Challenges in Debugging Bootstraps of Reflective Kernels

Carolina Hernández Phillips, Guillermo Polito, Luc Fabresse, Stéphane Ducasse, Noury Bouraqadi, Pablo Tesone

Challenges in Debugging Bootstraps of Reflective Kernels

Carolina Hernández Phillips
IMT Lille Douai, Univ. Lille, Unité de Recherche Informatique Automatique
France
Inria Lille-Nord Europe
France
carolina.hernandez-phillips@inria.fr

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

Luc Fabresse
IMT Lille Douai, Univ. Lille, Unité de Recherche Informatique Automatique
France
luc.fabresse@imt-lille-douai.fr

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

Noury Bouraqadi
IMT Lille Douai, Univ. Lille, Unité de Recherche Informatique Automatique
France
noury.bouraqadi@imt-lille-douai.fr

Pablo Tesone
Pharo Consortium
France
pablo.tesone@inria.fr

Abstract

The current explosion of embedded systems (*i.e.,* IoT, Edge Computing) implies the need for generating tailored and customized software for them. Instead of using specific runtimes (*e.g.,* MicroPython, eLua, mRuby), we advocate that bootstrapping specific language kernels is a promising higher-level approach because the process takes advantage of the generated language abstractions, easing the task for a language developer. Nevertheless, bootstrapping language kernels is still challenging because current debugging tools are not suitable for fixing the possible failures that occur during the process.

In this paper, we take the Pharo bootstrap process as an example to analyse the different challenges a language developer faces. We propose a taxonomy of failures appearing during bootstrap and their causes. Based on this analysis, we identify future research directions: (1) prevention measures based on the reification of implicit virtual machine contracts, and (2) hybrid debugging tools that unify the debugging of high-level code from the bootstrapped language with low-level code from the virtual machine.

Keywords bootstrap, language kernels, IoT

1 Introduction

Bootstraping consists in initializing a system through a process implemented in the same system it initializes. This is achieved by using a previous functional version of this system that we call *host*. A reflective system is a computational system which is capable of reasoning and acting upon itself [4]. Thus, the bootstrap process for a reflective system takes advantage of the self modification capabilities provided by the same language it is bootstrapping [10].

Defects in the definition of the bootstrapped language manifest as failures at different stages of the bootstrap process: *Generation*, *Loading* or *Execution*. During this process, code from different sources is executed by different execution machineries: the bytecode of the application that performs the bootstrap is interpreted by the host virtual machine (VM), the language definition non-compiled code is interpreted by an AST interpreter, and the virtual machine is executed as machine code. Traditional debugging tools are constrained to debug at only one level of execution. This makes them unsuitable for solving errors in a bootstrap process.

To illustrate this problem, consider a language definition that does not define the class LargeInteger. The system is successfully generated and loaded, and its main application starts to execute. If during its execution, adding two instances of SmallInteger produces an overflow, the virtual machine will try to fetch the class LargeInteger from the system memory to store the result, failing with a segmentation fault error. This behaviour is hardcoded in the VM implementation. Nowadays, the only way to debug this error is debugging the execution of the VM. For the case of the Pharo VM we use a C debugging tool such as GDB. But when debugging the execution of the VM, we have access only to generic low level tools to read the high level components of the bootstrapped language. This causes the lost of the bootstrapped language abstractions, making it difficult to find the corresponding defect back in the language definition.

In this paper we aim to respond the next research questions:

1. What are the problems a language developer faces during the bootstrapping of a reflective language kernel, and what is a useful classification for them?
2. What are the tools for preventing and for debugging these problems, considering the trade-off between constraining the new language to prevent possible problems at expense of losing freedom on the language definition.

For responding the previous questions:

1. As the product of experiencing bootstrapping a minimal Pharo-based kernel, we present a classification of defects in the language definition and their associated

IWST19, August 27–29, 2019, Cologne, Germany
2019.

causes regarding requirements originated in the VM or in the bootstrapped system's application.
2. We propose an initial approach to develop a toolset for bootstrapping reflective language kernels, including a language development framework and language-hybrid debugging tools.

To explain our solution we start by describing the process of bootstrapping a reflective kernel taking Pharo as example (Section 2). Showing different kinds of failures that occur during the process, we argue that the current debugging tools are unsuitable for debugging bootstraps (Section 3). Next we present our classification of defects in the language definition, relating them with its cause in the VM or application requirements (Section 4). Based on our defects classification we present different solutions that intend to solve each kind of defect (Section 5). We relate our analysis with different approaches for generating custom software for resource constrained devices (Section 6). Finally we present our conclusions and the future work (Section 7).

Assumptions. We make the next assumptions: The Bootstrapper has no defects. The VM has no defects. The bootstrapped kernels must execute in the same VM as the host system. Therefore adding debugging features to the VM is not part of our approach.

