IWST 2013

Program of the 5th edition of the International Workshop on Smalltalk Technologies

In conjunction with the

21th International Smalltalk Joint Conference

Annecy, september 2013

IWST, standing for International Workshop on Smalltalk Technologies, is a European Smalltalk User Group (ESUG) Conference joint event.

IWST was launched in 2009, in Brest, during the 17th ESUG Conference. The second edition took place in Barcelona, the third edition in Edinburgh in 2011 and the fourth edition was held in 2012 at the heart of historic Ghent. IWST 2013 is the fifth edition, and takes place in highly touristic Annecy.

ESUG gathers groups of professionals and hobbyists who share an interest in the Smalltalk programming languages and related technologies.

IWST was set up from a will to promote research activities - namely academic creative work undertaken on smalltalk use, and more generally on object technologies - apart from the ESUG main event.

The goal of the workshop is to create a forum around advances or experience in Smalltalk. IWST contributes to triggering discussions and exchanges of ideas. Contributions are welcome on all aspects, theoretical as well as practical, of Smalltalk related topics such as:

- Aspect-oriented programming,
- Meta-programming,
- Frameworks,
- Interaction with other languages,
- Implementation,
- New dialects or languages implemented in Smalltalk,
- Tools,
- Meta-modeling,
- Design patterns,
- Experience reports

This edition is an obvious milestone in the IWST's history; five years of existence demonstrates the viability of this event and the interest of the community. This edition is also the first one to receive an ACM SIGPLAN in-cooperation label.

Loïc Lagadec and Alain Plantec, co chairs

Program Committee

Loic Lagadec (chair)	LabSTICC UMR 6285 / MOCS - ENSTA Bretagne
Alain Plantec (chair)	LabSTICC UMR 6285 / MOCS - Universite de Bretagne Occidentale
Etien Anne	LIFL - universite Lille 1
Gabriela Arevalo	Facultad de Ingenieria - Universidad Austral
Alexandre Bergel	University of Chile
Johan Fabry	PLEIAD, University of Chile
Tudor Girba	CompuGroup Medical Schweiz
Mickaël Kerboeuf	LabSticc, Universite de Bretagne Occidentale
Mariano M.Peck	Ecole des Mines de Douai
Jorge Ressia	SCG, University of Bern
Bastian Steinert	HPI, Software Architecture Group
Hervé Verjus	University of Savoie
Erwan Wernli	SCG, University of Bern
Hernan Wilkinson	10Pines

Table of Contents

Full papers 9

Virtual Smalltalk Images: Model and Applications 11
Guillermo Polito, Stéphane Ducasse, Luc Fabresse and Noury Bouraqadi

Towards a flexible Pharo Compiler 27
Clement Bera and Marcus Denker

Early exploring design alternatives of smart sensor software with Model of Computation implemented with actors 37
Jean-Philippe Schneider, Zoé Drey and Jean-Christophe Le Lann

Representing Code History with Development Environment Events 45
Martín Dias, Damien Cassou and Stéphane Ducasse

Language-side Foreign Function Interfaces with NativeBoost 53
Camillo Bruni, Luc Fabresse, Stéphane Ducasse and Igor Stasenko

Pragmatic Visualizations for Roassal: a Florilegium 65
Mathieu Dehouck, Stéphane Ducasse, Usman Bhatti and Alexandre Bergel

Short papers 71

Identifying Equivalent Objects to Reduce Memory Consumption 73
Alejandro Infante, Juan Pablo Sandoval Alcocer and Alexandre Bergel

Part I
Full papers

The goal of the workshop is to create a forum around advances or experience in Smalltalk and to trigger discussions and exchanges of ideas. Participants are invited to submit research articles.
Full papers are long research papers with description of experiments and of research results.

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. The causal connections and meta-circularities makes difficult to change core parts of the system[DSD08]. For example, the array iteration method `Array»do:` is used by both user applications (the base level) and system infrastructure such as the compiler or debugger (the meta level). This method presents a causal connection in the sense that it is used by the tools in the process of changing/recompiling itself. Because of this causal connection, breaking such a method impacts not only in the final user code, but also on the libraries and tools that are essential in the system, causing the system to crash.

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 manipula-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IWST'13 September 9-13, 2013, Annecy, France
Copyright © 2013 ACM [to be supplied]. . . $10.00

tion of processes can be done only from an active process and thus, there is no possibility to activate directly a process from the language.

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: System Recovery

Working in an image based environment implies that our subject of work are the objects inside it instead of source code files. Every change in the system is expressed in terms of side-effects which are directly applied on its objects. Direct object manipulation provides as main advantage an immediate feedback to the user of the system.

However, manipulating the same image over and over again may leave it in a corrupt state, emerging when an image does suddenly not start. In such cases, all the information related to previous work sessions stays stored in a binary format inside the image file, including both application data (living domain objects) and code (methods and classes written during development). The recovery of all this information from a failing image is a tedious task, without a conclusive solution.

A typical example of corrupting an image is the wrong manipulation of the Pharo startup mechanism. The Pharo startup mechanism 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 system changes log appears as a first solution for system recovery. 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. This log allows the user to recover application code written between sessions, but not the recovery of application data stored inside the image.

Another ad-hoc solution that appeared to solve such a problem is to run the failing image with the virtual machine in debug mode. When debugging the system through the 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, easily accessible, and allow both recover application code and data.

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.

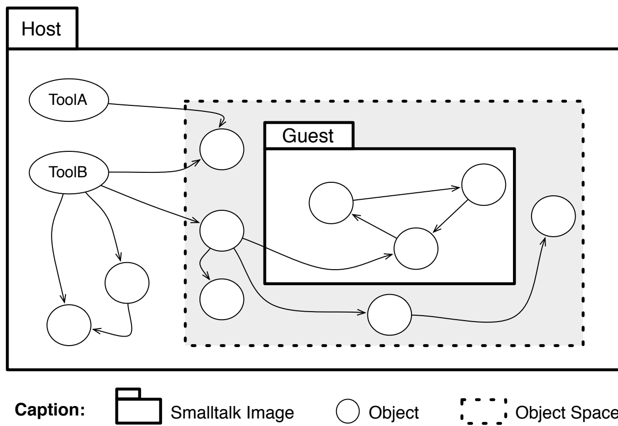


Figure 1. A host image contains an object space, represented as the region enclosed by the dotted line; the object space contains a guest Smalltalk image with its own object graph; the membrane is the gray region between the guest image and the dotted line; the tools inside the host interact with the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

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 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.

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:

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

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.

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

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. 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 ob-

jects 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.

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 4.5 and 4.6, 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 Pharo current infrastructure

To implement Oz we had to understand and the Pharo infrastructure (virtual machine and libraries), to transform it from a single-image to a multi-image solution. We describe the elements that we consider as key to understand our solution.

The special objects array. Pharo virtual machine holds the state of the image that is currently running into a *special objects array* object. The special objects array is a simple array object referencing special objects the virtual machines accesses and manipulates directly. For example, it references objects such as the boolean and numeric classes or the nil, true and false instances. Some elements inside the special objects array are optional, and therefore, may not be found in a Pharo image. We detail the contents and semantics of the Pharo special objects array in appendix A.

Concurrency through green threads. In Pharo, only one kernel (operating system) thread is used to execute code. Pharo processes are first class objects which share the same memory space as any other object in the system. The virtual machine internally handles and schedules them. Processes scheduled using this approach are also called *green threads* . Green threads provide process schedulling without native operative system support while limiting the proper usage of modern multicore CPUs.

Particularly, the special objects array contains a process scheduler object and its corresponding process objects, implementing the green threads.

Single interpreter assumptions. The virtual machine code makes many assumptions given the fact that the system is single-image. For example, the interpreter relies on constants and static variables, forbidding the ability to run two complete separate virtual machine interpreters in the same process. In addition, many of the virtual machine plugins such as the socket plugin handle their own internal state and store it outside of the image. This way, plugins state is shared for the whole virtual machine process, and would also be shared among the virtual images.

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

4.2 Oz 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. 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.

4.3 Oz 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. This method receives as arguments the object on which the primitive will be executed, an array of arguments, and the method 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:** . It 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.

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

Figure 4. Combining the two primitives to execute a primitive on a guest 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 OrderedCollection, 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.

4.4 Oz Process Manipulation and Scheduling

Processes inside an object space are first class objects as well as the ones inside Pharo. They 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. Suspending a process consists in putting the process in the corresponding suspension list of its process scheduler. The ProcessMirror and the ProcessSchedulerMirror handle the schedulling 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.

4.5 Oz Context Switch between Images

An object space has, as well as the host image, its own special objects array. Thus, for consistency, the execution of a piece of code inside an object space must use the corresponding special objects. For example, when evaluating the expression 'someObject isNil' inside an object space, the

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'**.

object referenced by the variable someObject must be compared against the nil object of the executing object space. We modified the virtual machine to be able to perform a context switch between the host image 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.

4.6 Creating an Oz 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 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.

4.7 Oz 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 (cf. Section 4.1)

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 (cf. Section 4.1). 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.

```
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 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 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.

```
((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.

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.

4.8 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 4.2). 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.,*

```
primitiveLoadImage
    | headerlength bytesRead newImageStart rootOffset oldBaseAddress 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

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.

```
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

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 is, however, work 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 4.2 and 4.8, 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 4.5), a Pharo virtual machine has single threaded execution with green threads (cf. Section 4.1). In our implementation, their usage allowed us to reuse the current virtual machine schedulling. 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.

In Kansas [SWU97], Smith et al. present a similar solution to the emergency kernel (cf. Section 5) for a collaborative environment based on Self [US07]. Smith et al. classify errors in three different categories: *benign* errors are the ones the user can solve by itself by using the typical debugging tools in the main system, *fatal* errors are those ones that prevent the system to continue, they lead to a system crash, and finally, a third category of errors that makes the system unusable from itself but does not cause a system crash. These last errors are trapped and solved in a separate alive environment, equals to Kansas, but called Oz, which does not fully share the same code base as the broken system. Once the problem is solved, users leave Oz and return to Kansas to continue their work. While Kansas makes focus on collaborative work, it is not addressed in the paper which level of isolation exist between Kansas and Oz, and what they share or not. In our work, both the host and guest images have each one their own and separate kernel, allowing to safely make changes into the guest image from the host.

The Squeak interpreter simulator [IKM+97, Mir11] was born as a project to enable the development of the Squeak image and virtual machine from Squeak itself. With the interpreter simulator, a Squeak virtual machine is programmed using Squeak objects. The simulator reifies virtual machine related concepts such as the object memory, execution stack, interpreter and process schedulling. Thus, the interpreter simulator allows to load a smalltalk image inside an object

memory instance and manipulate it freely from the host image. Regarding the internal details, the host virtual machine interprets the object memory instance as a single byte object. The host garbage collector does not traverse the graph inside the simulator object memory avoiding the problems of sharing the object memory as in Oz. The interpreter simulator has its own reference to the special objects array inside its object memory, for what no virtual machine changes are needed. From the external point of view, the interpreter simulator does not encapsulate properly the objects inside the object memory nor provides a high-level API for their manipulation as the membrane present in Oz.

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.

[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 *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997.

[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.

[Mir11] Eliot Miranda. The cog smalltalk virtual machine. In *Proceedings of VMIL 2011*, 2011.

[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.

[SWU97] Randall B. Smith, Mario Wolczko, and David Ungar. From kansas to oz: collaborative debugging when a shared world breaks. *Commun. ACM*, 40(4):72–78, April 1997.

[US07] David Ungar and Randall B. Smith. Self. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 9–1–9–50, New York, NY, USA, 2007. ACM.

[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 schedulling 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.

Towards a flexible Pharo Compiler

Clément Béra
RMOD - INRIA Lille Nord Europe
clement.bera@inria.fr

Marcus Denker
RMOD - INRIA Lille Nord Europe
marcus.denker@inria.fr

Abstract

The Pharo Smalltalk-inspired language and environment started its development with a codebase that can be traced back to the original Smalltalk-80 release from 1983. Over the last years, Pharo has been used as the basis of many research projects. Often these experiments needed changes related to the compiler infrastructure. However, they did not use the existing compiler and instead implemented their own experimental solutions. This shows that despite being an impressive achievement considering its age of over 35 years, the compiler infrastructure needs to be improved.

We identify three problems: (i) The architecture is not reusable, (ii) compiler can not be parametrized and (iii) the mapping between source code and bytecode is overly complex.

Solving these problems will not only help researchers to develop new language features, but also the enhanced power of the infrastructure allows many tools and frameworks to be built that are important even for day-to-day development, such as debuggers and code transformation tools.

In this paper we discuss the three problems, show how these are solved with a new Compiler model. We present an implementation, Opal, and show how Opal is used as the bases for many important tools for the everyday development of Pharo 3.

1. Introduction

A lot of research has been done with Pharo and Squeak in the past. Examples are Bytecode Manipulation with Bytesurgeon [DDT06], Advanced Reflection [DDLM07, RDT08, DSD08], Typesystems [HDN09], Transactional Memory [RN09] or Omniscient Debuggers [HDD06, LGN08].

