Ease Virtual Machine Level Tooling with Language Level Ordinary Object Pointers

Pierre Misse-Chanabier
pierre.misse-chanabier@inria.fr
Univ Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL.
Lille, France

Théo Rogliano
theo.rogliano@inria.fr
Univ Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL.
Lille, France

ABSTRACT

Virtual Machines (VMs) are critical language execution engines. When tooling the VM level, developers face an important abstraction gap. For instance, a VM supporting an Object-oriented Programming language often manipulates its memory using addresses whereas these addresses are hidden in the language this VM supports. This discourages tooling at the VM level.

We propose to use language level object ordinary pointer (LLOOP) to reduce the abstraction gap. LLOOP combine VM level and language level knowledge at the VM level to ease VM tooling. We present our implementation on the Pharo language, which is supported by the Pharo VM. Moreover, we created two tools solving two real-world major bugs in the Pharo environment. These tools required VM level support.

First, we investigate how to fix a meta error that was breaking a Pharo environment, preventing it to open. We repair the broken environment by tracking and fixing the language level method responsible for the error at the VM level. Second, we investigate a corrupted Pharo image. A few objects in the Pharo memory space were corrupted *i.e.,* the VM was not able to read and manipulate them. We are able to identify and remove the corrupted objects, fixing the Pharo environment.

CCS CONCEPTS

• **Software and its engineering** → **Runtime environments**; **Software maintenance tools**; **Software defect analysis**.

KEYWORDS

Virtual Machine, Language Virtual Machine, Virtual Machine Tooling, Abstraction Gap, Memory Corruption, Meta-circular Environment, Meta Error,

ACM Reference Format:
Pierre Misse-Chanabier and Théo Rogliano. 2022. Ease Virtual Machine Level Tooling with Language Level Ordinary Object Pointers. In *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '22), December 05, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3563838.3567676

1 INTRODUCTION

Language Virtual Machines (VMs) are critical language execution engines. Testing and debugging a VM is a laborious task without proper tooling [29]. Many VMs are developed in low level languages [40] or in low level style [7, 19, 20]. Additionally they are often used to support high level languages such as object-oriented programming languages [6]. Therefore there is an **abstraction gap** between a language and its VM [22].

Developers at the VM level often manipulate *Ordinary Object Pointers* (OOPs) whereas an object-oriented developer manipulates *objects* at the language level. Moreover, VMs constrain the kind of languages they execute. For instance, a VM may fixes a specific structures for classes implementation. This is VM level knowledge. The language implementation may vary, as long as it respects these VM constraints. VM level objects represents language level implementation details. Tooling the language at the VM level therefore does not only require developers to know the VM level, but it also require them to know the language level.

To bridge this gap, approaches such as meta-circular VMs propose to develop a VM in the same language it interprets [2, 3, 33, 36–38]. This reduces the semantic gap by allowing a language level developer to read and modify the VM code in the same language they use. Moreover many academic works investigate tooling languages at the VM level, however they mainly focus on specific applications. It has been used particularly for debugging [1, 9, 11, 12, 22, 40], profiling [22, 34] and memory visualization [5, 15, 30, 31] purposes. For instance, multiple works have been possible thanks to the Java Virtual Machine Tooling Interface (JVM TI) [9]. However this only exposes VM level knowledge. Moreover, many VMs do not provide a tooling interface. These multi-level knowledge requirements combined with the abstraction gap discourages VM level tooling and limits innovation from outside the VM community.

We propose to bridge the abstraction gap by manipulating Language Level OOPs (LLOOPs). A LLOOP is a reified representation of an OOP. It extracts not only VM level information from the OOP, but also language level information. It allows developers to have all the language level information at the VM level. Moreover, it provides developers with language level entities that manipulate the VM level memory. LLOOP therefore lowers VM level knowledge requirements for developers. For example a VM tool developer does not need to know if the VM uses free objects for its allocation scheme if she wants to offer a behavior only affecting living objects. Moreover a key advantage of LLOOP is that it can be added as a layer on top of OOPs without VM modifications.

This project objective fits effort around VMs like the Jikes Research VM [3]. The JikesRVM is a meta-circular JVM that aimed at enabling VM level research. We argue that easing access to the VM level will enable more people to create VM level tools in the same manner the JikesRVM enabled VM level research.

We implement LLOOPs on the Pharo VM in an implementation we call Polyphemus[1]. Pharo [4] general purpose language derived from Smalltalk [13]. The Pharo VM is written in a low level style in a restricted Pharo translated to C code [19, 20, 26]. This is an **abstraction gap** for Pharo developers wanting to tool the VM level.

We validate LLOOP by presenting multiple case studies. We present how Polyphemus allowed us to implement basic VM tooling such as an inspector and memory visualizers. Moreover, we present two real-world major bugs which we were able to fix with ease with Polyphemus. We describe how Polyphemus helped us create these tools at the VM level. First, we investigate how to fix a meta error that was breaking a Pharo image[2], preventing it to open. A meta-error is an error that prevents the environment to execute as expected [8]. We repair this broken image by tracking and fixing the method responsible for the error at the VM level. Second, we investigate a Pharo image containing a memory corruption [23]. A memory corruption happens when an OOP does not respect the object representation. Multiple objects in the Pharo memory were corrupted *i.e.,* the VM was not able to read them. We are able to identify and remove the corrupted objects, fixing the Pharo image.

The structure of this paper is as follow. First, we present why manipulating low level entities is cumbersome by searching for a class OOP (Section 2). Then we describe LLOOP and how it is implemented (Section 3). We then validate this work with multiple case studies explaining how LLOOP eases tooling the VM level on the Pharo VM (Section 4). We finish this paper with related work (Section 5), future works (Section 6) and the conclusion (Section 7).

2 ABSTRACTION GAP BY EXAMPLE

We present in this section an example of how to find a class object at the VM level. This example is based on the Pharo VM (Section 2.1.1). We provide some context about the VM level (Section 2.1.2) and the language level (Section 2.1.3) to ease the reading of this section. We then describe in detail how a developer would find a specific class at the VM level (Section 2.2). Finally, we highlight and analyze the abstraction gap present in this example (Section 2.3).

2.1 Example Context

2.1.1 The Pharo VM. Pharo [4] is an open-source, pure object-oriented, dynamically-typed, general purpose language derived from Smalltalk [13]. The Pharo VM implements at the core of its execution engine a threaded bytecode interpreter, a linear non-optimising JIT compiler named Cogit [25] that includes polymorphic inline caches [18] and a generational scavenger garbage collector that uses a copy collector for young objects and a mark-compact collector for older objects [35]. Pharo supports many industrial applications as well as research projects.[3]

[1]https://github.com/hogoww/Polyphemus
[2]An image is a memory snapshot.
[3]https://consortium.pharo.org

The Pharo VM is a continuation of the work of the OpenSmalltalk-VM[4]. It is written in Pharo itself and generated to C using a VM-specific translator [19]. Executing the Pharo VM in Pharo is called *simulating* the VM [20, 26]. Pharo VM developers work mostly in the simulation [29]. The full content of this paper takes place at the level of the simulated VM (Figure 1).