2 The current Pharo bootstrap process in a nutshell

The Pharo bootstrap process comprises the *generation*, *load* and *execution* of a custom object-memory (OM) starting from its *language definition* (Figure 1). The language definition is the static definition for all the classes in the bootstrapped language.

The generation stage is performed by a standard Pharo application, named *bootstrapper*, that runs in a Pharo system which serves as *host*. In this stage the OM is assembled in the host memory and by the end of it, the OM is serialized and written to disk.

Afterwards, the Pharo virtual machine (VM) loads and executes the generated OM, which fulfills the VM requirements.

Terminology. We introduce the next terms to be used during the paper. A *defect* or bug [2] as an unintended mistake introduced by the programmer in the source representation of a program. A *failure* is an unwanted executional behavior that is externally observable by a developer while the process is running [12]. A *critical failure* [13] is a failure that interrupts the execution of the process.

In the language definition, the definition of each class comprises its *structural definition* and the definition of its methods. The structural definition comprises the name of the class, its type and its instance variables (considering the order in which they are defined).

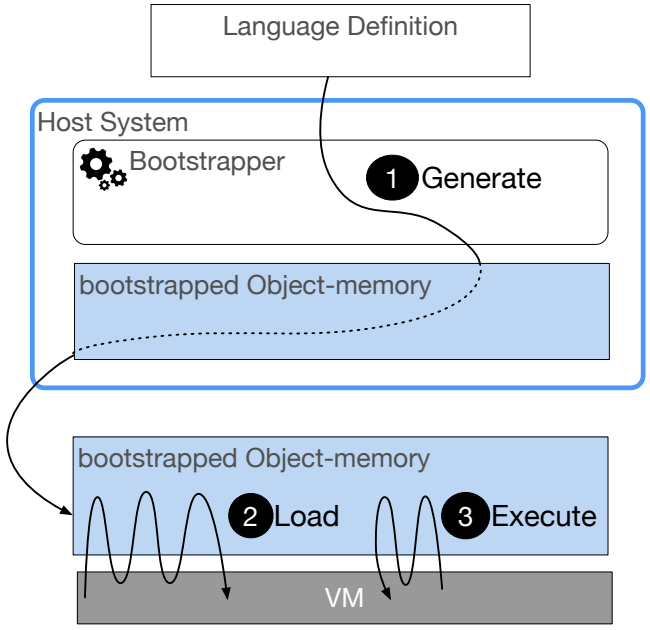

Figure 1. The three main stages of the Pharo bootstrap process.

2.1 Object-memory Generation

The bootstrapper creates an empty OM and fills it with all the classes and objects for the bootstrapped system. Afterwards, it serializes the OM and writes it to disk outside the host. No validations are performed when serializing and writing the OM, therefore we assume that the serializing and writing process is always successful.

The bootstrapper takes advantage of the reflective features of the bootstrapped language for generating the OM [11]. This means that it prefers executing operations in which the bootstrapped system modifies itself via *reflective operations*, instead of manipulating the OM directly via *non-reflective operations*. This strategy minimizes the coupling between the bootstrapper and the language it bootstraps, by partially delegating the logic of the bootstrap process to reflective methods in the language definition. For example, the language definition provides a ClassBuilder class that knows how to install classes in the generated language. The bootstrapper uses this class to install the final version of the system's classes.

We distinguish two kinds of operations performed during this stage:

Non-reflective operations. The bootstrapper installs classes and objects in the OM executing its own methods (methods defined in the host), which take as argument the structural definition of some classes in the language definition, and create objects in the OM.

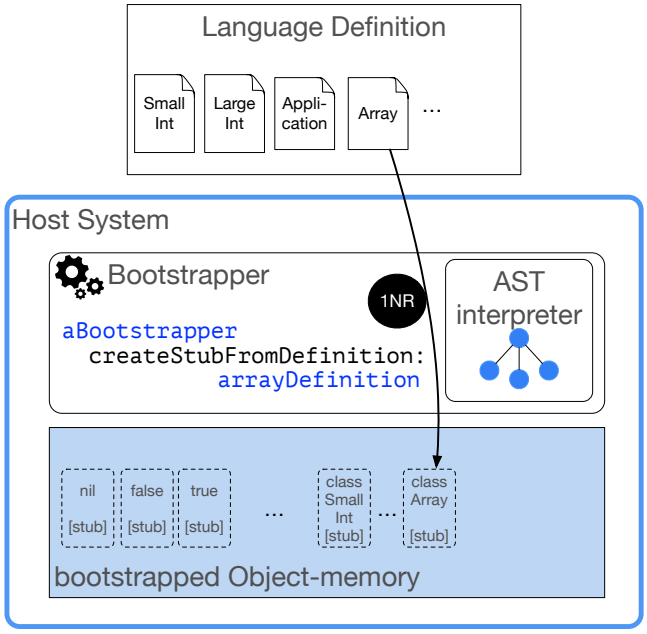

Figure 2. OM Generation: Non-Reflective operations. The bootstrapper modifies the OM by executing its own methods.

