Virtualization Support for Dynamic Core Library Update

Guillermo Polito
RMoD Project-Team, Inria
Lille–Nord Europe, France
CAR Team, Institut Mines-Telecom,
Mines Douai, France

Stéphane Ducasse
RMoD Project-Team, Inria
Lille–Nord Europe, France

Noury Bouraqadi
CAR Team, Institut Mines-Telecom,
Mines Douai, France

Luc Fabresse
CAR Team, Institut Mines-Telecom, Mines Douai,
France

Max Mattone
RMoD Project-Team, Inria Lille–Nord Europe,
France
CAR Team, Institut Mines-Telecom, Mines Douai,
France

Abstract

Dynamically updating language runtime and core libraries such as collections and threading is challenging since the update mechanism uses such libraries at the same time that it modifies them. To tackle this challenge, we present Dynamic Core Library Update (DCU) as an extension of Dynamic Software Update (DSU) and our approach based on a virtualization architecture. Our solution supports the update of core libraries as any other normal library, avoiding the circular dependencies between the updater and the core libraries. Our benchmarks show that there is no evident performance overhead in comparison with a default execution. Finally, we show that our approach can be applied to real life scenario by introducing a critical update inside a web application with 20 simulated concurrent users.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Extensible Languages

General Terms Languages, Object-Oriented Programming

Keywords Hot Updates, OOP, Pharo, Self-Modification

1. Introduction

Unanticipated software changes have led to the development of the *Dynamic Software Update* (DSU) field [HN05]. DSU targets to update applications at runtime without the need to restart them, thus minimizing application down-time and warm-up time. Software updates may exhibit critic aspects ranging from feature additions to critical security patches and bug-fixes.

This paper explores and proposes an approach to the problems of dynamically updating core libraries (DCU) in the context of a high-level object-oriented language. A core library is a library that contains essential behavior of a language *i.e.,* it contains built-in functions and data structures, or elements related to the execution of the language itself. We can name for example collections, threading or networking. Core libraries provide basic functionality that most applications rely on. Thus, issues present in these libraries are critical, as they may affect all users of the language. In addition, changing and updating core libraries is not a simple task. Recent research on DSU for high-level object-oriented languages, however, do not focus on core library changes. For example, the authors of Rubah [PVH14], a novel DSU solution which is efficient, flexible and non-disruptive, say the following on the topic:

> *For Rubah, the only classes that cannot be updated are the Java runtime classes and libraries (e.g., Java collections). Updatable classes can directly reference non-updatable classes but not the reverse, due to issues involving the bootstrap class path of a Java application.*

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

Onward!'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3688-8/15/10...
http://dx.doi.org/10.1145/2814228.2814233

Our analysis of core library evolution shows that (1) such libraries require hot updates as any other parts of a system, and that (2) it is challenging to achieve such updates. Let's take Pharo [BDN$^+$09], a smalltalk-inspired programming language that is constantly upgraded by using dynamic updates. During the last years, Pharo's engineering team faced different issues in the integration of changes such as changing the hashing of Dictionaries or the refactoring of the Set and Dictionary classes to introduce a common abstract superclass. These changes are challenging as the entire runtime, including the update mechanism, uses such classes to do the update. This list is not exhaustive and contains other non-trivial changes *e.g.,* the introduction of better closures, API changes in the class creation code, the merge of the classes that reify execution contexts (ContextPart and MethodContext), and the introduction of traits [SDNB03].

Languages such as Smalltalk and Lisp, and DSU systems such as Changeboxes [DGL$^+$07] and Rubah [PVH14] use reflection [Smi84] to perform *hot* updates. Reflection allows systems to introspect themselves and perform self-modifications in-situ. However, using reflection to update core libraries poses three main problems: a circular dependency between the updater and the core libraries, a tight dependency on the order in which changes must be applied and the restrictions imposed by the language or the VM that may prevent to apply some changes (cf. Section 2). However, the solution we explore in this article is not directly linked with reflection per se. We pose ourselves the following research question:

What is the infrastructure required to update core libraries at run-time when they are used by the update itself?

We propose Espell, an *application virtualization* approach that can be used complementary to DSU systems to support DCU (cf. Section 3). The application under update runs virtualized, under the control of a hypervisor that is in charge of applying the update. Both the virtualized application and hypervisor are separated, containing each one its own copy of the core-libraries. Espell presents the following properties:

Runtime independence. We avoid circular breakages by construction. The virtualized application and the hypervisor are indeed independent, containing each own its own copy of the core-libraries. Thus, the update never affects itself.

Atomic changes. All changes are applied from the hypervisor while the virtualized application is suspended. Then, from the virtualized application perspective, all changes are atomic and change ordering is avoided.

Full control. The hypervisor has full control on the virtualized application including even low-level VM-language wirings. On one side, this allows us to reconfigure the VM[1]. On the other side this avoids the insertion of unsafe code in the virtualized application.

Negligible performance overhead. Espell poses almost no performance penalties to the virtualized application when it is normally executed. The hypervisor is only active when an explicit update is available. Otherwise, the virtualized application runs at the full VM speed.

To validate Espell we built Chester, our Espell-based prototype DSU system. Chester implements manual data and control-flow migration. Chester updates use Espell's virtualized application API (cf. Section 4). For this, Chester accesses a virtualized application *meta-object* that provides operations that support code loading, instance migration, execution manipulation (needed for control-flow migration) and VM-language configuration.

We implement our solution in the Pharo language (cf. Section 7). We validated Espell and its implementation by running three different evaluations (cf. Section 5). First, we present functional validations that show that our solution provides effectively runtime independence, atomic changes and control on the low-level wirings of the application. Second, we benchmark Espell's virtualization infrastructure to show that there is no evident performance overhead in comparison with the Vanilla flavor of the Pharo VM. Finally, we show that Chester can be applied in a real life scenario by introducing three real case core library bug-fixes inside a web application with 20 simulated concurrent users.

2. Challenges of DCU

Dynamic Core Update (DCU) is the dynamic update of a language core libraries. Core libraries implement the basic features provided by a programming language (typically collections, networking and threading, amongst others) and are widely used by all the users of the language. Then, bugs or problems in these libraries impact a critical mass of users. DCU support opens the door to easily patch these critical problems on the fly.

DCU, as a subset of DSU, presents the same challenges as the latter. It does, however, present some challenges of its own. In this section we start by showing some real cases of DCU in the Pharo programming language and their problems. Then, we generalize these problems illustrating them with example.

