Back-in-time inspectors: an implementation with Collectors

Steven Costiou
Lab-STICC, UMR CNRS 6285
Université de Brest, France
costiou@univ-brest.fr

Clotilde Toullec
Independant Researcher
Brest, France
clotilde.toullec@gmail.com

Mickaël Kerboeuf
Lab-STICC, UMR CNRS 6285
Université de Brest, France
kerboeuf@univ-brest.fr

Alain Plantec
Lab-STICC, UMR CNRS 6285
Université de Brest, France
plantec@univ-brest.fr

ABSTRACT

Back-in-time debugging is an appealing solution to investigate bugs for which there is no obvious relation between their symptoms and their origin. In this paper we implement a tool named the Back-in-time inspector for Pharo, which provides an execution history of evaluated expressions selected from the source code. We evaluate the back-in-time inspector by investigating an unresolved bug which we are able to solve using our tool.

KEYWORDS

Object Oriented Debugging, Back-in-time Debugging, Debugging

Steven Costiou, Clotilde Toullec, Mickaël Kerboeuf, and Alain Plantec. 2018. Back-in-time inspectors: an implementation with Collectors. In *International Workshop on Smalltalk Technologies (IWST '18), September 10–14, 2018, Cagliari, Italy.* , 10 pages.

1 INTRODUCTION

One of the hardest difficulties in debugging is the distance between the cause of a bug and its symptoms [7]. For example, the state of a problematic object can be resulting from a distant origin, that does not appear clearly when the bug occurs. Sometimes a very small change in a program can lead to problems that seem completely unrelated. These bugs can be hard to understand and to fix. Omniscient debuggers [8] are interesting solutions to investigate such bugs, because they allow to explore and navigate forwards and backwards in time through the *post-mortem* execution of a program. These solutions usually provide dedicated debuggers and produce a large amount of recorded data issued from code instrumentations.

Collectors [4] is an ongoing work of the authors that allows the developer to define collection points in the control flow of a program. Collection points are non-intrusive instrumentations of specific expressions selected by the user. When these expressions are executed, the objects resulting from their computation are collected. The developer can define transformations for debugging purposes (*e.g.* logging) which are applied to the objects as they are collected during runtime. In this first model of Collectors, objects are not recorded, and are released as soon as they are claimed by the garbage collector.

We propose an extension of Collectors that records collected objects resulting from an instrumented expression and all its sub-expressions. We use this extension to implement a *back-in-time inspector*. The back-in-time inspector shows the recorded objects resulting from an expression execution. For each passage in the control flow of an instrumented expression, a record of the expression evaluation is created and stored in the program's memory. This record contains the object resulting from the expression evaluation and its associated meta-information (such as the execution stack). This allows to dynamically scope back-in-time records to very specific expressions in the program. This helps the developer to investigate suspicious segments of the code that could be part of a problem, and access contextual (meta) information of a particular expression evaluation.

The contribution of this paper is an extension of Collectors to support back-in-time capabilities and a *back-in-time inspector* to visualize object collection histories. We present an implementation of Collectors and the *back-in-time inspector* in Pharo [3]. We illustrate Collectors through a debugging session of a hard bug that we solved with the help of the Collectors *back-in-time inspector*. The source of this bug was unknown prior to our investigation.

IWST '18, September 10–14, 2018, Cagliari, Italy

The paper is structured as follows: section 2 describes the *Pillar bug*, an unsolved bug typical of the distant origin/symptom problem that can make debugging very hard. In section 3, we describe how we extended Collectors to support back-in-time histories of evaluated expressions and the associated tool, namely the back-in-time inspector. In section 4, we use the back-in-time inspector to investigate the *Pillar bug*, we find its source and we successfully devise a bug-fix. After a discussion of the debugging session in section 5, we discuss related work in section 6 and conclude in section 7.

2 MOTIVATING EXAMPLE: THE *PILLAR BUG* CASE

The *Pillar bug* problem is an error encountered by the *Pillar*[2] tool, a document generator from a *mark-up* type syntax implemented in the Pharo language. *Pillar* is a sophisticated tool that is tested through more than 3000 unit tests. The introduction of accessors for a new instance variable in a class leads to the failure of a particular but unrelated test. This problem is considered difficult to solve [6], as there is no obvious link between the context in which it appears and its symptoms.

The precise description and reproduction steps of the problem are publicly available[1]. A new instance variable named *disabledPhases* is added to a configuration class. Two accessors are introduced: *disabledPhases* for the instance variable reading, and *disabledPhases*: for the variable modification. By default, if the variable *disabledPhases* has not been initialized, its accessors initialize it as an empty array. Before the introduction of the accessors, the execution of all unit tests was successful. After the definition of these accessors, one of the unit tests fails. The error seems to be completely unrelated to the changes, which only consisted of a new instance variable and the definition of its accessors (without using them).

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

Figure 1: "*The symptom of the bug is that outputType is nil. However, this piece of failing code plus the fact that 3166 tests are still working, give us no clue about the relation between the change and the bug.*" [6]

[1] https://github.com/guillep/pillar-bug

The problem is also described by Dupriez et al. [6] (Figure 1). The error appears in the method *actionOn:* of the $PREPubMenuJustHeaderTransformer$ class. The accessor *outputType* of a configuration returns a *nil* value, which causes the test failure. Only a very reduced set of tests fail, and the problem symptom described by Figure 1 has no obvious relation to the direct modification made, namely the adding of the two accessors to the new variable *disabledPhases*.

