Virtual Smalltalk Images: Model and Applications

G. Polito
RMoD Project-Team, Inria
Lille–Nord Europe
Institut Mines-Telecom, Mines
Douai.
guillermo.polito@mines-douai.fr

S. Ducasse
RMoD Project-Team, Inria
Lille–Nord Europe
stephane.ducasse@inria.fr

L. Fabresse
Institut Mines-Telecom, Mines
Douai.
luc.fabresse@mines-douai.fr

N. Bouraqadi
Institut Mines-Telecom, Mines Douai.
noury.bouraqadi@mines-douai.fr

Abstract

Reflective architectures are a powerful solution for code browsing, debugging or in-language process handling. However, these reflective architectures show some limitations in edge cases of self-modification and self-monitoring. Modifying the modifier process or monitoring the monitor process in a reflective system alters the system itself, leading to the impossibility to perform some of those tasks properly. In this paper we analyze the problems of reflective architectures in the context of image based object-oriented languages and solve them by providing a first-class representation of an image: a virtualized image.

We present Oz, our virtual image solution. In Oz, a virtual image is represented by an object space. Through an object space, an image can manipulate the internal structure and control the execution of other images. An Oz object space allows one to introspect and modify execution information such as processes, contexts, existing classes and objects. We show how Oz solves the edge cases of reflective architectures by adding a third participant, and thus, removing the self-modification and self-observation constraints.

1. Introduction

In a Smalltalk environment, an image is a memory dump (snapshot) of all the objects of the system, and in particular all of the classes and methods at the moment of the dump. An image acts as a cache with preloaded packages and initialized objects. When the system is launched it takes an image as input and executes it from the place where the program counter was saved on previous save.

Smalltalk images are defined using a self-describing reflective architecture. Fully reflective architectures such as the one of CLOS [BGW93, Rho08] or Smalltalk [GR89] provide a simple and yet really powerful solution to develop tools such as full IDEs, code browsers, refactoring engines and debuggers [Riv96, Duc99]. Reification of the stack in addition to all the structural language elements allows one to manipulate program control flow as exemplified with modern web application frameworks such as Seaside [DLR07, GKVDHF01]. Indeed, a reflective system can be understood, changed and evolved using its own concepts and features. In addition, reflection is based on the notion of causal connection between the system and its meta-level [Mae87].

However, reflective architectures present some limitations. For example, kernel code such as array iteration method Array»do: is used by both base and meta levels of the system in a casually connected way. Because of this casual connection, breaking such a method impacts on libraries or tools that are essential in the system, causing the system to crash [DSD08].

Reflective architectures also suffer from the *observer effect* when doing analysis on the system. That is for example, observing its own running processes and their execution, or its consumed memory alters the observed element. Iterating the memory to count the amount of instances of a class, can create more objects in the iteration process. The manipulation of processes can be done only from an active process and thus, there is no possibility to activate directly a process from the language.

IWST '13, September 10 2013, Annecy, France

ACM 978-1-4503-2906-4/13/09 . . . $15.00
http://dx.doi.org/10.1145/2600128.2600130

To avoid this effect, the execution of these reflective operations is normally delegated to the virtual machine (VM). The virtual machine executes code atomically for the image's point of view. However, modifying the virtual machine to introduce new features is a tedious task, and there are not many developers experts in the area.

In this paper we propose to leverage this problems by creating an *image meta-level*. Our proposal is to move the control of this reflective operations from the virtual machine to another image. That is, an image will *contain* another image, and be able to reason about and act upon it. We call this *image virtualization*.

Contributions. The contribution of this paper is the introduction of Smalltalk *Virtual Images* to ease image analysis and evolution that is usually challenging in a reflective system (cf. Section 2). We describe Oz, an object space [CPDD09] based solution that we implemented on top of Pharo providing Smalltalk image virtualization (cf. Section 3). We also document the implementation details of both the language library and the virtual machine extensions we wrote (cf. Section 4). Then, we present some exemplar applications of this concept demonstrating that it solves the initial challenges (cf. Section 5). Finally, we discuss the solution and related work before concluding (cf. Section 6).

2. Reflective Architectures: Recurring Problems and State of the Art Solutions

Programming and evolving Pharo's core, several limitations and problems appear because of its reflective architecture. In the following subsections we illustrate some of these recurring problems, and describe their state of the art solutions.

2.1 Case 1: System self-brain surgery

Modifying Pharo's core parts from the system itself is a critical task. Core parts of a reflective system are in use while trying to modify them, generating an effect also known as self-brain surgery [CPDD09]. Doing so wrongly can put the system into an irrecoverable state since it may impact on elements that the system uses at the same time for running and applying the modifications. For example, that happens when changing methods such as Object»at: or Array»at:, adding new instance variables to core classes such as Process or Class, or even modifying tools like the debugger or browser. Introducing a bug at these places may make the system unusable, forbid the possibility to rollback the change and force a restart resulting in the loss of all the changes made.

Another issue while doing self-brain surgery on a system is that large system modifications cannot be performed in an atomic way. They should be split into several smaller changes, each of which may be critical on its own. Moreover, those changes also require to be applied in a specific order to be safe. Respecting a safe order constrains the development process, and therefore, restricts the developers working on the core of the system.

A typical case of self-brain surgery in Pharo is the modification of the debugger. The system automatically opens the debugger when an error occurs. The user performs actions with it like changing a method, evaluating an expression or even skip the error and proceed. However, making a mistake when rewriting a debugger's method may cause an irrecoverable infinite recursion. Indeed, an error launches the debugger, the trial for launching the debugger fails because of its bugged method, this debugger's failure leads to try to launch another debugger, and so on. Because of this infinite recursion, the user never gets the control back and cannot solve the original problem.