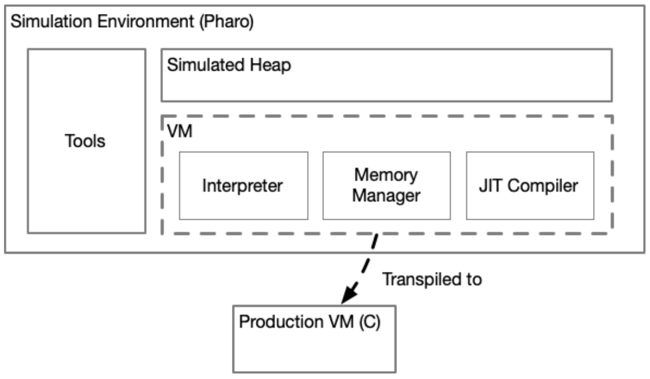


Figure 1: Simulated VM infrastructure for the Pharo VM. The main VM components are transpiled to C code to create the production Pharo VM. Tools are developed in Pharo and executed in the simulation environment.

Pharo implements a snapshot feature, which saves the state of its memory into a file *i.e.,* an **image**. This allows the VM to restart executions right at the point it was last saved. This enables Pharo VM developers to work on a frozen state of the dynamic memory of a program. To work on a given image, we load it in the simulation. Snapshot and images features have recently risen in popularity to support cloud computing [39].

2.1.2 VM level Context.

Objects are OOPs. The Pharo VM is developed to support an object-oriented programming language. The VM however, manipulates Ordinary Object Pointers (OOP). OOPs are pointers to a memory location containing the object. The VM represents objects as a header and the object content (Figure 2). The header contains information on how the content is encoded and how to decode it. For instance, it contains the layout, the number of elements and the class of the object represented by the OOP [16].

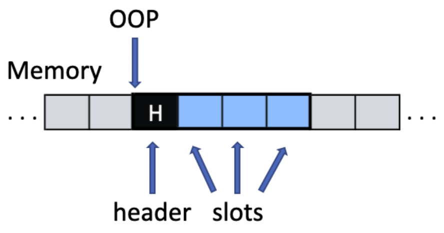


Figure 2: OOP structure. The VM represents objects as a header and the object content. An OOP points to the header.

[4]https://opensmalltalk.org

Object VM Constraints. The VM has a few requirements on the structure of some objects. These requirements are the hypothesis that the execution engine uses to execute code. As long as these constraints are met, any language is executable by the Pharo VM [17]. For example, the Pharo VM constrains the representation of classes. Each class is required to have in its first three slots[5], in this order: superclass, method dictionary and format. This allows the language to add other properties to their classes at the language level.

2.1.3 Language Level Context. Pharo classes add on top of the VM requirement a few properties that the VM is not aware of. For example, any Pharo class contains objects representing its name, its layout, and its subclasses. Unlike the VM level, the order here does not matter to Pharo because developers access slots by their names. The bytecode compiler manages the slots index for the developers.

2.2 Finding an Object on the VM Level

A VM developer finds an object that she suspects has a problem. For example, a common issue is for an object to be referenced in an unexpected place, causing memory leaks [14]. She wants to investigate how to debug such an object in the VM. Therefore she wants to find the corresponding object on the VM side. The question she asks herself is: *What is that object's OOP ?*

For the sake of simplicity, we describe here how to find a class OOP rather than any OOP. This section objective is to make the reader feel how cumbersome this process is at the VM level without LLOOPs. All the code snippets used in this section are custom-made for the sake of the example.

At the VM level, the developer only has OOPs. We therefore rely on properties that are available at the VM level. We are able to ask for the number of elements the object contains or the layout of the object. However filtering the memory with these properties yields too many results. Finding an OOP inside the class we look for is the same issue as finding the initial OOP. The most promising approach to find a class OOP is to filter classes by their name. This is possible because we have the language level knowledge that any class contains its name object in one of its slots.

2.2.1 Finding The Class Name OOP. To find the class name, we need the index of the slot in the class OOP representing the name of the class. The index of the slot of the object name is not known at the VM level. In Pharo, there are two classes kinds: classes and metaclasses. From the VM point of view, both are classes. These are language implementation details. However, we are able to access information that differentiate them at the VM level. The difference in memory is that a metaclass' class is the class Metaclass. Metaclasses names are in their sixth slot, whereas classes names are in their fifth slot (Listing 1) Unlike Pharo level, the indexes values at the VM level are 0-based.

```
1  classNameIndexForOop: anOop
2    | arrayClassOOP metaclassOOP isMetaClass |
3    arrayClassOOP := memory classAtIndex: (memory
       arrayClassIndex).
4    metaclassOOP := memory fetchClassOf: (memory
       fetchClassOf: arrayClassOOP).
5
6    isMetaClass := (memory fetchClassOf: anOop) =
       metaclassOOP.
7    ↑ isMetaClass
8      ifTrue: [ 6 "metaclass name index"]
9      ifFalse: [ 5 "class name index"]
```

[5]Slots are attributes in Pharo nomenclature

Listing 1: Custom method finding the class name OOP index for a class OOP at the VM level. To find the class Metaclass, we leverage the fact that the VM requires a few classes to exists, such as the class Array (Line 3). The class of Array is Array class and the class of Array class is Metaclass (Line 4). We then return the index depending on whether a class is a class or a metaclass (Lines 6 - 9).

```
1  convertStringOopToStringObject: aStringOop
2    | oopFormat isByteString numberOfElements
        aResultingString |
3    oopFormat := memory formatOf: aStringOop.
4    isByteString := oopFormat between: 16 and: 23.
5    numberOfElements := isByteString
6      ifTrue: [ memory numBytesOf: aStringOop ]
7      ifFalse: [ memory num32BitUnitsOf: aStringOop ].
8
9    aResultingString := ''.
10   0 to: numberOfElements - 1 do: [ :anIndex | | aCharacter |
11     aCharacter := isByteString
12       ifTrue: [ memory fetchByte: anIndex  ofObject:
         aStringOop ]
13       ifFalse: [ memory fetchLong32: anIndex ofObject:
         aStringOop ].
14     aResultingString := aResultingString , aCharacter
         asCharacter ].
15   ↑ aResultingString
```

Listing 2: Custom method converting a string OOP to a string object at the VM level. We start by getting the string OOP's format (Line 3) allowing us to determine whether the OOP is a ByteString or a WideString (Line 4). It also allows us to compute the size of the string (Line 5 - 7). We then fill a newly created string object (Line 9) with the characters extracted from the string OOP (Line 10 - 14). We retrieve the integer value corresponding to the character from the OOP (Lines 11 - 13). We convert the integer to a character and concatenate it to the result (Line 14). Finally we return the result (Line 15).

The next challenge is to convert the class name OOP to a readable String. Strings in Pharo are encoded in two different ways: ByteString and WideString. ByteStrings encode Strings containing only ASCII characters in 8 bits, and WideStrings encode Strings containing any non ASCII characters in 32 bits. Luckily this information does not require to get the correct String class, because this is also encoded in the header of the OOP. This is only true for objects we know are strings. Therefore we are finally able to decode classes name (Listing 2).

2.2.3 *Finding a Class.*

Finally, we are ready to search for a class by its name. The VM registers every classes created in the *Class Table*. Moreover it provides a method iterating the classes in the Class Table. With this method and the previously defined two custom methods, we are now able to filter the classes (Listing 3).

```
1  findClassNamed: aClassName
2    | classNameIndex classNameOop className |
3    memory classTableEntriesDo: [ :aClassOop |
4      aClassOop = memory nilOOP
5        ifTrue: [ "not a class, nothing to do" ]
6        ifFalse: [
7          classNameIndex :=self classNameIndexForOop:
     aClassOop.
8          classNameOop := memory fetchPointer:
     classNameIndex ofObject: address.
9          className := self convertStringOopToStringObject:
     classNameOop.
10         className = aClassName ifTrue: [ ↑ aClassOop ]]].
11   ↑ memory nilOOP
```

Listing 3: Custom method to find a class by its name at the VM level. Some entries of the class table are nilOOP, we therefore have to ensure we do not process them (Line 4 - 5). We compute the class name OOP index for the current iteration (Line 7), get the class name OOP (Line 8) and convert it to a string object (Line 9). Finally we compare the converted string with the argument (Line 10). If the class is not found, we simply return nil (Line 11).