This problem description seems rather abstract. However, this is where we are left after introducing only two accessors: a failing test, an error that is unrelated to the (trivial) modification, and not enough knowledge of the *Pillar* tool to know where to start. During a first investigation, traditional debugging tools did not help much. Exploring the stack from the exception showed absolutely no clue about what the problem was or how it was related to the modification. Breakpoints in the *disabledPhases* method were tedious to use, because the method was called many times and seemed to return a value which, as far as we could guess, was correct – or at least not suspicious. By inserting logging code, we saw that the getter method was called 42 times in a loop, and each call except the last one returned the same value (an array with 2 elements). We did put a conditional breakpoint to halt the execution on this particular value change, but again there was no indication of why this value was different from the other. Exploring the stack was not helpful either, because the more we got back-in-time in the stack, the more difficult it was to know the impact of the inspected code on the final observed result. As for the setter method, the same methodology only showed that it was called once and initialized the above mentionned array which seemed to be a correct value.

3 COLLECTORS: THE BACK-IN-TIME EXTENSION

In this section we describe the base Collectors concept [4] and its extension to record collected objects. When we talk about recording objects, we consider that the development environment and the program share the same memory space. Therefore, the developer can access saved objects after an execution. The impact of objects recording on the memory and on performance is not addressed, and this limitation is discussed in sections 3.5 and 5.

3.1 Object collection: recording objects from expression evaluations

The original Collectors model collects a single result of an evaluation of an expression. A collector targets an abstract syntax tree node, which is the abstract representation of a

source code expression. At runtime, each time this expression is evaluated, the resulting object will be collected and stored in memory. This is illustrated in Figure 2, in which the dotted line represents the target expression. In the original model [4], collected objects are released when they are claimed by the garbage collector. In this extension, objects are recorded and kept in the program memory. At the moment, there is no model describing how objects are saved and the *program memory* refers to the memory space that it shares with the development environment, as in live environments such as Pharo. Therefore, an object is "*saved*" because we keep a reference to this object during and after the program execution.

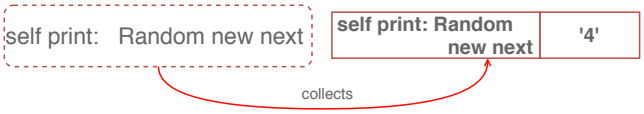

Figure 2: A collector: the dotted line represents the target *AST* node (on the left), while the resulting object is stored each time it is evaluated at runtime (on the right).

3.2 Deep collection: collecting results from sub-expression evaluations

To provide a local, scoped back-in-time feature to the object collection, we expand the concept of *object collection* through the *deep object collection*. When installing a collector on a node, the collector will try to install itself *as deep* as possible in the *AST*. Each child of the original target node will be targeted by the collector. As depicted by Figure 3, at runtime the resulting object of each child's evaluation will be collected. As the *AST* evaluation order is stable and deterministic, the intermediate results from the nodes evaluation are simply stored in an ordered collection. The last object stored in the collection is the result of the evaluation of the main instrumented expression. Each result in this collection annotates the node from which it was issued (Figure 3), and the whole collection is released (and the node annotations removed) when the collector itself is uninstalled. Each evaluation of the main target expression will produce such ordered collection, with a timestamp as a unique identifier.

3.3 The model

The Collector's extended model is illustrated in Figure 4. To instrument expressions with object collection behavior, a collector specifies one or more collection points. A *CollectPoint* is a position in the control flow of a program. It refers to an expression of the program, namely a node of an abstract syntax tree. When a collector targets a high level

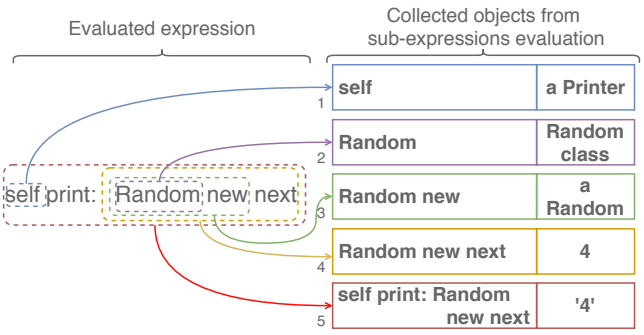

Figure 3: A deep collector: the collector is installed on all children of the target node (on the left), and objects resulting from each evaluation are collected at runtime (on the right). The collection order depends on the evaluation order of the *AST*.

entity or concept, *e.g.* an instance variable, it instantiates a collection point for every expression referencing this entity in source code, *e.g.* all assignments and all reads concerning an instance variable.

An *OmniscientRecord* holds collected objects for a particular *CollectiontPoint* execution. When a *CollectPoint* is executed, an *OmniscientRecord* is generated and stored in memory. Conceptually, an *OmniscientRecord* is a data annotation of a collection point. An *OmniscientRecord* contains a reference to the collected object, a deep copy of the object to capture its state at collection time, and meta data from the context of the current execution (*MetaData*). Throughout the program execution, a CollectPoint accumulates OmniscientRecords to form what we call an expression evaluation history.