Many different problems may arise when doing self-brain surgery and for each of them, many ad-hoc solutions or workarounds have been proposed. For example, instead of modifying directly the debugger, a developer may make a copy of it to work on. Then, the system debugger can be used to debug and test the one in development. Once finished, the new debugger can replace the original.

The current Pharo distribution includes within its libraries an *emergency evaluator* to solve some self-brain surgery cases. Whenever an error occurs and the normal graphical user interface cannot be displayed because of that error, the control falls back to the emergency evaluator. The emergency evaluator is a simple tool with almost no graphical dependencies used to evaluate expressions and revert the last method submission. However, it depends on the compiler, the event machinery and the collection library, and thus, breaking any of those dependencies makes the emergency evaluator unusable.

Finally, bootstrapping a system[PDF+on] or recreating it from scratch solves partially the problems of self-brain surgery. These processes create new images in an atomic way, overcoming many of the self-brain surgery limitations. However, the development process in that case gets interrupted: the surgery fixes should be introduced inside the specification of the image, the new image containing the fix is built from scratch, the current working image has to be discarded, and the development should be continued in the new image. Ongoing changes during former development, which reside in the old image, should be either ported to the new image or discarded.

Requirement. A solution for self-brain surgery problems should include the possibility to apply **atomic changes** in the system, keep the development process as **interactive** as possible and **scope the impact of side-effects**.

2.2 Case 2: Uncontrolled Computations

From time to time a Pharo image can become unresponsive. This problem may be caused by a bug in the processes priority configuration *i.e.,* a never ending process with high priority does never give chance to run to other processes, and thus, the user cannot regain control to modify it because the user interface process is blocked. Currently, the only existing solution to regain control in such situations is the usage of the interrupt key. The interrupt key is a key combination that when pressed forces the running image to pause one of its processes.

On the one hand, when the virtual machine detects this situation, it signals a semaphore that should awake a handler process inside the image to handle this situation. On the other hand, the current implementation of the interrupt key in Pharo uses the input event process to detect if the given key is pressed. This process runs at a fixed priority of 60 (of a total of 80).

The current state of the art of interrupting presents the two following problems:

Interruption runs on the same level as other processes. When the interruption succeeds, it activates a process that is supposed to suspend the problematic process and give back the control to the user. However, the activation of this interruption process suspends the problematic process placing it in its corresponding suspended queue, making it undistinguishable from other processes. Then, the interrupting process *must guess* which was the process that was interrupted.

Bad process configurations induce starvation. Since the event handling process, which implements interruption, runs at a priority of 60, processes with higher priority may never be interrupted. Then, higher priority processes can avoid interruption and make lower processes starve. One solution to this problem is changing the configuration of the interruption process to make it run in the highest priority. However, there may be cases in which the process configuration needs a process with higher priority than the input event process.

Requirement. There is a need for a solution allowing the **non intrusive and non constrained control over processes execution**.

2.3 Case 3: Error handling during system initialization

The initialization of a Pharo image is implemented in the language itself. At startup time the system iterates the *startup list* and sends the startUp: message to each of the objects it holds. Each object in the startup list handle their own startup. The startup runs before giving control to the user.

Language libraries can access and configure the startup list, providing a flexible and easily extensible configuration mechanism. However, the accessibility of this feature leads to misuse and errors. Resources initialized on startup can provoke irrecoverable errors if not well handled. For example, resources using low level code may cause the current operating system process crash and quit. Under this kind of errors, the image quits on startup without providing the user a way to recover the work he did in previous sessions.

The recovery of all the work done in a failing image is a tedious task, without a conclusive solution. The changes log is a file storing the operations performed on the image, including all changes made to class and methods definitions and executed expressions. When available, it can be accessed from other images to restore the work done.

Additionally, some ad-hoc solutions appeared such as running the failing image with the virtual machine in debug mode. When debugging the system through virtual machine, the developer must deal with low level code and work at the bytecode level. In exchange, he can control completely the execution: failing statements can be skipped, the image can get finally initialized and the he can obtain control to fix the bug and recover his work.

Requirement. The system recovery should be a **high level** process, and easily accessible.

3. The Oz model for Virtual Images

A virtual Smalltalk image is an image living inside another Smalltalk image. The container image, the host, observes the virtual image and has complete control over it. The main idea is that such tasks difficult to perform due to the reflective architecture are handled by the host image. We transform the critical "self-brain surgery" tasks into safe "brain surgery" ones, by delegating them to another Smalltalk image.

Oz is a virtual image model and implementation based on object spaces [CPDD09]. Casaccio et al. sketched object spaces to solve self-brain surgery. When doing self-brain surgery, the image under modification becomes a *patient* of a *surgeon* image. The patient is included inside the surgeon as an object space. Through this object space, the image gets manipulated by the surgeon, fixed and finally awoken.

In Oz, an object space is a subsystem of another image. It is an object graph composed by two main elements: a full Smalltalk image (cf. Section 3.1) and a *"membrane"* of objects controlling that image (cf. Section 3.2). The image containing an object space is its *host*, while the object space is its *guest*.

Figure 1 shows a host image with two tools (ToolA and ToolB) interacting with an object space. The object space is enclosed by the dotted line. It contains a guest image and a membrane. The host tools interact with the membrane objects, while the membrane objects manipulate the objects inside the image.

In Oz we extended the object space model to apply self-brain surgery (cf. Section 3.3) and control rigorously both communication and execution (cf. Sections 3.4 and 3.5). In this section we describe the concepts and design principles guiding our solution for virtual Smalltalk images.