For example: To install the class Array (Figure 2), the bootstrapper gets the definition for the structure of the class from the language definition and uses it as the argument (arrayDefinition) for the method createStubFromDefinition. This method is defined in the bootstrapper. It generates and installs a *stub* of the class Array directly in the OM.

Reflective operations. The bootstrapper uses a custom AST interpreter that executes *reflective methods* defined in the language definition to modify the OM. Methods defined in the language definition are not installed in the host. To execute them, the bootstrapper parses their code, obtaining their AST representation, which is given to the AST interpreter. This interpreter visits the AST and performs its execution with no need to install the method, nor to compile it.

For example, to generate a new instance of Array, instead using a method in the bootstrapper that creates the object through a non-reflective operation, we send the message new to the class Array installed in the OM. (Figure 3). Because the message we want to execute is defined inside the language definition and not in the host, the bootstrapper needs to use a custom AST interpreter to execute these instructions and limit the scope of its effects to the OM. By the end of this stage the OM is serialized and written to disk.

2.2 Object-memory Loading by Virtual Machine

When the VM starts, it first fetches a set of objects at specific locations in the OM. Failures occur at this stage if the generated OM does not fulfill the structural requirements of the VM.

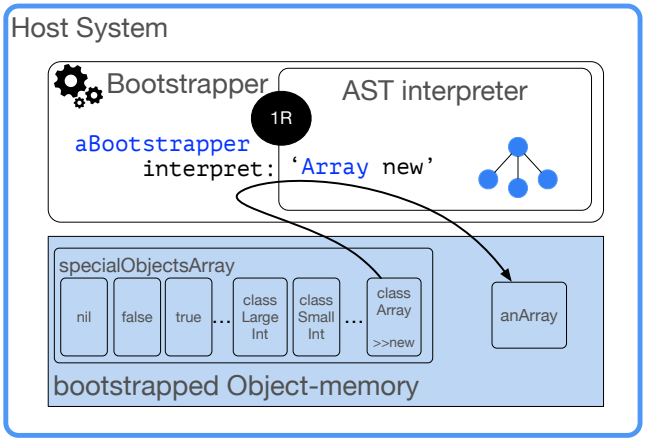

Figure 3. OM Generation: Reflective operations. The bootstrapper modifies the OM executing reflective methods defined in the language definition.

For example, nil, false and true are expected to be the first objects in memory (Figure 4).

Notice that no method from the generated language is interpreted (executed) by the VM during this stage.

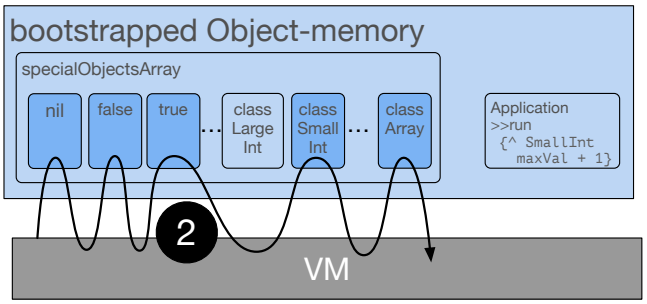

Figure 4. OM Loading. The VM loads the OM fetching specific objects from it. The expected structure for this objects is defined inside the VM.

2.3 Application Execution

Once the OM was loaded, the VM starts interpreting (executing) the system's Application code, whose entry point has been defined inside the language definition. In our examples the method run in the Application class is the system's entry point (Figure 5).

We consider the application execution as part of the bootstrap process to include the analysis of failures appearing during this stage. For the same reason we consider as part of the language definition the code associated to the Application.

Let us suppose our Application starts with the expression: SmallInteger maxVal + 1. The result of the expression SmallInteger maxVal is the maximum possible value stored in an

instance of SmallInteger. The addition operation for SmallInteger is defined in the language definition using a VM primitive, as follows:

```
Class SmallInteger {....
   >> + aNumber {
        <primitive: 1>
      }
...}
```

VM primitives are implemented inside the VM. When the VM encounters a primitive tag while interpreting Pharo code, it will execute its internal implementation for that primitive. Depending on the primitive, its implementation in the VM might require the presence of certain Pharo classes in the OM, as we explain next.

In the VM implementation of <primitive:1> when the result of the addition produces an overflow for SmallInteger, the returned value will be an instance of LargeInteger. Consequently, the VM fetches from a specific position in the specialObjectsArray, the reference to the class LargeInteger, and uses this class to instantiate the returned value.

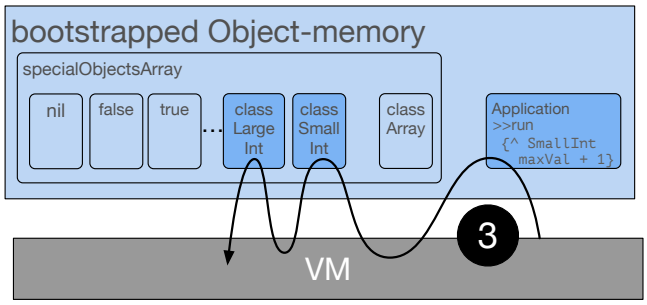

Figure 5. Application Executing. The VM executes the system's Application.