A collector can request meta data from the execution of its collection points. A *MetaData* is either a direct reification of the collection point execution context, or a transformation of objects from this context (*e.g.* the name of the active method or the execution stack). Requests for meta data are specified by the user or by a tool, as instances of *MetaDataRequest*. At collection time, these requests are performed by the collector and stored in the generated *OmniscientRecord*. A *MetaData* object containing transformations of context objects also contains copies of the source objects used in these transformations.

Finally, the *CollectBehavior* defines how a collector performs object collection when a particular *CollectPoint* is reached. When a target expression is evaluated at runtime, the collection behavior handles the deep object and meta data collection through sub-expression evaluation (Figure 3). The *CollectBehavior* manages how *OmniscientRecords* are timestamped, which defines how they relate to each other. The timestamp is only used as a mean to label collected data

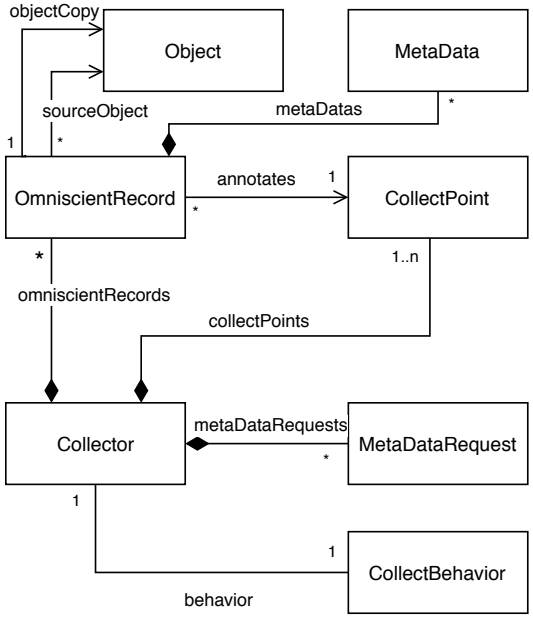

Figure 4: The Collectors *back-in-time* extension model.

with a unique identifier. Omniscient records sharing the same timestamp are all resulting from the same passage in the control flow of the evaluated expression. The *CollectBehavior* also restricts the scope of the available meta data requests. For example, for collection points based on abstract syntax trees, not all nodes provides the same reifications at runtime.

3.4 Tools for debugging

Collectors provide an *API* and a tool-set. Targeting *AST* nodes with Collectors is integrated into the development environment as dynamic source code interactions. A configuration tool helps defining Collectors and meta data requests, and the *Back-in-time* inspector (Figure 5) displays the collected objects with their local history, in regards to the node from which they originated. To define a collector, the user has to select expressions from the source code and specify the entity to collect (Figure 6). Further description of the tools and the *API* are available online[2].

3.5 Implementation

Our implementation[3] is based on Reflectivity [5] in Pharo [3]. Reflectivity provides metalinks, which are annotations of *AST* nodes. When an annotated node is evaluated, it triggers the execution of its annotations (*i.e.*, metalinks). Basically, a metalink references an object and a message selector. When the metalink is executed, it sends the corresponding message to the referenced object, thus executing meta-behavior. In this context, this object is called a meta-object. A metalink can also be configured with a list of requested meta-informations to be reified (*RFReification*). In that case when the node is executed, the metalink passes these reifications as parameters to the message sent to the meta-object. Which reifications are available depends on the kind of node on which the metalink is being installed.

For each collection point, the associated collector defines a single metalink to annotate the *AST*. An instance of *CollectBehavior* from Figure 4 plays the role of meta-object for all metalinks defined by a collector, and uses dedicated metalink installation strategies to target the collection points. In addition, the user can provide transformations from the reifications or from the collected object. These transformations produce user or tool specified meta-data, that can be used either for tool-based features or for the user's debugging needs. For example, the *Back-in-time* inspector uses transformations from the Reflectivity context reification to produce an execution scoped stack that displays the senders and the contextual values that lead to a particular object collection.

Reflectivity is the sole mandatory requirement to implement Collectors. We extended Reflectivity with an *API*[4] to ease the targeting of the nodes that we want to annotate, for example all assignments of an instance variable. Currently in Reflectivity all metalinks are lost when recompiling a method (*e.g.* when debugging the program). Our extension automatically reinstall links for high level targets (*e.g.* instance variables) after a method recompilation.

This prototype is limited in the back-in-time navigation of the recorded objects. We make deep copies of collected objects so the end user can observe how their state evolve throughout the program execution. We make shallow copies of contexts, because the deep copy behavior for contexts is not implemented in Pharo (although it could be done). Terefore we cannot see the changes to objects referenced by copied contexts from reified execution stacks. This is a limitation to the back-in-time power of this implementation.

Performance overhead and additional memory consumption were not evaluated for this prototype, although they constitute a known drawback for back-in-time approaches.

4 USING COLLECTORS TO INVESTIGATE THE *PILLAR BUG*

This section describes a debugging session with Collectors. We try to track the *Pillar bug* and to formulate a hypothesis about the reasons of the problem.