3.1 The Guest Image

The guest image inside an object space, as any other Smalltalk image, contains its own classes and its own special objects such as nil, true, false, processes and contexts. If we save this image on a filesystem, we can execute it as any

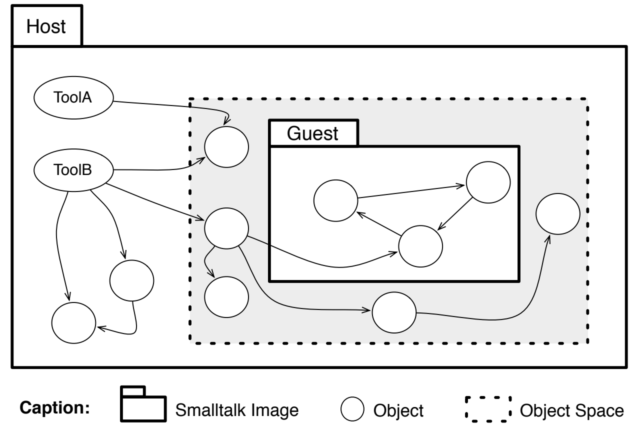

Figure 1. A host image contains an object space, represented as the region enclosed by the dotted line; the object space contains a guest Smalltalk image with its own object graph; the membrane is the gray region between the guest image and the dotted line; the tools inside the host interact with the objects in the membrane to manipulate the image.

other image. Indeed, it contains all objects that are necessary to run on its own dedicated virtual machine. Additionally, the guest image does not need to include any extra libraries or code for the host to include it. However, an image must fulfill a contract, as described in Section 3.6.

An object space's image contains an object graph satisfying the transitive closure property. That is, all objects inside the image reference only objects inside the same image. There are no references from the inside of the object space to its host. This is a key property to allow an image to be deployed both as an object space or as a standalone image on a dedicated virtual machine in a transparent way.

The object space enforces the isolation of its enclosed image in several ways. First, its membrane controls that no objects from the host are injected into the guest image. Second, it enforces that both guest and host images do not share any execution context. Finally, Pharo has no ability to forge object references [HCC+98] and therefore, the guest image can only refer to objects that are given to it explicitly, and not create arbitrary object references.

3.2 The Membrane

The membrane controls and enables the communication between the host and the guest objects. It encloses and encapsulates the guest image. This membrane is made up of objects which provide meta-operations to reason about and act upon the guest image. The host's objects cannot access the guest image but through the membrane's objects. The membrane objects are part of the host image and provided as a library in it.

The membrane contains objects to manipulate both the guest image as a whole and its inner objects individually. To manipulate the image as a whole, it provides one façade [GHJV95] object, the objectSpace. The objectSpace is a first-class object reifying the object space. Figure 2 shows the main methods conforming the API of an objectSpace object in Oz. To manipulate the individual objects inside the guest image in a controlled way, the objectSpace object provides mirrors, as described in Section 3.3.

```
ObjectSpace

"accessing"
nilObject
falseObject
trueObject
specialObjectsArray

classNamed:
classes
compactClassAt:
compactClassAt:ifNone:
globalNamed:

"conversion"
fromLocalByteString:
fromLocalByteSymbol:
fromLocalCharacter:
fromLocalCompiledMethod:

toLocalByteString:
toLocalByteSymbol:
toLocalCharacter:
toLocalCompiledMethod:

"process manipulation"
createProcessWithPriority:doing:
installAsActiveProcess:

transferControl
```

Figure 2. The API of an object space

3.3 Mirrors for Object Manipulation

The manipulation of objects inside the object space image cannot be achieved with a traditional message send mechanism. In the normal case, when a message send is performed, the virtual machine takes the selector symbol of the message and lookups in the class hierarchy method dictionaries of the receiver until it finds a method with the *same* (identical) selector. In our scenario, both host and guest images contain their own Symbol class and symbol table. Then, when performing a *cross image-message send* the method lookup mechanism takes a selector symbol from the host, lookups into the guest receiver's hierarchy, and finally fails because the selector in the guest is (while maybe equals) not identical to the selector in the host. Also, forcing a *cross image-message send* by using a guest's selector can leak host references to the guest: activating a guest method from the host gives the guest complete access to the host through the thisContext special variable which reifies the stack on-demand.

To encapsulate and control the basic object manipulation, the object space façade object provides mirrors [BU04]. Mirrors hide the internal representation of the objects inside the objectspace and expose reflective behavior. The guest is not aware of the existence of these mirrors.

A basic object mirror provides the following operations:

Field Manipulation. Operations to get and set values in both instance variables and variables fields of an object, such as at: and at:put:, or instVarAt: and instVarAt:put:.

Size calculation. Operations to get the size of an object expressed in the amount of instance variables and amount of variable fields, such as fixedSize and variableSize.

Class access. Operations to introspect and modify the behavior of an object, such as getClass and setClass:.

Special Objects Tests and Conversions. Operations to test if an object is a *primitive*[1] object such as nil, true or false, and to convert it to its equivalent in the host image, such as isNilObject, isSmallInteger or asBoolean.

All objects inside an object space and reachable by reference can be retrieved by host's objects through the object space facade and mirrors. There is no limitation nor restriction for object access. The host manipulates all objects in a homogeneous way through their mirrors.

Additionally, specific mirrors are provided to manipulate objects with a specific format and/or behavior such as Class, Metaclass, MethodDictionary, CompiledMethod, MethodContext, and Process.

3.4 Controlled Execution

An object space's execution is fully controllable from the host. The host can introspect and modify an object space processes via mirrors to obtain information such as the method currently on execution, the values on the stack or the current program counter. Besides from those reflective operations, an object space provides also operations to suspend, resume or terminate existing processes, and to install new ones.

The object space provides fine-grained control on the guest execution. An object space controls the amount of CPU used by the guest image. This way, a virtual image can be customized for scenarios like for example testing, CPU usage analysis, or old hardware simulation. For example, it may restrict its processes to run during only 300 milliseconds every second for either.

3.5 Controlled Communication

As explained in Section 3.1, an objectspace is an isolated object graph in the sense that from the guest image there is no way to reach host objects. However, the opposite relation is possible: the host can manipulate completely the object space.

