

Challenges in Debugging Bootstraps of Reflective Kernels

Carolina Hernández Phillips
INRIA Lille Nord Europe
IMT Lille Douai

Stéphane Ducasse
INRIA Lille Nord Europe

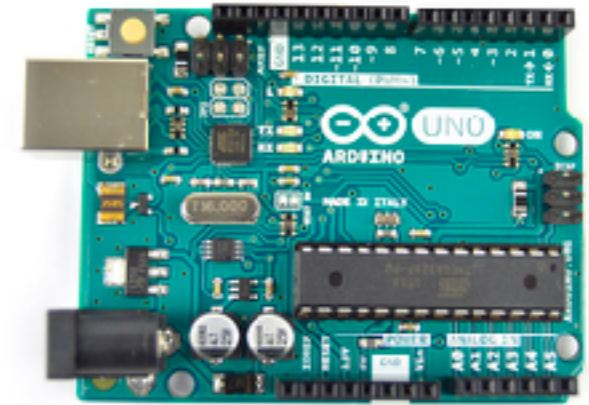
Luc Fabresse
IMT Lille Douai

Noury Bouraqadi
IMT Lille Douai

Guille Polito
Univ. Lille, CNRS, CRISTAL

Pablo Tesone
Pharo Consortium

Why generating custom application runtimes for IoT?



Small Hardware requires small software

Limited processing capabilities, storage, battery



Existing approaches: Generating lightweight implementations of Languages from scratch



MicroPython



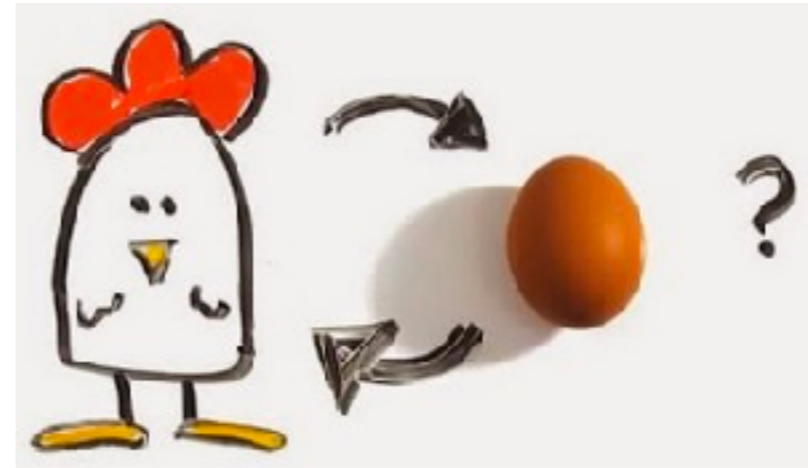
Implement from scratch: VM, base libraries, compiler

Implies complex low level implementation

Requires high expertise to develop!

Our high level approach: Bootstrapping reflective kernels

- Bootstrapping is to **generate a system using a previous version of the system** that is being generated

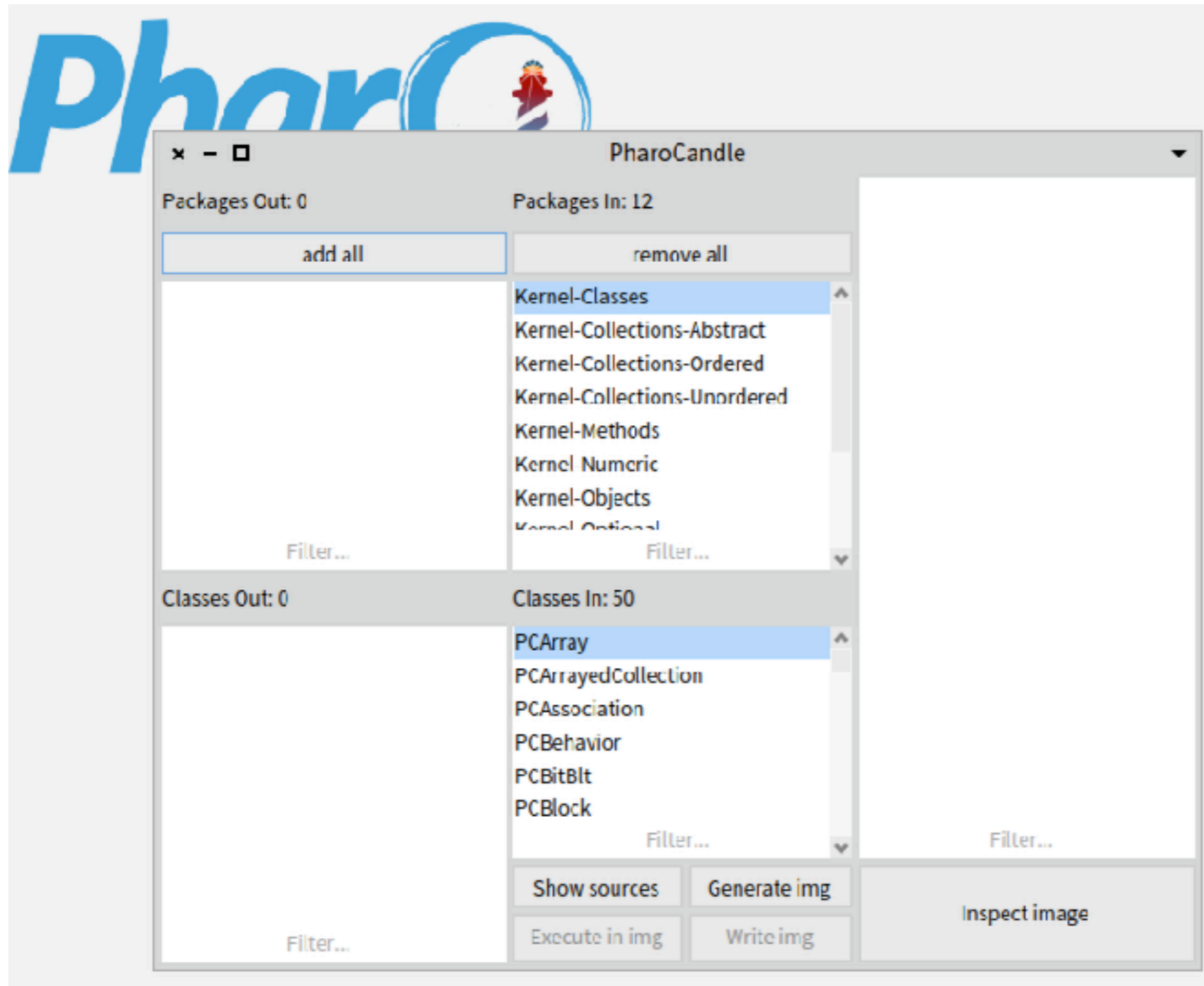


- Therefore we can use the **high level abstractions** and the **reflective capabilities** of both systems during the bootstrap
- The result is a **small Kernel** (an image in the case of Pharo) which can be executed by the same VM that executes its previous version

Demo

Let's Bootstrap PharoCandle
(a Pharo micro kernel)

Bootstrapper Application running in Pharo 7, reading a Language Definition of only 50 classes



Language Definition source code view

The screenshot displays the PharoCandle IDE interface. On the left, there are two panes for package and class management. The top pane, 'Packages In: 12', lists various kernel packages like 'Kernel-Classes', 'Kernel-Collections-Abstract', etc. The bottom pane, 'Classes In: 50', lists classes such as 'PCArray', 'PCArrayedCollection', and 'PCAssociation'. The main workspace is titled 'PCCompiledMethod>>numTemps In a ClyRing2Environment'. It features a class browser on the left showing a hierarchy of classes including 'PCCompiledMethod'. The right side of the workspace shows a list of instance variables: 'frameSize', 'header', 'initialPC', 'isCompiledMethod', 'numLiterals', 'numTemps', 'objectAt:', and 'objectAt:put:'. The 'numTemps' variable is selected. Below the workspace, there is a toolbar with options like 'Export as Tone!', 'Packages', 'Projects', 'Flat', 'Hier.', 'Inst. side', 'Class side', 'Methods', 'Vars', 'Class refs.', 'Implementors', and 'Send'. The source code editor shows the following code for the 'numTemps' method:

```
numTemps
  "Answer the number of temporary variables used by this method."

  ^ (self header bitShift: -18) bitAnd: 16r3F
```

At the bottom of the IDE, there is a status bar showing '1/4 [1]' and some icons.