[2]https://github.com/ClotildeToullec/Collectors/wiki
[3]https://github.com/ClotildeToullec/Collectors
[4]https://github.com/StevenCostiou/Reflectivity-dev

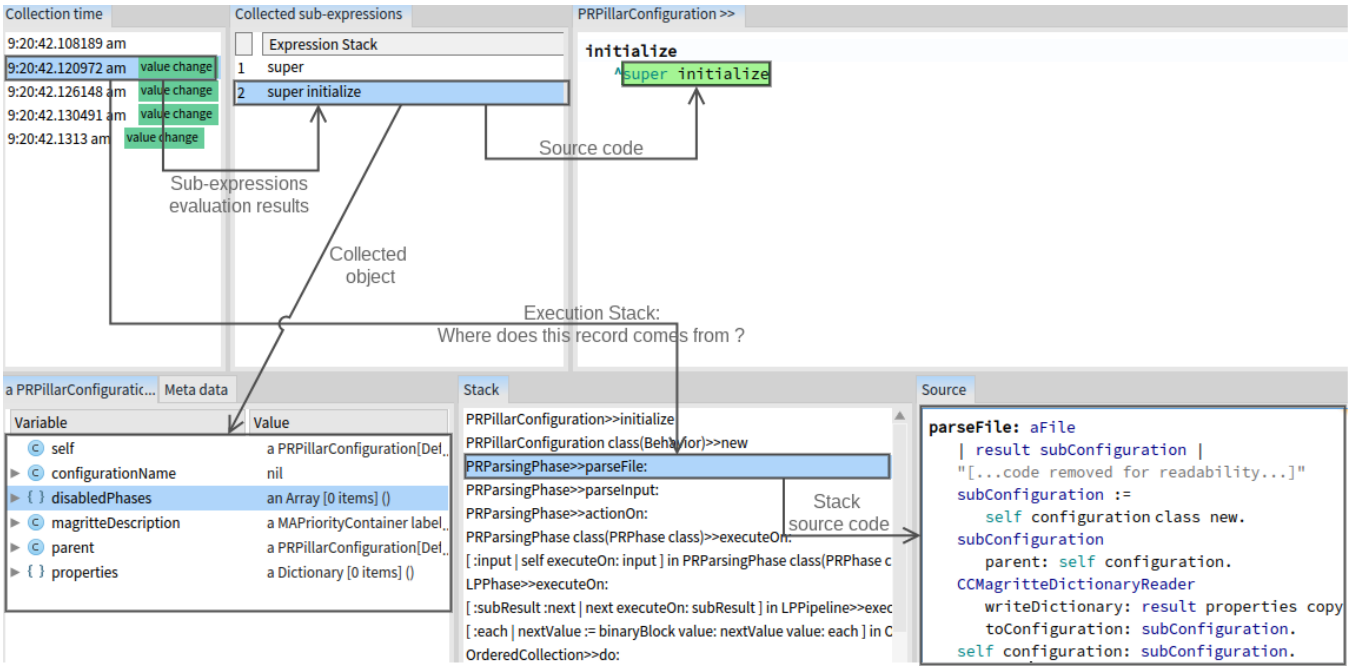

Figure 5: The Back-in-time inspector. The first row shows the different records from the instrumented expression evaluation (*Collection time*), the records from the subexpressions evaluations (*Collected sub-expressions*) and the source code visualization. The second row shows, in order: the recorded object for the selected sub-expression, the execution stack that lead to the evaluation of the whole instrumented expression and the source code of the selected method in the execution stack. The context of each method in the execution stack can be inspected through the contextual menu to see the available contextual information (*e.g.* the values of temporary variables).

4.1 Limitations of traditional debugging

The *Pillar bug* is difficult to understand. It happens in a specific unit test when an instance variable and its two accessor methods are added to a configuration class of *Pillar*. When the error occurs, there is no indication about how it may relate to these accessors.

Investigating this bug with traditional techniques, like breakpoints or simple logging, is unpractical and requires a lot of effort. Breakpoints are not efficient. We first need to guess where to put them, and only halt when the conditions under which we can observe the bug are met. The problem is we have no clue about what could these conditions be. Logging implies inserting statements in the source code. As we do not know where to put these statements, such investigation requires polluting the source code until we find a lead. Then we would have to remove all useless logging statements to focus on this lead, and compare a healthy execution with a faulty one.

4.2 Program observation with Collectors before the problematic modification

We briefly inspect the program before the introduction of the problematic accessors to study a correct execution of the failed test. We note that the accessors, though not existing, are referenced and called in the project source code. This call to the *disabledPhases* accessor is shown in Figure 7. We define a collector (Figure 6) to record the history of the values resulting from the execution of this call (the source code highlight in Figure 7). During a healthy execution, several object collections are performed, meaning that the program executes the instrumented code several times. At each passage in this control flow, the result of the code evaluation is stored. This expression always returns the same value, which is an array containing two strings : *'sections'* and *'justKeepHeaders'*. During a healthy execution, the evaluation result of this expression is therefore an invariant.