The communication mechanism between host and guest images is based on the *injection of objects* into the object space. The host may install from simple literal objects such as strings or numbers, up to more complex objects like classes, methods. An object space permits to *send messages to objects* inside itself by injecting process with the specified code. Injected processes may have any arbitrary expression. The membrane objects can retrieve the result from the process' context once the execution is finished.

The object space membrane ensures that object injection honors the transitive closure property. On one side, literal objects from the host are automatically translated to their representation in the object space. An object space implements the operations to transform literal objects (numbers, strings, symbols, some arrays and byte arrays) *from and to* its internal representation.

On the other side, non literal objects are actually not created in the host and injected in the object space. Non literal objects are directly created in the object space, so the task of injecting the new object inside a graph is safe.

3.6 A Guest Image Contract

The creation and set up of an object space is done by putting in place the guest image and setting up the corresponding membrane. The guest image can be created either from scratch or by loading an existing image file. One way to create a guest image from scratch is for example by bootstrapping it given a specification [PDF$^+$on]. On the other side, loading an existing image file consists in putting the object graph from that image inside the object space.

Once the guest image is available, the host only sees it as a big object graph, not being able of differentiate the objects inside it. Then, to be able to manipulate the internals of the object space, the objectSpace and mirror objects must be configured with information about the internal representation of the guest image objects. They need the following kind of information in order to discover the rest of the guest image:

Special instances. In order to write some tools, and do comparisons and testing methods, the object space needs to know how to reach special instances such as nil, true and false.

System Dictionary. For the object space give access to classes, traits and even global variables installed in its inner image, a description on how to reach them must be provided.

Processes. It is important, for execution manipulation (cf. Section 3.4), that the image provides access to its process machinery. The accessibility to processes in running, suspended or even terminated state is vital, while it is also desirable the access processes in failing state for process monitors and debuggers. Direct access to the process scheduler and the priority lists is also desirable.

[1] we mean by primitive objects those that represent the simplest elements in the language

Literal Classes Mappings. Communication between host and guest require the translation of literal objects from and to the internal representation of the guest image (cf. Section 3.5). To achieve this, the object space needs to know the classes and internal format of those objects and thus, a mapping specifying the transformation must be provided. For example, the object space should know which are the classes inside the guest image that correspond to the host ByteString and SmallInteger ones to transform them if necessary.

Special classes internal representation. In order to manipulate some special objects in the object space, such as classes, metaclasses, processes and contexts, the internal representation should be given. Their internal representation includes both the amount of instance variables and variable fields, their size in memory, and their meaning. For instance, a class object format must include which are the instance variables containing the class name and the instance variables list.

4. Oz implementation in the Pharo Platform

We implemented Oz[2] in the Pharo 2.0 platform. Our solution virtualizes Pharo images and provides, as already described, the ability to fully control their object graph, inject objects in a safe way and control their execution.

Our implementation includes a language side library resembling the membrane objects and an extension to the Stack virtual machine. We decided to extended the Stack virtual machine to avoid dealing with the complexity of the Just In Time (JIT) compiler. The virtual machine extensions, described in Sections 4.4 and 4.5, include the addition of three primitives (load an image into the object memory, transfer the execution to an object space, and install an image in an object space as host) and the modification of the function in charge of the context switch mechanism.

In this section, we explain the details of our solution's implementation. We intend this section to document both the features a programming platform (language and virtual machine) should provide to build this kind of solution and the way our solution uses those features.

4.1 Memory Layout

We decided to make an object space share the same memory space (the object memory) used by the host. Then, objects from both host and guest are mixed in the object memory, and not necessarily contiguous, as shown in Figure 3. This decision is funded on minimizing the changes made to the virtual machine, because of its complex state of the art. Our decision, while easing the development of our solution, has the following impact on it:

Reuse memory handling mechanisms. We use the same existing memory infrastructure as when no object spaces are used. Existing mechanisms for allocating objects or growing the object memory when a limit is reached can be reused transparently by our implementation.

Simplify the object reference mechanism. References from the membrane objects to the guest image objects are handled as simple object references. No extra support from the virtual machine was developed in this regard.

Shared garbage collection. Since objects from the host and guest are mixed in the object memory, and their boundaries are not clear from the memory point of view, the garbage collector (GC) is shared between them. Every GC run must iterate over all their objects, increasing its time to run.

Observer's effect on an object space's memory. Analyzing and controlling an object space's memory still suffers from the *observer's effect* in our solution: every action taken by the host on the object space modifies the shared memory, and therefore alters the process. Because of this, an object space's memory cannot be properly analyzed.

4.2 Mirror Implementation

Our implementation of mirrors manipulate the objects inside an object space by using already existing primitives. There was no need to implement new primitives in the virtual machine since the existence of two primitives:

Execute a given method on an object. Given a method, it is possible to execute it on an object, avoiding method lookup in the object. In the current virtual machine, this primitive is implemented in the method **receiver:withArguments:executeMethod:** of the CompiledMethod class, with number 188. This method receives as arguments the object on which the primitive will be executed, an array of arguments, and the methods to execute.

Execute a primitive on an object. It is possible to send a message to an object, so a primitive is executed on the receiver. This primitive is implemented in Pharo's ProtoObject class as **tryPrimitive:withArgs:** with number 118 and receives as argument the number of the primitive and an array or arguments.

Since the primitive tryPrimitive:withArgs: executes the given primitive on the receiver of the message, and we want our mirrors to avoid *cross image-message sends* (cf. Section 3.3), we combine both primitives. We use primitive receiver:withArguments:executeMethod: to execute the primitive method tryPrimitive:withArgs: on the object from the guest image, avoiding the *cross image-message send* and executing directly the primitive on the given object.