All these research experiments implemented by changing the compiler of Pharo/Squeak, and sometimes combined with virtual machine(VM)-level changes. In contrast to VM-level changes, compiler based experiments have many advantages: the compiler is implemented in Smalltalk, therefore the standard tools and debugging infrastructure can be used. In addition, the models realized in Smalltalk tend to hide technical and low level details compared to an implementation at VM-level.

One of the reasons why the Pharo Project was started originally is the idea to create a feedback loop for the development of the system itself: we revisit successful research results and integrate them back into the system. This way, future researchers can build on top of the results of prior work instead of always starting from scratch.

Opal is the compiler infrastructure used for past research experiments. The code-base has been used in experiments over the years.

The compiler framework described in this paper is the result of revisiting the experimental code with the result of a compiler that is stable and clean to be integrated in Pharo 3 with the goal of removing the old compiler in Pharo 4.

1.1 The Smalltalk Compiler

In a traditional Smalltalk system, Smalltalk code (text) is compiled to bytecode. This compilation happens on a per method basis when the programmer saves an edited method.

The Smalltalk bytecode is a traditional stack-based bytecode, the bytecode set of Pharo is taken from Squeak and therefore an extension of the original Smalltalk 80 bytecode with extensions to support block closures as implemented by the Cog VM [Mir11].

The Smalltalk bytecode set provides some ways for the compiler to optimize code statically: Loops and conditionals are compiled to jump instructions and conditions.

As the traditional Smalltalk system provides a decompiler from bytecode to text, no other optimizations are done. The goal is to be able to re-create the original source from bytecode, which would be impossible in the presence of any serious optimizations.

As such, Opal right now too compiles to exactly the same bytecode as the old compiler.

It should be noted that in a modern Smalltalk VM, there is a second compiler, a so-called JIT compiler, in the VM that compiles the bytecode to native code. This paper is not concerned at all with this VM-level compiler.

[Copyright notice will appear here once 'preprint' option is removed.]

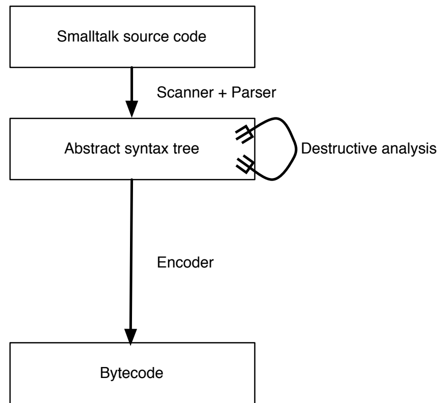


Figure 1. A representation of the old compiler toolchain

1.2 Smalltalk Language Tools

Besides the compiler, many IDE-level tools in Smalltalk reason about code. In the following we give a short overview.

Refactoring Engine. The prime example is the Refactoring Engine [RBJ97] which is the basis of all transformations related to refactoring. As the original AST of the Smalltalk compiler was not designed for transformations, the RB implements its own parser and AST.

Syntax Highlighting. As any modern IDE, Pharo supports syntax highlighting in real time while the developer types. Syntax highlighting is implemented using its own parser and representation of code, not reusing any parts of the compiler or the refactoring engine.

Debugger. The debugger is a central tool for the Smalltalk developer. Often development happens right in the debugger. To support highlighting of the execution when stepping, the debugger needs to have a mapping from bytecode to source code. This mapping can only be provided by the compiler, making the debugger reliant on the compiler in a non-trivial way.

In this paper we analyse the problems that the old compiler framework poses. We identify in Section 2 the following problems:

1. The architecture is not reusable,
2. The compilation can not be parametrized,
3. The mapping between source code and bytecode is overly complex.

We present a model for a new compiler (Section 3) to solve these problems. After we discuss implementation details (Section 4), we validate the new infrastructure by showing benchmarks and the tools that are build on top of Opal (Section 5). After a short discussion of related work (Section 6) we conclude with an overview of possible future work in Section 7.

2. Problems of the Current Compiler

Pharo uses nowadays a direct child of the original Smalltalk-80 compiler. Despite being an impressive piece of work for the eighties, the architecture shows its age. As a result, the compiler framework (scanner, parser, AST) are not used by other parts of the system. For example, the refactoring engine uses its own AST.

Reusable architecture. There are modularity problems in all the levels of the compilation toolchain. At the AST-level, Pharo uses different AST implementations. To be consistent (and for maintainability purposes), there should be only one AST for the whole system, not one AST per feature.

Then, on the semantic analysis-level, another problem is raised. The AST is dependent on the compiler to the point that the semantic analysis has side-effects and modifies the AST. After code-generation, the AST is therefore only usable by the compiler. Again, the semantic analysis should be reused in the system and implemented only once.

Lastly, at bytecode-level, no intermediate representation exists in the compiler. Therefore, the existing compiler back-end can not be used elsewhere.

Source code mapping. Another issue is the complexity of the debugging features. According to the current bytecode set of Squeak and Pharo, the bytecode representation is not aware of the names of the temporary variables, but only about their indexes. The bytecode representation does also not know about the highlighting range. To be able to get the temporary variable names and the highlighting range, we need to implement a mapping between the bytecode and the AST. This mapping is complex, especially for blocks and even more so for inlined blocks.

Parametrized compilation. Pharo developers would like to compile certain parts of the system differently, reaching from a bunch of methods up to a larger set of classes. For example, with the old compiler a set of classes can not be recompiled without automatic inlining of conditional messages. Another parametrization that would be interesting is to be able to plug in different classes for the compilation passes like parsing, semantic analysis or code generation.

Problem Summary. How can we have a flexible compiler with reusable and high-level intermediate representations?

The Opal compiler offers a solution to these problems. The flexibility comes from its pluggable architecture: you can easily change the object managing a part of the compilation chain. Opal relies on reusable representations as the RB AST or the bytecode-level intermediate representation (IR).

3. Opal: A Layered Compiler Chain

In this section we present the design of Opal from a high-level point of view.

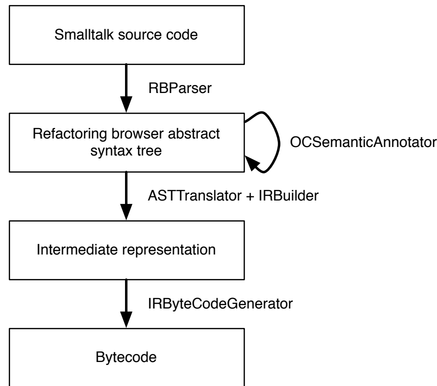


Figure 2. The Opal compilation toolchain's four stages

3.1 The Opal Model

Explicit compilation passes. As shown in Figure 2, the smalltalk code is parsed with RBParser to the RB AST. Then the OCSemanticAnalyser annotates it. The ASTTranslator visits it, building the IR with IRBuilder. Lastly, the IRByteCodeGenerator generates bytecode out of the IR.

3.2 Annotated RB AST: a Reusable Code Representation

Instead of creating a whole new representation, Opal reuses the AST from the refactoring browser. In addition, the semantic analysis does not change the AST, but only annotates it. This guarantees the reusability of the representation.

On the figure Figure 3, we can see the class diagram of the RB AST. All nodes inherit from the same superclass RBProgramNode. This way, they all have two main states: properties, which is a dictionary for annotations and parent, which is a back pointer to the outer node.

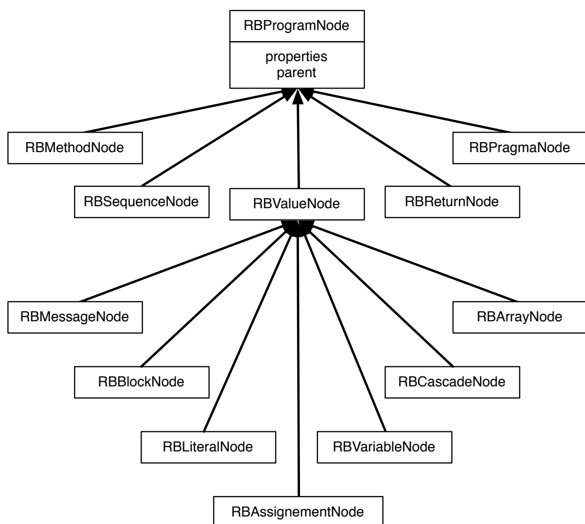


Figure 3. The refactoring browser AST class diagram

3.3 Compilation Context

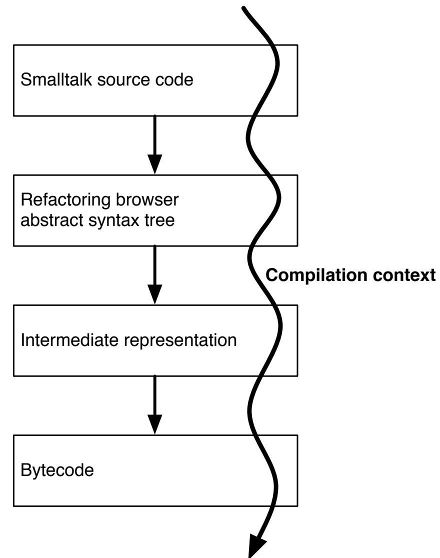


Figure 4. The compilation context in Opal compilation toolchain

When compiling a method to bytecode, we need to pass some objects along the compilation chain. For example, the old compiler used to pass:

- the requestor: this object corresponds to the UI element that holds the text to compile. This is needed because it permits for example to write error messages directly in the morph text instead of raising an error. For example, if we compile [:y | [:y | y]], we will get [:y | [Name already defined ->:y | y]] instead of an error.
- the failBlock: is executed when the compilation fails in interactive mode. Usually this happens because the source code was incorrect and an error was raised.

All this information is needed. But the issue with this approach is that it requires to always pass along these object through the whole compilation toolchain. The resulting methods with excessive numbers of arguments are hard to maintain and not nice to use. For example, we have in the old compiler:

```
Parser>>#parse:class:noPattern:context:notifying:ifFail:
Compiler>>#compileNoPattern:in:context:notifying:ifFail:
```

To increase the modularity of Opal, we needed to add even more arguments, most of them being booleans. We decided to add the CompilationContext object. This object holds all these arguments and in general all information that is of interest in later phases of the compiler. As Figure 4 shows, the context is passed through to the whole compilation chain.

3.4 IR: An Explicit Intermediate Representation

We discuss the intermediate representation (IR) of the Opal Compiler. The following shows the class hierarchy of the IR:

```
IRInstruction
    IRAccess
        IRInstVarAccess
        IRLiteralVariableAccess
        IRReceiverAccess
        IRTempAccess
            IRRemoteTempAccess
        IRThisContextAccess
    IRJump
        IRJumpIf
        IRPushClosureCopy
    IRPop
    IRPushArray
    IRPushDup
    IRPushLiteral
    IRReturn
        IRBlockReturnTop
    IRSend
    IRTempVector
IRMethod
IRSequence
```

This intermediate representation is modeling the bytecode yet abstracts away from details. It forms a Control Flow Graph (CFG). IRInstructions are forming basic blocks using IRSequence, these sequences are concatenated by the last instruction which is an IRJump.

Opal has an explicit low-level representation for three main reasons. Firstly, it gives to the user the possibility to easily transform the bytecode. Secondly, it simplifies a lot the debugging capabilities implementation of the system, as explained in Section 4.2. Lastly, this representation provides an abstraction over the bytecode, letting the whole compilation chain of Opal independent of details of the bytecode representation. A dedicated backend visits the IR (IRBytecodeGenerator, as shown in Figure 2).

3.5 Debugging Features

The AST with semantic analysis and its IR, provide the basis to map between all the representations. For example, mapping between bytecode offset and text. Details are explained in Section 4.2.2.

4. Opal Implementation Details

In this Section, we will present some of the implementation details of Opal. We will discuss two aspects: first the compilation context and how it enables Opal to be parametrizable and pluggable. Second we discuss in detail the infrastructure implemented for mapping text with AST, IR and low-level offsets in compiled code.

4.1 Compilation Context

The compilation context is an object that holds state that is of interest to later passes done by the compiler. The class definition is shown here:

```
Object subclass: #CompilationContext
  instanceVariableNames: 'requestor failBlock noPattern class
                category logged interactive options
              environment parserClass semanticAnalyzerClass
                astTranslatorClass bytecodeGeneratorClass'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'OpalCompiler-Core-FrontEnd'
```

The instance variables all help to make Opal more customizable and to change the compilation chain. We present them one by one.

Basic data.

requestor : this object corresponds to the user interface element that holds the text to compile.

failBlock : this block is executed when the compilation fails.

noPattern : this boolean tells if the text to compile starts with the method body or by the method selector.

class : the class of the compiled object to know how to compile accesses to instance variables.

category : the category where the compiled method should be located.

logged : will the new compiledMethod creation be logged in the changes file.

interactive : this compilation happens in interactive mode (Warnings are raised and stops the compilation) or in non interactive mode (Warnings are shown on console logging and does not stop the compilation).

environment : points to the Smalltalk environment (usually an instance of Smalltalk image) where you compile. This is used for example for remote compilation.