Note that this example is valid only under the hypothesis that the class name does not move, and always contains a string OOP.

2.3 Analyzing the Abstraction Gap

This example exhibits multiple occurrences of the abstraction gap (AG) a developer encounters when working in the Pharo VM level:

OOP Manipulation. (AG1) Both the VM and the language manipulate the same objects. However the VM manipulates them as addresses in a C style whereas the language manipulates objects with message sends and method lookup.

Encoded Information. (AG2) The memory is interpreted differently by the VM and the language. The VM level has access to every information encoded, but is only aware of a subset of what is encoded in the memory. For example, the VM level knows a given object address but does not know the class oop name. Therefore tooling at the VM level without language level knowledge misses many information.

Naming. (AG3) The naming of some concepts may differ depending on the level of abstraction. This is illustrated by the fact that what is called *format* at the VM level in this example, is called *instance specification* at the Pharo level.

Off By One Errors. (AG4) The VM level uses 0-based indexes whereas the language level uses 1-based indexes.

On top of the abstraction gap, this VM and language levels mixed process is cumbersome. The time and effort required finding a class alone is already significant. The process is worsened when looking for any other object that has less distinctive properties. All of those reasons make tooling the VM level difficult. We propose in this paper to ease these difficulties by interacting with an abstraction layer combining VM level and language level knowledge.

3 SOLUTION: LANGUAGE LEVEL ORDINARY OBJECT POINTERS

Creating LLOOP relies on three key points. We present them with the support of our implementation Polyphemus, on the Pharo VM (Section 3.1). We then present implementations details to adapt on other systems (Section 3.2). Finally, we present how to find a class at the VM level by using Polyphemus (Section 3.3).

3.1 LLOOP in a nutshell

Bridging the abstraction with LLOOPs resides in three key points:

(1) Language level entities;
(2) Imbued VM level knowledge;
(3) Imbued language level knowledge.

First, this enables developers to work on the same kinds of entities they are used to at the language level (1). Imbuing these entities with VM level knowledge enables the newly created entities to be manipulated at the VM level (2). Then, adding language level knowledge enables the developer to manipulate a VM level entity the same way she would manipulate them at the language level (3). The user interacts with an abstraction that provide access to these entities. We illustrate these points further on the Pharo language.

The Pharo VM uses OOPs. Pharo developers use objects. Therefore, we reify OOPs into LLOOPs (1). At the VM level, the only way to manipulate objects are with VM functions taking OOPs as parameter. To manipulate these newly created objects, we implement the necessary API to manipulate them. In our case, this relies on accessing the header and the object contents (2). Finally, LLOOPs are manipulated by slot names, similarly to the language level (3).

In Pharo, classes are objects. A class is either a class or metaclass. Therefore we offer two reifications: LLOOPClass and LLOOPMetaclass (1). Both implement the method #isMetaclass. LLOOPClass returns false whereas LLOOPMetaclass true (2). Finally we are able to ask for the slot names of any class (3).

3.2 Implementing LLOOP

We implement LLOOPs on top of the Pharo VM simulator. Creating an LLOOP requires to be able to identify what kind of entity is represented by an OOP (Section 3.2.1). Moreover for many applications, it is particularly useful to create LLOOPs for every OOP in memory for further analysis (Section 3.2.2). Finally we describe identified memories, the layer that stores the LLOOPs (Section 3.2.3).

3.2.1 *Identifying an OOP.*

The objects implement both a VM level view (such as addresses) and a language level view (such as classes names). Each OOP is reified as an object that reflects its role. This role is either from the VM point (class table) of view or the language point of view (metaclasses) or both (bit indexables). To identify how to reify a given entity, we mix VM level knowledge with language level knowledge (Listing 4). If no fitting identification is found, the object has no particular property that the user should be aware of and is identified as a regular object. Moreover, a few VM-level specific objects such as the Class Table are identified as their own objects kind to add VM specific knowledge.

```
"LLOOPClass class"
canHandle: anOOP memory: aMemory
  "VM level knowledge"
  (aMemory isInClassTable: anOOP) ifFalse: [ ↑ false ].
  "Language level knowledge"
  (LLOOPMetaclass canHandle: anAddress memory:
      aMemory) ifTrue: [ ↑ false ].
  ↑ true
```

Listing 4: This listing presents how Polyphemus identifies that an OOP is a class. At the VM level, a class OOP is an OOP that is registered in the class table (Line 4). If a class is actually a metaclass, then it is not a class (Line 6).

3.2.2 *Iterating Over the Memory.*

We iterate over all of the memory to identify every OOP (Figure 3). This is because we know where the memory starts and ends and that the VM works under the hypothesis that the managed memory is filled with OOPs. Any OOP inside the managed memory has the same structure: a header followed by its contents. This is not only true for objects, but also for VM specific *objectoïds* such as free objects which represent the available memory used during object allocation. The header of any OOPs knows the size in bytes of its contents. Therefore iterating the memory consists of taking the first address in memory, and adding the size of the current object to find the next one. All the identified objects are stored in an identified memory.

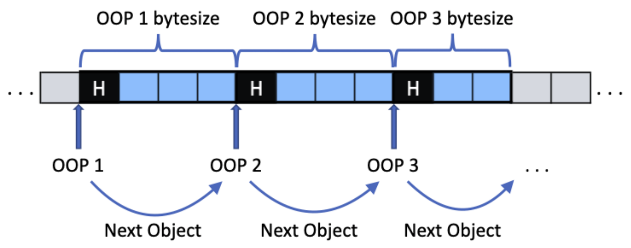


Figure 3: Process of iterating over the memory. Finding the next object is a matter of adding the size of the current object to the current OOP. This is possible because the VM works under the hypothesis that the managed memory respects the object representation.

3.2.3 *Identified Memory.*

Developers interact with a layer: an identified memory. Identifying the memory has two purposes.

First, it provides a more precise identification for multiple objects. For instance, the identified memory requires the OOP representing the class Metaclass to identify metaclasses 3.2.1. We therefore find a few specific classes OOPs before starting to iterate the memory by using VM level schemes. We presented how to get the class Metaclass in Listing 5.

```
  classNameIndexForOop: anOop
    | arrayClassOOP metaclassOOP |
    arrayClassOOP := memory classAtIndex: (memory
      arrayClassIndex).
    metaclassOOP := memory fetchClassOf: (memory
      fetchClassOf: arrayClassOOP).
```

Listing 5: VM level scheme to find the class Metaclass. This scheme is the same one we use in Section 2.

Second it creates a layer between the actual memory and the developer. This allows us to cache the LLOOPs already created which speeds up analysis. We use two identified memory heuristics.

Fully identified memory. The *fully-identified memory* identifies every single OOP at the time of its creation. A default Pharo image is about 50 megabytes, which is about 840 000 objects. Identifying all these entities takes time. Moreover, by default we only identify an entity once. Therefore in fully-identified memory we simply iterate over the memory and create all of the objects, regardless of whether they were already identified. This speeds up the creation of this memory significantly. Identifying precisely rare objects is removed from the identification process. They are created explicitly in a post iteration step. This is for instance the case of the class table which is a VM level object which has only one instance.