Our mirror system contains three main mirrors regarding the internal representation of objects: a mirror for objects containing just object references such as Array or OrderedCol-

[2] The code can be found under http://www.smalltalkhub.com/#!/~Guille/ObjectSpace with licence MIT

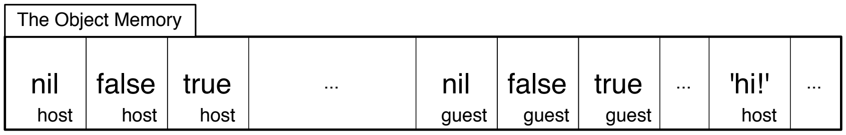

Figure 3. Objects from the host and guest are mixed in the object memory. In this figure, after the nil, true and false host instances, follow the corresponding ones of the guest, which can in order be followed by objects of the host, like the string **'hi'**.

```
CompiledMethod
    receiver: aGuestObject
    withArguments: { aPrimitiveNumber . anArrayOfArguments }
    executeMethod: (ProtoObject >> #tryPrimitive:withArgs:)
```

Figure 4. Combining the two *meta primitives* to execute a primitive on a guest object

lection, a mirror for objects with non-reference word fields such as Float or WordArray and a last one for objects with byte fields such as ByteArray or ByteString. In addition to them, we provide specialized mirrors for some kind of objects. The list of current mirrors we provide is the following: ObjectMirror, ByteObjectMirror, WordObjectMirror, ClassMirror, MetaclassMirror, ClassPoolMirror, MethodDictionaryMirror, MethodMirror, ContextMirror, ProcessSchedulerMirror and ProcessMirror.

4.3 Process Manipulation and Scheduling

Processes inside an object space are exposed to the host image as mirrors. Resuming/activating a process consists in removing it from the suspended list in its scheduler and put it as the active process in its image. Suspend a process consists in putting the process in the corresponding suspension list of its process scheduler. The ProcessMirror and the ProcessSchedulerMirror handle the scheduling in the guest image and keep the consistency in the object space process scheduler.

Using Oz, we can also create and install new processes inside an object space given a code expression. The creation of a process requires the creation of a compiled method with the code (bytecode) corresponding to the desired expression and a method context. The compiled method with the code to run is obtained by compiling the expression in the host and creating an object space compiled method. The object space compiled method is then provided with the compiled bytecode and its corresponding literals.

4.4 Context Switch between Images

Pharo virtual machine holds the state of the image running into the *special objects array* object. The special objects array is a simple array object referencing special objects directly accessed and used by the virtual machine. For example, it references objects such as the boolean and numeric classes or the nil, true and false instances. Particularly, the special objects array contains a process scheduler object and its corresponding process objects, implementing green threads. Pharo virtual machine has a single threaded nature and uses green threads to organize its execution.

An object space has its own special objects array, and thus, the code execution inside the object space must use its special objects and not the ones in the host. We modified the virtual machine to be able to perform a context switch between the host and the object space, and making it sensitive to the corresponding special objects array. We kept the single threaded nature of the vm, so the context switch between images puts the running image to sleep and awakens the new one. There are no concurrency problems between the different images.

Our modified VM has a special reference to the host's special objects array. To let an object space run, we implemented a primitive to explicitly give control to the object space by installing its special objects array. This primitive puts the current running process to sleep, changes the special objects array to the one request, and finally awakens the process installed as active in the object space. Figure 5 contains the VM code implementing this primitive.

Our implementation also supports the possibility to provide a controlled window of execution to an object space. The current VM possesses a heartbeat thread it uses to provoke a context switch every 20 milliseconds. Our implementation uses the heartbeat mechanism to pause the current object space process and give the control back to the host. We changed the VM function **checkForEventsMayContextSwitch:** adding the code in Figure 6, to use the behavior implemented in the primitiveResumeFromASpecialObjectsArray: primitive.

4.5 Creating an object space

An object space can be created either from scratch or by loading an existing image. Loading an existing image was implemented as a virtual machine primitive, because the image snapshot is actually a memory snapshot and therefore, easier to handle at VM level. This primitive, implemented with the code shown in Figure 7, reads the snapshot file, puts all objects into the object memory, updates the object references to make them coherent and finally returns the special objects array of the loaded image.

On the other side, creating an object space from scratch can be implemented as a bootstrap of the system, following the process defined in [PDF+on]. The object space provides

```
primitiveResumeFromASpecialObjectsArray:
                              aSpecialObjectsArray
  | oldProc activeContext  newProc |

  "we put to sleep the current running process"
  oldProcess := self activeProcess.
  statProcessSwitch := statProcessSwitch + 1.
  self push: instructionPointer.
  self externalWriteBackHeadFramePointers.
  activeContext := self
    ensureFrameIsMarried: framePointer
    SP: stackPointer.
  objectMemory
    storePointer: SuspendedContextIndex
    ofObject: oldProc
    withValue: activeContext.

  "we replace the special objects array"
  self replaceSpecialObjectsArrayWith: aSpecialObjectsArray.

  "we awake the process"
  newProc := self activeProcess.
    self  externalSetStackPageAndPointersForSuspendedCon-
textOfProcess: newProc.
  instructionPointer := self popStack

replaceSpecialObjectsArrayWith: newSpecialObjectsArray
  objectMemory specialObjectsOop: newSpecialObjectsArray.
  objectMemory nilObject:
       (objectMemory splObj: NilObject).
  objectMemory falseObject:
       (objectMemory splObj: FalseObject).
  objectMemory trueObject:
       (objectMemory splObj: TrueObject).

  "Reinitialize VM state to point to the correct nil object"
  method := objectMemory nilObject.
  messageSelector := objectMemory nilObject.
  newMethod := objectMemory nilObject.
  lkupClass := objectMemory nilObject.
```

Figure 5. VM functions written in Slang to transfer control to a virtualized image

the **createObjectWithFormat:** method to create an object respecting the given format but with an anonymous class, so we can consider it as a "classless" object. This method is used in the first stage of the bootstrap process, when no classes are available in the object space image yet, to create the nil instance (cf. Figure 8) and the first classes (cf. Figure 9). Later, when the classes are available, those objects are set their corresponding ones by using the **setClass:** message.

```
((hostSpecialObjectArray ~~ objectMemory nilObject)
  and:
[objectMemory specialObjectsOop ~~ hostSpecialObjectArray])
    ifTrue: [
      self primitiveResumeFromASpecialObjectsArray:
            hostSpecialObjectArray.
    ].
```

Figure 6. Additions to VM function **checkForEventsMayContextSwith:** written in Slang to give back control to the host image.