It should be noted that the current API follows to some extend the old implementation to make it easier to move the whole system to use Opal. In a second step, we plan to revisit the compiler API to simplify it.

Compiler options. The Opal compiler proposes options. A programmer can specify them either on a per class basis by overriding the compiler method or on method basis with a pragma. These options are passed with the compilation context through all stages of the compiler and can be read and reacted upon on any level. The first set of options concern optimizations of control-structures and loops:

- optionInlineIf
  optionInlineIfNil
  optionInlineAndOr
  optionInlineWhile

optionInlineToDo
optionInlineCase

This set of options controls automatic inlining of some message, such as ifTrue: and and:. There is no option to not optimize class as in Pharo the class is always a message send.

- optionLongIvarAccessBytecodes

  This option forces the compiler to generate long bytecodes for accessing instance variables. It is used for all classes related to MethodContext to support c-stack to Smalltalk stack mapping.

Compiler Plugins. In some cases it can be useful to replace parts of the compilation chain. Therefore the programmer can change which class is used for each compilation phase. One can redefine:

parserClass: changes the class that parses Smalltalk code and returns an RB AST. For instance a scannerless Parser could be used instead.

semanticAnalyzerClass: changes the class that is performing the semantic analysis on the RB AST nodes.

astTranslatorClass: changes who translates the RB AST to Opal IR intermediate representation.

bytecodeGeneratorClass: changes the generator class used to create bytecode from the Intermediate representation. This is especially useful when experimenting with new bytecode sets.

4.2 Opal Debugging Features

A central feature of any Smalltalk is its advanced debugger. To be able to implement a stepping debugger, there needs to be a mapping between the program counter on the bytecode-level and the text that the programmer wrote. In addition, we need to be able to access temporary variables by name.

4.2.1 Debugger Highlighting

The AST nodes know their source intervals as they are recorded when parsing. Then, each IR instruction knows the AST node that generated them. Lastly, from each bytecode you can get the IR quite easily, as each IR node knows the index of the corresponding generated bytecode.

Therefore the mapping can be done easily at AST/IR-level. Figure 5 shows a complete example of mapping an offset in the bytecode to the source. The idea is to generate the AST and IR from the compiled method, then to map from bytecode to IR to AST and lastly to the source interval. So the mapping does not build up special data structures, but instead relies on annotations on the AST and the IR that are generated by the compiler. It should be noted that we need to do a full compilation of the method from the sources to get the correct AST and IR mapping.

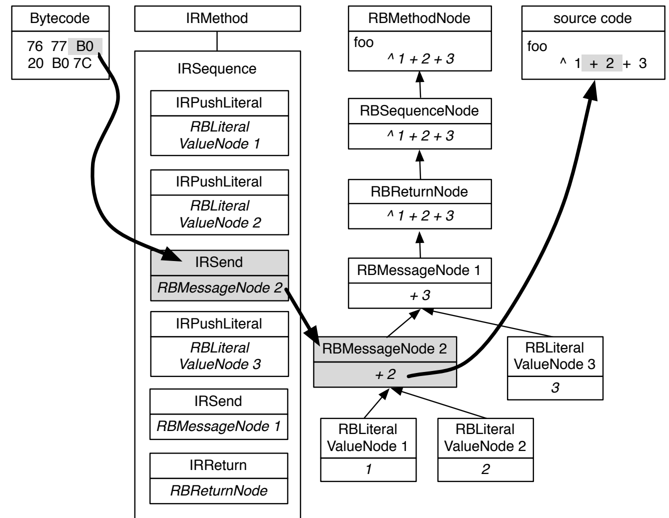


Figure 5. Bytecode to source code highlighting

4.2.2 Temporary Name Mapping

Temporary variables are accessed through an offset. For a simple temporary variable, the runtime representation of the method (called context) uses this offset to access the value of the variable. Moreover, Pharo supports blocks (commonly called closures). As these closures can live longer than their enclosing context, they also need their own runtime representation (context). Variables shared between closures and their enclosing contexts are stored in a heap allocated array named tempVector. Here is an example of a method with a block and shared variables:

```
SomeRandomClass>>foo
    | temp1 temp2 |
    temp1 := #bar.
    [ | closureTemp |
    closureTemp := #baz.
    temp2 := closureTemp ] value.
    ˆ temp2
```

We see in Figure 6 that the offset of the temporary variable temp1 is 2 in the method context. temp2, being shared by both the block and the method context, is stored in a temp vector. So its offset, while being accessed from the block or the method context, is 1 to reach the temp vector, then 1 which correspond to the offset in the temp vector.

To speed up the execution, one optimization is implemented. The temporary variables that are read-only in the closure are not stored on the external array but passed to the block context similarly to an argument.

Temporary name mapping is the correspondence between these offsets and the variable name. This mapping can be complex: in Smalltalk, one can have several nested blocks in a method and in each block there might be some read-only

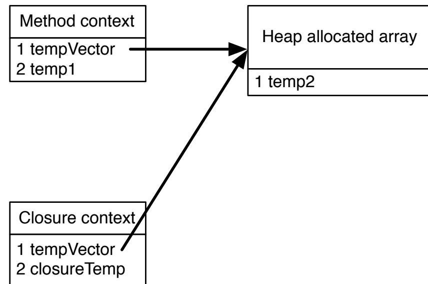


Figure 6. Runtime temporary variable storage

or written temporaries. This mapping is used for debugging (debugger and inspectors).

As an example for inspecting a context, the following code presents a simple temp access:

```
SomeRandomClass>>example
    | temp |
    temp := #bar.
    ^ temp
```

The offset of the temporary variable temp is 1. Therefore, when we inspect a context, the result is just an inspector on an object that has offsets, as we see in Figure 7.

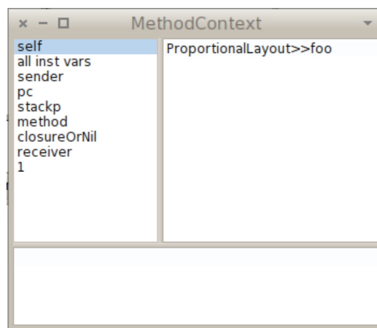


Figure 7. A basic inspector on a context

Programmers do not want to debug contexts with indexes of temporaries, but with temporary names (Figure 8).

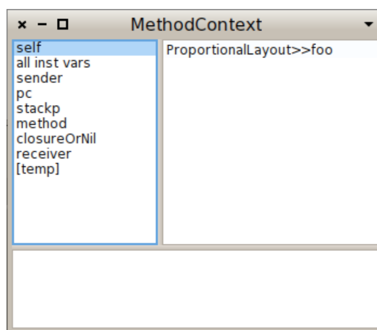


Figure 8. A specific inspector on a context

Temporary Names with Opal. Similarly to the highlighting implementation, we want to reify all needed information at the AST-level. We leverage the information added to the AST in the semantic analysis phase of the compiler. This includes objects modeling temporary names and offsets as well as so called scopes for each method or block AST node. These scopes store defined variables and are used for mapping names to the objects describing the variables. Each node can therefore access the corresponding scope due to the parent relationship in the AST.

In figure 8, we can click on the [temp] entry. This displays on the right panel the value. To do this, the context needs to know to which offset correspond the temporary name temp. The context knows in this case for the temporary variable offset the associated value. It knows that offset 1 is associated to the value #bar. The context, being a representation of the method, can access its corresponding method AST node. The node then provides through scopes the offset information about the variable. Of course, this simple example becomes exponentially complex when we have multiple closures inside the methods with shared variables that need to be stored in temp vectors.

The temporary name mapping, now working on AST-level, works the same way for optimized blocks (to:do:, ifTrue:ifFalse:, ifNil:ifNotNil:, and:, or:) and non optimized blocks.

5. Validation

We have discussed three problems of the old compiler infrastructure in Section 2. We will show in the following how the new design of Opal solves the problems.

To show that the resulting implementation is usable in practice, we show benchmarks of compilations speed.

5.1 Problem 1: Reusable Architecture

Pharo 3 is moving many subsystems to use parts of the Opal Infrastructure. We highlight some of them.

AST interpreter. We implemented a complete AST interpreter on top of the annotated AST. The AST interpreter is written in Pharo, permitting to prove the reusability of the annotated AST. The interpreter is able to interpret all the tests of the Pharo Kernel, and they all pass.

Hazelnut. In the case of Hazelnut [CDF+11], a bootstrap of a new Pharo image from sources, Guillermo Polito uses Opal for the flexible features of the semantic analysis tool. As he needs to compile some smalltalk code not for the current Pharo environment, but for a remote image, he needs to change the way variables are resolved. He implemented his own semantic analyzer, with different bindings for variables, replacing the one from Opal. Lastly, he used Opal to compile the methods, with a different semantic analyzer that the default Opal one.

Metalinks. Reflectivity is a tool to annotate AST nodes with Metalinks. A Metalink annotates an AST node. Metalinks can be executed before a node, instead of a node, after a node, if the execution of a node yields an error or if the execution of a node yields no error. Once the AST of a method is annotated with some metalinks, a method wrapper replaces the method to handle the compilation of an expanded version of the AST that takes metalink into account and then installs the resulting method. This tool rewrites the AST nodes according to their metalinks. The new AST is recompiled thanks to Opal.

Class builder. The new Pharo class builder [VBLN11, VSW11] avoids class hierarchy recompilation when adding instance variables in the superclass chain. On the low-level, this means that when adding an instance variable, some existing instance variables have to shift the instance variable offsets. This is done, in the case of a compilation with Opal, with IR transformations.

Smart suggestions. While we are coding we usually want to apply actions on the current edited element. For example if we have selected a variable we may want to rename it. To do this, IDEs often have large menus, including the correct feature, usually with lot of options that do not apply to the selected element.

Smart suggestions show only the options that you can apply to the selected AST node. We use the current AST to do this through RBParser»#parseFaultyMethod: and the Opal compiler semantic analysis. The best AST node corresponding to the selected text is calculated. Then the available suggestions are provided. The Opal semantic analysis provides the nature of a variable: temporary, instance or class to refine the suggestions.

Node navigation. Sometimes while browsing code we think in programming terms instead of text. For example we think in a message send or a statement instead of word, spaces or symbols. The idea is to use context information and let the programmer navigate the code thinking in those terms. In order to do this we find the best AST node and offer navigations in different directions:

Parent: The node that contains the selected one. For example if we have the code 'aNumber between: anotherNumber' and we are selecting the variable anotherNumber if we navigate to the parent the IDE highlights the message.

Sibling: The node in the same level that the selected. For example in a temporary variables definition: '| one two three |' if we are in the variable one we can navigate to the variable definitions two or three.

Child: Node contained by the selected. For example if we in a message send: 'aNumber between: anotherNumber' we will go the parameter anotherNumber.

Syntax highlighting as you type. We want to color the code we are writing using all the available information, in order to be able to select the scope where we are or to show associated information for a specific piece of code. To do that we use the AST and the semantic analysis (we need the semantic analysis because we want to show different kinds of variables with different colors, like undeclared variables), through the RBParser»#parseFaultyMethod: and RBParser»#parseFaultyExpression: to obtain the AST representation. The implementation is simple because we can define a new visitor defining the coloring algorithm. Once we define the coloring from each syntax representation we just visit the tree.

5.2 Problem 2: Source Mapping

To validate the new implementation of source code mapping, we use it as the basis for the debugger. Instead of implementing a dedicated map for the debugger (DebuggerMethodMap), we forward all requests to the AST (which is cached by the system). To test the performance, we perform a simple benchmark. We print the error message that prints the whole stack. This prints for each stack frame all the variables:

```
String streamContents: [ :str | thisContext errorReportOn: str ]
```

We execute this code in Pharo 1.4 which had a limited caching for the debugger map, as well as in Pharo 3 for both the old and the new compiler:

Pharo 1.4 (old compiler, simple cache)	11.7 per second
Pharo 2 (old compiler, no caching)	6.13 per second
Pharo 3 (new compiler, AST cache)	51.7 per second

As we can see, the Opal strategy of using the annotated AST structure is faster than even the old compiler using the simple debugger map cache.

5.3 Problem 3: Parametrized Compilation

No automatic inlining. To prove the flexibility of the Opal compiler, a good example is not to inline some messages in some classes or methods of the system. As an example, we can advice the compiler to not inline the if statement with a pragma. With the old compiler, the if condition, sent on a non boolean object, as a symbol, raises a NonBooleanReceiver error. On the opposite, with Opal compiler, the if condition, also sent on a symbol, raises a MessageNotUnderstood error.

```
MyClass>>foo
    <compilerOptions: - optionInlineIf>
    ^ #myNonBooleanObject ifTrue: [ 1 ] ifFalse: [ 0 ]

MyClass new foo
"With the old compiler, raises a runtime error NonBooleanReceiver"
"With Opal compiler, raises a MessageNotUnderstood error"
```

This aspect is useful for different reasons. For example, researchers might want to experiment with new boolean implementations. They could want a boolean system with true,

false and maybe. In this case, they needed to implement the new boolean messages with different names, creating a non readable smalltalk code, because they were not able to use the selector ifTrue:, ifFalse: or other optimized constructs. Other examples are proxies for booleans, symbolically executing code for type information and others.