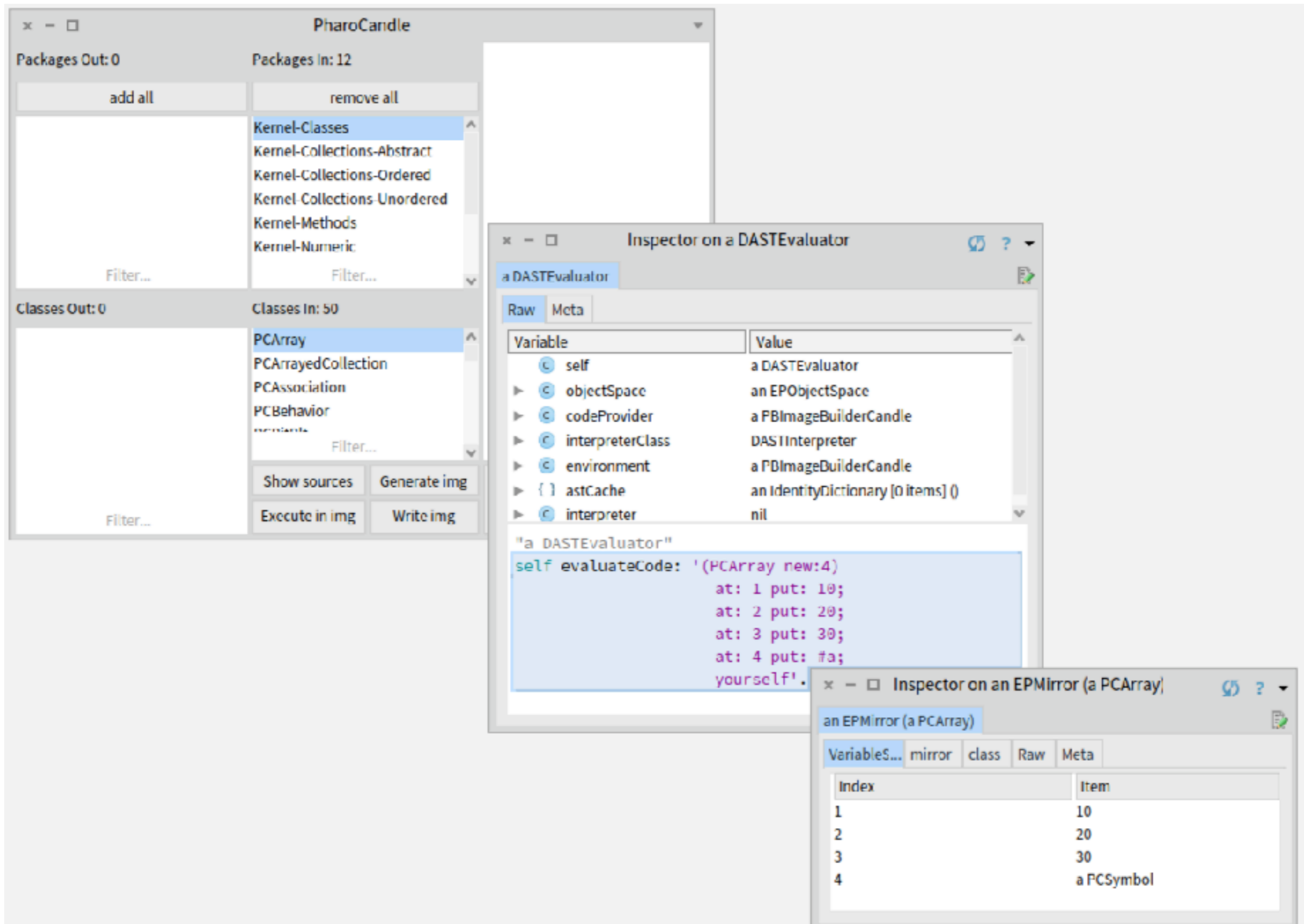
Language Definition source code view: Application entry point

The screenshot displays the Pharo IDE interface. On the left, the 'PharoCandle' environment is shown with package and class lists. The main window is titled 'PCSystem class >> start in a ClyRing2Environment'. The left pane shows a package browser with 'Kernel-System' selected. The right pane shows the class side with 'start' selected in the method list. The bottom pane shows the source code for the 'start' method:

```
start
  self log: 'Welcome to Pharo Candle edition!' .
  self log: self tinyBenchmarks.
  self snapshotAndQuit
```

The status bar at the bottom indicates '1/4 [1]' and 'as yet unclassified'.

Simulated execution of code from the language definition in the bootstrapped Kernel (before writing the Kernel to disk)



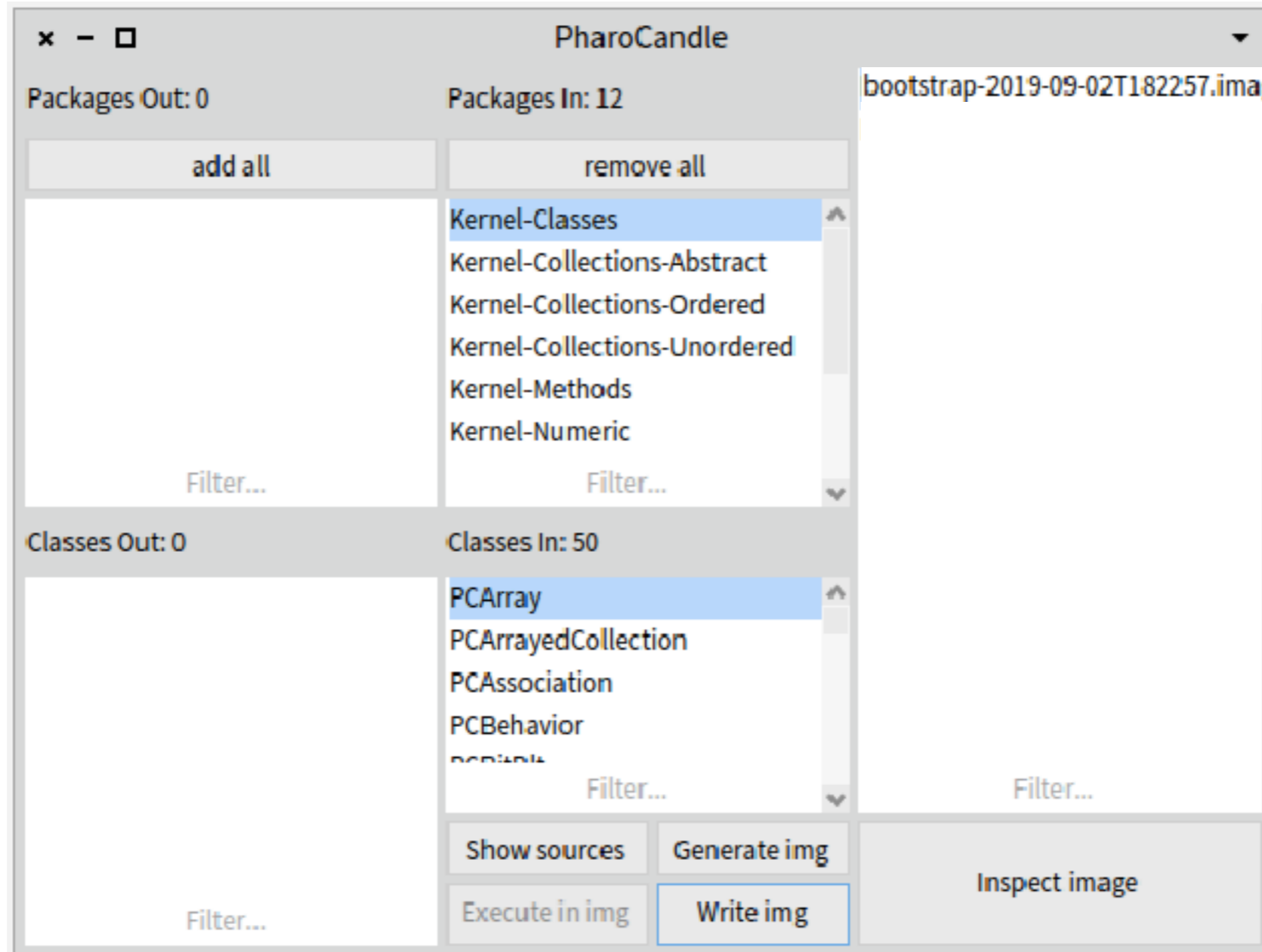
The screenshot displays the PharoCandle IDE interface, showing the simulated execution of code from the language definition in the bootstrapped kernel. The interface is divided into several panels:

- PharoCandle Package Manager:** Shows "Packages Out: 0" and "Packages In: 12". The "Packages In" list includes Kernel-Classes, Kernel-Collections-Abstract, Kernel-Collections-Ordered, Kernel-Collections-Unordered, Kernel-Methods, and Kernel-Numeric.
- PharoCandle Class Manager:** Shows "Classes Out: 0" and "Classes In: 50". The "Classes In" list includes PCArray, PCArrayedCollection, PCAssociation, and PCBehavior.
- Inspector on a DASTEvaluator:** Shows the state of a DASTEvaluator object. The "Raw" tab displays the following variables and values:

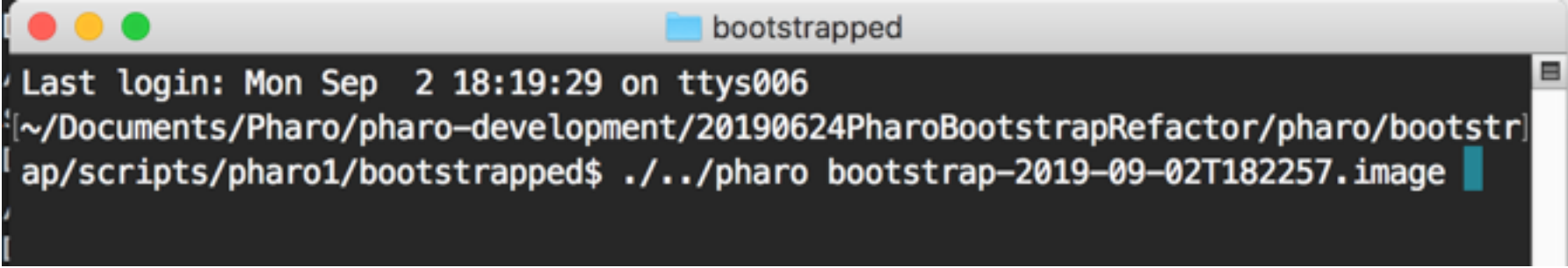
Variable	Value
self	a DASTEvaluator
objectSpace	an EObjectSpace
codeProvider	a FBImageBuilderCandle
interpreterClass	DASTInterpreter
environment	a FBImageBuilderCandle
astCache	an IdentityDictionary [0 items] ()
interpreter	nil
- Inspector on an EPMirror (a PCArray):** Shows the state of an EPMirror object (a PCArray). The "Raw" tab displays the following data:

Index	Item
1	10
2	20
3	30
4	a PCSymbol

Kernel written to disk as a Pharo image

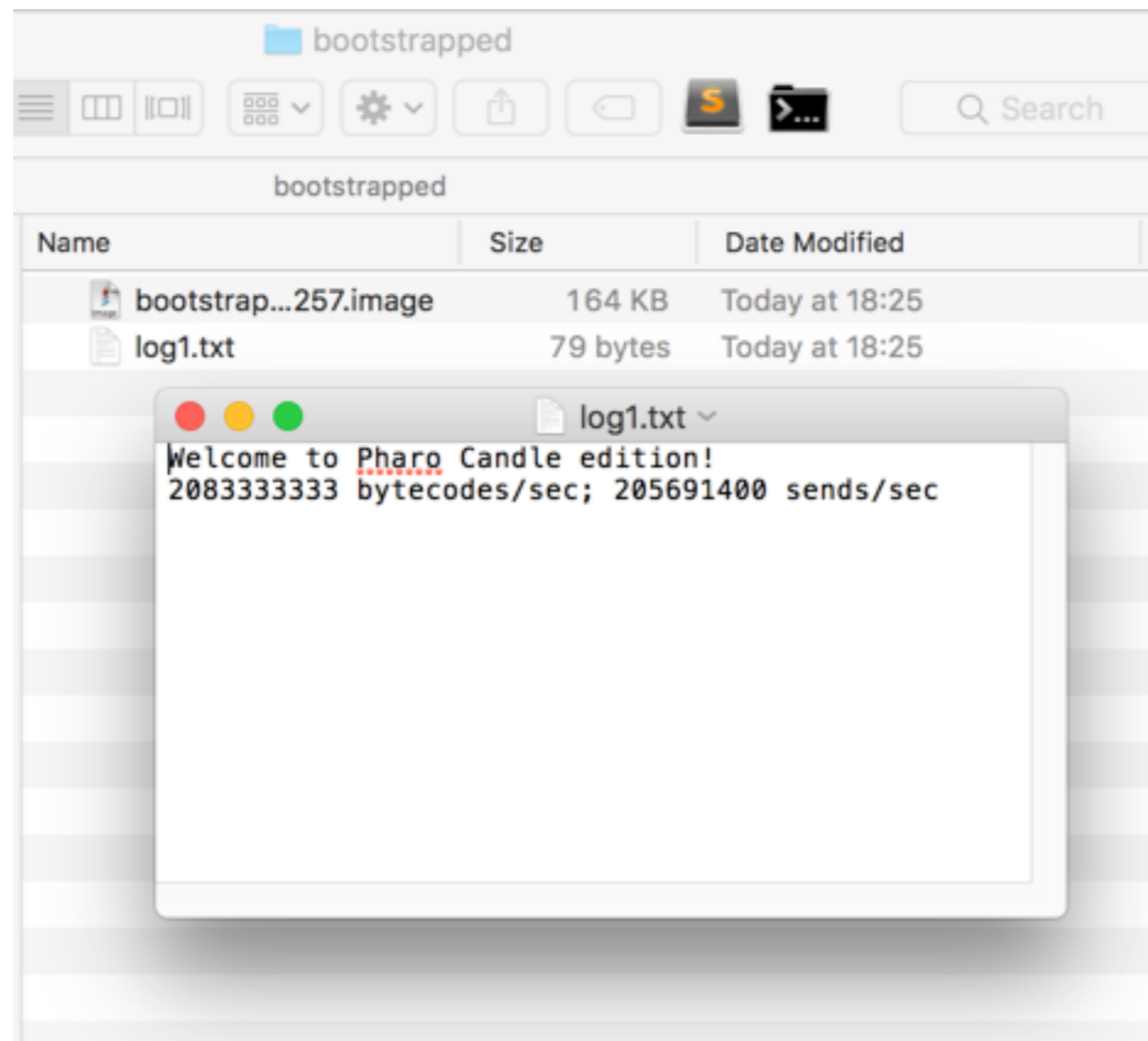


Executing the generated Image (Kernel) using the standard Pharo Virtual Machine

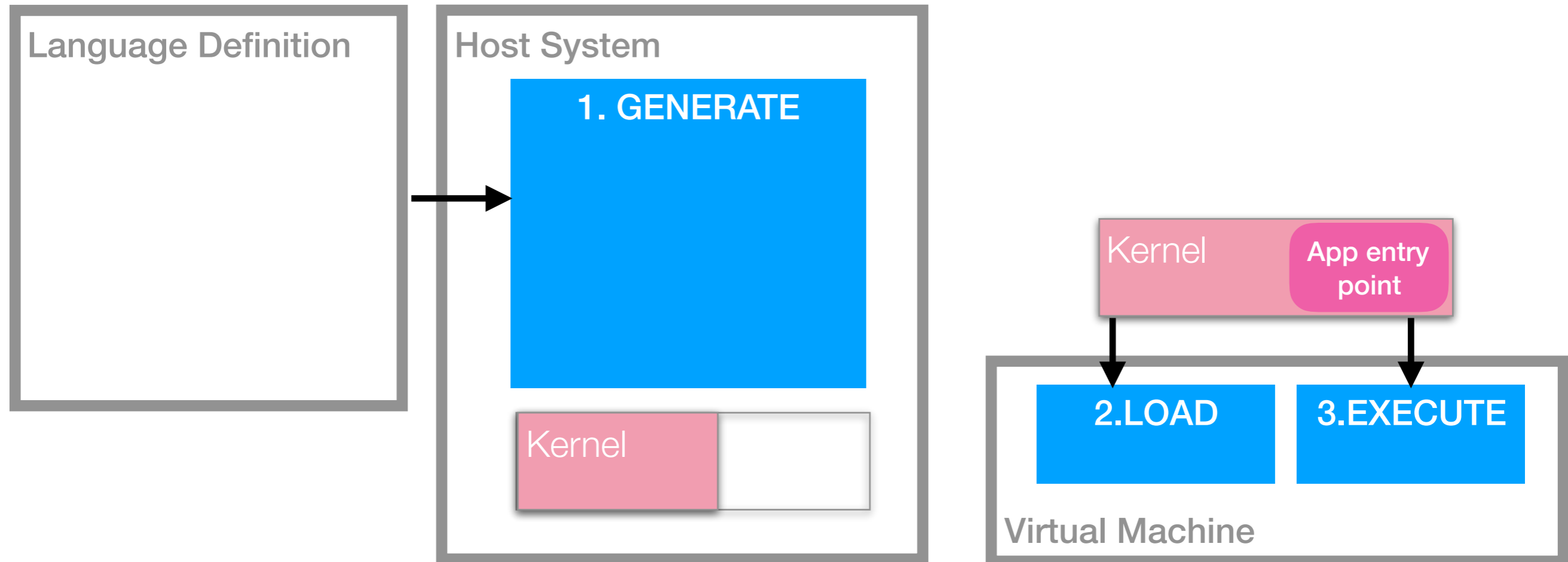


```
bootstrapped
Last login: Mon Sep  2 18:19:29 on ttys006
~/Documents/Pharo/pharo-development/20190624PharoBootstrapRefactor/pharo/bootstrapped$ ./../pharo bootstrap-2019-09-02T182257.image
```

The result of running the Image is the file log1.txt
The Image only weights 164KB!!