3 Problem: Current Debugging Tools are Unsuitable

Defects in the language definition manifest as different kinds of failures at different stages of the bootstrap process. When a failure occurs during the OM Generation, we use the host debugger to find its cause by debugging the bootstrapper application execution. When a failure occurs during the Loading or Execution stages we use a debugger, such as GDB, for debugging the VM execution.

We call *fix* to a change in the source representation of a program that eliminates a defect.

In the next subsections we illustrate with examples why the current debugging tools are not enough for debugging the bootstrapper process. For some failures we provide multiple possible causes in the form of defects in the language definition.

3.1 Debugging the OM Generation Stage

We use the native debugger of the host system to debug the bootstrapper while generating the OM.

3.1.1 Debugging Non-reflective operations

Non-reflective operations use methods defined in the bootstrapper. The host debugger does not provide enough information to find the cause of the failure. This is mainly because the defect belongs to the language definition, whose validity depends on the VM requirements, which are not explicit during the debugging process.

We illustrate these problems through the following two examples: Missing definition and Wrong format.

Missing definition. Let us suppose our language definition misses the definition of the class Array. The bootstrapper fails with a critical error, showing the message 'Class Array not found'. To solve this error we need to provide a structurally VM compatible definition for the class Array, such as:

```
Class Array {
   superclass: 'Object'
   instVars: [ ]
   type: 'variable'
... }
```

A valid definition must contain the name, type and instance variables expected by the VM. Because there's no formal specification of the VM requirements, the language developer needs to infer this information from the source code of the VM. We summarize this error in the next table:

F1. Missing Definition for Required Element	
Failure	Class Array not found
Level	Critical
Origin	VM requirement
Defect	Class Array is not defined

Wrong format for class type. Let us suppose that the language definition now contains the class Array, but its structure is not the one expected by the VM: the type is set as 'fixed' but it should be 'variable'.

```
Class Array {
   superclass: 'Object'
   instVars: [ ]
   type: 'fixed'  " incorrect type is a structural defect."
... }
```

The failure will only manifest when the bootstrapper tries to create an instance for this class. We summarize the error as follows:

F2. Instance Creation Fails	
Failure	AssertionFailure: new instance of Array is nil.
Level	Critical
Origin	Bootstrapper requirement
Defect	Incorrect structural definition for class Array

To infer that the defect is in the type of the class Array, the developer must deal with low level Pharo code belonging to

the VM simulator library, which reads and writes the OM. It also requires that the language developer is familiarized with the types required by the VM.

3.1.2 Debugging Reflective operations

Reflective operations use methods defined in the language definition, which are not installed in the host. These methods are executed using a custom AST interpreter. The only way to debug a failure that occurs during these operations, is to debug the AST interpreter execution. The interpreter execution hinders the execution trace of the interpreted code from the language definition. The next example illustrates a failure in kind of operations.

Semantic defect in reflective method. According to the VM requirements, each class stores its methods in an instance of the class MethodDictionary. Accordingly, the bootstrapper executes the reflective message MethodDictionary » at: put:, defined in the language definition, to store the methods for each class. Let us consider a definition for this method, which contains a semantic defect:

```
MethodDictionary { ...
  >> at: index put: value  {
     array at: (index + 1)  ¨ adding 1 is a semantic defect.¨
          put: value.
    }
... }
```

We get the failure:

F3. Interpreter Fails with Generic Error

Failure	Interpreter generic exception error
Level	Critical
Origin	VM requirement
Defect	Semantic defect in reflective method MethodDictionary»at:put:

The stack trace shows information about the execution of the interpreter, while the execution trace of the reflective method is hidden inside the interpreter's interpreter-failure-stack. (Figure 6)

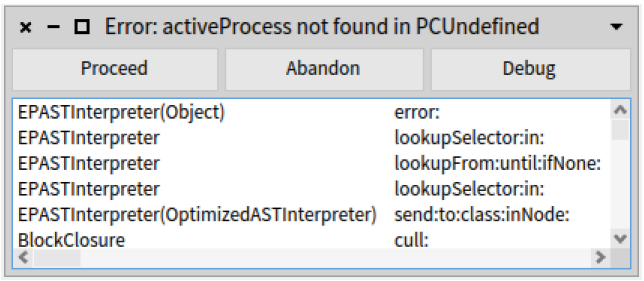

Figure 6. Stack trace for AST interpreter generic error.

Because the messages from the language definition are not installed in the host, they are not available from the host debugger either. The host debugger functionalities can not be used on methods from the language definition. Therefore it is not possible to place breakpoints inside the language definition methods, nor to perform the execution step by step.