2.1 DCU in the Real World

We studied the evolution of Pharo [BDN$^+$09], a smalltalk-inspired programming language and platform. The goal of Pharo is to explicitly revisit the complete language implementation from ground up. In addition Pharo update process is done by incremental dynamic updates modifying

[1] The Pharo VM, following the tradition of Smalltalk VMs, gives access from the language itself to the elements (specific classes *e.g.,* Array, message selectors *e.g.,* doesNotUnderstand:) required by the VM to work.

the system while it is running. Still Pharo is a real system with around 1000 users and for example Pharo 3.0 got 2364 closed issues[2]. Finally, Pharo is deployed and used in production so updates should be not let the system in an inconsistent state. Therefore Pharo evolution is a really good candidate to study DCU challenges. In addition, we interviewed one of the core developer and asked him about the problems the core team faced over the years. Here is an excerpt of the problems:

Changing Dictionary Hashing. Dictionaries are widely used objects not only in applications but in other core-libraries themselves. Changing their hashing implies rehashing existing dictionaries. Moreover, it means that if the update mechanism uses dictionaries, the update mechanism may break itself during the update because there is a moment where dictionary lookups may miss.

Introduction of the HashedCollection class. HashedCollection is a class introduced as an abstract superclass of collections that use a hash *e.g.,*Set and Dictionary. This change requires numerous refactorings such as moving methods up and down in the hierarchy. The complexity of such a change required to apply it in many steps to avoid breaking the update mechanism.

Renaming the Float class. For a future virtual machine (VM) update, the Pharo team faced a refactor in the Float class. Float became abstract and two new subclasses were added: ImmediateFloat (unused by the moment) and BoxedFloat (replacing the old Float class). Besides the code refactor, this change requires a *hot* VM reconfiguration. Indeed, the virtual machine needs to know the new BoxedFloat instead the (now abstract) Float class to perform floating point arithmetic and conversions.

This list is not exhaustive: the development of Pharo contains many other non-trivial changes *e.g.,* the introduction of better closures, API changes in the class creation code, introduction of traits in the language core and the merge of the classes that reify execution contexts (ContextPart and MethodContext).

2.2 Example

To clearly identify and illustrate the problems of DCU, let's analyze a toy example written in a Smalltalk-like language. This language contains the base class Object and an Array class representing vector objects. The Object class provides an at: method that returns the object at a given field of the receiver. Likewise, it provides a size method returning the amount of fields in the object. The class Array provides a do: iteration method that applies a closure to each element referenced by the array. This is an excerpt of code of the parts of this language that interest us:

```
Object >> at: index
  "field at index of the object"
  <primitiveAt>

Object >> size
  "number of fields of the object"
  <primitiveSize>

Array >> do: aClosure
  "apply aClosure to each element in the array"
  1 to: self size
    do: [ :index | aClosure value: (self at: index) ]
```

The next version of this language makes all objects in the language iterable by pushing the do: iteration method from the Array class to the Object class. We would like to apply this update at runtime. A first problem that arises is that these classes should be visible and modifiable from our update process. If they are hidden or fixed, as it happens in Java with its bootstrap class loader mechanism [LB98], we would not be able to change them. Next, there is a risk of letting the system in an unusable state if we do not correctly apply this change. Imagine that our update mechanism uses a globals dictionary that references all installed classes. Internally, that system dictionary object contains an array with its entries. A naive solution, using reflection [Smi84], would be to remove the do: method from Array and add it into Object.

```
(globals at: #Array) removeMethod: #do:.

(globals at: #Object) addMethod: ’do: aClosure
  "apply aClosure to each element in the array"
  1 to: self size
    do: [ :index | aClosure value: (self at: index) ]’
```

By first removing the do: method from the Array class, there is no means to iterate an Array anymore. Then, fetching the Object class from the globals dictionary in the second step fails, crashing the update mechanism and also the application.

One possible solution to this problem is to sort the update in a way that the additions are executed before the removals. This solution is however not reliable either: overwrites and inter-dependent methods could mean that there is not a correct order to apply them. An alternative solution to this problem is to use temporary structures to hold our changes. For example, we could make copies of the classes under modification, modify these copies and finally install them:

```
arrayUnderUpdate := (globals at: #Array) copy.
objectUnderUpdate := (globals at: #Object) copy.

arrayUnderUpdate removeMethod: #do:.
objectUnderUpdate addMethod: ’do: aClosure
  "apply aClosure to each element in the array"
  1 to: self size
    do: [ :index | aClosure value: (self at: index) ]’.
```

[2] http://pharo.org/news/pharo-3.0-released

```
"Install the copied classes into the globals table"
globals at: #Array put: arrayUnderUpdate.
globals at: #Object put: objectUnderUpdate.
```

While this solution allows us to change the Array and Object classes without caring about the order, the order problem is just pushed forward to the moment where we install the copies of the classes into the globals table. That is, if we install the new Array class before the new Object class, the application crashes again, for the same reasons.

2.3 Core Updates Main Challenges

From these cases mentioned earlier and the real world cases we studied, we identify three general problems that arise when updating core libraries: the circular dependencies between the updater and the core libraries, a tight dependency on the order in which changes are applied and the restrictions imposed by the language or the VM that may prevent to apply some changes.

Circular Breakages. There is a circular dependency between the update mechanism and the core libraries: the update mechanism uses the libraries it is updating. Then, changes in the core-library may impact the update mechanism itself. An update may break the update mechanism and leave the whole application in an inconsistent state. In our example, the update mechanism depends on the Array class that is under update.

Order Dependability. Applying critical changes on a running application is order dependent. A particular order may crash the application while others will smoothly update. Some changes may also require to be applied atomically to not crash the application. In our example, if our update mechanism would have installed the new Object»do: method before removing the old one, the application wouldn't have crashed after the removal.

Language and VM Restrictions. To update a core-library we require the support to access it and change it. Changing a language core classes and libraries at runtime may be prevented by a language in several ways. Reflective features [Smi84] may be not provided by some languages, or be restricted for safety reasons. Additionally, the changes we can apply may be constrained by the Virtual Machine (VM). In our example, we need to be able to modify critical classes such as Array and Object.

3. Dynamic Core Update trough Virtualization

This section presents our solution to DCU by using virtualization. First, we give an overview of Espell, our solution and its architecture. The toy example of the previous section is then presented.

3.1 Our Solution in a Nutshell

The architecture of Espell is based on running side by side two different applications: the application we want to update (*i.e.,* the virtualized application) and a *hypervisor*. The hypervisor and the virtualized application do not share the core libraries, each one has its own copy. Chester's *updater* (*i.e.,* the component in charge of applying the update) resides inside the hypervisor and manipulates the virtualized application through a first-class virtualized application object. Figure 1 shows an overview of the solution's architecture.