Bootstrap

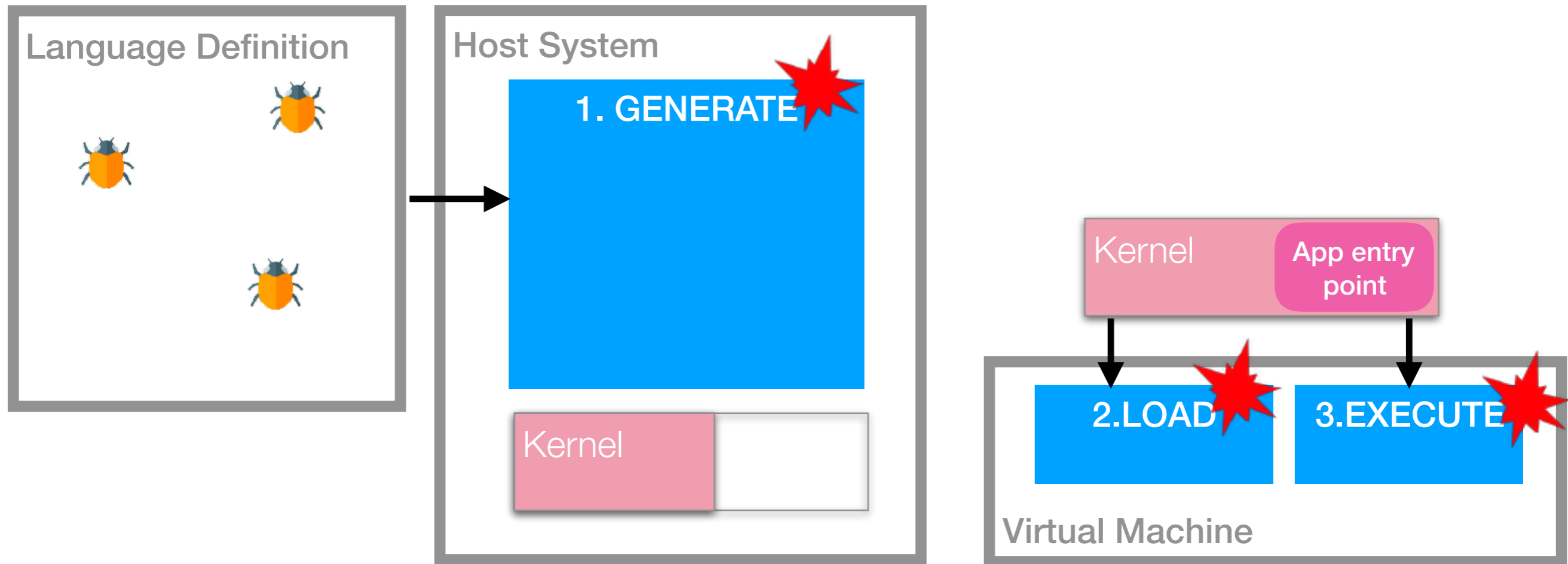


Defects and Failures

Defects & Failures

 Defect: error in Language Definition

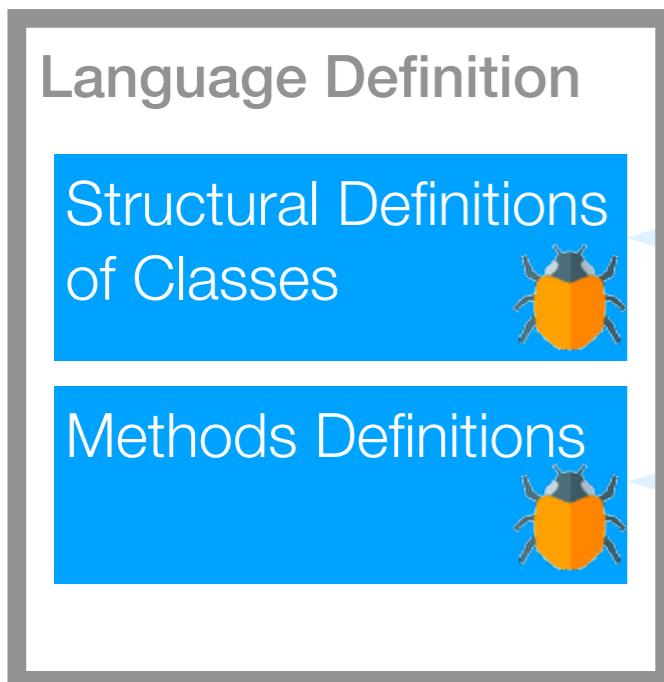
 Failure: incorrect result during the Bootstrap



Defects Classification



Defect: error in Language Definition



Class **PCPoint**

```
superclass : PCObject,  
instVars : { 'x', 'y' },  
type : variable fixed
```

Structural Defect

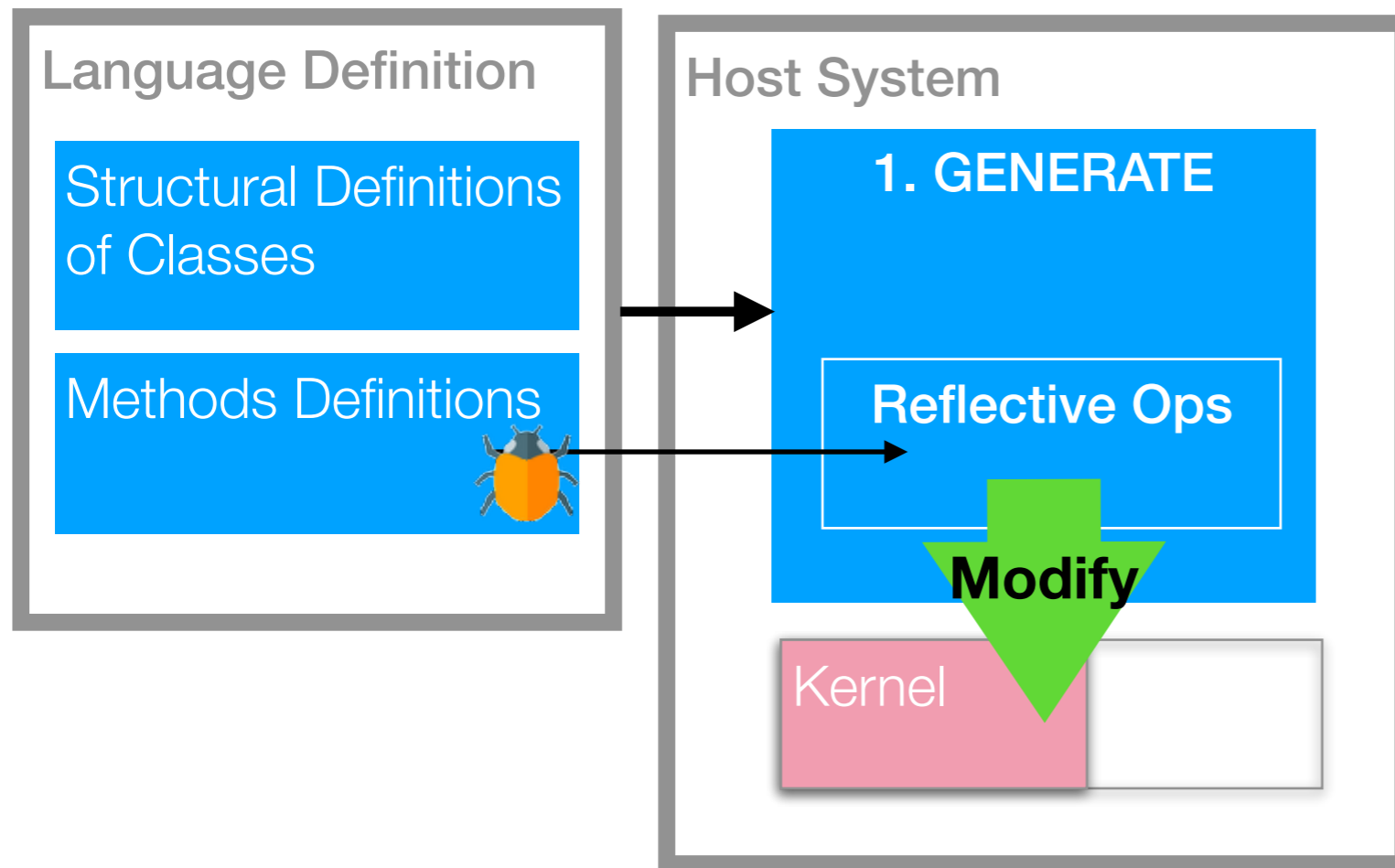
```
PCPoint >> + arg {  
  ^ (x + arg x) @ (y + arg y)  
}  
PCPoint >> crossProduct: aPoint {  
  ^ x * aPoint y - (y * aPoint x)  
  y  
  x  
}  
...
```

Semantic Defect

Semantic Defects are Dangerous



Defect: error in Language Definition



Semantic Defects in
reflective methods
modify the structural
definitions in the Kernel

```
PCClassBuilder >> installMethod: aCompiledMethod inClass: aClass {
```

```
    aClass methodDictionary add: aCompiledMethod
```

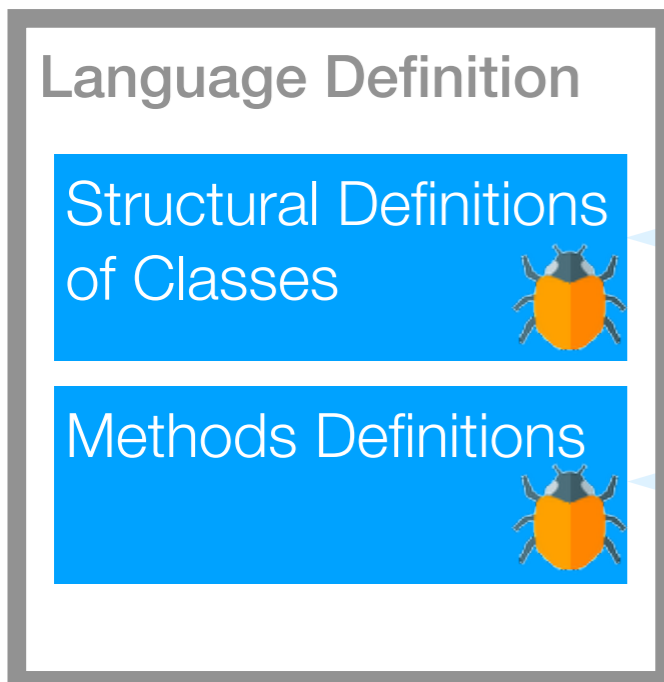
```
}
```

The why of defects

Defects



Defect: error in Language Definition



Class **PCPoint**

```
superclass : PCObject,  
instVars : { 'x', 'y' },  
type : variable fixed
```

Structural Defect

```
PCPoint >> + arg {  
  ^ (x + arg x) @ (y + arg y)  
}  
PCPoint >> crossProduct: aPoint {  
  ^ x * aPoint y - (y * aPoint x)  
  y   
  x  
}  
...
```