3.2 Debugging the OM Loading Stage

Failures during this stage are critical and prevent the VM to initialize. Therefore, to debug them we need a debugger for the VM (such as gdb or lldb).

Wrong format for instance variables. The VM requires that the class Behavior contains 3 instance variables: superclass, methodDict, format. Modifying the order in which these variables are defined introduces a structural error:

```
Class Behavior {
   superclass : 'Object'
   instVars: [ superclass format methodDict ]  ¨ wrong order for instance variables is a structural defect.¨
   type : 'fixed'
...}
```

We get the next failure:

F4. Wrong memory access

Failure	segmentation fault in C
Level	Critical
Origin	VM requirement
Defect	class Behavior instance variables format and methodDict are defined in the wrong order.

It is difficult to access objects in the OM using such a low-level debugger because it only provides generic low-level tools to read the memory. The debugger is agnostic to the abstractions of the bootstrapped language. There's no execution stack trace for the bootstrapped language.

3.3 Debugging the Application Execution

The VM has already initialized, and the application entry point has started its execution. However, if primitive executions make the process fail, the VM fails showing a segmentation fault error. This situation is analogus to the failure F4.

Another kind of failure for this stage is independent from the VM requirements and it's produced by semantic errors in code executed by the application, that has no connection with VM requirements. This kind of failures include critical and non-critical failures.

Semantic defect in application code. As an example, let us suppose our application entry point evaluates a variable before initializing it.

```
Class Application { ...
   >> run {
      | x |
      x := x + 1  ¨ use non-initialized variable is a semantic defect.¨
      }
...}
```

F5. Application Failure	
Failure	Stack overflow
Level	Critical
Origin	Application
Defect	Use non-initialized variable.

The debugger provides a Basic Smalltalk execution stack trace as follows:

```
Smalltalk stack dump:
0xbffc8fd0 M>species 0x6e4e350: a(n) bad class
0xbffc7c0c M>copyReplaceFrom:to:with: 0x6e4e350: a(n) bad class
0xbffc7c30 M>, 0x6e4e350: a(n) bad class
0xbffc7c5c I>doesNotUnderstand: activeProcess 0x6e2f7c0: a(n) bad class
0xbffc7c88 I>doesNotUnderstand: activeProcess 0x6e2f7c0: a(n) bad class
...
```

This stack trace is produced by the VM implementation, and it is shown besides the VM execution stack trace in C which starts as follows:

```
C stack backtrace & registers:
0   Pharo           0x000b4d1e reportStackState + 813
1   Pharo           0x000b49dc error + 28
2   Pharo           0x0007ae82 copyToOldSpacebytesformat + 90
3   Pharo           0x00061e50 copyAndForward + 194
4   Pharo           0x0007abdd scavengeReferentsOf + 289
...
```

The debugger only provides generic low level tools to read the OM, consequently the abstractions of the language to access and manipulate objects are lost.

4 Classification of Defects in the Language Definition

After experimenting with the Pharo bootstrap process, we analyzed the encountered failures and their associated defects, and we propose a classification for them. We analyze defects found only in the language definition and not in the bootstrapper, nor in the VM; because we assume that the bootstrapper and the VM are free of defects. Our analysis does not consider syntax defects because they are solved using the current debugging tools in the host. We classify these defects as follows:

4.1 Structural Defects

They occur when the language definition misses the definition of a class required by the VM, or when the structural definition of a required class doesn't match the format expected by the VM.

Structural defects always cause critical failures in the stage they arise (Generation, Loading or Executing). Meaning that they always interrupt the execution of the process.

From the examples presented in Section 3, failures F1, F2 and F4 are caused by structural defects.

It follows that to prevent this kind of defects ***the language definition must define the minimal set of classes required by the VM, and that these classes must comply with the format expected by the VM***.

4.2 Semantic Defects

A method in the language definition has a correct syntax (meaning that it could be compiled), but its execution produces a wrong result. Semantic defects cause critical and non-critical failures, which arise in any of the 3 stages of the process. From the examples presented in Section 3, failures F3 and F5 are produced by semantic defects.

We make the distinction between semantic defects manifested during reflective operations in the OM Generation stage (*e.g.,*failure F3), and semantic defects manifested during the Application Execution stage (*e.g.,*failure F5). Notice that during the OM Loading stage, no methods belonging to the bootstrapped language are interpreted by the VM; instead, only parts of the OM structure (special objects and classes) are fetched by the VM.

4.2.1 Semantic Defects Manifested during the OM Generation

During the OM Generation the bootstrapper executes two kinds of operations: non-reflective, which take as input structural information from the language definition, but do not execute methods from the language definition; and reflective, which execute methods defined in the language definition.

Methods from the language definition that participate in reflective operations during the OM Generation modify the OM. Thus, a semantic defect in one of these methods affects the process of generating the OM, whether by interrupting it such as in failure F3, or by producing a corrupted OM that fails in a posterior stage such as in failure F5.