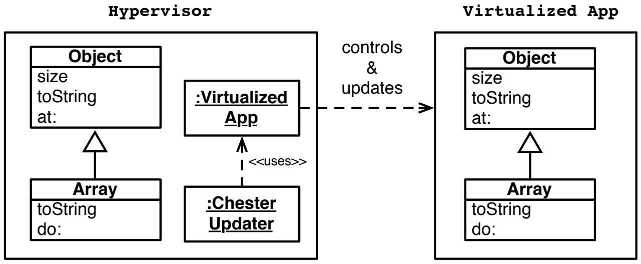

Figure 1. Solution Architecture Overview. Chester's updater inhabits a different application than the updated application. It controls the updated application through a first-class virtualized application object.

Using virtualization, the updater gets the following properties:

Runtime Independence. The updater changes core classes from the virtualized application safely. Indeed, the updater does not depend any more on the core libraries of the virtualized application: it has its own copy. This breaks the circular dependency between the updater and the core-libraries of the update application, allowing it to perform the update without affecting itself.

Atomic Changes. All changes performed by the updater take place while the virtualized application is paused. From the managed application's perspective these changes are applied atomically. Thus, the order in which we apply the changes is not of relevance anymore.

Full Control. The updater is an external entity to the virtualized application and has full control on the latter through the virtualized application meta-object. It is able to apply changes that the virtualized application could not apply by itself because the language restricts it.

3.2 Update Example

Let's imagine a desktop application with a UI thread that process events and renders UI elements inside an infinite loop. We place an update point [HN05] inside the loop indicating that we can safely update before each iteration:

```
UIProcess >> run
  [ true ] whileTrue: [
    Chester updatePoint.
    self processEvents.
    self render. ]
```

The virtualized application runs uninterruptedly until it reaches an update point. At that moment, if updates are available Chester activates the hypervisor. The hypervisor pauses the UI thread at the update point and starts Chester's updater. The updater fetches the Array class from the virtualized application and remove its do: method. Next, it fetches the Object class and installs a new version of the do: method. Finally, once the update is applied the UI thread of the virtualized application is resumed and the hypervisor is suspended again (until the next update point).

```
ChesterUpdater >> update
  "The virtualized application is paused"
  arrayClass := virtualizedApplication classNamed: #Array.
  arrayClass removeMethod: #do:.

  objectClass := virtualizedApplication classNamed: #Object.
  objectClass compileMethod: 'do: aClosure
    1 to: self size do: [ :i |
      aClosure value: (self at: i) ]'.

  virtualizedApplication continue.
```

Notice that while the code performing the update in our example looks trivial, the main difference resides in the architecture. We can easily manipulate the Array class of the virtualized application without breaking the updater since it is different from the hypervisor's Array class. The order is not important because both applications are independent and the virtualized application is paused at that moment. Indeed, we can remove the do: method from the Array class before we add the new method in the Object class, even introducing a moment in the execution where there is no do: method in the virtualized application. Finally, our virtualization mechanism provides access to the Array and Object classes from the virtualized application, even if such access is forbidden from within the virtualized application itself.

3.3 Update Workflow

To focus on core-library updates, our solution uses explicit update points such as the ones present in existing solutions *e.g.,* Rubah [PVH14]. Figure 2 shows the workflow of a virtualized application when it reaches an update point. First, it checks if updates are available. In that case, it suspends itself and gives the control to the hypervisor. The hypervisor performs the update while the virtualized runtime is suspended. Finally, it resumes the execution of the update application by returning it the control and resuming all threads.

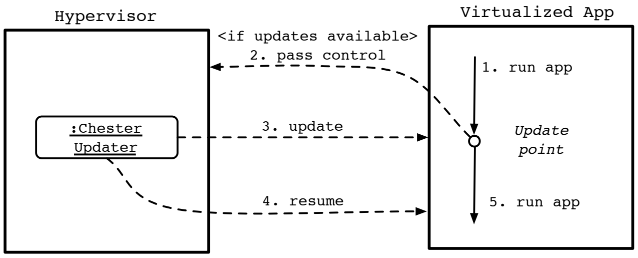

Figure 2. Update Workflow.

In the case where the application under update is a multithreaded application, Chester waits each thread to arrive to an update point. If updates are available, threads are suspended when they reach an update point. Once all threads are suspended, the control is passed to the hypervisor to perform the update. Notice that Chester is not a mature DSU system but a prototype we build to evaluate Espell's virtualization support. A more mature DSU system may use alternative algorithms to detect when an update is feasible [VEBD07].

3.4 Control Flow and Data Migration

Chester migrates data and control flow with scripts. For this, Espell provides a virtualized application *meta-object* in the hypervisor that provides such support through its meta-object protocol (MOP) [KdRB91]. In this section, we give some examples of how control flow and data is migrated using Chester. We explain further our virtualized application MOP in Section 4.

Data Migration. Chester uses scripts to manually migrate application data. Chester scripts transform objects from one version to the other. Let's take for example the migration of a Point class from a 2D representation (with variables x and y) to a 3D representation (with variables x, y and z). Chester iterates all the instances of the old version of the class, creates a new instance, migrates the data, and swaps the identity of the old point objects by the new ones.

```
ChesterUpdater >> updateFrom: oldPointClass to: newPointClass
  oldPointClass allInstances do: [ :oldPoint |
    | newPoint |
    newPoint := newPointClass basicNew.
    newPoint slotAt: 'x' put: (oldPoint slotAt: 'x').
    newPoint slotAt: 'y' put: (oldPoint slotAt: 'y').
    newPoint slotAt: 'z' put: 0.
    oldPoint swapIdentityWith: newPoint.
  ]
```

Control Flow Migration. Chester provides support for application control migration at several levels. The simplest control-flow migration mechanism is the manual startup and termination of threads. For example, we want to change the control-flow of the UIProcess»run that executes the infinite loop of the UI thread. To do so we can terminate the thread that was suspended in the update point and create a new thread that will restart from the new version of this method definition.

```
ChesterUpdater >> updateUIThread: uiThread
  uiThread stop.
  virtualizedApplication
    createNewThreadSending: #run
    to: uiProcessObject.
```