Lazily-identified memory. The *lazily-identified memory* creates LLOOPS lazily. Upon asking for an object in the identified memory, if the object was already identified, it is returned. If it was not, it is then identified before being returned. Similarly to the fully identified memory, multiple specific objects such as the class table LLOOP are explicitly instantiated for performance reasons. Therefore they are created when the lazily identified memory is created. This is useful when a tool does not need to access every objects, such as for the class browser 6.

```
findClassName: aClassName
  identifiedMemory classes do: [ :aClassOop |
    aClassOop oopClassName asString = aClassName ifTrue:
      [ ↑ aClassOop ]].
  ↑ identifiedMemory reifiedNilObject
```

Listing 6: Custom method to find a class by name by using LLOOPs in Polyphemus. We iterate over the classes (Line 2). If we find the class oop, we return it (Line 3). If we do not, we return the nil LLOOP. This method is significantly smaller than the required work presented in Section 2 and in particular Listing 3. Moreover, its style is closer to what a Pharo developer encounters on a daily basis.

3.3 Finding a Class in Polyphemus

Searching for a class object by its name at the VM level in Polyphemus uses a object-oriented programming style. Listing 6 presents Polyphemus way of finding a class.

We solve almost all the abstraction gaps we identified in Section 2.3. Polyphemus presents users with LLOOPs rather than OOPs (AG1). Moreover the available API depends on the kind of LLOOPs we are working on *e.g.,* classes provide accessors to their class name (AG2) and string OOPs provide a conversion of themselves to Pharo strings (AG2). Finally the LLOOPs layer uses indexes that are 1-based, reflecting Pharo level collections behavior (AG4). The naming issue however (AG3), is difficult to overcome because our approach resides in mixing the language level and the VM level.

LLOOP and Polyphemus allowed us to create multiple tooling at the VM level for various purpose. We validate our approach by presenting in the following multiple case studies of how Polyphemus eased the development of those tools.

4 CASE STUDIES

To validate that LLOOP eases VM level tooling development we present multiple tools we developed by leveraging Polyphemus. We present how Polyphemus allowed us to implement basic VM tooling with ease such as an inspector and a memory visualizer (Section 4.1). Moreover, we present two real-world major bugs which we were able to fix with ease thanks to Polyphemus. These bugs previously resulted in developers loosing their work. We describe how Polyphemus helped us to create these tools at the VM level.

First, we investigate how to fix a meta error that was breaking a Pharo image, preventing it to open [8]. We repair the broken image by tracking and fixing the method responsible for the error at the VM level. Second, we investigate a Pharo image containing a memory corruption [23]. A memory corruption happens when an OOP does not respect the object representation anymore. A few objects in the Pharo memory space were corrupted *i.e.,* the VM was not able to read them. We were able to identify and remove the corrupted objects (Section 4.3).

4.1 Improving VM Development

We start by presenting multiple basic VM tools we created with Polyphemus. First we present how naming VM entities eases VM development (Section 4.1.1). We then present two basic tool complementing the manipulation operations over the memory (Section 4.1.2). Finally, we present two visualizations we created to look at the memory (Section 4.1.3).

4.1.1 Named VM entities. In Polyphemus, we have added additional VM level knowledge. Particularly, we name and even reify VM level special entities. The Pharo VM uses multiple VM level objects that are either completely hidden from the Pharo language (*e.g.,* Class Table, Remember Set) or available but with restricted use (*e.g.,* Special Object Array) at the language level. This allows new developers to immediately know that an OOP is used at the VM level, with a particular name, and a precise purpose.

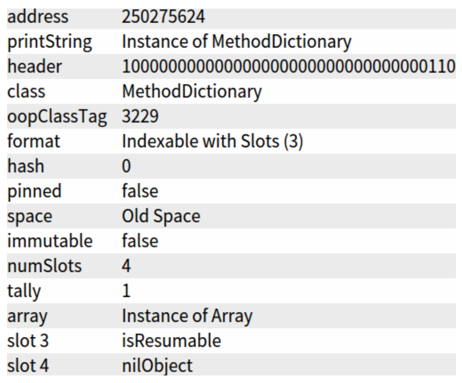
Special LLOOP. The *class table* is a VM level OOP. The class table registers all classes. This is how the Pharo VM recognizes whether an OOP is a class. Moreover, a few VM level entities are stored in the last slots of the class table. As this is a central object at the VM level it has a custom reification. This reification provides a custom API that reflects the VM manipulation of the VM of the class table.

Understanding VM Level Entities. When we first visualized an image (Section 4.1.3), we were wondering why there was a specific object in the middle of the memory. It has particular properties making it recognizable such as being unmovable, the second biggest object in memory and is one of the rare 64-bit indexable layout objects. Multiple Pharo VM experts, including us, did not recognize out of the box which object this was, in spite of its recognizable properties. We found that this was the *remembered set.* The *remembered set* is a VM level object that retains references to old space OOPs referencing young space OOPs. This is used by the generational garbage collection scheme used by the Pharo VM.

VM Level Variable Duplication. The *Special Object Array* is an array of special objects that the VM requires to perform some operations. It allows the VM to find quickly specific objects such as the string containing the method identifier #doesNotUnderstand: which allows the VM to send this message to objects which do not understand a method identifier. By simply naming the VM level entities, we detected a bug in the implementation of the Espell tool [17, 27]. Espell is a tool to generate images with different properties for the Pharo VM. We discovered that the special object array in images created by Espell is duplicated. Another object is allocated and filled properly whereas the object created for this purpose is left in memory without purpose. The correctly filled special object array is both recognized by the VM and the language level even though it is in the wrong place in memory.

4.1.2 Basic Tools.

LLOOP Inspecting. The Pharo IDE provides developers with an inspection system. The inspection system enables developers to explore objects and their contents. The simulated memory is an object. An OOP points to somewhere inside a simulated memory object. Therefore inspecting an OOP requires simulated memory object. Unlike an OOP which is an integer, a LLOOP contains the whole context necessary to inspect it. Therefore a developer is able to inspect any given LLOOP (Section 4). A VM developer would need to query the simulator to show each of the VM level values displayed. Moreover, the inspector displays language level information that it extracts from the underlying OOP such as the slot names and the class name. The VM level has a complicated custom function that provides a similar behavior (Listing 7).



address	250275624
printString	Instance of MethodDictionary
header	100000000000000000000000000000000110
class	MethodDictionary
oopClassTag	3229
format	Indexable with Slots (3)
hash	0
pinned	false
space	Old Space
immutable	false
numSlots	4
tally	1
array	Instance of Array
slot 3	isResumable
slot 4	nilObject

Figure 4: LLOOP inspector. The inspector displays the VM level information in a readable manner,such as the number of slots, the format and the hash. The inspector proposes to inspect the slots with their names and references to the corresponding objects. This is the case here of the slots tally and array. Slot 3 and 4 are numbered because they are unnamed indexed references, accessible only by their indexes.

```
16rEEAE728: a(n) MethodDictionary (16rC9D=>16rEE8AA88)
        format 16r3 size 2 hdr8 ..... hash 16r0
0       16r9 =1 (16r1)
1  16rEF16690 an Array
2  16rEF14C18 #isResumable
3  16rEE73720 nil
```

Listing 7: #printOop:. VM method displaying information about the same OOP inspecting in Figure 4. It lacks not only the interactive properties Pharo developers are used to, it lacks the ease of access to information that our inspector provides. Moreover, the inspector is easily customized for a given receiver.