The downside is that the non-optimized code is just produced for the methods or classes explicitly compiled with this option. To scale this to all code of the system, we recompile non-optimized code when a mustBeBoolean: exception is raised. The nice property of this solution is that it only slows down the case where an optimized construct is called on a non-Boolean object.

5.4 Compilation Benchmarks

Even though the Opal model is introducing an Intermedia Representation (IR) and using multiple visitor passes, the resulting compiler is comparable in speed.

The benchmarks were run on a MacBook pro, on Mac OS X (Version 10.8). The machine had 8 Gb of RAM (1600MHz DDR3) and a Intel Core i5 processor of 2.5 Ghz. The SMark framework provides a precise average time of each benchmark run including error margin.

Compilation Speed. We first compare the two compilers with regard to compilation speed when recompiling classes. This exercises the whole compiler toolchain from parser down to bytecode generation and therefore gives a real world view on compilation speed. In the following table we compare recompiling the whole image and recompiling Object class:

Recompile	Opal Compiler	Old Compiler
Object class (ms)	296.66 ± 0.98	222.9 ± 2.4
Whole image (ms)	72120 ± 189	49908 ± 240

As we can see, the factor between the compilers is around 1.4. Considering that Opal generates a reusable AST with annotations for semantic analysis and uses a dedicated IR representation for the bytecode generation, this performance is acceptable. Especially considering that we can use the low-level IR backend in cases where speed matters, as the next benchmark shows.

IR Benchmarks. The intermediate representation of Opal allows the programmer to manipulate the high-level IR instead of the low-level bytecode, AST or text. Manipulating bytecode directly is not practical due to hard coded jump offsets and the need to create new method objects if things like number of literals or the max depth of the stack changes. Using the high-level text or AST model for manipulating code can lead to performance problems, even when using the faster old compiler. An example for this is recompilations of class hierarchies when adding instance variables high up in the class hierarchy.

Opal provides the possibility to manipulate the IR representation instead. To assess the performance, we benchmark the speed of the IR backend. We show the times for

- decompiling bytecode to IR,
- a full IR based roundtrip of decompiling and regenerating bytecode,
- generating bytecode from IR (difference of the first two).

All these are done on all methods of the complete system.

BC -> IR (ms)	BC -> IR -> BC (ms)	IR -> BC (ms)
2827.2 ± 4.0	10533 ± 13	7706 ± 17

The benchmarks prove that manipulating IR is much faster than recompiling source code, both with the old or the new compiler. We can regenerate the whole bytecode of the Pharo 3 image in just 10 seconds instead of 50 seconds when recompiling with the old compiler.

This fast way to manipulate methods will be used by the new class builder when adding instance variables.

Runtime Speed. It should be noted that as the compiler generates the same bytecode, execution speed of the generated code is identical. We do therefore not provide any benchmarks.

6. Related Work

Smalltalk like languages implement a compiler from text to bytecode in Smalltalk and make it available reflectively. This is not the case with many other languages. In most languages, the compiler is a stand-alone application not visible for compiled programs. As such, all the compiler, IDE and tools are seen as distinct and sharing implementations between them is not common. In turn, compiler frameworks that enable experiments are done as external tools without the goal of replacing the main compiler of the language.

Polyglot [NCM03] is an extensible compiler framework that supports the easy creation of compilers for languages similar to Java. A newer example is JastAdd [EH07], a compiler framework for Java that is build using a special Java-like AOP language. It has seen a lot of use in recent implementations of AOP systems in Java.

All Smaltalk-like languages contain a compiler very similar to the old Compiler of Pharo. It is available in the language, but changing it is difficult. The easiest way to reuse the compiler is to copy the code and change it. And example of this is the Tweak extension of Smalltak used in Croquet [SKRR03].

7. Future Work and Conclusion

In this paper we have presented Opal, a new Compiler infrastructure for Pharo. Opal solves some problems that where found when using Pharo for numerous research prototypes: (i) The architecture is not reusable, (ii) compilation can not

be parametrized and (iii) the mapping between source code and bytecode is overly complex. As shown, Opal solves these problem by being bases on a modular design using a compilation context and keeping the mapping explicit.

We have validated Opal by presenting benchmarks and shown a number of tools and frameworks that are build using it. Opal is already used as the default compiler of Pharo 3.

There are many possible direction for future work, for example:

Compilation time optimizations. As seen in Section 5.4, Opal compiler is now 1.4 times slower than the old one. With this, Opal is already fast enough for productive use. However, we plan to conduct extensive profiling and optimization passes after the compiler has been integrated in Pharo 3.

Optimizations on IR. Currently, optimizations are done by the ASTTranslator. For example, the inlining of block for ifTrue: or whileTrue: is done by analyzing the AST. However, the AST makes it hard to analyze since there is no explicit representation of control flow. Therefore, the correct place for these optimizations would be on the IR-level as the IR is a CFG (Control Flow Graph). We plan to simplify the AST to IR translation and to move the optimizations to the IR-level.

Experiment with Opal. The flexible features of Opal permits to conduct experiment more easily. We would like to experiment statically available information for optimizations. For example, it is easy to inline message sends to globals. In addition, simple limited type inference schemes are interested to explore.

Acknowledgements

We thank Gisela Decuzzi for her work and comments about AST based tools, Jorge Ressia for his work on porting Opal to the new closure model and Anthony Hanan for creating the ClosureCompiler project years ago that was the starting point for the explorations that became Opal. We thank Stéphane Ducasse and Camillo Bruni for their reviews.

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011- IRSES MEALS.

References

[CDF+11] Gwenael Casaccio, Stéphane Ducasse, Luc Fabresse, Jean-Baptiste Arnaud, and Benjamin van Ryseghem. Bootstrapping a smalltalk. In *Proceedings of Smalltalks 2011 International Workshop*, Bernal, Buenos Aires, Argentina, 2011.

[DDLM07] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-method reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.

[DDT06] Marcus Denker, Stéphane Ducasse, and Éric Tanter. Runtime bytecode transformation for Smalltalk. *Journal of Computer Languages, Systems and Structures*, 32(2-3):125–139, July 2006.

[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.

[EH07] Torbjörn Ekman and Görel Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1-3):14–26, 2007.

[HDD06] Christoph Hofer, Marcus Denker, and Stéphane Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*, volume P-88 of *Lecture Notes in Informatics*, pages 17–32. Gesellschaft für Informatik (GI), September 2006.

[HDN09] Niklaus Haldimann, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types for a dynamic language. *Journal of Computer Languages, Systems and Structures*, 35(1):48–64, April 2009.

[LGN08] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[Mir11] Eliot Miranda. The cog smalltalk virtual machine. In *Proceedings of VMIL 2011*, 2011.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, 2003.

[RBJ97] Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[RDT08] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008.

[RN09] Lukas Renggli and Oscar Nierstrasz. Transactional memory in a dynamic language. *Journal of Computer Languages, Systems and Structures*, 35(1):21–30, April 2009.

[SKRR03] David A. Smith, Alan Kay, Andreas Raab, and David P. Reed. Croquet, a collaboration system architecture. In *Proceedings of the First Conference on Creating, Connecting and Collaborating through Computing*, pages 2–9, 2003.

[VBLN11] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot

access. In *Proceedings of 26th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*, pages 959–972, New York, NY, USA, 2011. ACM.

[VSW11] Toon Verwaest, Niko Schwarz, and Erwann Wernli. Runtime class updates using modification models. In *Proceedings of the TOOLS 2011 8th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'11)*, 2011.

Early exploring design alternatives of smart sensor software with Model of Computation implemented with actors

Jean-Philippe Schneider Zoé Drey Jean-Christophe Le Lann
UMR 6285 Lab-STICC, ENSTA Bretagne
jean-philippe.schneider@ensta-bretagne.fr

Abstract

A cabled sea floor observatory is an infrastructure composed of a network of sensors that aims to study the oceans.Such networks present two drawbacks: the large amount of acquired data prevents manual processing, and a lot of irrelevant data degrades the treatment efficiency. To smartly reject irrelevant data, filtering must happen at the source. This requires the design of complex sensors that support concurrent operations to mix acquisition, treatment and dissemination of data. Such sensors are called *smart sensors*.

Simulation using adequate models of concurrency can help in the design activity of smart sensors. In this paper, we present a framework based on Models of Computation (MoCs) to model concurrency in a smart sensor software that are implemented using actors. This framework enables to rapidly test alternatives of architecture in the early stages of the design process. Some actors implement the computation behavior, some others implement the communication behavior. This distinction promotes agility during the exploration phase. Several MoCs are investigated among which, Kahn Process Network (KPN), Communicating Sequential Process (CSP) and Synchronous Data Flow (SDF).

The framework implementation relies on Scala at first, while Smalltalk and then the Biniou framework are used to speed up the process, through alleviating the need for a software simulator and compiling to hardware platforms instead.

Keywords Actor, Model of Computation, Smalltalk, Scala

1. Introduction

In 2008 the European Council published the directive 2008/56/EC which obliges European states to check the water quality in their coastal area on the long term. This raises the interest for coastal sea floor observatories. An example of such a project is the MeDON (Marine eData Observatory Network) observatory deployed near Brest in France[14]. The usual structure of a cabled sea floor observatory is a network of sensors and one or more servers. Sensors acquire data and servers perform computations on this data. The acquisition and the computation processes can be achieved concurrently. A drawback of a cabled sea floor observatory is that it generates a huge amount of data to process and to store. Moreover, the data exhibits a lot of redundancy and noise, impeding the computation efficiency of the servers. A way to address this issue is to delegate some parts of the computation to smart sensors, as suggested by Spencer *et al.* [30].

A smart sensor consists of a sensing device that acquires some data and a microprocessor that performs computations on this data. For example, an hydrophone monitors the underwater sound level. The hydrophone can compare real-time acquired data to a computed average sound level. If the current level is higher than the average one, the hydrophone can directly switch a camera on without resorting to a server.

Delegating some of the data computation to smart sensors reduces the amount of the data to be processed by the servers. A smart sensor can perform multiple tasks such as the acquisition, the processing, and the dissemination of data. For example, a smart sensor built around an HD camera acquires images, detects shapes in these images, and broadcasts the shape classification over the network. When an image is acquired, it can be either computed by a single process or by multiple processes, depending on its size or its complexity. In the latter case, the designer has to decide (1) how to divide the input image and dispatch sub-images among processes and (2) how to recombine the results of the computations on the sub-images. Such operations require concurrency to be carefully handled between the processes of the smart sensor. To handle multiple processes, different concurrent software architectures can be envisioned; the designer has to choose the one that best fit his needs, in terms of *e.g.,* speed, memory usage, and image precision.

In this paper, we present an experimental framework to rapidly test different alternatives of concurrent software architectures. This framework relies on the actor model as described by Agha [2]. The actor model offers a simple programming structure to define concurrent applications. Our framework implements a range of existing models of concurrency and communication that provide various ways to exchange data and synchronize processes. These models, also called Models of Computation are: Kahn Process Network (KPN) [16], Communicating Sequential Process (CSP) [13] and Synchronous Data Flow (SDF) [21]. Specifically, the framework aims to help designers to:

- test the functionalities that they implemented in a smart sensor;
- simulate different alternatives of concurrent software architectures;
- adjust the alternatives until reaching a solution satisfying their needs.

A first implementation, written in Scala, serves as a baseline, allowing to meter the gains that using Smalltalk offers. The Scala language, which integrates the actor model, has the advantages of running on the Java Virtual Machine, then supports integration of existing pieces of code written in Java, and to be wide-spread in the industry.

[Copyright notice will appear here once 'preprint' option is removed.]

Smalltalk has already been used in the context of multi-agent systems[25]. The agent formalism can be used to describe networks of smart sensors, where smart sensors are viewed as autonomous entities (the agents) that communicate with each other in order to perform a mission. Smalltalk also addresses the issue of programming virtual machines that support multiple concurrency mechanisms [23].

Also Smalltalk has been used to [19] set up Biniou, a framework that describes applications as a set of concurrent processes, prior to synthesizing the application onto a reconfigurable device. This process, often referred as high-level synthesis (HLS), can either rely on global scheduling or on distributed scheduling, with either a known behavior or inter modules arbitrary synchronizations. The benefit of using Biniou is to easily stress some design options, while providing a hardware speedup compared to a pure software execution. Besides, as Biniou embeds debugging (observability, controllability) features within the generated hardware, this speed up comes at no cost in term of exploration and analysis.