Chester supports also fine-grained control flow migration thanks to Espell. Threads in Espell contain a chain of execution contexts or stack frames that we can manipulate. Chester can insert and remove stack frames in a thread, and even revert a thread to a previous stack frame to restart it, without the need to create a new thread. Then, an alternative implementation of the control flow migration of this example would be as follows:

```
ChesterUpdater >> updateUIThread: uiThread
  | runMethod |
  runMethod := uiThread actualStackFrame.
  unthread actualStackFrame: (runMethod parentStackFrame).
```

4. A Virtualized Application MOP

Updating an application requires access and control over its internal representation. We need the means to create, remove and modify classes, methods and objects of the virtualized application. Our solution focuses on the support to avoid circular breakages, order dependability and VM restrictions and does not cover the automatic migration of application state and control flow. For this reason in Chester we do migration through scripts that manipulate a virtualized application *meta-object*. This meta-object eases such manipulations through its meta-object protocol (MOP) [KdRB91]. This MOP is based on our previous work on the object space model [PDFB13]. Following, we present the basic operations of the virtual application MOP that we needed to do manual migrations during DCU. We distinguish four different update stages, which are reflected in our MOP: *code loading* is used to install new code, *instance migration* serves to the purpose of updating existing objects to the new loaded code, *execution manipulation* is required to adapt the code that was running to the loaded code, and the *VM-language interface* is used to update the relationship between the VM and the language when modifying essential objects and classes.

4.1 Code Loading MOP

The most basic operations of DSU must support the installation of new code. In our context of object-oriented applications, this implies the proper installation (and deinstallation) of classes and methods.

create class *<name>*, *<spec>*. It creates a class named <*name*> whose instances will follow the specification <*spec*> *i.e.,* their type and number of slots. It returns the meta-object of the newly created class. This class is not installed in the virtualized application *i.e.,* it is not visible to other classes.

install class *<class>*. It installs <*class*> in the virtualized application: Make <*class*> available to the rest of the code in the virtualized application.

remove class *<class>*. It uninstalls <*class*> from the virtualized application: Make <*class*> unavailable to the rest of the code in the virtualized application.

get class by name *<name>*. It returns the meta-object of the class named <*name*>.

compile method *<source code>*. It creates a new method by compiling <*source code*>. Names (*e.g.,* class names, globals) inside <*source code*> are linked to the corresponding objects and classes inside the virtualized application. It returns the meta-object of the newly created method.

install method *<class>*, *<method>*. It installs <*method*> as part of <*class*>: make available this method to the rest of the code in the virtualized application.

remove method *<class>*, *<method>*. It removes <*method*> from <*class*>: make unavailable this method to the rest of the code in the virtualized application.

4.2 Instance Migration MOP

When new code is loaded, there is a possibility that instances of the old version of the code still exist. In such cases, after loading the new code the update mechanism must also migrate existing instances to their new representation. The following MOP provides the basic operations for such a migration.

create instance *<class>*. Creates an instance of <*class*> *i.e.,* an instance that conforms to the spec of <*class*>, containing the number and type of slots described in it. It returns a meta-object of the newly created instance.

instances of *<class>*. It returns a list of the instances of <*class*>.

get class of *<object>*. It returns the meta-object that corresponds to the class of <*object*>.

set class *<object>*, *<class>*. It changes the class of <*object*> to <*class*>, if both classes have the same spec.

get slot *<object>*, *<slot name>*. It returns the meta-object that corresponds to the object referenced by the slot named <*slot name*> of <*object*>.

set slot *<object>*, *<slot name>*, *<new value>*. It sets the slot named <*slot name*> of <*object*> to <*new value*>.

swap identity *<old object>*, *<new object>*. It replaces all references to <*old object*> by references to <*new object*>. This operation is important to guarantee that the migration of <*old object*> to <*new object*> is complete.

4.3 Execution Manipulation MOP

Loading code may invalidate and replace old code. In turn, this invalidates the threads that were suspended on the execution of old code. These *old code threads* should be migrated to new code threads. The following MOP provides the basic operations for such a migration.

get threads. It returns a list of meta-objects for each of the threads in the virtualized application.

create thread <*object*> <*message*>. It creates a thread that will start running by sending <*message*> to <*object*>. It returns a meta-object that corresponds for the newly created thread.

install thread <*thread*>. It installs <*thread*> in the virtualized application. When the virtualized application is resumed, this new thread will start.

terminate thread <*thread*>. It forces the termination of <*thread*> in the virtualized application. This thread will not execute anymore once the virtualized application is resumed.

get stack frames <*thread*>. It returns the list of meta-objects that correspond to the activated stack frames of <*thread*>. This operation serves to understand where the virtualized application was suspended and what was the code executed at the moment.

set current stack frame <*thread*>, <*stack frame*>. It sets <*stack frame*> as the current stack frame of <*thread*>. When <*thread*> is resumed, it will continue its execution from <*stack frame*>.

create stack frame <*object*>, <*message*>. It creates and returns a stack frame that will send the message <*message*> to <*object*> when activated. The created stack frame is not inserted in the execution.

insert stack frame <*stack frame before*>, <*stack frame after*>. It inserts <*stack frame after*> as the next stack frame of <*stack frame before*>. This operation allows us to modify the call stack of a given thread.

4.4 VM-Language Configuration MOP

The VM and language are tightly coupled to execute code. We can generalize this relationship as a set of the well-known objects of the language that the VM requires at runtime. Examples of such well-known objects are Boolean objects true and false required to evaluate boolean expressions, or the Array class that may be used internally by the VM. In scenarios where these objects are not meant to change, they are configured only during the startup of the application. To apply DCU, we include in our MOP two basic operations that allows us to modify the interface between the language and the VM.

get special object <*name*>. It returns a meta-object that corresponds to <*name*>. This operation enables the introspection of the current VM-language configuration.

set special object <*name*> <*object*>. It replaces the object at <*name*> by <*object*>. This operation enables the reconfiguration of the VM-language interface.

5. Experimental Validation

We divide the validation of our solution in three parts. First, a functional validation presents that our solution can overcome the challenges of DCU. Following we present a series of benchmarks based on the Benchmark Game Suite showing that the overhead of our virtualization solution is in general negligible compared to a vanilla Pharo Stack VM. Finally, we measure the overhead of applying real bug-fixes in a web application, showing that the application only degrades temporarily its performance without loosing requests and finally recovers its normal performance.

5.1 Functional Validation

Two simple experiments show that our solution correctly solves the problems stated in Section 2. Our first experiment shows that the changes applied by Chester are atomic and independent from Chester. The second experiment shows that Chester can do deep modifications such as VM reconfigurations.