4.6 Image Contract and Membrane Configuration

Section 3.6 states the need for establishing a contract between an image and the object space in order to build the object space membrane. This contract has, in our understanding, two complementary parts: the services an image provides, and the format to access them.

Image services. In order for the host to manipulate the image inside an object space, the guest image must provide the required services. Those services are exposed as objects to the host, and their availability is given by how reachable they are in the object graph. For example, to get the list of classes inside an object space or to manipulate its processes, its system dictionary and its processor should, respectively, be reachable in the image's object graph.

Given a Pharo image from the current distribution, the reachability is constrained by its special objects array. The special objects array is the only object directly accessible of an image, since an image file contains in its header an explicit reference to it. So far, we understand the objects served by an image are the ones in the special objects array, which we detail in appendix A.

The special objects array contains references to many of the objects the membrane needs: nil, true, false, the processor, the numeric classes, the System dictionary, the compact classes, and some but not all literal classes. However, some elements in the special objects array are not mandatory in Pharo. For example, the System Dictionary may not available and then, there is no easy way to find all classes in the system.

The current special objects array in Pharo does not provide all necessary services. It has to be extended to support, for example, the recovery of process objects suspended because of an error. These processes currently are only referenced by graphical debuggers, and thus not easily reachable from the special objects array.

The image format. Given an object in the guest image, its enclosing object space requires its internal representation and format to manipulate it correctly. We mean by internal representation its size, its amount of variable and

```
primitiveLoadImage
    | headerlength bytesRead newImageStart rootOffset old-
BaseAddress dataSize rootOop fileObject |

  "get the reference to the file object"
  fileObject := self stackValue: 0.

  "Where will we put the new objects"
  newImageStart := objectMemory startOfFreeSpace.

  "read image header"
  self readLongFrom: fileObject.
  headerlength := self readLongFrom: fileObject.
  dataSize := self readLongFrom: fileObject.
  oldBaseAddress := self readLongFrom: fileObject.
  rootOffset :=
      (self readLongFrom: fileObject) - oldBaseAddress.

  "seek into the file the start of the objects"
  self seek: headerlength onFile: fileObject.

  "grow the heap in the ammount of the image size"
  objectMemory growObjectMemory: dataSize.

  "read the file into the free part of the memory"
  bytesRead := self
            fromFile: fileObject
            Read: dataSize
            Into: newImageStart.

  "tell the vm the free space is now after the loaded objects"
  objectMemory advanceFreeSpace: dataSize.

  "update the pointers of the loaded objects"
  self
      updatePointersForObjectsPreviouslyIn: oldBaseAddress
      from: newImageStart
      until: newImageStart + dataSize.

  "return the special objects array"
  rootOop := newImageStart + rootOffset.
  self pop: 2 thenPush: rootOop.
```

Figure 7. Implementation of primitive **primitiveLoadImage** that loads an image snapshot into the object memory written in Slang

fixed slots, the kind of and size of those slots, and in some cases their meaning.

First, the semantics associated to the special objects array and its contents should be provided. That is, what does each index of the array mean.

```
theNil := objectSpace createObjectWithFormat: nilFormat.
objectSpace nilObject: theNil.
```

Figure 8. Bootstrapping an object space: Creating a "classless" nil when there are no classes

```
metaclassMirror := objectSpace
   createClassWithFormat: classFormat
   forInstancesOfFormat: metaclassFormat.
metaclassClassMirror := objectSpace
   createClassWithFormat: metaclassFormat
   forInstancesOfFormat: classFormat.

metaclassMirror           setClass: metaclassClassMirror.
metaclassClassMirror    setClass: metaclassMirror.
```

Figure 9. Bootstrapping an object space: Creating "classless" Metaclass and Metaclass class when there are still no classes

Second, the guest image may differ from the host Pharo image. Then, the object space needs to make a correlation between the literal classes inside both host and guest to transform instances from and to the object space format. The classes subject to this correlation in our current implementation are ByteString, ByteSymbol, Array, SmallInteger, Character and Association. Such correlation is done by providing the corresponding transformation methods.

Finally, some mirrors must manipulate the internal state of special objects, and thus they must know their internal structure. The membrane configuration must provide the meaning of the instance variables of such special objects *i.e.,* the ProcessSchedulerMirror needs the index of the activeProcess and processList, and the ClassMirror needs the index of the superclass, method dictionary and name instance variables.

4.7 Non Implemented Aspects

For the sake of completion, we document in this subsection the aspects that have not been yet implemented in our solution.

Our current implementation does not handle properly the release of resources such as files or network connections (sockets). In Pharo, the finalization and release of such resources is made in the language side. Given the single-threaded nature of our solution, an image running can provoke the garbage collection of any object in the memory even if they belong to another image, since the object memory is shared by all images (cf. Section 4.1). However, garbage collection only activates in the current implementation the finalization process that belongs to the running image. The finalization processes of other images are ignored. Then, resources may leak, since they can be garbage collected but not properly finalized and released.

Another yet not implemented aspect regarding resources are global limitations imposed by the virtual machine. For example, the virtual machine memory is accounted globally without distinguish the usage per image; the virtual machine network plugin accounts and limits the amount of open sockets in a global way. In this sense, an image can use resources indiscriminately and restricting their use to other images *i.e.,* if there is a total of 100 sockets and an image opens 70, the rest of the images in the system have to share the 30 left.