LLOOP Referencers. OOPs contain references to other OOPs. However, an OOP does not know which OOPs reference it. Therefore we needed a tool to find these references. This is achieved quite simply with Polyphemus by iterating over the heap and looking at the slots of each object. This is because each LLOOP knows which of its slots are references. Particularly, only part of the slots of a compiled method contain references, and indexable OOPs contain no references. This tool is the base that is used in other contexts. Particularly, this allowed us to confirm easily that the old special object array in Section 4.1.1 was indeed unreferenced. This allowed us to investigate the biggest objects present in Pharo to reduce the base Pharo image size.

4.1.3 *Visualization Tools.*

Memory Visualization. Another objective we had was to visualize the memory in multiple ways. Particularly we created a few visualizations at the LLOOP granularity (Figure 5).

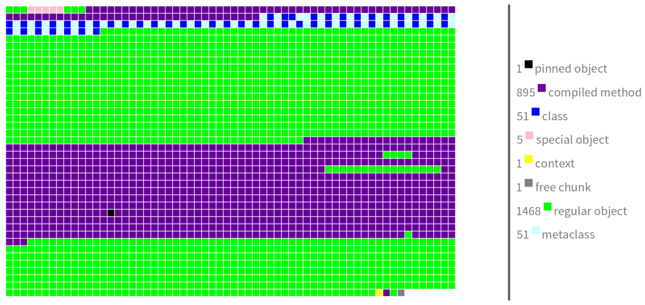
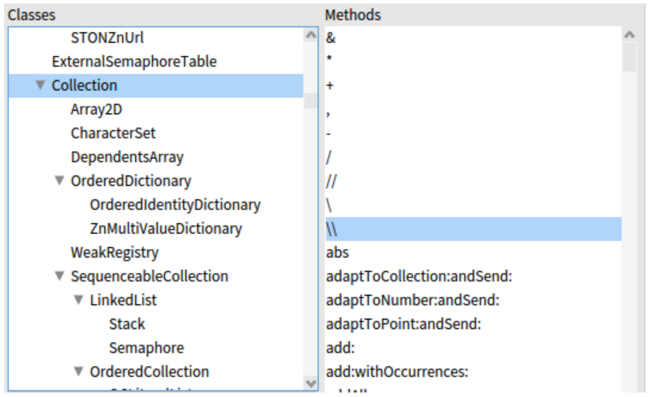


Figure 5: Memory inspector presenting a very small Pharo image [17]. Each box represents a LLOOP. The statistic and legends are dynamically computed and presented on the right panel. A developer is able to click on a particular box to inspect it.

For instance, the Pharo VM implements unmovable objects (pinned objects). These objects are not moved during the compaction phase of the garbage collector. Understanding how the compaction and unmovable objects are interacting was difficult. Using the visualization, it became trivial.

Class Browser. The Pharo IDE browses classes and their methods in a meta-circular browser. We provide a similar browser on the image loaded in the simulator (Figure 6).



Classes	Methods
STONZnUrl	&
ExternalSemaphoreTable	*
▼ Collection	+
Array2D	,
CharacterSet	-
DependentsArray	/
▼ OrderedDictionary	//
OrderedIdentityDictionary	\
ZnMultiValueDictionary	\\
WeakRegistry	abs
▼ SequenceableCollection	adaptToCollection:andSend:
▼ LinkedList	adaptToNumber:andSend:
Stack	adaptToPoint:andSend:
Semaphore	add:
▼ OrderedCollection	add:withOccurrences:

Figure 6: Class browser on the simulated memory.

4.2 Repairing a Meta Error

A meta-error is an error that prevents the environment to execute as expected. For instance an error occurring in the debugger, opens a debugger, which also encounters that error, and so on [8].

We were presented with a Pharo image containing a meta error in the Pharo UI. Most meta errors are minor and are fixed within the image. However a meta error in some components such as the UI, compiler or debugger breaks the image and requires VM level support to be fixed. Pharo developers encountering such meta error simply lose their work. This is worsened by the fact that Pharo developers work lives in a live programming environment that is in the dynamic memory rather than in multiple files. Therefore all of the current classes and methods are created inside the image, and saved all at once.

We present in this section how a Pharo developer encountered such a meta error (Section 4.2.1). We then explain how Polyphemus allowed us to find the object responsible for this meta error and fix it (Section 4.2.2). Finally we explain how Polyphemus allowed us to only use language level knowledge and detail the VM level knowledge that a Pharo developer would have needed to achieve this behavior without LLOOP (Section 4.2.3).

4.2.1 *Meta Error by Example.*

In our research team, we had a student that was working on improving the meta-circular Pharo UI from within a Pharo image. She was modifying sensitive user interface code. At some point, she added a halt[6] to explore a method. She proceeded to save her image to ensure she would be able to recover her execution at a safe point. She then closed the Pharo image. When she tried to open the image later, she was faced with a debug stack (Figure 7). The debug stack presents the state of the frozen execution. It shows that the error freezing the execution is the halt that the student put. Moreover, the debug stack shows which method is throwing the halt.

This image therefore was encountering a *meta error* and could not open nor compile anymore. Such errors are also described in other contexts such as debuggers [8]. Moreover Pharo allows developers to modify any method in the system. Therefore any developer may inadvertently modify a method used by Pharo core tools with unexpected consequences. Many Pharo developers hack the system

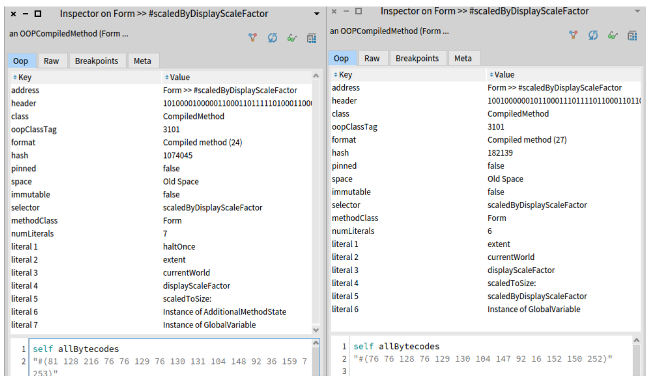
[6]halts are Pharo breakpoints

```
Halt
SmallInteger(Object)>>haltOnce
Form>>scaledByDisplayScaleFactor
ThemeIcons>>iconNamed:
MorphicRootRenderer(Object)>>iconNamed:
MorphicRootRenderer(OSWorldRenderer)>>setAttributesDefault
MorphicRootRenderer class(OSWorldRenderer class)>>forWorld:
[ :arg5 | tmp2 := arg5 forWorld: arg1 ] in AbstractWorldRendere
FullBlockClosure(BlockClosure)>>cull:
[ :arg4 | (arg1 value: arg4) ifTrue: [ ^ arg2 cull: arg4 ] ] in
 arg2 cull...etc...
OrderedCollection>>do:
OrderedCollection(Collection)>>detect:ifFound:ifNone:
OrderedCollection(Collection)>>detect:ifFound:
AbstractWorldRenderer class>>detectCorrectOneForWorld:
```

Figure 7: Debug stack printed by the Pharo VM when trying to open the broken Pharo image from a command line. The last method executed is at the top. This shows that (1) the error preventing the user interface to open is a halt (line 1) and (2) that the method throwing the halt is the method #scaleByDisplayScaleFactor in the class Form (line 3).

for many purposes, and encounter meta errors regularly. Upon encountering a meta error breaking an image, the image is considered **dead**. At this point, the only solution to Pharo developers is to throw the image away and to start fresh from a new image. Developer loses all the code that was not saved nor committed: usually at least the last edited method.