Our contributions are as follows :

an actor-based framework for prototyping smart sensors We have defined a framework based on the actor model to ease the prototyping of smart sensors. This framework separates the computation concerns from the communication ones in the software design of a smart sensor.

two implementations of our framework We have experimented the underlying model of our framework in two languages: Scala and Smalltalk.

a case study of our framework We have applied our framework on a case study useful for seafloor observatories. This case study is a basic application to detect edges in an image captured by a camera.

The rest of the paper is organized as follows. Section 2 first describes some related work and summarizes the definitions used in the paper for the main concepts. Section 3 provides more details on the context and our motivations. Section 4 delves into the choices made for the implementation of our framework. Section 5 demonstrates the usage of our framework on a simplified example of Smart Sensor.

2. Related Work

The modeling of embedded systems in general and of sensors in particular has been studied in various ways. This section provides background and focus on the notion of Model of Computation (MoC) and Actors, as they both drive our modeling.

2.1 Background

SensorML [4] has been used for describing the specification of sensors with a XML format. The XML Schema of SensorML is standardized by the Open Geospatial Consortium. In SensorML a sensor has a series of attributes and may be composed of processes linked together. Robin and Botts [26] demonstrated the use of SensorML to describe chains of processes and to analyze acquired data. SensorML describes the relationships between processes but it neither provides a description of the communication nor the synchronization mechanisms between processes.

Diallo *et al.* [7] leverage MoCs to describe the communication semantics within models in the Model-Driven Engineering context. They define a modeling language called Cometa that enables to model the communication and synchronization mechanism defined by a MoC.

ThingML [9] is a Domain-Specific Language for modeling resource-constrained systems such as smart sensors. A Model Driven Engineering approach is promoted, using nested state machines to model the behavior. Several model-to-code transformations are defined and target the Java programming language, the Arduino family of board and the Atmel AVR and TI MSP chips. As the final aim of ThingML is to generate code, the designers of the system must be highly confident in their model and transformation engines.

FlowTalk [3] is a high-level programming language derived from Smalltalk to program sensors. FlowTalk identifies the use of time-consuming operations such as reading sensor values or access to the network. Code in NesC (a derivative of C) that optimizes the use of the computing resources is then generated. FlowTalk focuses on the sequential programming of the sensors. Our approach focuses on the concurrent architecture of the software of the sensor.

Several tools or language libraries implement specific MoCs. Communicating Sequential Process is implemented in the Java Programming Language with JCSP or in Scala with Communicating Scala Objects [32]. A SDF-based approach can be found in industrial tools like Scade Suite [33]. However, these tools are restricted to one Model of Computation. Since we want to explore architecture that may mix different MoCs we need a tool that implements various MoCs.

Ptolemy [8] is an analysis tool for heterogeneous systems based on Models of Computation. Ptolemy is a graphical system modeling tool. Models are executable through a thread-based Java implementation. In Ptolemy, designers can choose only one MoC to model communication for a given network of processes. In contrast, our actor-based framework allows designers to use multiple MoCs to manage different styles of communication between processes.

ForSyDe [27] is both a tool and a Model-Driven Engineering methodology for the design of system on chips (SoC) based on Models of Computation. MoCs in ForSyDe are implemented in Haskell. The proposed methodology is based on the use of modeling during the whole design cycle. The model is refined incrementally until the designer gets a model ready for implementation on a SoC.

Our approach is synergetic with these works, as we describe concurrency between processes and we make explicit their communications and synchronizations. We offer a practical tooling to implement and simulate several alternatives of architecture enabling to verify various functional scenarios for concurrent applications. The simulations enable to select the design alternatives that best suit the needs of the system.

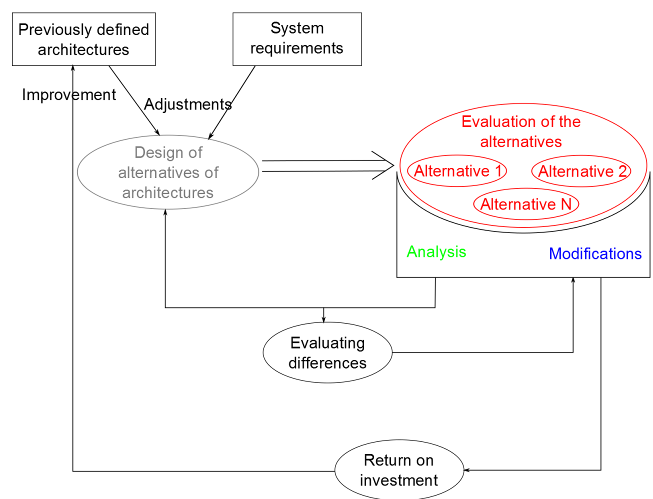


Figure 1. Overview of a design process flow of a system architecture

In Figure 1, we illustrate the design process flow of a system architecture. Our environment intends to serve as an exploration step (central left bubble) with fast, but coarse grained evaluations. Simulating architecture alternatives helps the designer to fully capture the functional requirements. Once done, architecture models are evaluated with other tools such as ForSyDe or Ptolemy to compare the results of the different analysis.

2.2 Models of Computation

A Model of Computation (MoC) for concurrent applications defines [22]:

1. the composition of the concurrent components in an application including the specification of computations inside a component;
2. the concurrency mechanisms that govern the execution of the components;
3. the communication mechanisms between the components.

MoCs are used to design embedded systems [15] such as smart sensors. They provide an abstraction of the concurrency that enables to design a system without having to take into account the implementation intricacies of concurrency on the actual system. This ability is particularly useful in the first phase of the design cycle when the final platform may not have been chosen yet.

2.3 Actor Model

The actor model is a paradigm that defines the components of a concurrent system as entities, *i.e.,* the actors, that run in parallel and communicate asynchronously with each other [2]. An actor is able to send messages to other actors and to receive messages from them. Each actor has its own buffer to receive messages. When an actor sends a message to another it is never blocked. On the contrary, when reading data from its buffer, an actor is blocked if the buffer is empty.

In some languages such as Erlang [1] or Scala [11], actors are a language feature. In other languages such as Java (e.g. Kilim [31]) or Smalltalk (Actalk [5]), actors are provided through libraries.

In a former work, we showed that each concurrent processes of a smart sensor can be made of a thread with a FIFO to store received data [29]. Our framework goes a step further in the use of actors to implement the processes of a smart sensor.

3. Context and Motivations

Unlike standard sensors, smart sensors have a microprocessor that enables to provide them with intelligence [30]. A smart sensor is able to acquire data, to perform computations on these data and to send the result of the computations over a network. A smart sensor is also able to modify its behavior according to data sent by other smart sensors. In the context of sea floor observatories, smart sensors are studied because their embedded intelligence enable them to automatically register into the sensor network of the observatory [34]. They are also able to reduce the amount of data that is sent on the network.

Because a smart sensor mixes hardware and software, faults and errors in such a system come either from software or hardware, or interactions between software and hardware. In sea floor observatories, recovering from a failure may require an on-site intervention using expensive equipments. As the underwater environment is very hostile, hardware failures due to unexpected causes have a high rate of appearance. As the software of a smart sensor is the easiest manageable part, a lot of pressure is put on the software engineers. They have to reduce the risks of pure software failures.

In order to ensure the quality of their softwares, software engineers use rigourous development methodologies supported by tools. For example, the International Council of System Engineering (INCOSE) [24] details such a system engineering methodology in its Systems Engineering Handbook [12]. In this methodology, the step of the system architecture design consists in defining different candidates of architecture and then in validating them. The validated architectures are compared to choose the one that best fits the system requirements. In order to validate the candidate architectures, simulation can be used. We are interested in the concurrent behavior of the system, the communication and synchronization. Simulation requires a way to ease the modeling of these aspects.

In small-size systems, it is tempting to develop the software of the smart sensor directly on the final platform. The software may mix portions of code at different levels of maturity. This can be the source of failures in the final system. In a complex system, it is essential to separate prototyping from final coding in order to obtain the required level of quality.

It is also required to abstract from the smart sensor platform. MoCs enable reasoning about the system at a high level, abstracting over the low-level (platform specific) details. This abstraction has the following advantages as shown in Figure 2:

- it enables to generate software for different platforms from the same model;
- it enables to perform simulation of the system on a platform agnostic simulation framework to validate functional properties;
- it ensures the coherence of the different generated software as communication and synchronization are well-defined.

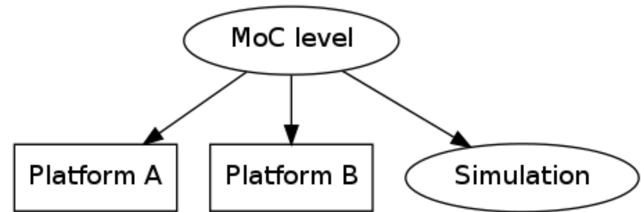


Figure 2. Benefits of using Models of Computations

To design a smart sensor, architects must identify the functionalities of the system, as well as the way data are exchanged in the system. A prototyping framework should help the designers in these tasks. Prototyping is a way for designers to discuss with the client with a concrete support. Functions of the system can be incrementally prototyped with the client.

The exchange of data among the functions of the system has a great influence on the architecture of the system. For this reason, designers need to stress the different communication mechanisms to evaluate their influence on the functions of the smart sensor.

4. Implementation Choices

Actors provide a convenient way to program a concurrent application. We have experimented an implementation of MoCs with actors. This section highlights the choices we have made during the implementation.

4.1 Framework Architecture

Choosing the appropriate communication mechanisms in a concurrent application is challenging. For example, it requires to manage deadlocks or concurrent access to shared variables.

Instead of implementing a scheduler to simulate concurrency, we rely on actors provided by the implementation language of our framework. In doing so, we focus on the implementation of different communication mechanisms defined by MoC.

A disadvantage of actors is that they only have a single FIFO to receive the messages from other actors. Karmani and Agha [18] points out that this mechanism does not guarantee the order of reception of messages by an actor. They suggest the creation of dedicated communication channels between actors. Our framework follows this proposal; each channel of communication is associated to a Model of Computation to describe how the messages should be exchanged. This solution provides modularity: by reifing channels of communication, switching MoCs with each other is facilitated and the communication and computation concerns are separated.

A naive example of synchronization of concurrent processes is illustrated by Listing 1. This code creates two processes that communicate through a *SharedQueue* in Smalltalk. Since reading is blocking and writing is not, this implicitly describes a Khan Process Network synchronization scheme.

```
|channel|
channel := SharedQueue new.
[[true] whileTrue:[channel nextPut:
        self produceData]] fork.
[[true] whileTrue:[self process:
        channel next]] fork.
```

Listing 1. A naïve implementation of Khan Process network

In order to be modular and to enforce reusability, we use the Adapter design pattern [10]. We implement a MoC as an actor that is reactive to two messages:

- Put: message sent by a writer process to send data on the channel;
- Get: message sent by a reader process to read data on the channel.

Figure 3 shows the principles of the architecture of our prototype.

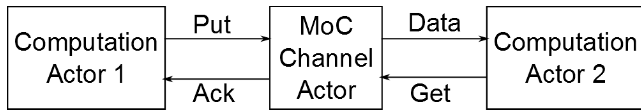


Figure 3. Principle of the implementation

As illustrated in Figure 3, we have defined a common protocol for data exchange between a concurrent block and a MoC block. To send data, a concurrent block (Computation Actor 1) should send a *Put* message to the MoC block and wait for an *Ack* message. To read data, a concurrent block should send a *Get* and wait for a *Data* message. In the case of MoCs that describe non-blocking write operations such as KPN or SDF, the MoC actor sends the *Ack* message as soon as it receives a *Put* message. As the MoC actor does not wait for the reception of a *Get* message, we can simulate a non-blocking write operation. Otherwise, the *Ack* message is sent when the synchronization between the writer and the reader processes can occur.

A simple implementation is proposed, based on three main classes: Actor, Channel and MocActor. The code in the Actor class does not depend on the used MoC as shown by Listing 2.

```
read
"Get"
(self channels at: #input) postRequest: #req.
"Data"
^(self channels at: #input) getAck

write: aData
"Put"
(self channels at: #output) postRequest:
        aData.
"Ack"
(self channels at: #output) getAck
```

Listing 2. The Actor's read and write operations

4.2 Implementation

We now illustrate the implementation of Communicating Sequential Processes, Kahn Process Networks and Synchronous Data Flows.

4.2.1 Implementing Communicating Sequential Process

Communicating Sequential Process (CSP) were introduced by Hoare and describes a network of concurrent processes, where communications are rendez-vous based [13]. Both write and read operations are blocking. CSP also offers a conditional rendez-vous mechanism that introduces non-determinism in the Model of Computation. To achieve non-determinism, multiple rendez-vous are started concurrently. Only the first rendez-vous that can be made is taken in account. The other rendez-vous are canceled as soon as the first one succeeded. In order to select which conditional rendez-vous must be started, a guard is used.

CSP is well suited for modeling systems that require tight synchronization between processes. An example is the dining philosopher problem that can be easily solved with CSP.