A quick observation of the code shows that, when the accessor *disabledPhases* is not implemented, the reception of this message triggers a *doesNotUnderstand* exception. In the impacted configuration class, the *doesNotUnderstand*

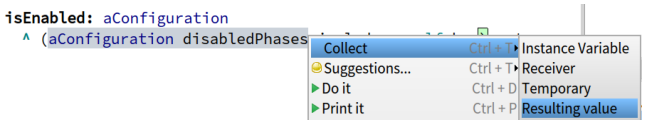

Figure 6: Definition of a *CollectPoint*. The tool integrates in the environment to ease the instrumentations.

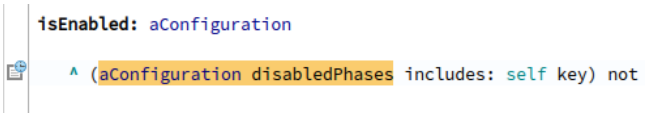

Figure 7: Collection point on the call to the *disabledPhases* accessor : each time the program will execute this control flow, the evaluation result of the expression will be collected. The button on the left size opens the execution history for the highlighted expression.

behavior is instrumented to capture some messages (for example *disabledPhases*) and redirect them to a dictionary containing the configuration properties. A configuration can also have a parent configuration. If a property is not found in the current configuration, for example *disabledPhases*, the property is looked up in its parent configuration until the property is found or until the topmost parent is reached. If the property is still not found in the topmost parent, the *nil* value is returned by the *doesNotUnderstand* instrumentation. Inserting exception handling code or putting a breakpoint to investigate the behavior of the *disabledPhases* property lookup in the *doesNotUnderstand*: method is tedious, because this mechanism is used widely by *Pillar* for all properties of its configurations.

4.3 Program observation after the error was introduced

After the introduction of the accessors in the source code, the test execution fails. The history of the instrumented expression in Figure 7 shows that throughout the test execution, the collected objects do not always have the same state (Figure 8). The invariant was *broken*, and replaced by an empty array. This is an evident difference between the healthy execution of the program and its problematic execution, and it establishes a first investigation lead.

The history presented in Figure 8 contains the results of the instrumented expression evaluations, but also of the evaluations of its subexpressions. Thereby, when the collector collects the result of sending the *disabledPhases* message to the *aConfiguration* object (Figure 7), it also collects the evaluation result of the subexpression *aConfiguration*. We

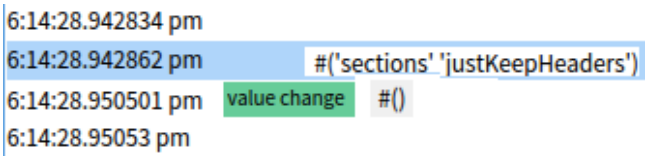

Figure 8: State of the collected objects does vary: the history of the collector indicates with a label that the collected value changed (the collected values from the inspector have been pasted to the right of the list for better lisibility). The object collected by the instrumented expression seems not change during a healthy execution for this particular test.

can therefore compare not only the problematic result of the global expression, but also the values and the states of the objects and evaluations used in that expression. The comparison of the configurations before and after the invariant alteration shows that the configuration state was not modified but that the two objects are different. The original configuration is replaced during the test execution by another one which state is not what we expect.

4.4 Tracking and correction of the problem

This new lead consists of looking for the moment when the configuration with a correct state is replaced by a new configuration with an incorrect state. We decide to collect all the objects of this particular configuration class at their creation. We introduce for this purpose an *initialize* method in our configuration class to be able to easily collect its result, that is for every new instance of our configuration. We make sure that the addition of this method has no impact on the test result by executing the whole test sequence in a healthy version of the application, without the problematic accessors.

After a new execution of the failing test, the configuration objects creation history shows several instantiations of the class (Figure 5). The inspection of the first collected configuration shows a correct state of the *disabledPhases* corresponding to the invariant. The other collections harvested configurations with an incorrect state: an empty array that does not correspond to the expected invariant. For each collection, it is possible to consult the execution stack that lead to the instrumented expression execution.

The execution stack of the first collected configuration shows that it is the one created at the test initialization. It is notably configured via the *configuration* method that we find in the stack and that initializes the *disabledPhases* invariant (Figure 9). The execution stack of the second configuration shows that the latter is created by a completely different path (Figure 5). In particular, the configuration instantiation is called from a *parseFile*: method. This method

```
configuration
    ^ super configuration
    "we disable these phases as they pollute the tests"
        disabledPhases: #('sections'  'justKeepHeaders');
        yourself
```

Figure 9: Method originally called by the unit test to initialize the configuration, found in the execution stack of the first collected configuration : the *disabledPhases* invariant is correctly initialized.

instantiates a new configuration and gives it the previous configuration – in that case the original configuration, as parent. This new configuration is not initialized and notably no property or instance variable of this configuration is modified.

We are therefore able to state a hypothesis. In the non-problematic case before the introduction of the accessors in the configuration class (Figure 10), every send of the messages *disabledPhases* and *disabledPhases*: triggers a *doesNotUnderstand* exception. The behavior of the latter is instrumented to look in the configuration properties for the *disabledPhases* property. In the case that we observe in the stack shown in Figure 5, the newly created configuration is not initialized. Every attempt to look for the *disabledPhases* property will end up in the *doesNotUnderstand*, that will look for the property in the parent configuration. This behavior explains why the property is always correctly initialized and why its value is invariant.