Runtime Independence and Atomic Changes. To test that Chester can deploy an update in an independent and atomic way by removing a core method. Removing normally this method should provoke a crash in the system. This validation shows that indeed Chester can remove such method and leave the updated application temporarily inconsistent. Then, it can re-deploy the method and put again the application in a consistent state. Chester was not affected by this core change as it has its own copy of the core libraries. From the updated's application perspective this change happened atomically, not perceiving that it was temporarily inconsistent. The code conforming our experiment is the following.

```
"remove old method"
(virtualizedApp classNamed: #SequenceableCollection)
   methodDictionary removeKey: #do:.

"Add again method"
objectClass := virtualizedApp classNamed: #Object.
objectClass compileAndInstall: ’do: aBlock
   "Refer to the comment in Collection|do:."
   1 to: self size do:
      [:index | aBlock value: (self at: index)]’.
```

Language-VM reconfiguration. To test that we can correctly modify objects and classes that are part of the VM-Language interface, we modified the SmallInteger class. SmallInteger is the class of integer objects that are represented on 31 bits. Pharo VM represents them as a tagged reference (*i.e.,* an object reference with a flagged bit) instead of occupying the place of an object inside the heap.

We replaced the SmallInteger class by a new subclass of it (so clients are not broken) that implementing a new method of our own and then reconfigured the VM to use our new class. The code applying such a change is the following.

```
newClass := virtualizedApp
   createClassNamed: #MySmallInteger.
newClass subclassOf:
   (virtualizedApp classNamed: #SmallInteger).

newClass compileAndInstall: 'asPoint
  ^ Point x: self y: self'.

virtualizedApp smallIntegerClass: newClass.
```

5.2 Virtualization General Benchmarks

To estimate the cost of our virtualization infrastructure, we run the computer language benchmarks game[3] in two different scenarios. We first benchmarked the Vanilla Pharo Stack VM without any of our changes, to make it our basis of comparison. The Stack VM makes a fair basis of comparison since our virtualization infrastructure does not yet count with a JIT compiler. Afterwards, we benchmarked a virtualized application without update points. In the latter, the virtualized application runs in our specialized version of the Pharo Stack VM that contained also a hypervisor. These measurements were made on a 2.2 Ghz Intel Core i7 machine with 8 GB memory 1333 Mhz DDR3.The results (cf. Table 1) show for most of the benchmarks that our virtualization infrastructure performs in a similar range than the vanilla Pharo Stack VM implementation.

5.3 Real Case Updates

To evaluate our solution in a more realistic scenario, we chose three issues from Pharo's issue tracker. These issues represent two bugs in the core of the language (where one is a severe problem in exception handling) and one feature addition in BlockClosure. Particularly the exception handling bug was staged in five different steps to avoid turning the environment unusable. Table 2 lists the three bugs fixes, the number of classes and method affected and the number of steps that Pharo integrators had to perform to *hot* apply any of these issues normally in comparison with Chester's atomic update.

Benchmark	Vanilla VM (1x)	Virtualized	Overhead
SpectralNorm	77.31ms +/-0.25	78.56ms +/-0.22	1.02x
KNucleotide	94.6ms +/-1.2	100.5ms +/-6.5	1.06x
Cham. Redux	179.34ms +/-0.31	176.91ms +/-0.35	0.99x
BinaryTrees	36.53ms +/-0.21	35.02ms +/-0.20	0.96x
RegexDNA	6982ms +/-93	5751.1ms +/-3.0	0.82x
Mandelbrot	400.16ms +/-0.41	451.17ms +/-0.40	1.13x
Reverse Comp.	4.90ms +/-0.16	5.79ms +/-0.18	1.18x
ThreadRing	2.29ms +/-0.14	2.53ms +/-0.17	1.10x
PiDigits	0.63ms +/-0.17	0.66ms +/-0.17	1.05x
Meteor	3182ms +/-68	2144.3ms +/-1.9	0.67x
NBody	28.95ms +/-0.28	21.09ms +/-0.19	0.73x
Fann. Redux	0.110ms +/-0.081	0.070ms +/-0.067	0.63x
Chameleons	19.64ms +/-0.23	21.86ms +/-0.21	1.11x
Fasta	7.80ms +/-0.20	4.93ms +/-0.14	0.63x

Table 1. Virtualization Overhead. Comparing the execution time of several benchmarks over the vanilla VM and its virtualized version - run 100 times each.

Issue	Classes Affected	Methods Affected	Normal Steps	Chester Steps
Exception handler[4]	4	13	5	1
Block Memoization[5]	1	2	1	1
Float debugging[6]	1	3	1	1

Table 2. Integration steps of three bug fixes. Comparison of the number of steps required to integrate a particular issue into the Pharo programming language without and with Chester.

We applied these updates to the SmalltalkHub web application. SmalltalkHub[7] is a web-based code repository service for the Monticello version control system (the VCS used by Smalltalk implementations such as Pharo and Squeak). The productive SmalltalkHub application contains over 2000 repositories and has over 1700 users. For testing, we used a modified SmalltalkHub's web server. We added in it an update point before a connection is served (Figure 3). Our benchmarking methodology is as follows. We started our SmalltalkHub application and we performed several requests to it simulating 20 concurrent users during 90 seconds. At second 30 we introduced the update from the example *i.e.,* pushing up in the hierarchy the do: iteration method.

Figure 4 shows the response time graphs for our testing Smalltalk Hub application when deploying such updates. We benchmarked our application during 90 seconds, sampling the response times every half a second. Our graphs show the moment of update (around the second 30 of each graph). We observe that applying these changes generate little slowdowns in our application, affecting the response time of our application. Other slow downs or *hiccups* are due

[3] http://benchmarksgame.alioth.debian.org/.
Pharo Implementation: http://www.smalltalkhub.com/#!/~StefanMarr/SMark