In our implementation, the rendez-vous between a producer and a consumer is managed by a dedicated Actor called *CspRendezVous*. The *CspRendezVous* actor implements several methods that handle incoming messages and execute a rendez-vous. Each method follows the same principles:

- it is called on reception of a trigger event;
- after reception of a message from a consumer, the method waits for a message from a producer and vice-versa;
- if a conditional rendez-vous is canceled then the original message must be put in queue again;

Figure 4 is an excerpt of the state machine implemented in the *CspRendezVous* actor. It details the reception of a *Get* message from a reading process.

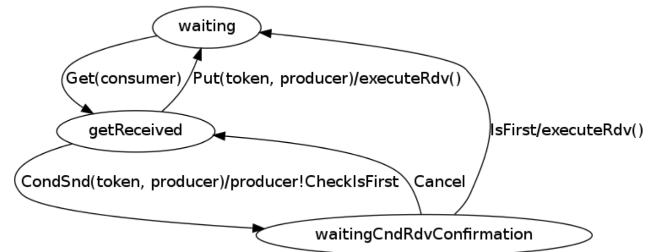


Figure 4. Part of the state machine of *CspRendezVous* to manage the reception of a *Get* message

A method named *executeGet* is called after the reception of a *Get* message and it waits either for a *Put* message or a *CondSnd* message.

When a *Put* message is received the method *executeRdv* is called. This method performs the rendez-vous by sending an *Ack* message to the producer and a *Data* message to the consumer.

When a *CondSnd* message is received, a *CheckIsFirst* message is sent to the producer. This message informs the producer that the rendez-vous is possible. The producer notifies the first ready

CspRendezVous by sending an *IsFirst* message to the actor, which calls the *executeRdv* method. An example of the sequence of messages that are sent during a conditional communication is shown Figure 5.

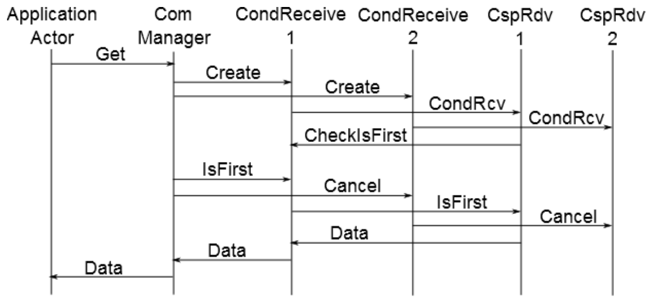


Figure 5. Example of sequence of messages sent during a Conditional communication

Conditional rendez-vous are initiated by a specific actor that is able to start several actors which are competing to perform a rendez-vous. This actor reacts to a *Get* or a *Put* message sent by an application actor. It also checks which rendez-vous is successful at the first place and cancels the others.

4.2.2 Implementing Kahn Process Networks

The Kahn Process Network MoC describes a network of communicating parallel processes as an oriented graph [16]. In KPN, the communication are made through non-blocking write operations and blocking read operations. The communications are made through channels with an infinite FIFO. Processes in KPN can be created during the execution of the process network.

Kahn Process Network is useful in the modeling of systems based on data flows. Signal processing or scientific computing applications are example of data flow applications.

Our implementation is based on Kahn and McQueen's implementation proposal [17]. The FIFO of a channel is managed by a dedicated actor called *KpnChannel*. The attributes of this actor are:

queue The FIFO managed by the actor.

hungryConsumer A consumer of data that is blocked on a read operation on the FIFO.

isFinishReceived The flag that indicates that no more data will be received from the producer.

monitor A specific actor that manages the different FIFOs.

The KpnChannel actor may receive three different messages: *Get*, *Put*, and *Finish*.

Figure 6 describes the state machine implemented in the actor *KpnChannel*.

When receiving a *Get* message, the *KpnChannel* actor checks if its *queue* is empty. If there is data in the *queue*, the *KpnChannel* sends the head of the *queue* to the consumer in a *Data* message. Otherwise, the *KpnChannel* checks if the *isFinishReceived* flag is set. In such a case, the *KpnChannel* sends the *Finish* message to the consumer and the *KpnChannel* kills itself. Otherwise, the attribute *blockedConsumer* is set and a message *ReadBlocked* is sent to the *monitor*. Then the *KpnChannel* waits for new messages.

When receiving a *Put* message, the *KpnChannel* first sends an *Ack* message to the producer. Then it checks if there is an *hungryConsumer*. In such a case, the *KpnChannel* sends a message *Data* to the *hungryConsumer* and a message *ReadUnblocked* to the monitor. The *hungryConsumer* attribute is set to *null* and the *KpnChannel* waits for new messages. Otherwise, the data received from the producer is enqueued and the *KpnChannel* waits for new messages.

When receiving a *Finish* message, if there is an *hungryConsumer*, the *KpnChannel* sends to it a *Finish* message and kills itself. Otherwise, the flag *isFinishReceived* is set to true and the *KpnChannel* goes on waiting for new messages.

The attribute *monitor* is a reference to an actor of kind *KpnMonitor*. Its role is to detect deadlocks. A deadlock occurs when all active process are blocked into a read operation. So the *KpnMonitor* keeps a count of the active process and of those blocked in a read operation.

4.2.3 Implementing Synchronous Data Flow

Like KPN, Synchronous Data Flow (SDF) describes communication with non-blocking write and blocking read operations [21]. SDF adds the constraint that the production or consumption rates on the communication links are constant and well known. As a result SDF offers the ability to pre-determine a scheduling of the involved processes. A SDF process begins by reading the required amount of data on each of its inputs, then performs its processing and finally writes a given amount of data on each of its outputs.

Unlike KPN, SDF can only be used when the data production rate of the different processes are known, as is the case in most of the signal processing applications.

Our implementation use a multithreaded dynamic schedule as described by Schaumont [28]. The actors of the application work concurrently. The scheduling of the actors is left to the underlying virtual machine.

The actor *SdfChannel* has an attribute *nbElemToRead* which defines how much data must be read by the consumer at each read operation. If there is not enough data in the FIFO then the read operation is blocked until the required amount of data is reached.

The *SdfChannel* actor receives either a *Get* or a *Put* message. The content of a *Put* message is a list of data. This list is dequeued to be inserted in the FIFO.

When a Get message is received, the size of the FIFO is compared to the *nbElemToRead* attribute. If it is superior or equal, *nbElemToRead* elements of the FIFO are dequeued to populate a List of data that is sent to the consumer. Otherwise, the consumer is blocked until enough data is available.

4.3 Using Scala and Smalltalk

Scala and Smalltalk both provide interesting programming mechanisms. An example is the collection manipulation methods such as *map* and *filter* in Scala and *collect* and *select* in Smalltalk. These methods ease the manipulation of data structures. An advantage of Scala is that it has built-in actors. In various projects of sea floor observatories [6, 14] the Java language is used. Since Scala runs on the Java Virtual Machine, it is possible to reuse existing code and to integrate third-party code written in Java in our framework.

A big difference between Scala and Smalltalk lies in the type system. Even if Scala implements a type inference mechanism, it remains a statically typed language. It has the advantage of raising some errors at compile time. However, the type of each variable has to be either inferred from the context or made explicit by the programmer. It has an impact on the use of generic messages for the communication between actors. For example, the *Put* message has an argument with the generic type *Any* (equivalent of *Object* in Smalltalk). The processing of the *Any* message in a consumer process requires type checking and casting before a clean use of the value carried by the *Data* message.

On the contrary, Smalltalk is dynamically typed: there is no need for type checking and casting. This produces a more readable code than in a statically typed language. Moreover, dynamic typing

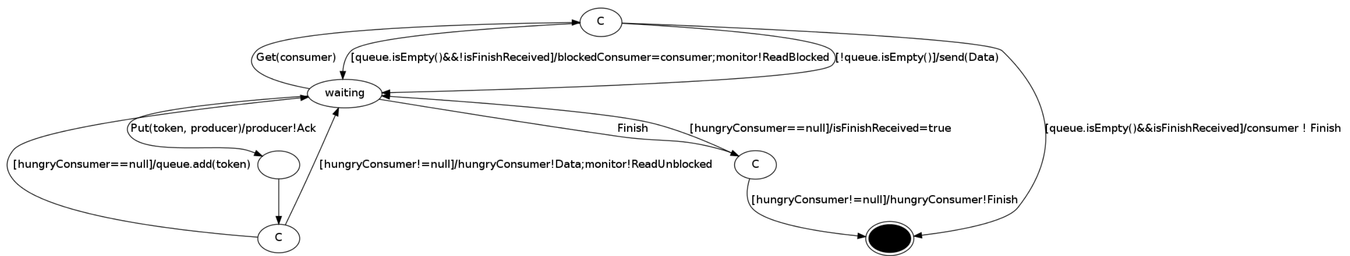


Figure 6. State machine of the actor *KpnChannel*

seems more suited for fast prototyping purposes as changing the type of a variable does not affect the whole code.

```
process

[[| ack data |
"read request, may be blocking"
ack :=(self channels at: #output) getRequest.
"read data, may be blocking"
data :=(self channels at: #input) getRequest.
(self channels at: #output) postAck: data.
(self channels at: #input) postAck: ack]
repeat] fork
```

Listing 3. The CSP MoC in Smalltalk

```
process
"Two processes"
[[     "read data, may be blocking"
self fifo nextPut:
    (self channels at: #input) getRequest.
"automatic acknoledge"
(self channels at: #input)
    postAck: #acknoledge
]repeat] fork.

[[     "read request, may be blocking"
(self channels at: #output) getRequest.
"fifo access request, may be blocking"
(self channels at: #output) postAck:
       self fifo next
]repeat] fork
```

Listing 4. The KPN MoC in Smalltalk

The listings 3 and 4 illustrate how the behavior divergence is implemented. The MocActor object is an abstract common super class of CSPMocActor and KPNMocActor; it implements the process method, which schedules the get/ack operations over the IOs channels. Every channel has two SharedQueues and support posting and requesting operations over them. Additionally, the MocActor owns an auxiliary SharedQueue named FIFO. This shared queue supports temporary storage of data for KPN and SDF. It's useless for CSP, though.

5. Experimentation

5.1 Description of the Experimentation

In the MeDON project[14], we deployed a high-definition camera. This camera produces a lot of images that are manually processed. We create a smart sensor using the HD camera as the sensing device. The smart sensor embeds image processing algorithms. These algorithms are used on the acquired data before they are sent to a ground-based server. In the following, we simplified the example. The acquisition part consists in reading an image file on the disk. An example of image processing algorithm is the Sobel edge detection algorithm. The data sending consists in writing the result of the Sobel algorithm on the disk.

5.2 Exploring the Logical Architecture Alternatives

The exploration phase intends to determine the best solution when describing the architecture as a set of communicating processes. Not all of the processes have a direct mapping with a physical device or sensor. Instead, the analysis focuses on functions being executed, with a tradeoff to be found out, between simplicity of simulation and accuracy of the modeled behavior.

5.2.1 Presentation of the Architecture Proposals

Multiple architectures may be used for this example. The simplest alternative makes each block of the functional architecture become a concurrent entity. Another possible breakdown is to decompose the Sobel algorithm into multiple concurrent entities. This architectural alternative is shown Figure 7.

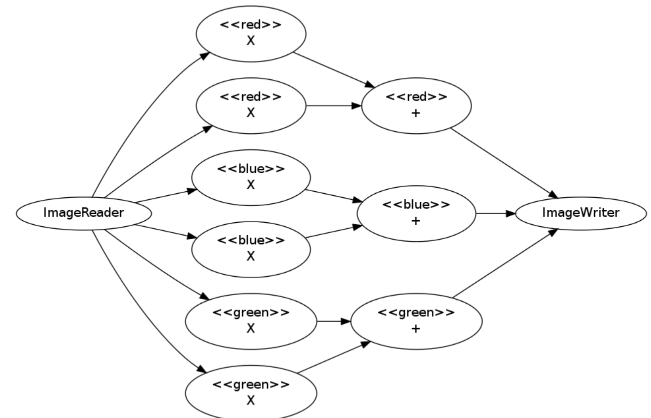


Figure 7. Alternative for the logical architecture

The *ImageReader* is responsible for reading the image from a file. For each pixel and for each color plane, the *ImageReader* creates a list of nine elements containing the different values associated to a pixel and its neighbors.

For each color plane, two convolution and a sum are required. Each of them are transformed into concurrent entities. The *ImageWriter* is responsible for creating an image from the values coming from the sums.

As there are several concurrent entities, we need to ensure the communication and synchronization of these entities. For example, we need to ensure that the sum is performed on data that concerns the right pixel.

5.2.2 Implementation using only CSP

The topology described in Figure 7 can be implemented directly using actors. However, the order of the messages received by the different actors is not guaranteed.