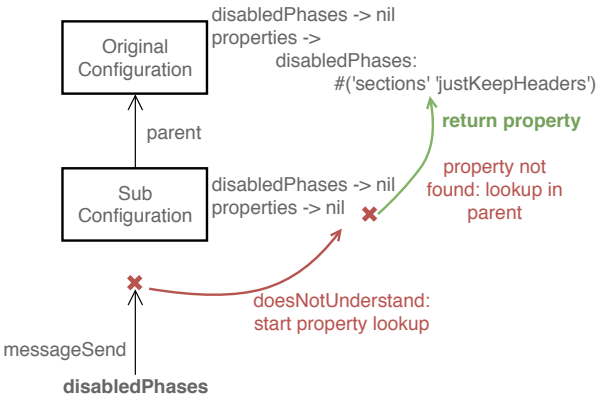

Figure 10: *Pillar* configurations lookup: the *disabledPhases* is found through a parallel lookup in the properties of the configuration and its parent.

In the problematic case after the introduction of the accessors in the configuration class (Figure 11), the sending of

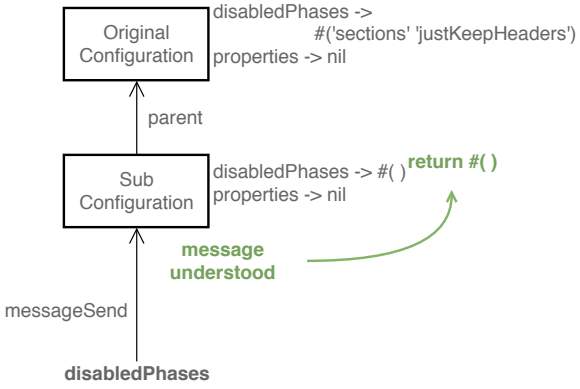

Figure 11: After the introduction of the *disabledPhases* accessor, the message is always understood by the configurations and the standard lookup always returns the value of the *disabledPhases* instance variable.

the messages *disabledPhases* and *disabledPhases*: does not trigger a *doesNotUnderstand* anymore, because the message is now understood by the configuration object. When it receives this message, the new configuration immediately executes the accessor that returns its instance variable initialized as an empty array by default. There is no more lookup for the property in the parent configurations, since the instrumented behavior of the *doesNotUnderstand*: method is not executed anymore. The configuration does not respond to this message in a coherent way, and the returned value is not invariant anymore.

```
subConfiguration disabledPhases:
  self configuration disabledPhases
```

Figure 12: Experimented correction after the creation of a new sub-configuration : copy of the *disabledPhases* from the parent configuration.

Furthermore, the comment in the code of Figure 9 indicates that the test requires by design this specific setting of the *disabledPhases* property. The error occurs from the particular semantics placed on the *doesNotUnderstand* behavior of configurations, that is broken by the introduction of the accessors in the configuration class. We experiment a simple correction (Figure 12), that consists of updating the *disabledPhases* variable state of every new configuration from its parent configuration. The unit test does not trigger any more errors and the complete sequence (more than 3000 tests) also ends successfully.

5 DISCUSSION

The *Pillar bug* is a hard problem, because the semantics established by *Pillar* are not obvious for someone external to the project. An apparently harmless modification provokes an error, the difficulty which lies in the fact that the relationship between the modification made to the system and the apparent symptom of the error are very distant from one another [7].

The use of Collectors and their tools allowed us to find the error origin and to understand it in a few minutes. The tools just use the language model to implement the advantages of the *back-in-time* or *omniscients* debuggers [8]. The visualization of the program execution recordings allows one to follow object evolution and interactions. But the contribution of Collectors lies in the used approach, in which the developer uses his or her knowledge of the program and the resulting intuition to manually select the expressions to be instrumented. The evaluation result of such expressions will be collected every time they are executed, as well as all the objects resulting from the evaluation of its subexpressions. For each collection, elements contextual to the execution are recorded to help the developer analyze the results. For example, the following elements have been crucial during this investigation :

- The observation of a broken invariant state between a healthy execution and a problematic execution helped us understand that the configuration objects changed, and that we had to track down where the configuration with the erratic state was instantiated.
- The access to the execution stack (with its contexts) leading to the collection of a problematic configuration allowed us to find the cause of the incoherent state almost immediately.

These steps stand however on a preliminary analysis of the studied program code, in order to gain sufficient knowledge and comprehension to make hypotheses. The tool does not give a solution to the problem, but supplements this knowledge of the program by giving the possibility to instrument suspicious expressions promptly and intuitively. The easy setup of Collectors and its integration in the development environment are very important for this purpose.

The definition of the debugging scope, *i.e.* which code should be instrumented and which objects should be analyzed, is facilitated by the possibility to specify collectors from the source code following the developer's intuition.