[4] Issue 11996: https://pharo.fogbugz.com/f/cases/11996/

[5] Issue 14458: https://pharo.fogbugz.com/f/cases/14458/

[6] Issue 13761: https://pharo.fogbugz.com/f/cases/13761/

[7] http://www.smalltalkhub.com/

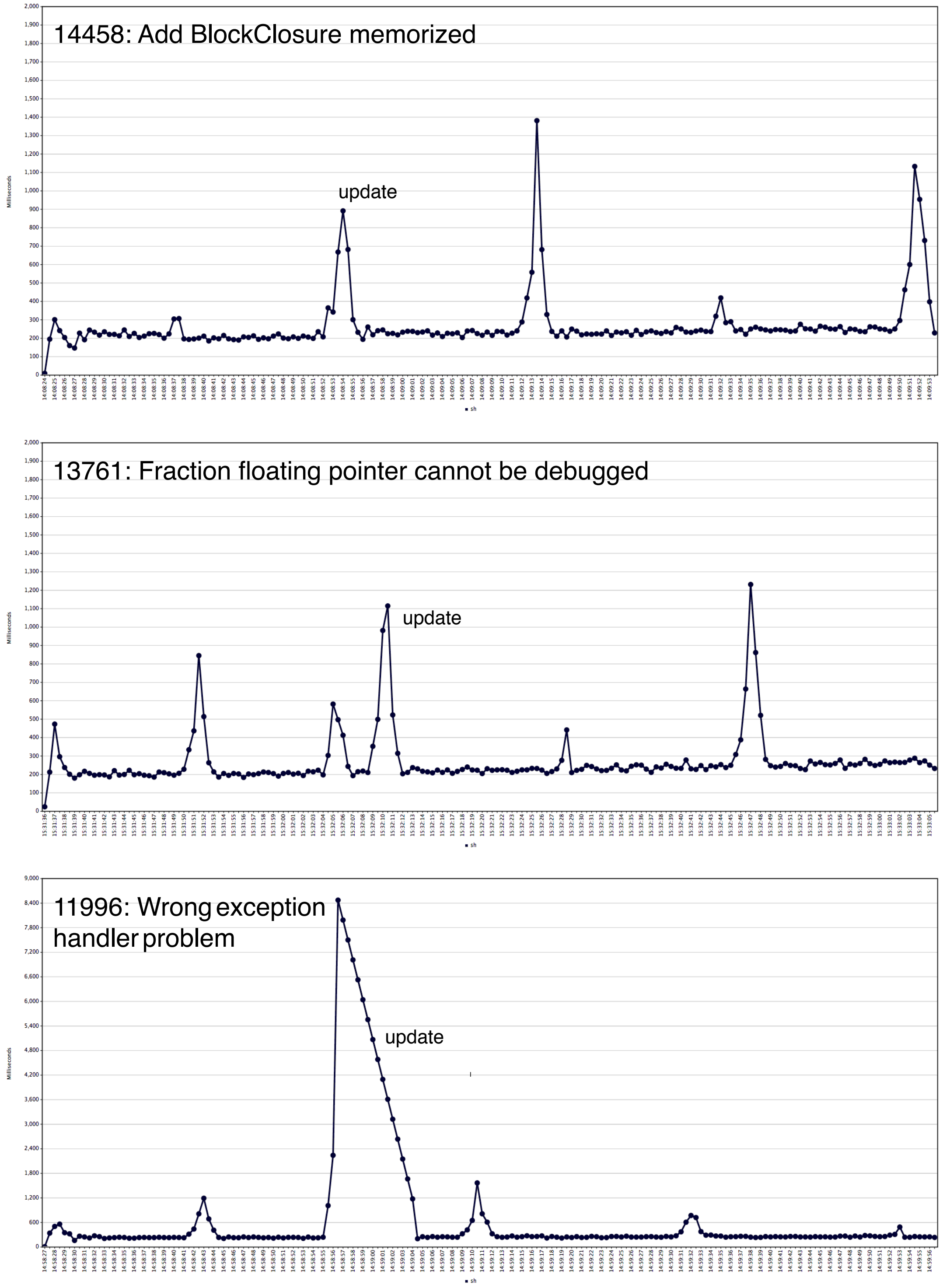

Figure 4. Response Time Graph of SmalltalkHub Application. Running an application during 90 seconds with an update at second 30. Simple updates create pauses similar to garbage collections. After the update the application returns to its usual performance level.

```
ZnMultiThreadedServer >> listenLoop
  self initializeServerSocket.
  [
    [ Chester updatePoint.
    self serveConnectionsOn: serverSocket
    ] repeat
  ] ifCurtailed: [ self releaseServerSocket ]
```

Figure 3. Adding an update point to SmalltalkHub.

to garbage collection pauses. We observe that updating the fixes for issues 13761 and 14458 produce slowdowns comparable to a garbage collection pause. Issue 11996 on the other hand produces a pause that slows down requests for eight seconds maximum. The nature of the change forces the system to scan the stacks of all the current (stopped) processes. In addition, these slowdowns can be optimized by pre-calculating the patches to apply and precompiling the source code to install such as it is done in Rubah[PVH14] and Kitsune [HN05]. These optimizations are however out of the scope of this paper.

6. Discussion

6.1 Flexibility and Frequence of Changes

Flexibility is one of the desirable features of a DSU system [HN05]. We believe that a flexible DSU system should not limit the software elements that can be updated, and thus, it should provide with DCU support to be a complete solution. At the same time, we could argue that usually core libraries do not change as fast as application code. This is however not the case of Pharo. Pharo's open-source community has an open culture and embraces the modification of core libraries where *e.g.,* Pharo 3.0 got 2364 closed issues[8]. We believe that a language with good DCU support would adopt changes and refactorings in core libraries more often and have shorter release cycles.

6.2 Runtime Independence vs Memory Consumption

Espell's atomicity, runtime independence and full control are a trade-off to more memory consumption due to the extra copy of the core libraries. While this trade-off may become a problem in memory constrained environments, this is not the case of the systems where most DSU systems are aimed (typically service-like applications). In addition, the memory consumption of the hypervisor can be reduced by using in it a specialized version of the core libraries. For example, in previous experiments we produced minimal core libraries with a memory footprint of no more than 100KB [PDF$^+$14] and we automatically tailored existing core libraries for specific usages, producing object heaps that starting in 10KB of memory footprint [PDBF11].

7. Implementation

We implemented Espell on top of the Pharo language, based on the idea of object spaces [PDFB13]. Object spaces are isolate-like applications as in the Multitasking Virtual Machine [CD01]. That is, two different Pharo applications run on top of the same VM. They are, however, independent *i.e.,* each one has its own copy of core-libraries and objects. In our implementation the hypervisor and the virtualized application reside each one inside a different object space. Figure 5 shows an overview of the implementation and Table 3 gives an idea of the effort of this implementation measuring its number of lines of code.