Semantic Defect

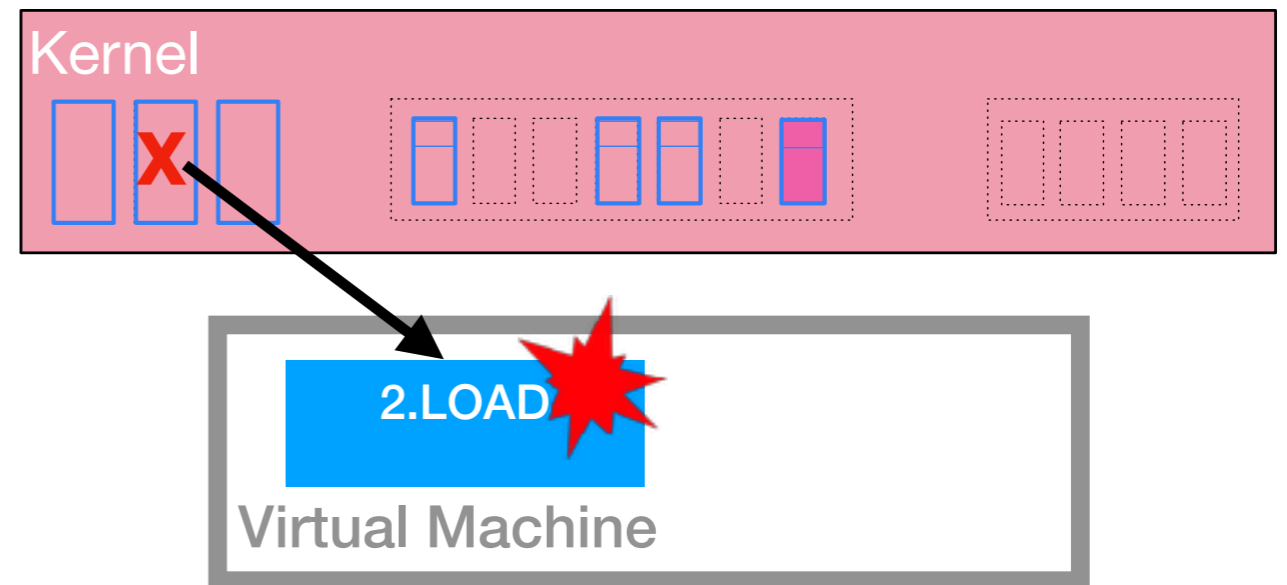
The why of Defects

- Virtual Machine requirements

Class **PCArray**

```
superclass : PCObject,  
instVars : { },  
Type : variable
```

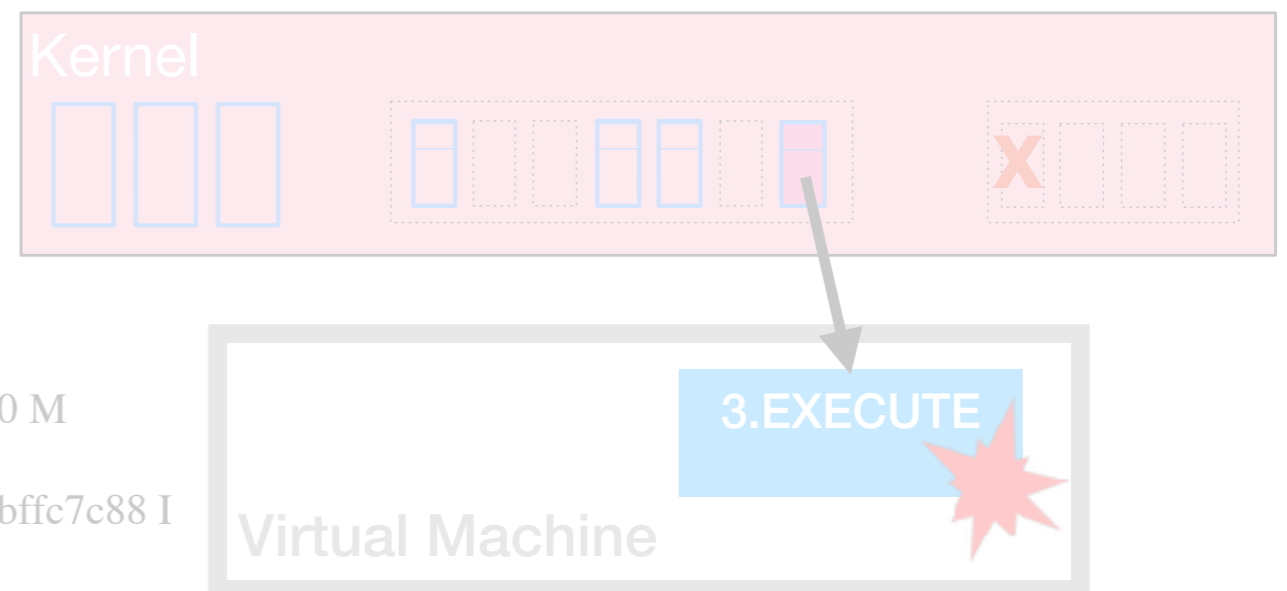
Segmentation Fault



- Application requirements

```
PCMainApplication >> entryPoint {  
  PCMyClass doSomething  
}
```

```
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
```

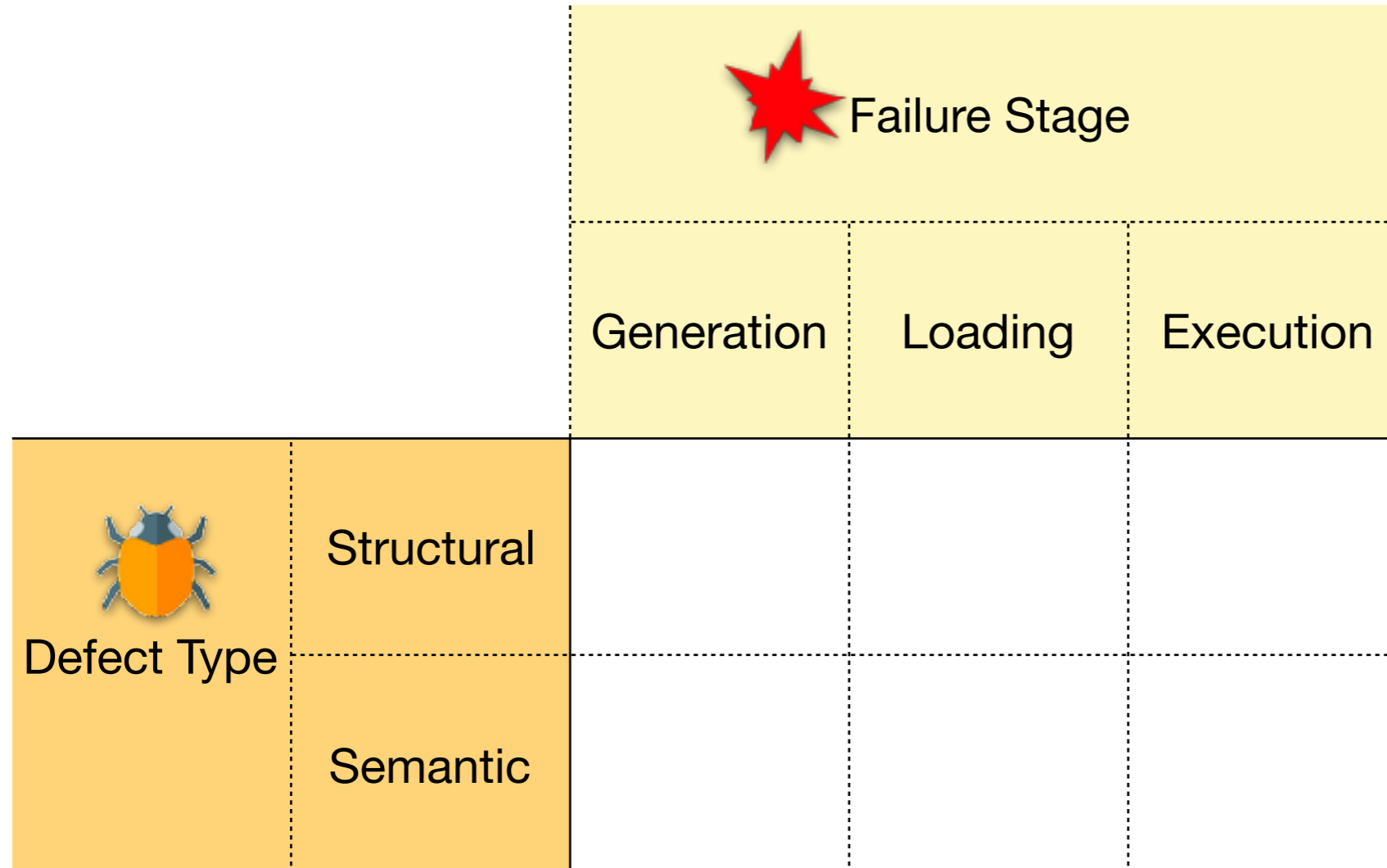


Why is it hard to find the defects back?