4.2.2 Meta Error Repair Steps. We opened the dead image inside a VM simulator. Using the class browser described in Section 4.1.2, we are able to track down and inspect the method containing the halt graphically and quickly. Once we had the method LLOOP, we were able to open a graphical inspector displaying many information. It allowed us to compare the broken method with a version without the halt graphically (Figure 8).


Inspector on Form >> #scaledByDisplayScaleFactor
an OOPCompiledMethod (Form ...
Oop
Raw
Breakpoints
Meta

Key	Value
address	Form >> #scaledByDisplayScaleFactor
header	10100001000001100011011111010001100
class	CompiledMethod
oopClassTag	3101
format	Compiled method (24)
hash	1074045
pinned	false
space	Old Space
immutable	false
selector	scaledByDisplayScaleFactor
methodClass	Form
numLiterals	7
literal 1	haltOnce
literal 2	extent
literal 3	currentWorld
literal 4	displayScaleFactor
literal 5	scaledToSize:
literal 6	Instance of AdditionalMethodState
literal 7	Instance of GlobalVariable

```
self allBytecodes
"#(81 128 216 76 76 129 76 130 131 104 148 92 36 159 7
253)"
```

Key	Value
address	Form >> #scaledByDisplayScaleFactor
header	10010000001011000111011110110001101
class	CompiledMethod
oopClassTag	3101
format	Compiled method (27)
hash	182139
pinned	false
space	Old Space
immutable	false
selector	scaledByDisplayScaleFactor
methodClass	Form
numLiterals	6
literal 1	extent
literal 2	currentWorld
literal 3	displayScaleFactor
literal 4	scaledToSize:
literal 5	scaledByDisplayScaleFactor
literal 6	Instance of GlobalVariable

```
self allBytecodes
"#(76 76 128 76 129 130 104 147 92 16 152 150 252)"
```

Figure 8: Inspector of the broken compiled method LLOOP on the left and the working compiled method LLOOP on the right. The broken method has three more bytecodes, which send the message haltOnce. It also contains an extra literal holding the selector *#haltOnce* in the first literal index which is used by the message send bytecode.

We first had to remove the message send triggering the halt from the bytecodes (Listing 8). We replaced each bytecode with the bytecode of the working version. Moreover, the object structural properties cannot be changed easily and we could not reduce its bytecodes size. However, the last bytecode is always a return. The extra bytecodes are after a return, and will not be executed.

```
replacementBytecodes := #(76 76 128 76 129 130 104 147 92
   16 152 150 252).
replacementBytecodes doWithIndex: [ :aBytecode :anIndex
   |
  brokenMethod bytecodeAt: anIndex put: aBytecode ]
```

Listing 8: Code patching the method bytecodes. The variable replacementsBytecodes is initialized to the working method bytecodes. We then replace each bytecode in the method OOP by sending messages to its LLOOP. This removes the bytecodes sending the message triggering the halt.

Moreover, to send a message, the bytecode pushes the selector of that message on the execution stack. This selector reference is stored inside the method literals. In this case, we see that the *#haltOnce* selector is stored in the first literal (Figure 8). Since the bytecode of the working method version considers only six literals instead of seven, the literals indexes in the bytecode are off by one. We replace each literal with the previous one to fix the literals.

After the patch is applied on the memory, we snapshot the image which replaces the image file. This repairs the Pharo image and allows the developer to recover her work.

4.2.3 Repair Knowledge Requirement Analysis. Repairing this dead image required only the following Pharo level knowledge:

- Methods are objects containing bytecodes and literals;
- Sending a message in a method, puts a reference to the message selector into the method literals;
- The last bytecode of a method is always a return.

Similarly, the VM level knowledge burden that would have been left to the tool developer without Polyphemus were:

- Finding a given method at the VM level (see Section 2.2);
- Manipulating different parts of a compiled method. The compiled method format which is a combination of an extra header, indexable slots containing literals and 8 bit indexable containing bytecode;
- Decryption of the literals;
- Decryption of the bytecodes.

With the abstractions provided by LLOOP and Polyphemus, it took us less than an hour of work to figure out how to fix this image and recover days of work for the student.

Using Polyphemus we were able to create a tool at the VM level; that required VM level; using only Pharo level knowledge. Moreover, this tool tackles an issue that previously resulted in developers loosing their work in Pharo.

4.3 Recovering From a Memory Corruption

We were presented with a Pharo image containing a memory corruption. Similarly to the previous case study, Pharo developers encountering memory corruptions simply lose their work. Thankfully, memory corruption are exceedingly rare.

Using Polyphemus, we are able to identify the corrupted objects (*i.e.,* objects which do not respect the memory representation) and extract them. This eases understanding how the corruption happened and allowed us to fix the corrupted image to restore the object representation hypothesis. This allows the developer to recover and to resume her work.

4.3.1 Memory Corruption in Managed Language. A VM defines an object representation. This object representation defines the object structure and how the VM understands what it contains. The VM emits the hypothesis that any given OOP in memory respects the object representation. This works well under the assumption that there is no bug in the VM code that breaks this hypothesis. However, bugs inside VM code may create a particular kind of errors called *memory corruptions* [23]. A memory corruption means that somewhere in memory, some bytes have values that breaks the object representation hypothesis. Therefore the memory does not make sense to the VM anymore. Upon encountering a memory corruption, the VM behavior is undefined.

The Pharo VM simulation has been used to debug memory corruptions [20, 29]. This was possible because they had a reproducible case. However, investigating how a given memory has gotten corrupted is particularly difficult to track down, because they usually happen a long time before they trigger the crash.

4.3.2 Snapshoting a Memory Corruption. A corrupted Pharo image behavior is undefined. It often (albeit not always) results in a crash. However, a corrupted Pharo image will not immediately crash. Most likely, it will continue working until the VM manipulates a corrupted object. The VM manipulates every object in its memory during garbage collection. Moreover, upon saving the memory state, the VM starts a garbage collection to only save objects still in use. Therefore if the garbage collector creates a memory corruption while saving the memory state, the resulting image file is corrupted. The VM is able to open the corrupted memory, but is not able to save it. However while loading the memory in the simulated VM, more checks are applied on the memory. Some of these checks detected the memory corruption we were presented with.

4.3.3 Iterating a Corrupted Memory. The first step to enable LLOOP was to load the corrupted image in spite of the memory corruption. Thankfully, the simulator prevents the image to load in the simulator with assertion errors. We catch the assertion errors and force the computation to resume. At this point the memory is loaded, and the simulator state is partially corrupted. However we are able to manipulate the corrupted memory.

Polyphemus relies on iterating the memory. However, upon encountering a corruption, the iteration stops (Figure 9). We do not have enough information to continue finding objects with this strategy. Rather than looking at the memory as a line of bytes, we look at it as a graph of objects (Figure 10). This strategy iterates over the contents of each discovered object. If we encounter a corrupted object while looking at the content of an object, we ignore it and continue by looking at the next slot. Although this strategy is less sensitive to memory corruption iteration, some objects may still not be recovered. Particularly, an object that is referenced only by a corrupted object is lost.