5. Image Virtualization solving the Reflective Architecture Problems

Virtualizing an image, and therefore obtaining fine grained control on it from the language has several applications. In this section we describe some applications that solve common problems, although our solution is not constrained to them.

5.1 Image Surgery and Emergency Kernel Layer

Oz solves typical image surgery scenarios [CPDD09] such as the self-modification of the kernel and the recovery of broken images, described in sections 2.1 and 2.3. Using object spaces turn self-brain surgery into simple brain surgery, by introducing the role of the surgeon with a host image. Broken images can be loaded inside an object space to be subject of surgery in an **atomic** way. The host contains **high-level** tools such as a browser, an object inspector and a debugger to manipulate the object space and ease the surgery.

By using virtual images we can also provide a rich and **interactive** *Emergency Kernel*: whenever an error occurs in the running Pharo system because of self-brain surgery, the system can give the control to a fallback image. This fallback image is a full image containing the failing image inside an object space, and tools to act upon it, so it can perform surgery to solve the problem. The fallback image is to the system an *Emergency Kernel* which compared to the original emergency evaluator solution, has no dependencies on the failing image and therefore avoids its self-brain surgery problems. After the surgery, the main system can get back the control and continue running.

5.2 Controlled Interruption

Image virtualization can provide a solution for process interruption (cf. Section 2.2). When an object space is interrupted, its host obtains the control letting the interrupted object space untouched. This way, the interruption process has its two problems solved:

Non intrusive interruption. The state of the object space when the interruption took place remains unchanged. The problematic process can be found easily since is not moved to a suspended list, but remain as active process in the asleep object space.

Non restricted interruption. Since interruption is handled by the host image, there are no restrictions on which processes can be interrupted by the interrupt key combination.

5.3 Sandboxing

Oz can be used to sandbox applications by **limiting the scope of side effects** and the CPU consumption.

For example, running the some test suites of Pharo lets the system in a dirty state because of side effects. For example, the test case MCWorkingCopyTest unloads the MonticelloMocks package and reloads it again as Monticellomocks, without respecting the original casing. Oz leverages this problem by isolating the side effects inside the object space. The host stays unaffected and can analyze the test results when they finish to run. Finally, the object space under testing can be discarded while the user can continue working with the host.

6. Discussion and Related Work

In the field of virtualizing reflective object oriented languages and their runtimes, we did not find so far a work directly related with our solution. There are, however, works on isolation related with some parts of it, specially with the internal low level implementation details.

The memory layout we implemented has, as we stated in sections 4.1 and 4.7, many advantages regarding the development of our solution, but presents also many drawbacks. Sharing the object memory between different images implies that there is no need for special support on *cross-image references*, and that the existing memory management in the virtual machine can be used transparently. However, this solution forbids the host to analyze the object space memory usage, and has an impact on the GC.

J-Kernel [HCC+98] and Luna [HvE02] present a solution similar to ours regarding the memory usage. They are Java solution for isolating object graphs with security purposes. In them, each object graph is called a *protection domain*. All protection domains loaded in a system, and their objects, share the same memory space.

The J-Kernel enforces the separation between domains by using the Java type system, the inability of the Java language to forge object references, and by providing capability objects[Lev84, MRC03, Spo00] enabling remote messaging and controlling the communication. This same separation in Luna [HvE02] is achieved by the modification of the type system and the addition in the virtual machine of the *remote reference* concept. In our solution, the separation is given by the same inability to forge object references and the membrane objects that control the communication.

KaffeOS [BHL00] makes an explicit domain separation in memory by using different memory heaps in the virtual machine. They enforce domain separation by using memory write barriers. Cross-domain references become cross-heap references, and thus, they need special virtual machine support.

Regarding the threading model (cf. Section 4.4), a Pharo virtual machine has single threaded execution. Only one operating system thread is used to execute Pharo code, so process scheduling is handled internally by the virtual machine. Processes scheduled using this approach are also called *green threads* . Green threads provide process scheduling without native operative system support while limiting the proper usage of modern multicore CPUs. In our implementation, their usage allowed us to reuse the current virtual machine scheduling. We also use a green thread approach to schedule image execution. All images are executed in the same single thread, one at a time. This model simplifies our implementation because it avoids concurrency problems between host and guest images.

KaffeOS presents a model where resource accounting is handled at the level of the virtual machine. Our solution aims to control and account resources at the language level. However, our implementation is not complete yet on this front.

Worlds [WK08] scope side-effects of Javascript programs by reifying the notion of its state. Our solution takes a similar approach by reifying images. In our solution, images have a notion of their own state just like Javascript Worlds, but include also its manipulation from the outside.

7. Conclusion and Future Work

This paper explores image virtualization for object oriented reflective systems such as Smalltalk. We present Oz, an object space based solution for image virtualization. Oz object spaces provides services to control and manipulate Smalltalk images, without enforcing the inclusion of extra libraries inside them. In particular, Oz object spaces allow image surgery and the manipulation of an image's execution from the language.

Oz object spaces encapsulate and enclose their inner image by creating a membrane of objects responsible for its communication and control. The membrane is composed by a façade object which reifies the object space, and mirrors that control the communication between the host and single objects inside the object space. This façade and mirrors hide the internal details of the object space, such as its internal representation, memory layout or threading model. This encapsulation property may allow to implement alternative Oz object spaces, polymorphic with the current one. For future research we would like to explore the object space API for controlling remote images and how it relates to distributed images.

Oz presents a green thread scheme of execution. It virtualizes processes and avoids concurrency problems by enforcing mutual-exclusion of the execution of different images. As future work, we want to explore the introduction of operating system threads to take advantage on the latest multicore CPUs, take control of them through the objectspace and account their consumed resources through the language.