To illustrate this, let us consider a semantic defect in the class ClassBuilder, which is used during the Generation stage to install the definitive version of all classes in the OM. The semantic defect is in the method ClassBuilder » instSpec, which returns an integer that encodes the type for a class. The value associated to each type is defined by a VM requirement. Thus, the expected value 16 for classes of type words is a requirement of the VM. The defect we show next is returning the value 10 instead of 16 for the type words.

```
Class ClassBuilder {...
  >> instSpec{
    ...
      (self isBytes) ifTrue: [ ^ 16 ].
      (self isWords) ifTrue: [ ^ 16 ].   ¨ semantic defect, the expected value is 10, not 16.¨
    ...
    }
....}
```

In this example it is clear that to prevent this kind of defect ***methods in the language definition that collaborate with the OM Generation must be aware of the VM requirements***.

The presented defect produces a critical failure during the Loading stage, in which the VM will crash showing a segmentation fault error, because it reads the memory of the

instances of a class type words, such as WordArray, as if they were instances of a class type bytes.

4.2.2 Semantic Defects Manifested during the Application Execution

Another kind of semantic defects are those that exist inside methods that are executed during Application Execution. ***These defects are independent from the VM requirements***, except when they relate to the usage of VM primitives, such as in our example of adding two instances of SmallInteger that cause an overflow, shown in Section 1.

Defects of this kind produce critical or non-critical failures during the Execution stage.

Example: The application entry point references a class that is not defined.

5 Towards an Unified Toolset for Developing Reflective Languages

In this section we propose solutions for each kind of defect described in Section 4. The proposed solutions are complementary, intended to work together, conforming a unified toolset for bootstrapping reflective language kernels. The solutions proposed in Section 5.1 and in Section 5.3 are preventive and they intend to avoid the appearance of failures during bootstrap, which could be related to semantic or structural defects. While the solutions proposed in Section 5.2 deal with those failures which can not be prevented, this kind of defects are always semantic.

5.1 A Framework for Developing VM-compatible Language Kernels

Every VM requirement over the OM is structural. They arise from assumptions the VM makes about specific objects existing in the OM and their byte level structure.

These assumptions manifest in the next two scenarios:

- During the OM Loading stage: the VM always fetches the same objects in the same order, until the Application entry point is reached.
- During the execution of VM Primitives: the VM fetches specific classes for instantiating objects. The number of primitives defined in the VM is finite (around 200).

VM assumptions are finite and therefore they comprise a finite number of objects that must exist in the OM and that must have a specific byte level structure. Discovering the minimal set of VM requirements is the first step towards defining static tests that apply on the OM and moreover on the language definition. Additionally, to help the process of defining a language from scratch, we could provide a base-language definition that produces a VM compatible OM, and which can be modified or extended by the language developer to create its own language.

This framework intents to solve structural-defects described in Section 4.1. Each part of the framework is explained in the next paragraphs.

Static tests for the Language Definition. The VM requirements will be explicitly reflected in these tests. These tests will be applied on the language definition before the bootstrap process starts. They will check the existence and format of the classes required by the VM. Classes required during the OM Loading stage are always the same and therefore our tests will always check for their presence. This is because if one of them is missing, the bootstrap process will fail during the OM Generation or during the Load stage.

Classes required by the execution of VM primitives depend on which primitives do the bootstrapped system application executes. We will use a static analysis on the language definition to obtain all the primitives used inside its methods. A dynamic analysis using the simulated execution environment described in Section 5.3 will complement the previous one, reducing even more the list of required primitives. However the primitives list obtained using our dynamic analysis is a subset of the list of primitives the application requires when executing in a real world environment. Consequently the question of how to fix bugs associated to the absence of classes required by primitives that executed during real world execution, but are missing in the dynamic analysis primitives list, remains open and will be explored by us in future work.

As a first step, our tests will check for all the classes required by the VM for loading the OM and by all the VM primitives. Each test will indicate if its tested class is:

- required by the loading stage. If one of these tests fail, the generated OM will fail to load.
- required by one of the primitives obtained by the dynamic analysis. If one of these tests fail, the system will fail during its execution stage.
- required by one of the primitives obtained from the static analysis. If one of these tests fail, the system may still execute and never fail, or it can fail during its execution.
- required by one of the primitives that does not belong to the language definition. The failure of this kind of tests says nothing about possible failures of the system.

Static tests for the OM. VM-compatible OMs share a common structure. Therefore it is possible to define an Object-memory format for performing static tests on the OM as it is generated. By studying this structure we will determine a standard file format for VM compatible object-memories, in a similar way the Executable and Linkable Format (ELF) [1] defines a standard format for executable files, widely used in UNIX systems. This format will be used to perform static validations on the OM, which will be specially useful during its Generation stage and at the end of it. We currently have an inspector for the OM which is capable of obtaining its

classes and special objects at different steps during the OM Generation stage. We are working on creating static tests on these retrieved classes and objects.