One possible solution to this issue is to use a highly synchronized system (rendez-vous based). This ensures that producer and consumer work at the same rate. As no data is produced until the previous one is not consumed there will be no inversion of data. However, this imposes strong constraints on the real system. It limits the number of processes able to run in parallel.

Each ellipse in Figure 7 is implemented as an actor with a computation role. Each arrow is implemented by a *CspChannel* actor. Each *CspChannel* actor handle the rendez-vous between two computation actors as described by CSP. The code of a computation actor contains instructions to send messages to and receive messages from *CspChannel* actors. These instructions are the only differences due to the use of our framework rather than a purely actor based implementation. An excerpt of the modified topology of the application, modeled based on CSP, is shown in Figure 8. The *CspChannel* actors that are added are represented in black.

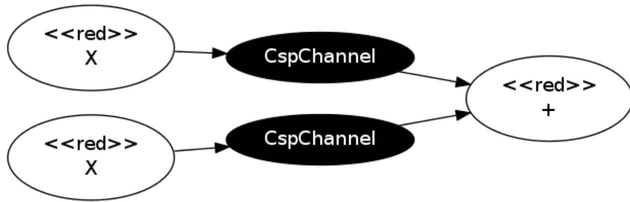


Figure 8. Topology of the application using CSP

The use of CSP prevents an actor from acting on data before the tasks of the previous actors in the flow are not completed on their own input data. As a result, the order of the data at the inputs of the *SumActor* is kept.

The use of our implementation does not affect the logic of the actors composing the system but only the way data are exchanged. This agility only requires that for each sending of data we add code to receive the acknowledge from the MoC actors. In the case of receiving data, we have to add code to send a get request to the MoC actor and the code to wait for its answer.

5.2.3 Implementation using only KPN

In our previous implementation, we used CSP to make the different actors communicate. It ensures the correct treatment of the pixels making the input image. CSP puts strong constraints on the system. As the synchronization between the application actors is obtained through rendez-vous, the number of actors that are able to work concurrently is limited. This may reduce the overall performance of the application.

These constraints are useful when developing an application. They simplify the debugging process through offering a more sequential scheme to the designer. This level of constraints may not be necessary in the final application. Exploration - as previously stated - remains one of our more critical motivations beyond this work. The ability to switch between different MoCs is a key facility to support agility and incremental refinements at no cost in term of readability and understanding versus final performance tradeoff.

Another alternative to CSP channels, is to use a FIFO per required communications between two computation actors. This reduces the risks of reading data coming from the same sender twice. Besides, blocking read operation ensures that the consumer process will wait for incoming data. This corresponds to the KPN MoC.

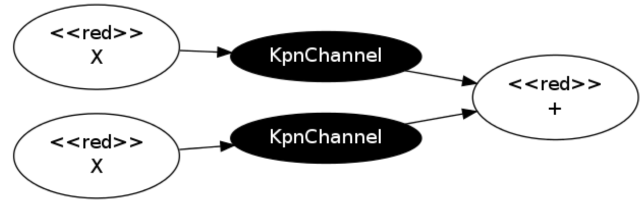


Figure 9. Modification of the topology using KPN channels

In contrast to CSP, KPN enables a process that produces data to keep running while the consumer of its data is also running. This allows to have more processes running in parallel than with CSP. However, there is a lost of control over the communications. It is not possible to ensure that the system will have enough memory for the different FIFOs.

We made an implementation of this alternative with our framework using *KpnChannel* actors. No modification is required on the application actors as we defined a common interface for all implementation of MoCs. The only changes occur in the description of the topology of the application to obtain the topology shown in Figure 9. New instances of kind *KpnChannel* have to be created to replace instances of *CspChannel*. We are able to quickly change the type of communication or synchronization between actors in the application we would like to prototype.

5.3 Benefits of the Smalltalk debugger

From our point of view, in these experiments the main advantage of Smalltalk over Scala is its debugger.

Firstly the Smalltalk debugger is integrated in the development environment. When a fault occurs in an application the debugger is started. It enables to perform a post-mortem analysis of the application. It is possible to check the state of the different components of the application. On the contrary, Scala only provides a message corresponding to the exception that occurs in the running program. The advantage of Smalltalk is the ability to analyze deeply the reason of a failure as all information are available.

Secondly the Smalltalk debugger enables to make on-line modification on the code of the running application and on the values of the different variables. The Scala debugger only to modify the values of the variables. The Smalltalk's debugger provides the ability to make a correction on the application and to continue the execution. It is an asset when performing fast prototyping as there is no need to perform the *Code - Compile - Run* cycle again.

The Redpill [20] environment reproduces most of the Smalltalk debugger features at a hardware level. This is critical as only hardware emulation can support scalability; at mid-term, we want to address massive sensor networks. Since Redpill has been developed using Cincom Visualworks, and is MoC oriented - despite only CSP is supported at this time - it offers a sound path to integrate our modeling and evaluation framework with hardware synthesis. Not only extending the set of supported MoCs makes sense, but it is part of our strategic research plan and lies at the heart of several key projects.

6. Conclusion and Future Work

Smart sensors appear as a solution for coping with the large amount of data acquired by a sea floor observatory. However they come at the price of an increased complexity. When designing the embedded software of a smart sensor the biggest challenges come from its inherent parallelism and concurrency.

In this paper we explored the use of different Models of Computations for modeling smart sensors. We created dedicated actors for

the behavior of the communication described by the MoCs. This enables to separate the computations from the communications and synchronizations. Moreover, we defined a common protocol of data exchange between the computation actors and the MoC actors. This enables to have a modular simulation framework for concurrent applications defined with Models of Computations. This framework can be used to help to define a candidate architecture for concurrent applications.

In future work we will realize an hardware emulation of such smart sensors, with no loss in term of observability and controllability of the execution. This will offer both faster execution and scalable modeling. This direction takes a direct benefit from the RedPill framework. Next, system integration will be considered. Multiple abstraction layers and on-demand refinements will support addressing (smart) sensor networks. It's a second dimension for scalability, with qualitative enhancements in addition to quantitative scaling. This second direction will benefit from previous work that we have led on the Cometa and Biniou frameworks.

Acknowledgments

This work has been done with the financial support of the French Délégation Générale de l'Armement and of the Région Bretagne.

References

[1] Erlang programming language. URL `http://www.erlang.org/`.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[3] A. Bergel, W. Harrison, V. Cahill, and S. Clarke. Flowtalk: language support for long-latency operations in embedded devices. *Software Engineering, IEEE Transactions on*, 37(4):526–543, 2011.

[4] M. Botts and A. Robin. Opengis sensor model language (sensorml) implementation specification. *OpenGIS Implementation Specification OGC*, pages 07–000, 2007.

[5] J.-P. Briot. Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment. In *Proceedings ECOOP*, volume 89, pages 109–129, 1989.

[6] Data Management and Archiving System Team. Neptune and venus data management and archiving system (dmas) preliminary design review. Technical report, Neptune Canada, 2006.

[7] P. I. Diallo, J. Champeau, and V. Leilde. An approach for describing concurrency and communication of heterogeneous systems. In *Proceedings of the Third Workshop on Behavioural Modelling*, BM-FA '11, pages 56–63, 2011. ISBN 978-1-4503-0617-1.

[8] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. In *Proceedings of the IEEE*, pages 127–144, 2003.

[9] F. Fleurey, B. Morin, A. Solberg, and O. Barais. Mde to manage communications with and between resource-constrained systems. In *Model Driven Engineering Languages and Systems*, pages 349–363. Springer, 2011.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[11] P. Haller and F. Sommers. *Actors in Scala. Concurrent Programming for the multi-core era*. Artima, 2012.

[12] C. Haskins, K. Forsberg, and M. Krueger. Systems engineering handbook. *INCOSE. Version*, 3.2, 2010.

[13] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. URL `http://doi.acm.org/10.1145/359576.359585`.

[14] Interreg IVA. Marine edata observatory network, 2013. URL `http://medon.info/`.

[15] A. Jantsch and I. Sander. Models of computation in the design process. 2005.

[16] G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[17] G. Kahn and D. Macqueen. Coroutines and Networks of Parallel Processes. Rapport de recherche, 1976. URL `http://hal.inria.fr/inria-00306565`.

[18] R. K. Karmani and G. Agha. Actors. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1–11. Springer, 2011. ISBN 978-0-387-09765-7.

[19] L. Lagadec and D. Picard. Software-like debugging methodology for reconfigurable platforms. In *IPDPS*, pages 1–4. IEEE, 2009.

[20] L. Lagadec and D. Picard. Smalltalk debug lives in the matrix. In *International Workshop on Smalltalk Technologies*, IWST '10, pages 11–16, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0497-9. doi: 10.1145/1942790.1942792. URL `http://doi.acm.org/10.1145/1942790.1942792`.

[21] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987. ISSN 0018-9219. doi: 10.1109/PROC.1987.13876.

[22] E. A. Lee and S. A. Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Lee & Seshia, 2011.

[23] S. Marr. *Supporting Concurrency Abstractions in High-level Language Virtual Machines*. PhD thesis, Software Languages Lab, Vrije Universiteit Brussel, January 2013.

[24] I. C. on Systems Engineering. Incose, 2013. URL `http://www.incose.org/`.

[25] R. Robbes, N. Bouraqadi, and S. Stinckwich. An aspect-based multi-agent system. *ESUG 2004 Research Track*, page 65, 2004.

[26] A. Robin and M. E. Botts. Creation of specific sensorml process models. *Earth System Science Center-NSSTC, University of Alabama in Huntsville (UAH), HUNTSVILLE, AL*, 35899, 2006.

[27] I. Sander and A. Jantsch. System modeling and transformational design refinement in forsyde [formal system design]. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23 (1):17–32, 2004.

[28] P. R. Schaumont. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2012.

[29] J.-P. Schneider, J. Champeau, D. Kerjean, O. K. Zein, Y. Auffret, and L. Dufrechou. Domain specific modelling applied to smart sensors. In *OCEANS, 2011 IEEE-Spain*, pages 1–6. IEEE, 2011.

[30] B. Spencer Jr, M. Ruiz-Sandoval, and N. Kurata. Smart sensing technology: opportunities and challenges. *Structural Control and Health Monitoring*, 11(4):349–368, 2004.

[31] S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP 2008–Object-Oriented Programming*, pages 104–128. Springer, 2008.

[32] B. Sufrin. Communicating Scala Objects. In P. H. Welch, S. Stepney, F. Polack, F. R. M. Barnes, A. A. McEwan, G. S. Stiles, J. F. Broenink, and A. T. Sampson, editors, *Communicating Process Architectures 2008*, pages 35–54, sep 2008. ISBN 978-1-58603-907-3.

[33] E. Technologies. Scade suite, 2013. URL `www.esterel-technologies.com/products/scade-suite/`.

[34] D. M. Toma, T. O'Reilly, J. del Rio, K. Headley, A. Manuel, A. Broring, and D. Edgington. Smart sensors for interoperable smart ocean environment. In *OCEANS, 2011 IEEE-Spain*, pages 1–4. IEEE, 2011.

Representing Code History with Development Environment Events

Martín Dias Damien Cassou Stéphane Ducasse

RMoD

Inria Lille–Nord Europe — University of Lille — Lifl

Abstract

Modern development environments handle information about the intent of the programmer: for example, they use abstract syntax trees for providing high-level code manipulation such as refactorings; nevertheless, they do not keep track of this information in a way that would simplify code sharing and change understanding. In most Smalltalk systems, source code modifications are immediately registered in a transaction log often called a ChangeSet. Such mechanism has proven reliability, but it has several limitations. In this paper we analyse such limitations and describe scenarios and requirements for tracking fine-grained code history with a semantic representation. We present Epicea, an early prototype implementation. We want to enrich code sharing with extra information from the IDE, which will help understanding the intention of the changes and let a new generation of tools act in consequence.

Keywords Source-code change meta-model; Collaboration; Continuous Versioning; Explore-first Programming

1. Introduction

Modern integrated development environments (IDEs) can have information about the intent of the programmer: they use abstract syntax trees (ASTs) and provide high-level code manipulation (such as refactorings [3]). Nevertheless, they do not keep track of this information in a way that would simplify code sharing and change understanding. For example, after a few hours of work, developers might want to separately share the different changes they have worked on: documentation improvements, bug fixes, and feature additions are better committed separately to facilitate review and backtracking. If each change were semantically recorded, making separate commits would be much simpler: for example, a method renamed could be seen as just one high-level operation instead of many lines removes and added.

In this paper we describe scenarios, requirements, and an early prototype, named Epicea,[1] for tracking code history with a semantic representation. Based on Epicea, we want to enrich code sharing with extra information from the IDE, which will help understanding the intention of the changes and let tools act in consequence. For example, when a library developer updates an API (*e.g.,* by renaming a method), he can provide a dedicated semantic change to the library users so that they can update their client code automatically.

Structure of the paper. In Section 2 we describe the problem in current Smalltalk systems. In Section 3 a series of scenarios illustrate the key requirements for tracking changes semantically. We summarise such requirements in Section 4. We present the design of our prototype in Section 5. Section 6 has screenshots of our prototype in action. After a short overview of related work in Section 7 we conclude in Section 8.