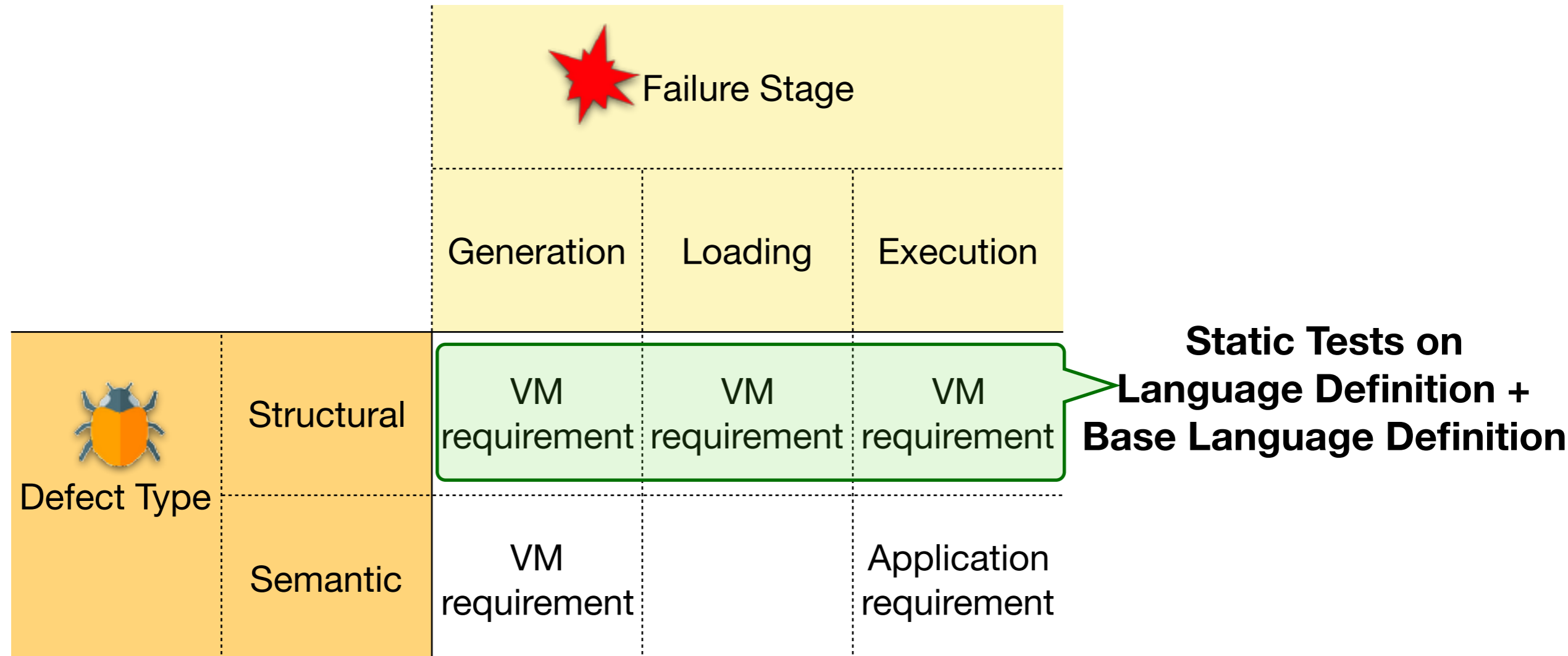
- We are **debugging the VM**
- We **lose great part of the abstractions** of the generated language