The set of tests for the OM and for the Language definition will explicit the VM requirements for the language developer.

An extendable base language definition. It is possible to provide a *base language definition* that contains the minimal set of valid structural definitions required by the VM. This base definition will provide a very simple application entry point, for example it could call to the primitive for quitting the system. When the static tests are applied to the base definition, all of them they will pass. The base definition is bootstrapped, the result is a VM-compatible OM which is load by the VM and quits as soon as it starts its execution.

The language developer will be free to extend this base definition adding new classes and methods, or to modify it as much as he wants, to the point of deciding not to use any of its classes and instead to provide a definition built by himself from scratch. Naturally, the developer runs the risk of introducing defects to the language definition when extending base language definition. Structural defects will be caught by the static tests we just described, however semantical defects will require appropriate debugging tools to be fixed.

We are working on the definition of the VM requirements and also on the definition of an extendable minimal base language definition. We need to do more experimentation and research to determine a good language base, and/or an appropriate way to extend it, that minimizes the constrains over the range of bootstrapped languages that will be extended from it.

A generic bootstrapper application. We are working on implementing a generic bootstrapper that takes as input an extended version of the base language definition.

The language developer may introduce structural defects in the language definition while extending it. Therefore it is necessary to statically test the language definition checking that it contains the minimal set of valid structural definitions required by the VM. If the structural requirements are fulfilled, the bootstrapper will be able to install in the OM the stubs for those classes required to perform reflective operations.

The bootstrapper must be configurable regarding the execution of reflective operations provided by the language definition. In case of semantic defects, the bootstrapped will use the hybrid debugger described in section 5.2 to find their cause. Reflective operations defined in the language definition are used to install classes, methods and the application entry point.

5.2 Debugger for the OM Generation

The following components of our solution solve semantic-defects manifested in reflective operations during the OM Generation, described in Section 4.2.1.

Hybrid-Debugger for OM Generation. Semantic defects manifested during reflective operations interrupt the generation stage or corrupt the OM.

Tools for developing and debugging these reflective functionalities are necessary. But they can not be tested in isolation, to make sense they need the bootstrapper execution environment and the VM simulator environment. We want an integrated debugging environment that mixes VM simulator execution, bootstrapper execution and generated language execution.

We propose the use of an Hybrid Debugger for debugging the reflective code used during the bootstrap, the host code from the bootstrapper and the VM simulation code.

- For the bootstrapper code we use the existing debugger in the host.
- The reflective code is executed using a steppable AST interpreter, we require to have a debugger that allows us to debug the language definition code.
- For the VM sym code, we use the existing debugger in the host, but we need an extension to be able to inspect and manipulate structures used by the VM. (*e.g.,* inspect and manipulate objects in the OM).

5.3 Simulated High Level Execution Environment

A high level execution and debugging environment implemented in the host system is a useful tool for fixing defects manifested during the Application Execution, as described in Section 4.2.2. It will allow us to test the application code in an environment where the VM is implemented as a high level library, which could be implemented on top of the current VM simulator library.

This solution comprises two parts:

- This environment will use an extended version of our custom steppable AST interpreter to interpret the application code. The interpretation process will use our high-level VM simulator for primitive evaluation.
- The environment must also provide and a high level implementation for the object-memory. Our custom steppable AST interpreter will use this high-level object memory as the memory of the system. Our implementation could be based in the current VM simulator library for manipulating low level object-memories.

We envisage that there is potential in instrumentalizing our AST interpreter. As an example, the interpreter could register which primitives does the main application requires to run. This information could be used to remove components from the VM, reducing its size.

6 Related Work

Nowadays there are different approaches for generating custom software for constrained systems.

An automated-approach to miniaturize JavaScript (JS) applications to run in IoT devices was developed by Morales

et al. [6]. They do not modify the application code, but they only configure the JS interpreter turning on and off specific features of the interpreter that have impact on performance. The MoMIT solution searches empirically the appropriate configuration of the JS interpreter to run a specific application. Because this solution is automated, debugging tools are not applicable.

A non-automated approach is MicroPython [5], which is a lean and efficient implementation of Python 3 that includes a small subset of the Python standard library, datatypes and functionalities. MicroPython is optimised to run in constrained environments. The system sources are written in C and they include different modules, among them the system's virtual machine, which is completely independent from the Python 3 virtual machine. The system modules are compiled under different configurations to produce different versions of MicroPython, named ports.

Developing a new module for extending the MicroPython language requires rebuilding the port in which the module is intended to work. The new module must be written in C. In case of failures during the building, the error must be debugged using a C debugger. At this level the high level abstractions of the MicroPython language are not available.

MicroPython doesn't provide debugging tools for developing applications, therefore debugging code at the MicroPython level requires to use a debugger to debug the VM execution at the C code level. However MicroPython provides a REPL which, according to some user experience, partially compensates the need for debugging tools because it allows to execute code interactively in the client.