This approach does not invalidate the use of a complementary standard debugger. For example, the use of breakpoints at keys locations in the code allows to stop and inspect the program. There is however no guarantee that the halt occurs before the problematic change of state, or that it occurs within a reasonable time during a manual step by step execution in the debugger. If the breakpoint – or the step by step execution – misses the critical point in the program execution where a state is altered, for example here the *disabledPhases* property, the developer has to start the debugging session again. The breakpoint does not allow to easily visualize the state changes of an object between different executions and to compare some of their specific aspects to their context – for example the execution stacks. Furthermore, the instrumentation to observe a particular object can be complex and need code insertion on different locations in the program. The non-intrusive aspect of the Collectors, inherited from its Reflectivity [5] implementation layer, avoids such insertions and any code pollution by debugging instructions.
We did not evaluate performance overhead nor additional memory consumption resulting from the use of the *back-in-time* inspector. In the context of a single unit test, there was no visible slowdown from the developer's point of view. To put these concerns aside was important during the design and the implementation of the Collectors extension. It allowed us to freely explore the back-in-time features we were interested in for debugging. However, we fear that it is a weakness of the current implementation, and that aspect remains to be investigated. Research has been done to address the efficiency and the memory usage of back-in-time debuggers [11, 12, 14]. That would be our starting point to improve our model and our prototype.

6 RELATED WORK

Back-in-time or *Omniscient* debugging is not a new idea [8]. It consists on remembering the result of every execution of the running program, including state, objects, contexts and stacks, etc. The main disadvantages of this technique are its memory consumption and its performance overhead, a great deal of additional resources being necessary to record every step of an execution. It still remains a powerful technique to solve hard bugs, and to understand their root cause. It gives the developer the ability to go backwards in time and to explore how the state of a program evolved to reach a critical situation – like a bug.

A lot of research has been contributing to the technique and its evolution; for example, *object and flow-centric* back-in-time debugging [9–11] to track the origin and the flow of objects in a program execution. Work on *flow-centric* debugging focuses on providing answers about how an object arrived in a particular place and the history of its state. An effort has also been made to provide practical tools, both in the post-mortem visualization of the flow of objects and the overall performance of the provided tools. Other work focused on scalability and scoping of the generated traces [13, 14], and also shares a concern for practicability of the

debugging technique. The provided debugger allows to step forward and backwards in time after the program has run and all execution has been saved in a database. The developer can define the scope of the traces, either by specifying which class should be subject to traces, or by activating the traces through a manual switch. A few commercial debuggers are also available as professional tools for back-in-time debugging, like Chronon [1]. Orthogonal to back-in-time debugging, the particular problem of saving objects efficiently has been studied and prototyped in Pharo [12]. Limitations of Collectors like performance overhead, memory consumption or object state recording, are orthogonal problems to the back-in-time debugging features that could be tackled by reusing existing solutions from the state of the art.

Our back-in-time implementation with Collectors is a limited subset of the features provided by the aforementioned solutions. However what is recorded are surgically scoped expressions evaluations, and a selection of their contextual (possibly meta) informations. This scoping and configuration operations are designed to be simple and intuitive from the developer's point of view, in the hope to lower the learning cost of the tools. Although we also provide a dedicated visualization tool, it relies on a mechanism that is integrated in the language and as such, debugger extensions could be built and seamlessly integrated in the development environment. Similar work [17] captures the execution of a program for *post-mortem* analysis. Domain model objects are specified by the user and snapshotted at runtime. The evolution of these objects throughout the program execution can be visualized afterwards. It rather focuses on program execution analysis rather than specifically on debugging, but understanding a program is part of the debugging process. We believe that the *Pillar bug* could be easily solved with such technique. The main difference with Collectors is the semantics and the grain of the recorded elements. While [17] focuses on recording how domain objects are connected and evolve during the execution, Collectors record objects and meta-information resulting from the evaluation of an expression (and its sub-expressions). When going back-in-time, we see less information because only manually selected expressions are recorded. We cannot know which other objects were referencing a collected object, nor when theses references were created. But we also see more information, because transient objects computed within the body of a complex expression are recorded and can be accessed during the *post-mortem* analysis.

The original work describing the *Pillar bug* also advocates for more advanced breakpoints [6]. Specifically, it is argued that breakpoints could improve the debugging activity with more contextual and accurate information. In its current form, the Collectors back-in-time feature could be integrated as a new kind of breakpoint, or an extended *watchpoint*, but providing a more accurate history execution than just a single recorded value. This idea of a local history is also emerging in other approaches and tools [15, 16], which integrate a record of the last local evaluations in the debugger. This history is however only accessible if the program halts, for example through a breakpoint or following an exception. The Collectors back-in-time inspector shows a full history of the instrumented expression, allowing a local comparison of healthy and problematic executions.

7 CONCLUSION AND FUTURE WORK

We described Collectors for back-in-time debugging through in-memory logging: the developer can instrument an expression from the source code, and at runtime the object resulting from the evaluation of the expression is collected and recorded. Meta-information is also recorded, such as the execution stack that leads to a particular evaluation. This provides an evaluation history of the instrumented expression, and the ability to navigate back-in-time in the evaluation(s) of this expression. We presented the back-in-time inspector, which provides views on the expression evaluation history to ease the analysis of the recorded objects and meta-information.