Taxonomy of Errors and proposed Solutions

Taxonomy of Errors and Solutions



Taxonomy of Errors and Solutions

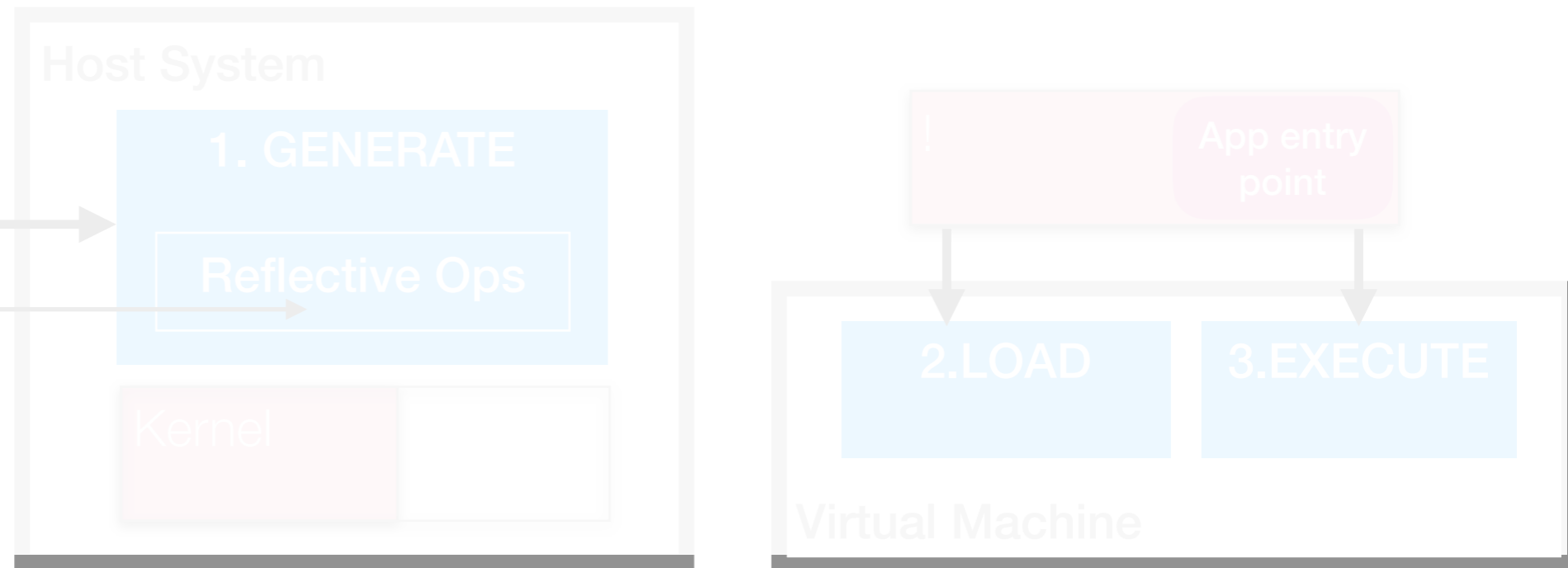


Language Definition

Structural Definitions
of Classes

Methods Definitions

Extensible Base Language Definition



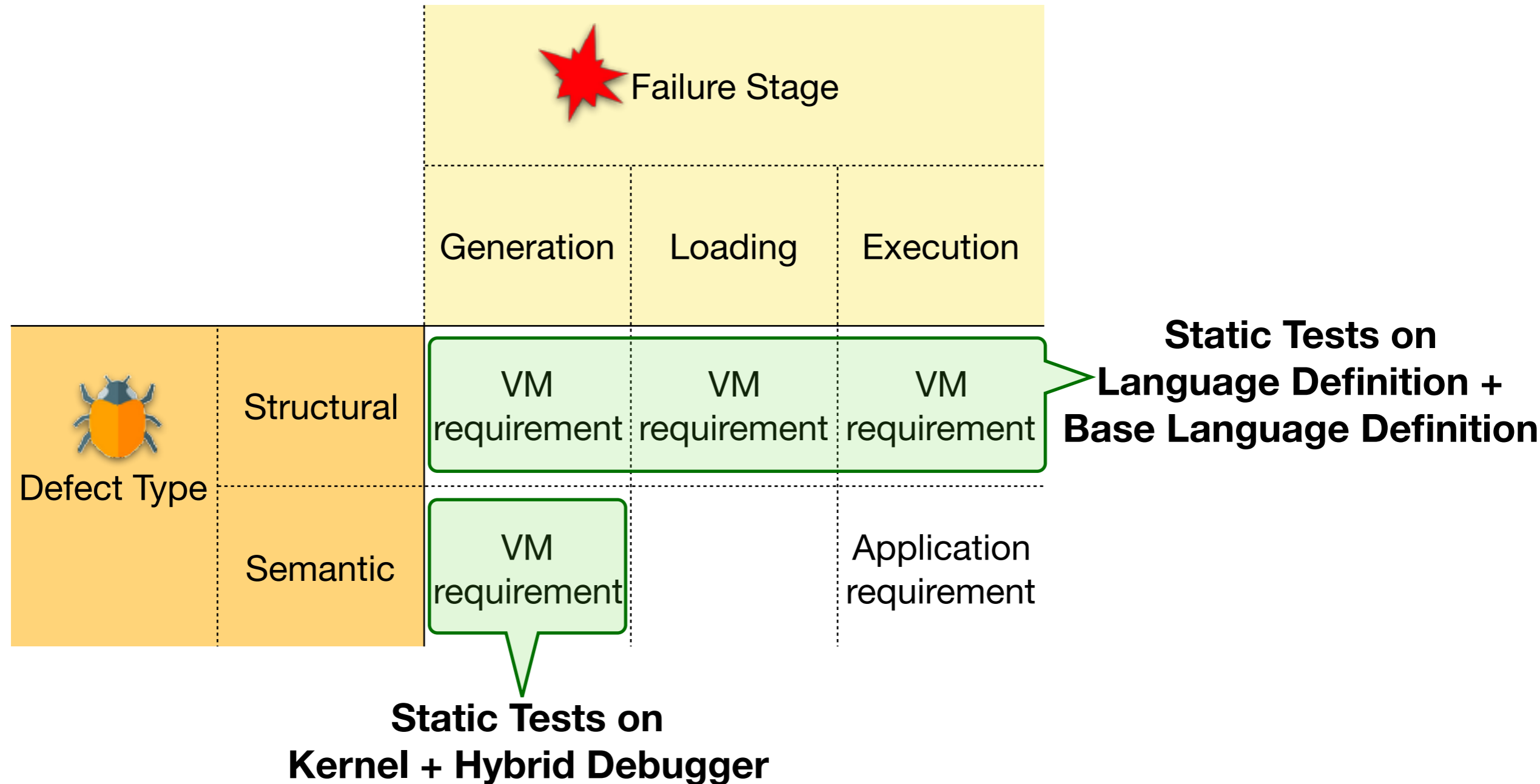
Static Tests on Language Definition

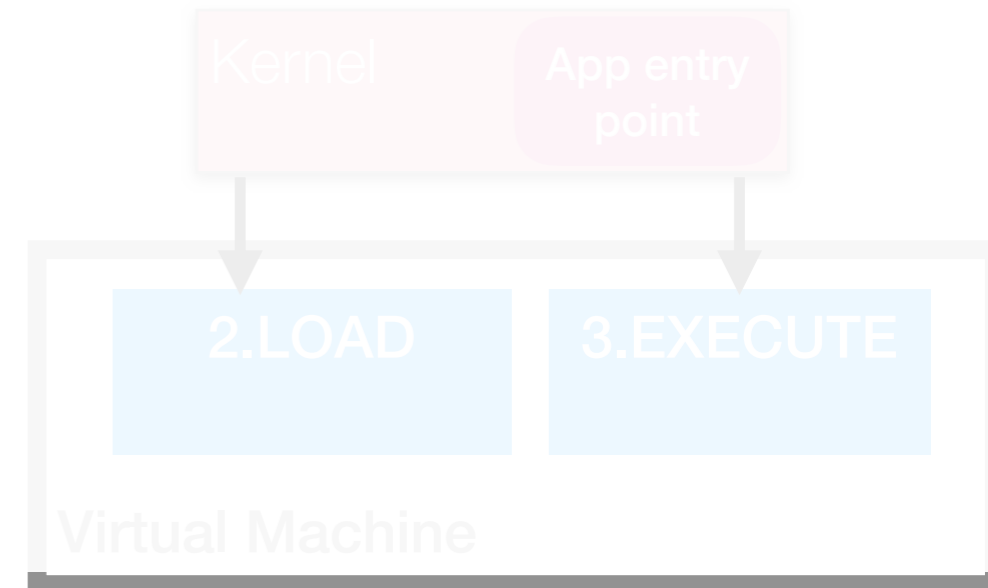
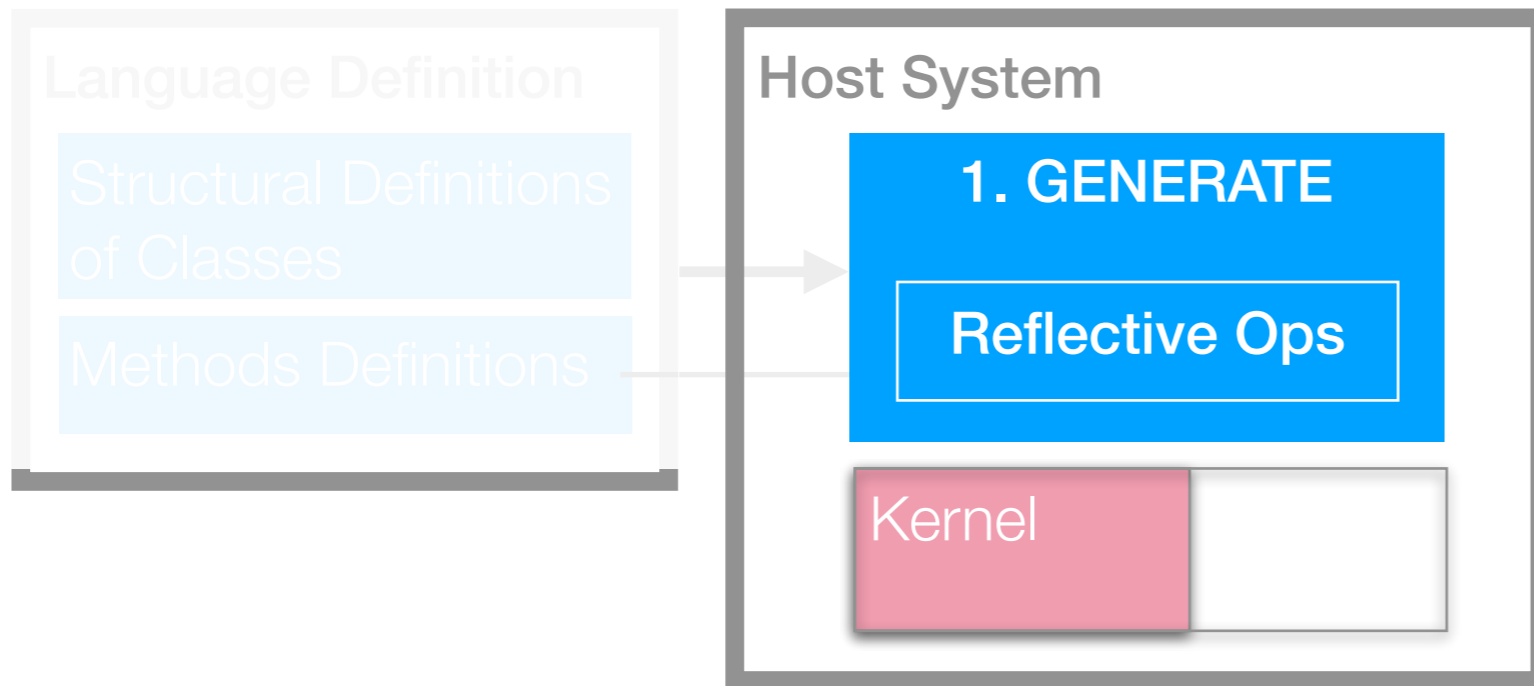
(They reify the
VM requirements)

The screenshot shows a test runner interface for 'PBLanguageDefinitionTests'. The left pane shows a tree view with 'PBLanguageDefinitionTests' selected. The right pane shows a list of test methods, including 'testArrayClassIsVariable', 'testByteSymbolClassIsVariable', 'testEveryClassClassClassIsMe', 'testEveryClassRoleHasAClassL', 'testEveryInheritanceChainEn', 'testEveryMetaClassClassIsMe', 'testLargeNegativeIntegerClass', 'testLargePositiveIntegerClass', and 'testMetaClassClassClassIsItse'. The 'testArrayClassIsVariable' method is selected, and its code is displayed in the bottom pane: `self assert: language classArray isVariable`. The interface also shows various filters and a comment box.