Another approach is eLua [3], which offers the full implementation of the Lua Programming Language to the embedded world.

eLua provides a modular structure offering diverse components among which we find the interpreter and a console for developing and transfering files. eLua is written in C, and its sources contain platform-specific modules for various microcontrollers. The set of used components is specified when the project is compiled and can be adjusted to reduce the amount of the code. In case of failure during the building the developer must debug C code.

Same as MicroPython, for extending eLua a new module must be written in C. In case the building process fails, it is necessary to debug the C code.

For developing applications, the eLua project aims to provide a development environment, including debugging tools, on the microcontroller itself, without the need to install a specific development environment on the PC side. However, the Debugging feature (remote/on target) is not yet implemented in the eLua project.

mRuby [7] is the lightweight implementation of the Ruby language, its syntax is Ruby 2.x compatible. In mRuby, the compiler and the interpreter are separated, this leads to reduction in the memory requirements during program executions. When building the interpreter the developer must choose the corresponding toolchain, comprising the mRuby compiler and its corresponding architecture.

mRuby can be extended by creating extensions in C and/or Ruby. It is necessary to use the library manager mrbgems, specifically created to integrate extensions into mruby. If there is missing dependencies, mrbgem dependencies solver will reference centralized repositories. In case of failures when the extension is written in C, it must be debugged in C, presenting the same problems described in this paper.

mRuby provides a debugger for debugging code at the mRuby code level [8].

7 Conclusion and Future Work

Taking the Pharo bootstrap process as example, we presented an analysis of the challenges of bootstrapping reflective language kernels and using the analysis we propose solutions for easing the developing process.

We divided the process in 3 stages: Object-memory Generation, Object-memory Loading and Application Execution. We make the distinction between Reflective operations and Non-reflective operations, performed during the Generation stage. We use this them as guide to classify the different kinds of failures that arise during the process.

Linking failures and their defects in the language definition, and analysing their origin in the requirements of the VM or the system application, we created a suitable classification of defects. In this classification we recognize 3 kinds of defects:

- Structural defects, originated in VM requirements.
- Semantic defects in reflective code, also originated in VM requirements.
- Semantic defects in application code, originated in application requirements.

We use this classification of defects to guide the design of solutions for solving them.

Finally we propose research directions for finding solutions that solve each one of the described defects.

- For preventing structural errors we propose an Extensible Base Language Definition that contains all the structural definitions required by the VM. This definition will be extended by the language developer.
- For debugging reflective operations as part of the bootstrapper execution, we propose a Hybrid-Language Debugger that provide debugging support at the level of execution for the VM simulator, the bootstrapper and the generated language.
- For debugging the application code we propose a high level execution and debugging environment in which the VM and the OM are implemented in high level. This implementation could be based in the current VM simulator code.

As a future work, we plan to continue working on the research directions proposed in this paper.

Acknowledgements

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

References

[1] 1997. *System V Application Binary Interface: Intel386 TM Architecture Processor Supplement*. Technical Report 4th ed. The Santa Cruz Operation, Inc.

[2] Boris Beizer. 1990. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA.

[3] eLua [n. d.]. eLua. http://www.eluaproject.net/. http://www.eluaproject.net/

[4] Pattie Maes. 1987. Concepts and Experiments in Computational Reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, Vol. 22. 147–155.

[5] MicroPython [n. d.]. MicroPython. https://micropython.org/. https://micropython.org/

[6] Rodrigo Morales, Rubén Saborido, and Yann-Gaël Guéhéneuc. 2019. MoMIT: Porting a JavaScript Interpreter on a Quarter Coin. *CoRR* abs/1906.03304 (2019). arXiv:1906.03304 http://arxiv.org/abs/1906.03304

[7] mRuby [n. d.]. mRuby. https://mruby.org/. https://mruby.org/

[8] mRubyDebuggerRep [n. d.]. mRuby Debugger Github. https://github.com/mruby/mruby/blob/master/doc/guides/debugger.md. https://github.com/mruby/mruby/blob/master/doc/guides/debugger.md

[9] mRubyHowToDebug [n. d.]. How to Debug mRuby. https://es.slideshare.net/yamanekko/rubyconftw2014. https://es.slideshare.net/yamanekko/rubyconftw2014

[10] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Noury Bouraqadi. 2015. A Bootstrapping Infrastructure to Build and Extend Pharo-Like Languages. In *Onward! 2015*. http://rmod.inria.fr/archives/papers/Poli15a-Onward-Bootstrapping.pdf

[11] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Benjamin van Ryseghem. 2014. Bootstrapping Reflective Systems: The Case of Pharo. *Science of Computer Programming* (2014). http://rmod.inria.fr/archives/papers/Poli14c-BootstrappingASmalltalk-ScienceOfComputerProgramming.pdf

[12] Diomidis Spinellis. 2018. Modern Debugging: The Art of Finding a Needle in a Haystack. *Commun. ACM* 61, 11 (Oct. 2018), 124–134. https://doi.org/10.1145/3186278

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