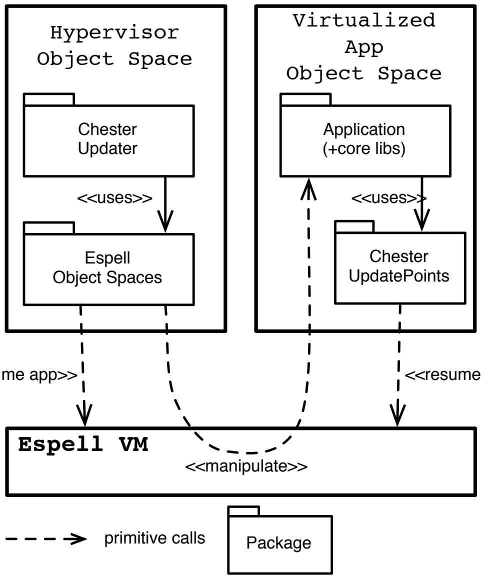

Figure 5. Implementation Overview. Two isolated object spaces run on top of the same Espell VM. The Espell Object Spaces library allows the hypervisor to manipulate the virtualized application. Chester is built on top of Espell virtualization support.

Our Espell implementation comprises some modifications on Pharo Stack VM and a hypervisor-side library. The VM modifications provide support to run several object spaces on top of the same VM (and having for example different copies of the core libraries). These changes include the primitives to resume the hypervisor/virtualized application and patch some primitives such as the ones that iterate over the heap (to only iterate over the correct object space). Espell's hypervisor-side library implements the object space meta-object and its MOP. Espell object spaces manipulate the virtualized application through VM primitives and provide a high level interface through mirror objects [BU04]. Most of the primitives we use were already existing in the VM. Implementing Espell supposed a one-time effort of 461 lines of code extending the existing VM and a bit more of 4700 lines of Smalltalk code for the hypervisor-side library. Our VM modifications are portable between different op-

[8] http://pharo.org/news/pharo-3.0-released

erating systems as they are completely written in Slang, a Smalltalk subset used to build Pharo's VM, so not platform specific code is used [IKM+97].

Chester is a DSU library on top of Espell. Chester is fully implemented in Pharo, without extra VM support. It is divided in a hypervisor-side and a virtualized application-side libraries. On the virtualized application-side, the Chester Update Points library provides support for declaring update points and deciding when an update should be applied. When an update point is activated from the application, Chester invokes an Espell VM primitive that suspends the virtualized application threads and activates the hypervisor. Chester's updater is a hypervisor-side library in charge of executing the update scripts on the object space *meta-object* and then resume the application's execution. When the virtualized application is resumed, all threads from the hypervisor are suspended. Chester occupies roughly 48 lines of code, 22 in the virtualized application side and 26 in the hypervisor side. Note that Smalltalk code is compact - method average line of code is 6.

Component	Lines of code
Espell VM	461
Espell Library	4735
Chester Updater	26
Chester UpdatePoints	22

Table 3. Implementation Effort. Implementation effort of our solution measured in lines of code. All of them are one-time efforts.

8. Related Work

8.1 Reflective Systems and Reflectivity

Reflective systems and languages provide support for accessing to a program's representation and change it at runtime [Smi84]. To enable reflection, mainstream languages such as Java, Ruby or JavaScript count with a reflective architecture [Mae87]. A reflective architecture relies on the idea of *causal connections i.e.,* the programming language incorporates structures that represents aspects of itself (*e.g.,* classes, objects), in such a way that if one structure changes the aspect it represents is updated accordingly, and vice-versa. However, the introduction of these structures introduces a circular dependency between the reflective and the base layer of the language: the reflective layer uses the same base layer it is modifying. This property introduces a risk of infinite meta-recursions when the meta-level instruments code that it relies upon [CKL96]. This meta-recursion problem is a particular case of a circular breakage as presented in this paper.

Denker et al. partially solve this problem in Reflectivity [DSD08]. Reflectivity is a reflective framework that avoids meta-recursions by tracking the degree of metaness of the execution context. In each reflective call, a MetaContext object is activated and it accounts the meta-level jump. Likewise, when the reflective call returns, the MetaContext is deactivated. Using the accounted meta-level jumps of the MetaContext, meta-objects do only reflect on objects of a lower metaness (and not greater or equal metaness). Thus, the program has contextual information and can scope reflective behavior. Reflectivity succeeds to modify and scope behavioral changes for different meta-levels inside the same reflective architecture. However, it does not provide with support for structural changes of the code the reflective framework depends upon, such as changing the structure of classes or removing methods and classes.

Mirrors [BU04] present a general solution for metaprogramming and reflection. A mirror is an explicit meta-object providing reflective operations on base-level objects. In such a manner, mirrors decouple the meta and base levels of an application. Mirrors have been used previously in the context of debugging and meta-programming purposes. Our solution implements mirrors to cleanly describe its meta object protocol.

8.2 DSU Systems

DSU solutions for several languages investigate only application-level runtime updating and do not concern themselves with core language updates.

Rubah [PVH14], for example, assumes not being able to update Java runtime classes as it must stick to the language limitations. On one hand, Rubah needs the application to be suspended at a quiescent point to apply changes. However, objects from the Java core libraries never reach such quiescent points because Rubah runs in the same environment and uses them. On the other hand, instance migration process in Rubah can be performed lazily and more generally the data migration process is independent from the update process. Our virtualization solution can only affect the system while the execution is suspended.

DCEVM [WWS10] and JRebel [Zer12] aim to enhance the developer experience bringing live updating to java at development time: developers do not have to recompile and redeploy the project to see the effect of a change. Such behavior is standard in Smalltalk and Lisp system since 1980. DCEVM fully relies on the Java Debug Wire Protocol and so it can only be used with the debugger enabled. JRebel only runs on standard Java application servers. Neither of them support a full state transfer process and cannot ensure the migration of data through versions.

Jvolve [SHM09] provides a new Java virtual machine supporting dynamic updates. It can automatically find safe update points in the code. Jvolve will inspect the execution stack of a process and insert a return barrier that will be executed as the process is safe to be updated. It will thus be suspended at a safe update point. Therefore, no additional code is required from developers to know when to perform the update. As Jvolve, Espell supports to be suspended automatically at special VM conditions (particularly we implemented suspension in send and buckjump bytecodes). Then, transparent update points could be added in Chester to make it easier to use. We chose however to implement explicit update points to give the user the control on them and because it they make easier to assume the state of the call stack while migrating the application's control flow.

Changeboxes [DGL$^+$07] is a change model designed to encapsulate and scope changes. Its main purpose is to allow several versions of a system to coexist at runtime *i.e.,* the existence of different versions of the same system in the same environment. In Changeboxes, a *changebox* encapsulates the changes made on classes and methods and an executable version of the system with these changes applied. The system can contain many changeboxes at the same time, and applications can be scoped to run within different changeboxes. This notion of dynamically scoping an application to a changebox allows one to have co-existing environments (*e.g.,* testing, development, production), increasing the developer's efficiency. Furthermore, it eases application update and migration to new versions, and reduces its update down-time as the application does not have to be stopped to be updated. The Changeboxes model proves sound to update and migrate application and framework classes. However, it has the main drawback of not affecting critical classes in the system. The Changeboxes prototype does not work on classes such as Array or CompiledMethod as the underlying infrastructure (the VM) restricts the system to the existence of only one of them at the same time. The change model does not provide either a solution for this problem, as it focuses on application code update, leaving this as an open problem.