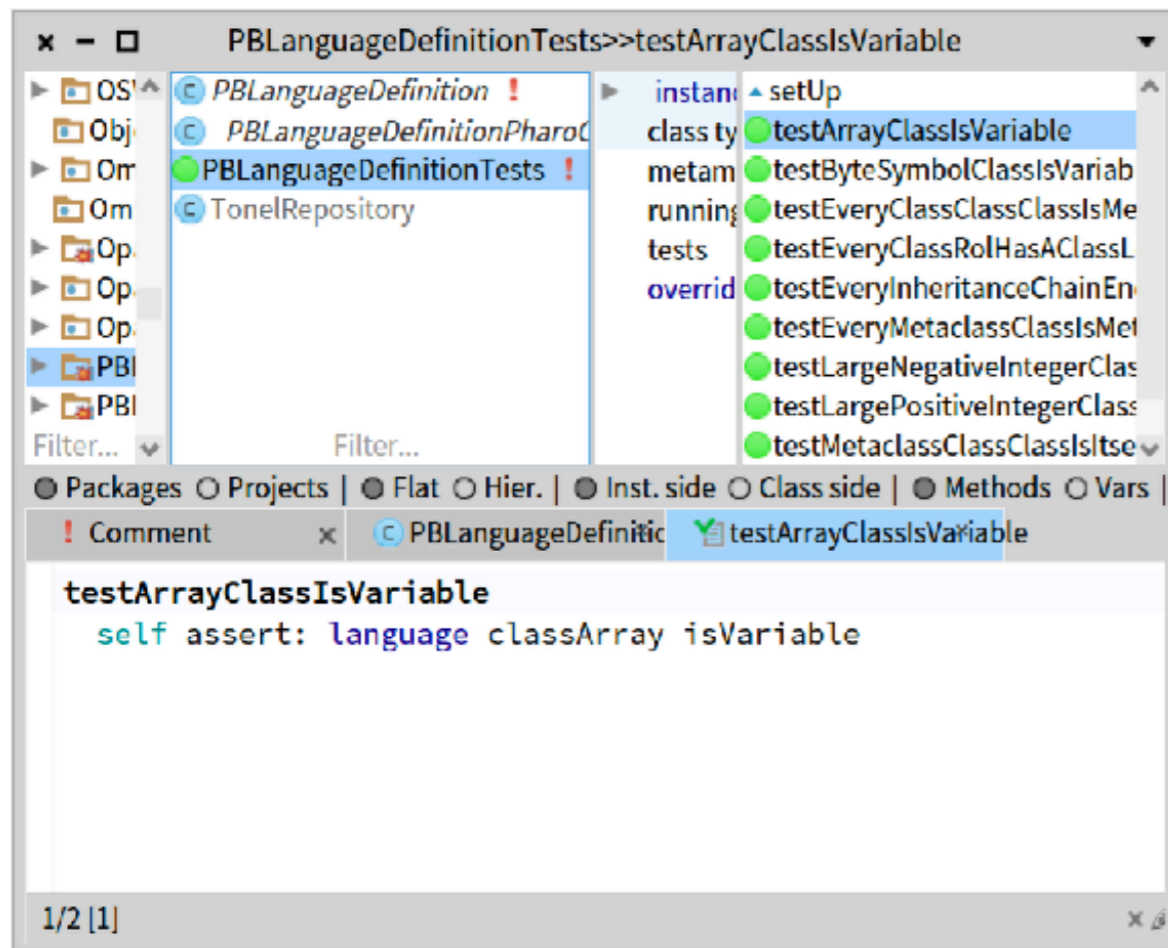


Taxonomy of Errors and Solutions





Static Tests on Language Kernel



+

Hybrid Debugger

3 Execution levels:

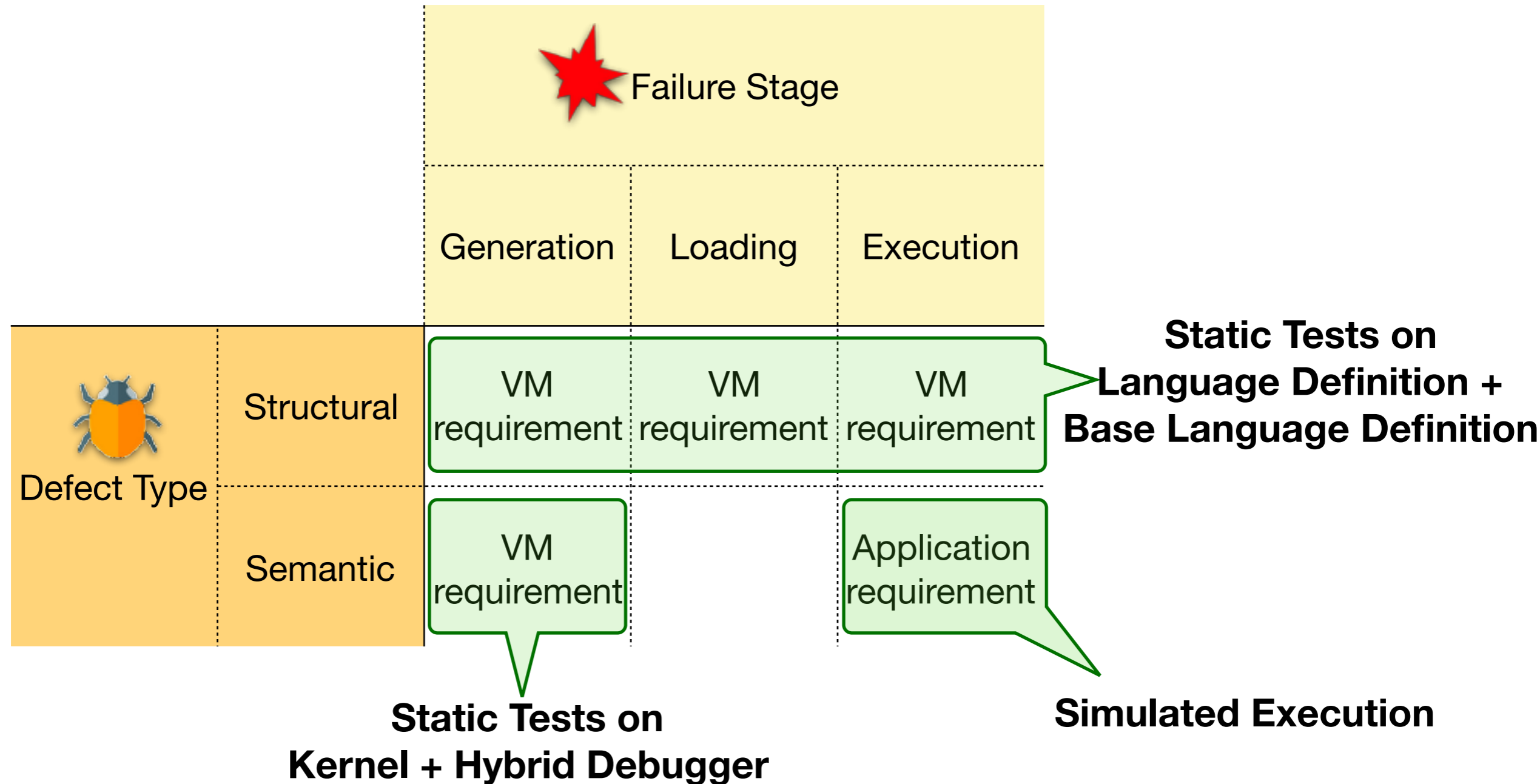
- Language definition code
- Pharo code
- VM code

2 new Debugging Operations

- Step Down
- Step Up



Taxonomy of Errors and Solutions



Language Definition

Structural Definitions of Classes

Methods Definitions

App entry point

Lookup

Execution Simulator

AST interpreter + VM simulator

R/W objects

Kernel

Kernel

App entry point

2.LOAD

3.EXECUTE

Virtual Machine

PharoCandle

Packages Out: 0 Packages In: 12

add all remove all

- Kernel-Classes
- Kernel-Collections-Abstract
- Kernel-Collections-Ordered
- Kernel-Collections-Unordered
- Kernel-Methods

Filter...

Classes Out: 0 Classes In: 50

- PCArray
- PCArrayedCollection
- PCAssociation
- PCBehavior

Filter...

Show sources Generate img

Execute in img Write img

Inspector on a DASTEvaluator

a DASTEvaluator

Raw Meta

Variable	Value
self	a DASTEvaluator
objectSpace	an EObjectSpace
codeProvider	a PImageBuilderCandle
interpreterClass	DASTInterpreter

```
self evaluateCode: '(PCArray new:3)
                    at:1 put:1;
                    at:2 put:2;
                    at:3 put:3;
                    yourself'.
```

Inspector on an EPMirror (a PCArray)

an EPMirror (a PCArray)

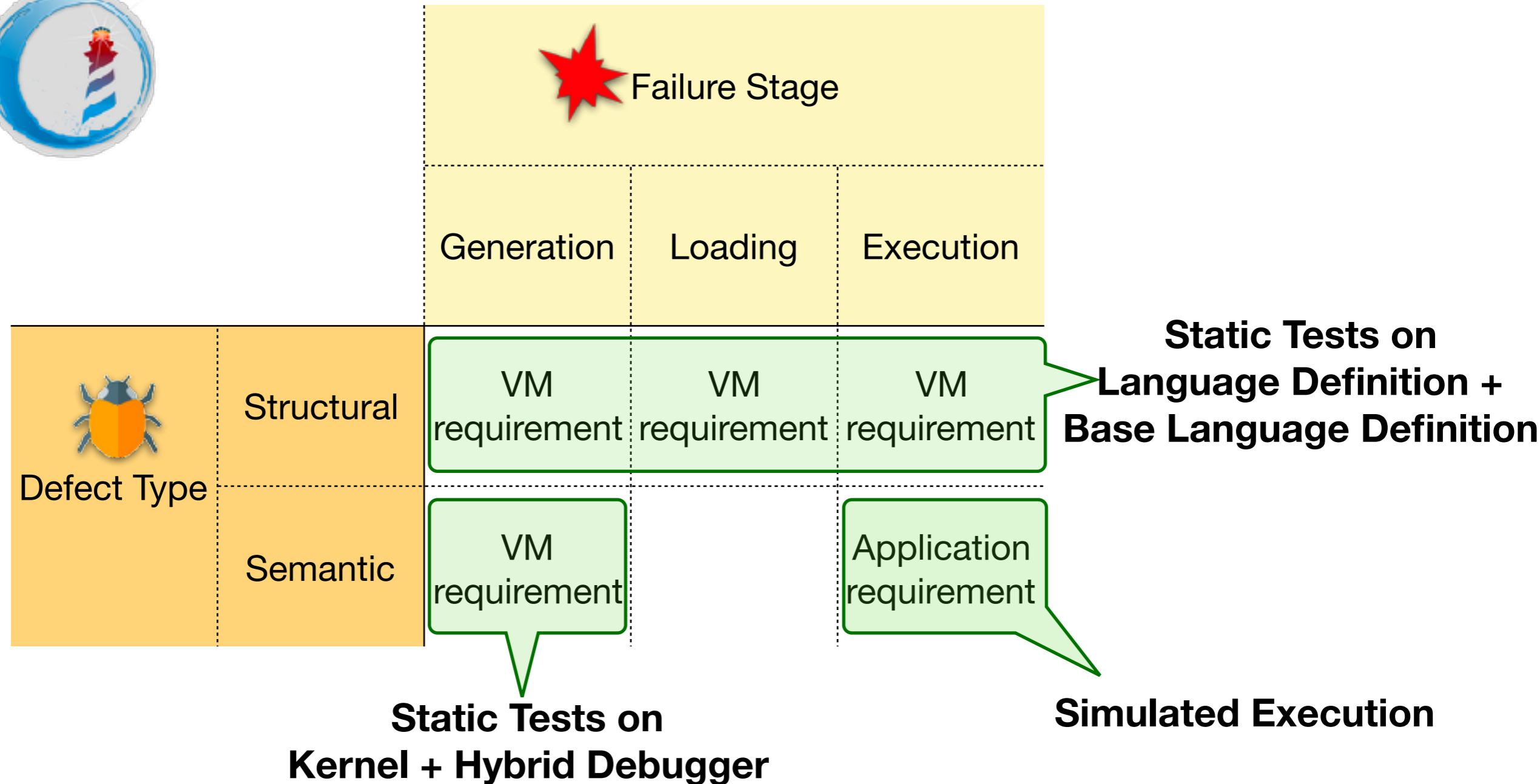
VariableS... mirror class Raw Meta

Index	Item
1	1
2	2
3	3



Taxonomy of Errors and Solutions

All these solutions can be used to debug the current Pharo bootstrap process!!



Research Directions

- Define the Pharo VM requirements, and model them for future modifications in future VM implementations
- Maximise the flexibility of the extensible base language definition, to maximise the range of languages that we can define from it
- Explore what is a good design for the hybrid debugger, so it contains the correct abstractions for debugging the bootstrap process
- Explore the limitations for the simulated execution environment
- Explore a way to debug failures hard to reproduce and which occur in production environment
- Shrinking the VM by removing unused plugins, which will be determined by dynamically analysing the simulated execution and its interaction with the VM simulator

Conclusions

- Analysis of Pharo Bootstrap process
- Taxonomy of Defects and Failures
- Proposed Solutions for each kind of error

Carolina Hernández Phillips
carolina.hernandez-phillips@inria.fr

Thanks for your attention

Image Inspector:

<https://github.com/carolahp/PharoImageInspector>

Steppable AST Interpreter:

<https://github.com/carolahp/DebuggableASTInterpreter>

Carolina Hernández Phillips