For future work, we would like to explore Oz as an infrastructure for developing customized Smalltalk kernels and software analysis.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region (CPER) 2007-2013.

References

[BGW93] Daniel G. Bobrow, Richard P. Gabriel, and J.L. White. CLOS in context — the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

[BHL00] G. Back, W. Hsieh, and J. Lepreau. Processes in kaffeos: Isolation, resource management and sharing in java. In *4th USENIX International Symposium on Operating System Design and Implementation (OSDI)*, 2000.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[CPDD09] Gwenaël Casaccio, Damien Pollet, Marcus Denker, and Stéphane Ducasse. Object spaces for safe image surgery. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST'09)*, pages 77–81, New York, USA, 2009. ACM digital library.

[DLR07] Stéphane Ducasse, Adrian Lienhard, and Lukas Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

[DSD08] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The meta in meta-object architectures. In *Proceedings of TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 218–237. Springer-Verlag, 2008.

[Duc99] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GKVDHF01] Paul Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the web with high-level programming languages. In *Proceedings of ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 122–136, 2001.

[GR89] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[HCC+98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in java. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 22–22, Berkeley, CA, USA, 1998. USENIX Association.

[HvE02] C. Hawblitzel and T. von Eicken. Luna: a flexible java protection system. *ACM SIGOPS Operating Systems Review*, 36(SI):391–403, 2002.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.

[MRC03] Todd Millstein, Mark Reay, and Craig Chambers. Relaxed multijava: balancing extensibility and modular typechecking. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 224–240. ACM Press, 2003.

[PDF$^+$on] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Benjamin Van Ryseghem. Bootstrapping reflective systems: The case of pharo. *Journal on Science of Computer Programming - Special Issue: Smalltalk Based Systems*, 2012, under submission.

[Rho08] Christophe Rhodes. Sbcl: A sanely-bootstrappable common lisp. In *International Workshop on Self Sustainable Systems (S3)*, pages 74–86, 2008.

[Riv96] Fred Rivard. Pour un lien d'instanciation dynamique dans les langages à classes. In *JFLA96*. INRIA — collection didactique, January 1996.

[Spo00] Lex Spoon. Objects as capabilities in squeak, 2000.

[WK08] Alessandro Warth and Alan Kay. Worlds: Controlling the scope of side effects. Technical Report RN-2008-001, Viewpoints Research, 2008.

A. Appendix: The Special Objects Array

In this appendix the present an overview of the special objects array used by the Pharo platform. We present for each of its indices: (a) the object to be found, (b) if that object is mandatory for the virtual machine and (c) relevant comments. If the object is not mandatory for the virtual machine, a nil reference will took the place most certainly.

We emphasize in **bold** the objects required so far in Oz in order to be able to introspect an image. The availability of literal classes can be replaced by the availability of the *system dictionary* and the required class names in the membrane configuration.

Array Index	*Required in Pharo Stack VM core*	*Object*	*Details*
1	x	**nil**	
2	x	**false**	
3	x	**true**	
4	x	**Scheduler association**	
5		Bitmap class	Required only for graphics.
6	x	**SmallInteger class**	
7	x	**ByteString class**	
8	x	**Array class**	
9		**System dictionary**	Elemental: without it, Oz cannot reach all classes in the image.
10	x	**Float class**	
11	x	**MethodContext class**	
12		BlockContext class	This class does not exist any more in Pharo.
13	x	Point class	
14		LargePositiveInteger class	
15		Display class	Required only for graphics.
16	x	Message class	
17		**CompiledMethod class**	Not used by the Virtual Machine
18		Low space semaphore	Used to signal low space
19	x	Semaphore class.	
20	x	**Character class**	
21	x	doesNotUnderstand: selector	
22	x	cannotReturn: selector	
23		Low space process	The Virtual Machine uses this internally. Not used by the language.
24	x	Special selectors array	An array of the 32 selectors compiled as special bytecodes.
25	x	Character table	An array of the 255 Characters in ascii order.
26	x	mustBeBoolean selector	
27		ByteArray class	
28		**Process class**	Not used by the Virtual Machine.
29	x	Compact classes array	An array of up to 31 classes whose instances have compact headers.
30		Delay semaphore	Used if scheduling timers only.
31		Interrupt semaphore	Used for VM side interruption.
32		Float prototype	Not used by the Virtual Machine.
33		LargePositiveInteger prototype	Not used by the Virtual Machine.
34		Point prototype	Not used by the Virtual Machine.

Array Index	*Required in Pharo Stack VM core*	*Object*	*Details*
35	x	cannotInterpret: selector	Used in case method dictionary in a class is nil.
36		MethodContext prototype	Not used by the Virtual Machine.
37	x	BlockClosure class	
38		BlockContext prototype	Not used by the Virtual Machine .
39	x	External objects array	Array of objects referred by external code.
40		Mutex	Not used by the Virtual Machine.
41		LinkedList for overlapped calls in CogMT	Used by another Virtual Machine implementation.
42		Finalization Semaphore	
43		LargeNegativeInteger class	
44		ExternalAddress class	Used for FFI calls.
45		ExternalStructure class	Used for FFI calls.
46		ExternalData class	Used for FFI calls.
47		ExternalFunction class	Used for FFI calls.
48		ExternalLibrary class	Used for FFI calls.
49	x	aboutToReturn:through: selector	Used to notify of unwind contexts.
50	x	run:with:in: selector	For objects as methods usage.
51		Immutability message	Not used in Pharo.
52		FFI errors array	Not used by the Virtual Machine.
53		Alien class	Used for FFI callbacks.
54		invokeCallback:stack:registers:jmpbuf: selector	Used for FFI callbacks.
55		UnsafeAlien class	Used for FFI callbacks.
56		WeakFinalizer class.	Used in Weak finalization.