program. The relative debugger is responsible for executing the two program versions in parallel and report any difference between the marked data structures. The two major differences with the Echo-Debugger are that 1) relative debugging deals with state differences while the Echo-Debugger deals with control-flow differences and 2) relative debugging requires manual interventions of the developer to mark the data structures they want to compare, and at which lines of code in the two programs to perform the comparison.

In the WhyLine [9, 10] tool, the developer asks questions about a recorded execution. The tool exploit traces to answer the questions, and tell why a particular variable has or has not a given value. Recorded divergences and convergences in the echo-debugger could be leveraged to ask questions about the execution in order to bring a better understanding of why two execution diverge.

Pinocchio [20] is a proof-of-concept implementation of a first-class code interpreter. Developers subclass the default interpreter to add behaviors to the code execution. An example use case is the creation of a *parallel debugger*, running two interpreters in parallel and comparing their state after each step. As opposed to the Echo-Debugger, the two interpreters of Pinocchio runs in the same runtime, and can only compare two executions on the same code base.

Algorithmic debugging. Algorithmic debugging is a technique proposed in 1982 by E. Y. Shapiro in the context of logic programming [17, 18]. Algorithmic debugging requires an oracle to compare execution outputs. These techniques try to isolate faulty code based on how developers assert the outputs of faulty and successful executions. An oracle could be used in the echo-debugger, to ask the developer to assert if a given convergence or divergence is normal or not (*e.g.*, between two program versions). This would help to focus on convergences and divergences that are relevant for the user, and ignore mundane differences like semantic-preserving refactoring.

8 Future Work

As future work to expand the echo-debugger presented in this paper, we identified 3 main axis.

State Differences. The Echo-debugger presented in this paper focuses on control-flow differences between the echo-executions. Another dimension in which two executions can differ is state (for example in the content of their variables). Incorporating state differences into the Echo-debugger, possibly inspired by the work of Henry Liberman [12], will make it paint a more complete picture of the differences between the executions.

Automated Setup. Setting up an echo-debugging session is a multi step process that can be tedious. An improvement axis consists in developing an automated setup tool to create the three runtimes, load the echo-debugger and its dependencies, run the debugger servers and client, and link them over HTTP. This tool could for example take as input a link to a git repository and two commit ids.

Using a Back-in-time Debugger as Back-end. Back-in-time debuggers are specifically designed to allow faithful replays of executions. The Echo-debugger requires this feature, and currently implements it by naïvely replaying the executions. This works for deterministic, isolated executions, but not for more complex executions. Using a back-in-time debugger as back-end will lift this limitation of the Echo-Debugger.

9 Conclusion

In this paper, we tackled the challenge of debugging two similar executions in parallel. We proposed the echo-debugger: an interactive debugger to debug two similar executions running in different runtimes. We also proposed the Convergence Divergence Mapping algorithm (CDM), an algorithm that fully runs both executions and compares the AST nodes they are executing to build a map of when they diverge and converge in terms of control-flow. This map records how many steps each echo-execution took to reach each event. The echo-debugger can then restart the echo-executions and step them to any event of this map the developer wants to inspect. We showed on an example how the echo-debugger helps finding the cause of a vicious bug.

The main limitation of the echo-debugger is that it focuses on the control-flow differences between the executions, but ignores the potential difference of state. This constitutes the main improvement direction of the echo-debugger. Additionally, since the echo-debugger should be able to replay an execution, the execution should be deterministic. Combining it with a back-in-time debugger would lift this limitation.

Acknowledgements

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.

References

[1] D. Abramson, I. Foster, J. Michalakes, and R. Sosic. Relative debugging and its application to the development of large numerical models. In *In proceedings of IEEE supercomputing*, 1995.

[2] T. Arloing, Y. Dubois, D. Cassou, and S. Ducasse. Pillar: A versatile and extensible lightweight markup language. In *International Workshop on Smalltalk Technologies IWST'16*, Prague, Czech Republic, Aug. 2016.

[3] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[4] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *In Proceedings of the USENIX Security Conference. USENIX Association*, 2007.

[5] S. Costiou. *Unanticipated behavior adaptation : application to the debugging of running programs*. Theses, Université de Bretagne occidentale - Brest, Nov. 2018.

[6] S. Ducasse, L. Renggli, and R. Wuyts. SmallWiki — a meta-described collaborative content management system. In *Proceedings ACM International Symposium on Wikis (WikiSym'05)*, pages 75–82, New York, NY, USA, 2005. ACM Computer Society.

[7] T. Dupriez, G. Polito, S. Costiou, V. Aranega, and S. Ducasse. Sindarin: A versatile scripting api for the pharo debugger. In *DLS'19, Dynamic Language Symposium*, 2019.

[8] C. Hofer. Implementing a backward-in-time debugger. Master's thesis, University of Bern, Sept. 2006.

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

[10] A. J. Ko and B. A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *In Proceedings of the 30th International Conference on Software Engineering, ICSE 08*, 2008.

[11] B. Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG'03)*, Oct. 2003.

[12] H. Lieberman. Steps towards better debugging tools for lisp. *ACM Symposium on Lisp and Functional Programming*, 1984.

[13] A. Lienhard, T. Gîrba, and O. 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.

[14] H. Palikareva, T. Kuchta, and C. Cadar. Shadow of a doubt: Testing for divergences between software versions. In *International Conference on Software Engineering (ICSE 2016)*, 2016.

[15] 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.

[16] G. Pothier, E. Tanter, and J. 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.

[17] E. Y. Shapiro. Algorithmic program diagnosis. In *Proceedings of Principles of Programming Languages (Popl)*, pages 299–308. ACM, 1982.

[18] J. Silva. A survey on algorithmic debugging strategies. *Advances in engineering software*, 42(11):976–991, 2011.

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

[20] T. Verwaest, C. Bruni, D. Gurtner, A. Lienhard, and O. Niestrasz. Pinocchio: bringing reflection to life with first-class interpreters. In *OOPSLA '10: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, 2010.

[21] E. W. Wong, R. Gao, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[22] A. Zeller. Yesterday, my program worked. today, it does not. why? In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 253–267, London, UK, 1999. Springer-Verlag.

[23] A. Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10, New York, NY, USA, 2002. ACM Press.

[24] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oct. 2005.