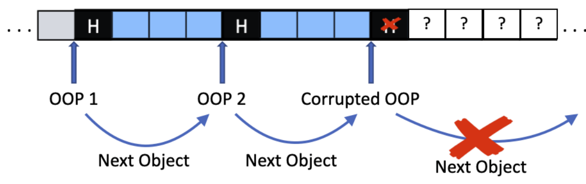


Figure 9: Iterating over the corrupted memory. Upon encountering a corruption, the iteration stops. We cannot compute what objects are after a corrupted object.

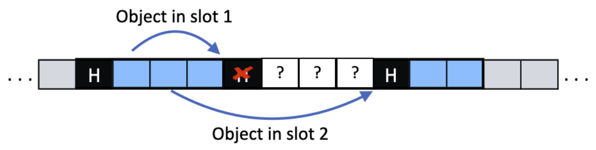


Figure 10: Iterating over the corrupted object graph. If a corrupted object reference is encountered while looking at the content of an object, we ignore it and continue by looking at the next slot.

4.3.4 Corruption Details. In this case, we found two different corruptions.

Corrupted Free Objects. To respect the object representation, the memory is filled with OOPs. Not all OOPs are objects. Some OOPs are *objectoïds*. Such OOPs are VM level entities that have the same structure as objects but are not object. This is the case of *free objects*. Free objects are used in the Pharo allocation and garbage collection schemes. Free objects are registered in two VM level objects: the *free lists* and the *free tree*. In our case, both the free lists and the free tree are corrupted. By comparing the discovered free objects to the free lists free object, Polyphemus found corrupted free objects. Particularly, a free object was found inside another free object which should never happen.

Object Containing Corrupted References. By iterating over the objects and their slots, we found objects that contained invalid references. The addresses referenced in some objects slots were not valid OOPs in the memory space. Although not an OOP, or even a valid reference, the referenced object is reified as an abnormal entity. This allows us to further analyze these entities. Particularly, we were able to discover that there were multiple objects referencing a single invalid OOP.

For both kinds of corrupted objects, Polyphemus keeps a reference to each of them for further analysis and repair.

4.3.5 Corruption Cleansing. Corrupted references in corrupted objects are replaced with another existing object. By default, we replace it with the nil oop. However we are able to replace any objects with correct domain objects if a developer is able to provide one. After having recovered as many objects as possible, Polyphemus recreates all of the free objects in memory as well as in the free lists and free tree.

By fixing the corrupted references, the corrupted free lists and tree, the image was working properly again. We tested that the image was working properly by executing the garbage collection [20]. Assuming that the corruption is not on a critical object, the image is now ready to be used again. Otherwise, the image is in a state that the tool we presented in Section 4.2 is able to fix.

4.3.6 Repair Knowledge Requirement Analysis. Repairing a memory corruption is more complicated than a meta error. Creating such a tool does require VM level knowledge. Moreover, writing this tool led us to improve Polyphemus further to reduce these VM level knowledge requirement. Particularly, we had to enable loading and iterating of a corrupted image as well as isolating the corrupted objects (Section 4.3.3). The limits between the tool repairing a memory corruption and Polyphemus are blurrier than the meta error case study because one could argue that Polyphemus should be able to do these operations. We report that using Polyphemus eased repairing the memory corruption with:

- Objects' slots iteration;
- Reference patching;
- Re-computation of the free lists and the free tree;
- Learning what was this particular memory corruption structure and how to fix it.

With the abstractions provided by LLOOP and Polyphemus, we were able to create a tool that was able to repair a corrupted Pharo memory. Moreover, this tool also tackles an issue that previously resulted in developers loosing their works in Pharo.

5 RELATED WORK

Several solutions have been proposed to ease VM development. VM frameworks and meta-circular VMs have offered for a long time simulation environments that helped in testing and debugging VMs. Such VMs are for example is the case of the CLisp [24], Java VMs [2, 3, 10, 33, 37, 38], Self [36], Python [32], Smalltalk [19, 20] and Maxine [38]. These solutions use full-system simulations to ease debugging at the cost of slower and less-precise executions. We take advantage of the simulation as well. However a key advantage of LLOOP is that it is a layer that is added on top of a an API exposing low level objects, without VM modification.

Multiple tools in Java requires VM level support. Therefore it supports VM level tooling with JVM Tooling Infrastructure [9]. This allowed academic works to develop tools for debugging [1, 12], profiling [34] and memory visualization [5, 15, 30, 31] purposes on the JVM. JVM TI provides an access to the JVM in a low level manner. LLOOP however aims not only at offering a VM tooling API, but also at bridging the VM level and the language level.

Recently, Maxine's team reported a QEMU test-based infrastructure for cross-ISA testing and debugging [21]. They reported that this infrastructure helped them in porting their VM to AArch32 (ARMv7). Still, their debugging happens in gdb in an abstraction level far-away from the original source code which uses OOPs. Moreover, they reported they still cannot cover many parts of their codebase. We believe that many of these limitations come from the fact that Maxine is a meta-circular VM that uses its JIT compiler to compile itself to binary. Similarly, the Pharo team described a QEMU test-based infrastructure they have used to port their VM to AArch64 [29]. They leverage their simulation environment execute test cases before they have access to the hardware. In both cases, they rely on their tools which use OOPs to ease VM development. However, developer need to bridge the abstraction gap.

Espell [27] assembles brand new application runtimes. This is used to grow a given memory to bootstrap systems [17] and to dynamically tailor applications runtimes [28]. This is achieved by creating a memory using the Pharo VM simulator and filling it up with the simulator support. Polyphemus is also based on the Pharo VM simulator, but it treats any runtimes satisfying the Pharo VM constraints, including runtimes created with Espell. Moreover, Espell provides representations that is at the language level whereas Polyphemus mixes language level and VM level which offers more control to the developer.

6 FUTURE WORK

First, Polyphemus is focused on memory and its concepts. However, a VM is more than its memory. We plan to take into account the execution engine and its variables. Particularly, it would be useful to be able to have access to stack pages to enable developers to modify them. This would for example allow us to recover OOPs only accessible from the execution engine in the case of a corrupted Pharo image. Moreover, we would like to extend their utilization to be used in a VM level debugger.

Second, the Pharo VM uses a baseline just in time (JIT) compiler. The JIT compiler compiles bytecode to binary at run time. A method that is compiled by the JIT compiler is stored in a particular OOP: a jitted method. Moreover they are stored in their own space in memory. Currently Polyphemus does not support jited objects.

Finally, we want to apply LLOOP on top of JVM TI.

7 CONCLUSION

In this paper, we presented LLOOP and its implementation Polyphemus on the Pharo VM. We presented how to create tooling on the Pharo VM and showed that it is cumbersome. This example showed that there is an abstraction gap between the language level and the VM level. We then presented LLOOP with the support of Pharo VM examples. LLOOP bridged this gap by identifying the OOPs at the VM level. These LLOOPs described both VM level and language level knowledge to ease VM level tooling. Finally, we have presented multiple case studies showing how we were able to create multiple analysis tools at the VM level. Particularly we have presented in depth how LLOOP eased the creation of two tools targeting real-world major bugs. The first tool allowed us to repair a meta error and the second tool allowed us to repair a memory corruption in Pharo images. These tools allowed developer to recover what was previously considered lost work. We argue that easing access to the VM level will enable more people to create VM level tools in the same manner the JikesRVM enabled VM level research.