Using Collectors and the back-in-time inspector, we have been able to quickly understand and solve a complex bug, namely the *Pillar bug*, for which there was no solution prior to our investigation. The *Pillar bug* illustrates a kind of bug in which a trivial modification leads to an error which seems completely unrelated to the modification. It makes such bug difficult to understand and to fix.

Because of the integration of the tool in the language and the fine grained scoping of the back-in-time features, we could select which expression(s) to instrument in only a few non-intrusive actions performed on the source code. After the test execution, the recorded meta-data, such as the execution stack and the collected objects from the evaluation of the sub-expressions of instrumented expressions, provided crucial information to understand the source of the bug.

In future work, we want to study the probable performance and memory bottlenecks. Such problems are already studied in the literature, which will be the starting point for our future investigation.

We plan to evaluate further the tool and its applicability to debugging following two directions. First, we want to benchmark the tool on a bug database. We would like to study the tool performance compared to traditional debugging, or their complementarity, on different kind of bugs. The Pharo bug list, from the official Pharo language bug tracker, could serve as a reference database for such benchmarking. Second, we would like to perform a control experiment with developers – some debugging with our tool and some without –

to assess the concrete impact of our tool on the debugging activity.

We also plan to experiment more variations of the tool for debugging, based on the Collectors model. First, we would like a fully integrated into the debugging environment as a new kind of breakpoint or as an extended watchpoint. Second, the model could be used to provide access to recorded objects from any breakpoint at runtime. These objects could serve in conditional expressions of other traditional breakpoints. Finally, recorded objects could be used as replay values, to replace the result of specific expressions evaluations at runtime. That could bring an interesting feature in a runtime program in which we need to systematically reproduce the same result of a non-deterministic computation that provokes a program failure.

ACKNOWLEDGMENTS

We would like to thank Guillermo Polito, Stéphane Ducasse, Marcus Denker, and Thomas Dupriez for their very helpful comments on the early revisions of this paper and for the detailed description of the *Pillar bug*. Special thanks to Alexandre Bergel for the fruitful discussions about what could be done with Collectors and for testing early prototypes. We would also like to thank the anonymous reviewers for their extensive and very helpful review of the paper.

REFERENCES

[1] 2018. Chronon Time Travelling Debugger | Chronon. (2018). http://chrononsystems.com/products/chronon-time-travelling-debugger (accessed 2018-06-05).

[2] Thibault Arloing, Yann Dubois, Stéphane Ducasse, and Damien Cassou. 2016. Pillar: A Versatile and Extensible Lightweight Markup Language. In *Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies*. ACM, 25.

[3] Andrew P Black, Oscar Nierstrasz, Stéphane Ducasse, and Damien Pollet. 2010. *Pharo by example*. Lulu. com.

[4] Steven Costiou, Mickaël Kerboeuf, Alain Plantec, and Marcus Denker. 2018. Collectors (to be published). In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (Programming '18 Companion)*. ACM, 9. (authors version available at: https://kloum.io/costiou/collectors/collectors-authors-version.pdf).

[5] Marcus Denker. 2008. Sub-method Structural and Behavioral Reflection. (2008).

[6] Thomas Dupriez, Guillermo Polito, and Stéphane Ducasse. 2017. Analysis and exploration for new generation debuggers. In *Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies*. ACM, 5.

[7] Marc Eisenstadt. 1997. My hairiest bug war stories. *Commun. ACM* 40, 4 (1997), 30–37.

[8] Bil Lewis. 2003. Debugging Backwards in Time. *CoRR* cs.SE/0310016 (2003). http://arxiv.org/abs/cs.SE/0310016

[9] Adrian Lienhard, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. 2006. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*. 39–43.

[10] Adrian Lienhard, Julien Fierz, and Oscar Nierstrasz. 2009. Flow-centric, back-in-time debugging. In *International Conference on Objects, Components, Models and Patterns*. Springer, 272–288.

[11] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. 2008. Practical object-oriented back-in-time debugging. In *European Conference on Object-Oriented Programming*. Springer, 592–615.

[12] Frédéric Pluquet, Stefan Langerman, and Roel Wuyts. 2009. Executing Code in the Past: Efficient In-memory Object Graph Versioning. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 771–772.

[13] Guillaume Pothier and Éric Tanter. 2009. Back to the future: Omniscient debugging. *IEEE software* 26, 6 (2009).

[14] Guillaume Pothier, Éric Tanter, and José Piquer. 2007. Scalable omniscient debugging. *ACM SIGPLAN Notices* 42, 10 (2007), 535–552.

[15] Stefan Schulz. 2017. Back-In-Time Evaluation: Towards Online Trace-Based Debugging. In *Companion to the First International Conference on the Art, Science and Engineering of Programming (Programming '17)*. ACM, New York, NY, USA, 40:1–40:2. DOI: http://dx.doi.org/10.1145/3079368.3079373

[16] Stefan Schulz and Christoph Bockisch. 2017. RedShell: Online Back-In-Time Debugging. In *Companion to the first International Conference on the Art, Science and Engineering of Programming*. ACM, 1.

[17] Peter Uhnák and Robert Pergl. 2017. Ad-hoc Runtime Object Structure Visualizations with MetaLinks. In *Proceedings of the 12th edition of the International Workshop on Smalltalk Technologies*. ACM, 7.