8.3 JVMTI

The Java Virtual Machine Tool Interface (JVMTI) [JVM] presents an architecture similar to our virtualization system but including also some main differences. JVMTI is the interface offered by the Java VM for its manipulation and control. It was originally called Java Platform Debugger Architecture (JPDA) and used in the context of debugging. JVMTI provides introspection and some limited intercession facilities at VM and language level. In particular it provides memory and heap management, thread control, execution stack manipulation, object and class manipulation and breakpoint support. JVMTI is used mainly for debugging, monitoring and analysis purposes, particularly profiling and thread analysis. JVMTI exposes this behavior as C functions whose client is called an *Agent*. JVMTI agents are meant to be written in C, to be as compact as possible and allow maximal control with minimal intrusion.

Our object space MOP provides a range of operations that is similar to the one present in JVMTI. We observe however that our solution is less restrictive and offers more altering operations. We think this is maybe due to a JVM design decision for security. Another key difference is the programming level of the *agents*. Our solution allows the programmer to describe its manipulations in high-level Smalltalk, having access to all the benefits of programming in such a language.

8.4 Virtualization Approaches

In Espell we try to reconcile language *virtual* machine technology with operating system virtualization. High-level language VMs abstract the developer from the complexities of the underlying physical machine. A language VM provides a language with an execution model closer to its semantics as well as several services such as automatic memory management, cross-cutting optimizations and portability. Although these language VMs are indeed *Virtual*, state of the art production-ready VMs do not provide by themselves the typical advantages of virtualized operating systems.

On the operating system side, we can find virtualization approaches like Xen [Chi07]. Xen is a Virtual Machine Monitor (VMM) that allows one to control and manage VMs in a high performance and resource-managed way. This approach targets the virtualization of full and unmodified operating systems (OSs), to facilitate their adoption in industrial/productive environments. Operating System virtualization technology is characterized by the existence of a *hypervisor* (named after the Operating System *supervisor* that controls the OS processes). The hypervisor is the VM component that allows one to observe or control the internals of one or many VMs. A VM hypervisor provides amongst others, VM co-location, resource control, security and application mobility.

Espell introduces co-location of applications in a language VM, as in the Multitasking Virtual Machine [CD01], and introduces also the idea of hypervisor: a virtual machine monitor that can observe and manipulate the internals of virtualized applications.

9. Conclusion and Future Work

This paper proposes to add dynamic core update support to DSU systems through runtime virtualization. We show how Espell, our virtualization infrastructure can deploy changes in core libraries in an atomic and order-independent way, making it also possible to update critical classes even those used by the VM. Espell works by allowing two independent Pharo applications running on top of the same virtual machine: the application to update and a hypervisor. A runtime meta-object, namely an object space, provides the hypervisor with a MOP to manipulate the application under update. This MOP includes operations to load and remove code, modify objects and reconfigure the VM-language relationship of the updated application.

We validated Espell with Chester, a DSU library featuring explicit update points and manual control-flow and data migration. First, we show how Chester uses the virtualization features to update critical classes in a safe way. We show that our virtualization model using explicit update points has no visible performance overhead when no update points are present. Finally, we deployed three real bug-fixes into a real web application used by 20 simulated concurrent users showing we can simplify updates that took several integration steps with slow-downs similar to a garbage collection pause.

For future work, we aim at extending this infrastructure in two different ways. First, we would like to raise Chester to the level of a real DSU system introducing alternative data and control-flow migration mechanisms and update points. Second, we would like to explore the hot-update of VMs using this infrastructure.

Acknowledgments

We thank the European Smalltalk User Group for their support (www.esug.org).

References

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BU04] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *OOPSLA'04*, pages 331–344. ACM Press, 2004.

[CD01] G. Czajkowski and L. Daynés. Multitasking Without Compromise: a Virtual Machine Evolution. *ACM SIGPLAN Notices*, 36(11):125–138, 2001.

[Chi07] David Chisnall. *The Definitive Guide to the Xen Hypervisor*. Open Source Software Development Series. Prentice Hall, 2007.

[CKL96] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding Confusion in Metacircularity: The Meta-Helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.

[DGL+07] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and Exploiting Change with Changeboxes. In *ICDL 2007*, pages 25–49. ACM Digital Library, 2007.

[DSD08] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The Meta in Meta-object Architectures. In *TOOLS EUROPE'08*, volume 11 of *LNBIP*, pages 218–237. Springer-Verlag, 2008.

[HN05] Michael Hicks and Scott Nettles. Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems*, 27(6):1049–1096, nov 2005.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *OOPSLA'97*, 1997.

[JVM] Sun Microsystems, Inc. JVM Profiler Interface (JVMPI).

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[LB98] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of OOPSLA '98*, pages 36–44, 1998.

[Mae87] Pattie Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.

[PDBF11] Guillermo Polito, Stéphane Ducasse, Noury Bouraqadi, and Luc Fabresse. Extended results of Tornado: A Run-Fail-Grow Approach for Dynamic Application Tayloring. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[PDF+14] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Benjamin van Ryseghem. Bootstrapping Reflective Systems: The Case of Pharo. *Science of Computer Programming*, 2014.

[PDFB13] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Noury Bouraqadi. Virtual Smalltalk Images: Model and Applications. In *IWST'13*, 2013.

[PVH14] Luís Pina, Luís Veiga, and Mickael Hicks. Rubah: DSU for Java on a stock JVM. In *OOPSLA'14, ACM SIGPLAN Notices*, 2014.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behavior. In *ECOOP'03*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[SHM09] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic Software Updates: A VM-centric Approach. *SIGPLAN Not.*, 44(6):1–12, June 2009.

[Smi84] Brian Cantwell Smith. Reflection and Semantics in Lisp. In *POPL'84*, pages 23–3, 1984.

[VEBD07] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates. *Transactions On Software Engineering*, 33(12), 2007.

[WWS10] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. Dynamic code evolution for Java. PPPJ '10, 2010.

[Zer12] ZeroTurnAround. What Developers Want: The End of Application Redeployes. http://files.zeroturnaround.com/pdf/JRebelWhitePaper2012-1.pdf, 2012.