Acknowledgements. This work was supported by Ministry of Higher Education and Research, Hauts de France Regional Council.

REFERENCES

[1] Pedro Pablo Garrido Abenza, Angel Grediaga Olivo, and Bernardo Ledesma Latorre. 2008. VisualJVM: a visual tool for teaching Java technology. *IEEE Transactions on Education* 51, 1 (2008), 86–92.
[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J. D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The Jalapeño virtual machine. *IBM Systems Journal* 39, 1 (2000), 211–238. https://doi.org/10.1147/sj.391.0211
[3] Bowen Alpern, Steve Augart, Stephen M Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, et al. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal* 44, 2 (2005), 399–417.
[4] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland. 333 pages. http://books.pharo.org
[5] Alison Fernandez Blanco, Alexandre Bergel, and Juan Pablo Sandoval Alcocer. 2022. Software Visualizations to Analyze Memory Consumption: A Literature Review. *ACM Computing Surveys (CSUR)* 55, 1 (2022), 1–34.
[6] Carl Friedrich Bolz and Armin Rigo. 2007. How to not write Virtual Machines for Dynamic Languages. In *3rd Workshop on Dynamic Languages and Applications*. http://dyla2007.unibe.ch/?download=dyla07-HowToNotWriteVMs.pdf
[7] Guido Chari, Diego Garbervetsky, Stefan Marr, and Stéphane Ducasse. 2015. Towards Fully Reflective Environments. In *Onward! 2015*. 240–253. https://doi.org/10.1145/2814228.2814241
[8] Steven Costiou, Thomas Dupriez, and Damien Pollet. 2020. Handling Error-Handling Errors: dealing with debugger bugs in Pharo. In *International Workshop on Smalltalk Technologies - IWST 2020*. https://hal.inria.fr/hal-02992644
[9] Java Documentation. 2020. Java Virtual Machine Tool Interface (JVM TI). https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/index.html
[10] C Flack, Antony L Hosking, and Jan Vitek. 2003. Idioms in OVM. (2003).
[11] Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, Eliot, and Sergey I. Salishev. 2009. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments* (Washington, DC, USA) *(VEE '09)*. ACM, New York, NY, USA, 81–90. https://doi.org/10.1145/1508293.1508305
[12] D Glez-Peña and F Fdez-Riverola. 2006. Understanding JPDA (debugging) & JVMTI (profiling) Java APIs within JavaTraceIt!. In *IADIS International Conference WWW/Internet 2006*. 334–338.
[13] Adele Goldberg and David Robson. 1983. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass. http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf
[14] Reed Hastings. 1992. Purify: Fast detection of memory leaks and access errors. In *Proc. 1992 Winter USENIX Conference*. 125–136.
[15] Matthias Hauswirth, Peter F Sweeney, Amer Diwan, and Michael Hind. 2004. Vertical profiling: understanding the behavior of object-priented applications. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 251–269.
[16] Carolina Hernández Phillips. 2020. Objects Layout in the Pharo VM Memory. https://rmod-files.lille.inria.fr/Team/Presentations/VMPresentations/2020-09-29-VM-Hernandez-objectsLayout.pdf
[17] Carolina Hernández Phillips. 2021. *Bootstrap-Based Language Development: Turning an existing VM into a polyglot VM*. Ph. D. Dissertation. Université de Lille.
[18] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *European Conference on Object-Oriented Programming (ECOOP '91)*. https://doi.org/10.1007/BFb0057013
[19] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*. ACM Press, 318–326. https://doi.org/10.1145/263700.263754
[20] Daniel Ingalls, Eliot Miranda, Clément Béra, and Elisa Gonzalez Boix. 2020. Two decades of live coding and debugging of virtual machines through simulation. *Software: Practice and Experience* 50, 9 (2020), 1629–1650.
[21] Christos Kotselidis, Andy Nisbet, Foivos S Zakkak, and Nikos Foutris. 2017. Cross-ISA debugging in meta-circular VMs. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. 1–9. https://doi.org/10.1145/3141871.3141872
[22] Bastian Kruck, Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. 2017. Crossing abstraction barriers when debugging in dynamic languages. In *Proceedings of the Symposium on Applied Computing*. 1498–1504.
[23] Oscar Llorente-Vazquez, Igor Santos, Iker Pastor-Lopez, and Pablo Garcia Bringas. 2021. The Neverending Story: Memory Corruption 30 Years Later. In *Computational Intelligence in Security for Information Systems Conference*. Springer, 136–145.
[24] John McCarthy. 1978. History of LISP. In *History of programming languages*. 173–185.
[25] Eliot Miranda. 2011. The Cog Smalltalk Virtual Machine. In *Proceedings of VMIL 2011*.
[26] Pierre Misse-chanabier, Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse, and Pablo Tesone. 2022. Differential Testing of Simulation-Based Virtual Machine Generators. In *International Conference on Software and Software Reuse*. Springer, 103–119.
[27] Guillermo Polito. 2015. *Virtualization Support for Application Runtime Specialization and Extension*. Ph. D. Dissertation. University Lille 1 - Sciences et Technologies - France.
[28] Guillermo Polito, Stéphane Ducasse, Noury Bouraqadi, and Luc Fabresse. 2011. *Extended results of Tornado: A Run-Fail-Grow approach for Dynamic Application Tayloring*. Technical Report. RMod – INRIA Lille-Nord Europe.
[29] Guillermo Polito, Pablo Tesone, Stéphane Ducasse, Luc Fabresse, Théo Rogliano, Pierre Misse-Chanabier, and Carolina Hernandez Phillips. 2021. Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8. In *MPLR '21, Germany*. Münster, Germany. https://doi.org/10.1145/3475738.3480715
[30] Tony Printezis and Richard Jones. 2002. GCspy: An adaptable heap visualisation framework. *ACM SIGPLAN Notices* 37, 11 (2002), 343–358.
[31] Steven P Reiss. 2009. Visualizing the Java heap to detect memory problems. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 73–80.
[32] Armin Rigo and Samuele Pedroni. 2006. PyPy's approach to virtual machine construction. In *Proceedings of the 2006 conference on Dynamic languages symposium* (Portland, Oregon, USA). ACM, New York, NY, USA, 944–953. https://doi.org/10.1145/1176617.1176753
[33] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. 2006. Java on the bare metal of wireless sensor devices: the Squawk Java virtual machine. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments* (Ottawa, Ontario, Canada). ACM Press, New York, NY, USA, 78–88. https://doi.org/10.1145/1134760.1134773
[34] Jeremy Singer and Chris Kirkham. 2008. Dynamic analysis of Java program concepts for visualization and profiling. *Science of Computer Programming* 70, 2-3 (2008), 111–126.
[35] Dave Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices* 19, 5 (1984), 157–167. https://doi.org/10.1145/390011.808261
[36] David Ungar, Adam Spitz, and Alex Ausch. 2005. Constructing a metacircular Virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1094855.1094865
[37] John Whaley. 2003. Joeq: A virtual machine and compiler infrastructure. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*. 58–66.
[38] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An approachable virtual machine for, and in, java. *ACM Transaction Architecture Code Optimization* 9, 4 (Jan. 2013). https://doi.org/10.1145/2400682.2400689
[39] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.
[40] Thomas "Würthinger, Michael L. Van De Vanter, and Doug" Simon. 2010. Multi-level Virtual Machine Debugging Using the Java Platform Debugger Architecture. In *Perspectives of Systems Informatics*, Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 401–412. https://doi.org/10.1007/978-3-642-11486-1_34