2. Analysis of Current Smalltalk Systems

In most Smalltalk systems [4] source code modifications are logged immediately after any editing operation in a transaction log, often called a ChangeSet.[2] This transaction log acts as a tape recording source code changes. The programmer can navigate different versions of the code without requiring a traditional version control system (VCS), such as git, svn and Monticello. In addition, if the execution of the system is interrupted (*e.g.,* the virtual machine crashes or the process is killed), then such a log can be explored to recover and replay the sequence of changes.

While this log mechanism has proven to be reliable over the years, it has the following problems:

Barely structured text. There is a lack of abstraction. The log is a text file where each new event is appended at the end, as a sequence of chunks. Instead of represent-

[1] http://smalltalkhub.com/#!/~MartinDias/Epicea

[2] http://wiki.squeak.org/squeak/674

[Copyright notice will appear here once 'preprint' option is removed.]

ing the events in a declarative format, the events are written as executable commands. The idea is that by re-evaluating them the original change is reproduced. This format makes it difficult for tools to recover semantic information.

Elementary model. A ChangeSet records only class, package and method definitions. As a result, ChangeSet lacks information about class modifications or high-level events such as refactorings.

Mixing sources and system events. ChangeSets mix source management (the state of a system) with system event recording (the steps to go from one state to the next). The same model and format is used for ChangeSets and the traditional in Smalltalk fileIn/fileOut mechanism. As a result, not all the events can be recorded (*e.g.,* refactorings, package loading). In addition, the granularity of the events is often too coarse, leading to problems on recovery. For example, instance variable addition and class addition are indistinguishable.

Losing intermediate states. ChangeSets only keeps track of what entities (*e.g.,* a class or method) has been modified. The intermediate states of such entities cannot be recovered but just the current one.

In this paper we introduce the notions of *Log* and *View* to fix the above-mentioned problems.

3. Scenarios for Changes as Programming Activity Traces

In this section we present several scenarios that illustrate the use of logs and their interplay in the IDE. We first define the vocabulary used in the rest of this paper.

Image. In a Smalltalk environment, an image is a snapshot of all the objects of the system, *i.e.,* a memory dump: this includes both the objects of the software under execution but also the classes and methods at the moment of the snapshot. An image acts as a cache with preloaded packages and initialised objects.

Session. An image can be launched, modified, and saved many times. We call each one of these periods a session.

Operation. We refer with this word to an action performed in a session. An operation can either have a duration in time (*e.g.,* an expression evaluation) or be a punctual fact (*e.g.,* a class addition). An operation can trigger other operations. In Figure 1, the list in the top represents a session where the developer has done three operations: (1) he has loaded the version 1 of a package named P using a VCS; (2) he has undone the addition of the class A from package P; (3) he has added a new class named B to package P. The light grey bullets and the horizontal alignment of the elements represent triggering (undoing the addition of class A has triggered the removal of A).

Event. We define an event as a representation of an operation. Some events represent a modification in the source code; we refer to them as *code changes*. Sometimes we say that an event triggered another event when the operation that the former event represents triggered the operation that the latter event represents.

Log. A log contains events recorded from the IDE. This includes, for example, class additions, method redefinitions, and refactorings. If the user does not save or if the system crashes, the log and the image will become desynchronised: *i.e.,* the log will contain information that is not in the image.

Code unit. In this paper we call code unit to a package, class, trait or method.

View. The log can have an overwhelming amount of information recorded about the system. This makes it difficult to understand the changes in a particular code unit. To solve this problem we include the concept of *view*. In Figure 1, views for the class A and package P are shown. The history of A is simple: it was added and then removed. The view of P is more complex: first, the class A was added, then this change got undone, and finally the class B got added (creating an implicit branch in the view). Each view has a head, marked as ◁ h [X], which represents the current state in the system for the code unit X. The current head will be the parent of the next change that affects this code unit and the head will be updated to point to this new change.

Commit. We call *commit* a particular version of source code stored in a VCS. In Figure 1, we mark the last change performed during the load of version 1 with the tag P version 1.

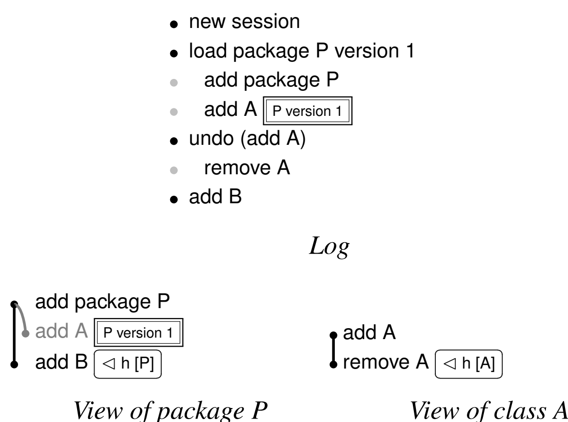


Figure 1. Example.

3.1 Logs Transcend Sessions

Since a code unit can be edited over multiple sessions, the history of a code unit transcend history of images. In this section we discuss some scenarios that crosscut sessions.

Tie the events of several sessions. In Figure 2 we show the history of the package P accumulated over three sessions. The view ignore session boundaries.

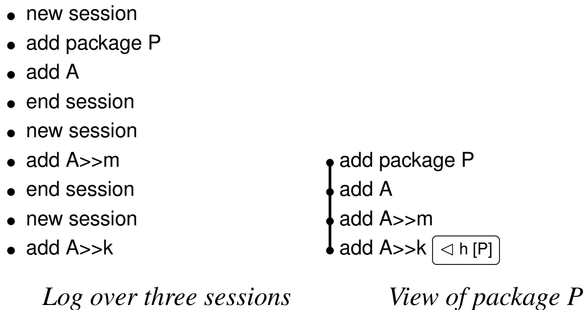


Figure 2. Views ignore session boundaries.

Recover lost changes after the IDE crashed. In Figure 3, the user created a package P with a class A and committed the package P to a VCS. After adding methods m and k, the IDE crashes. The user reopens the IDE, visualises the log of the crashed session, and redoes the lost changes. Such redone changes are shown as a new branch in the view. Each of those redone changes has a [redone] tag. Such a tag always references the original entry so the developer can analyse the event in the context where it was originally logged.

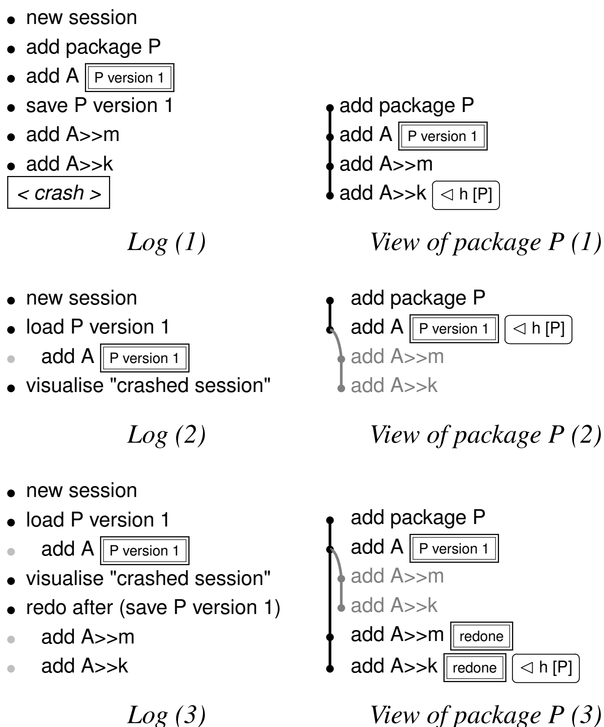


Figure 3. Redo lost changes after the IDE crashed.

Reload in fresh image. Since during experimentation images sometimes become unstable, it is a good practice to regularly rebuild from scratch the current head of development in a fresh image. Current infrastructure supports such practice by loading the code from the VCS, at the expense of losing the versions that occurred between two commits. The log overcomes such problems.

3.2 Code Operations

In this section we discuss some scenarios where navigation to previous versions of code or reorganisation of changes are important.

Undoing a code change. In Figure 4 we show that reverting the addition of method A»m has different effects on the different views. In the package and class views, the original method additions are shown in grey as a branch. In that way, the original history of events with the original chronology is available to be browsed. In the A»m view, the undo operation is seen as a removal of the method. For the A»k view the operation has no impact. Note that in each view there is a head pointing to a different event.

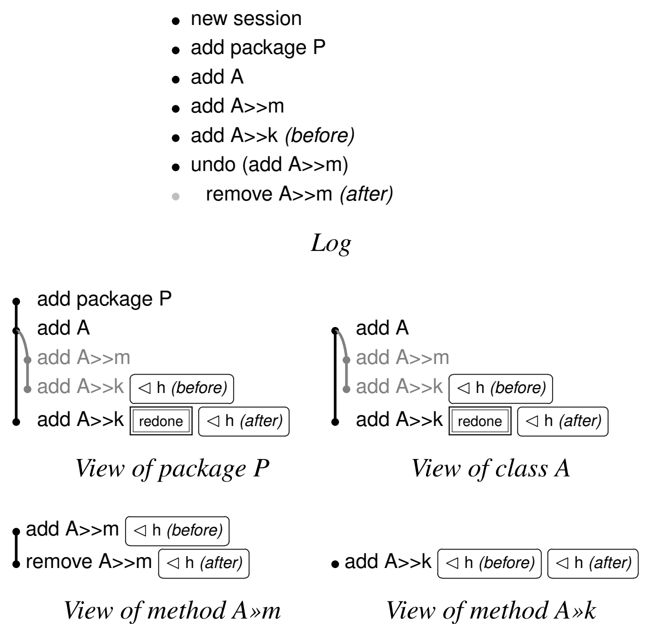


Figure 4. Undoing the addition of A»m. The operation has different effects at package, class and method level.

Grouping changes before committing. When a developer is working for some time on a project, chances are that he will perform multiple independent tasks. This happens even when there is a concrete goal such as implementing a new feature or fixing a bug: either a typo, or some code that deserves a refactoring, or any other change that is unrelated to the goal can appear. Tools should make it easy for a developer to fix the off-topic issue and let him either mark it or split it to a different branch so the main branch stays focused and cohesive. We need a kind of cherry picking of the elements we want to commit. In Figure 5 we show an example of changes done in the package P, where the

developer added a class B with some methods, and in the middle found and fixed a typo in the comment of A»m. He decides to create a new branch to keep this change separated from the other ones. He also adds a comment to the separated change (modify A»m) with a 'typo fix' tag.

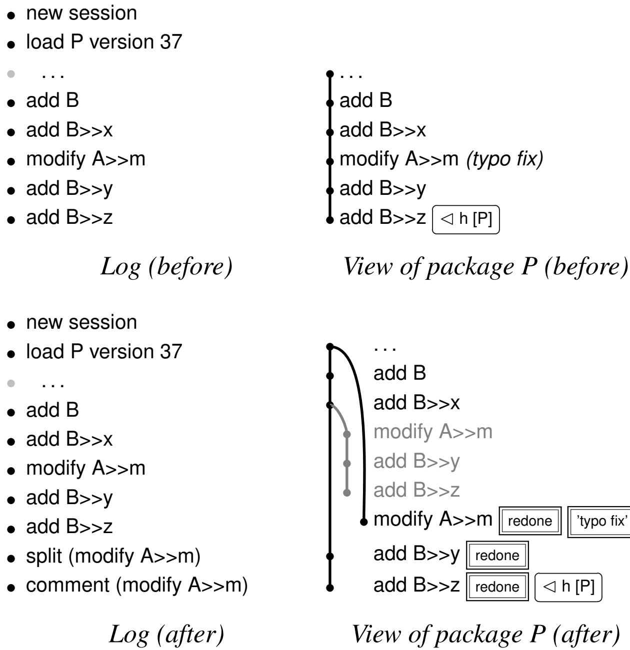


Figure 5. Split changes for doing meaningful commits.

Commenting events. The developer can write arbitrary comments on an event (or group of events) to facilitate later understanding. We mentioned this feature in Figure 5, with the 'typo fix' tag. Additionally, the system can help the developer writing comments based on what triggered the related event.

Condensing code changes. The log might have changes that neutralise themselves (*e.g.,* a method is added and removed). In addition there are cases where the programmer may want to forget current history of certain entities. In Figure 6, we show in an example how the condense operation works when applied to the package P. Without any optimisation, the operation is done in two main steps: first, undo the events until the older neutralised event (remove B, add C, and add B); second, redo only the needed changes (add C).

Recording refactoring information. Some high-level operations, such as refactorings, group events. In Figure 7, a method is renamed (A»m) and all senders (B»k) of this method are updated. Each event related to the refactoring have a dedicated tag that references the high-level operation.

3.3 Sharing Events

Logs and events can be shared between developers, projects, and images.

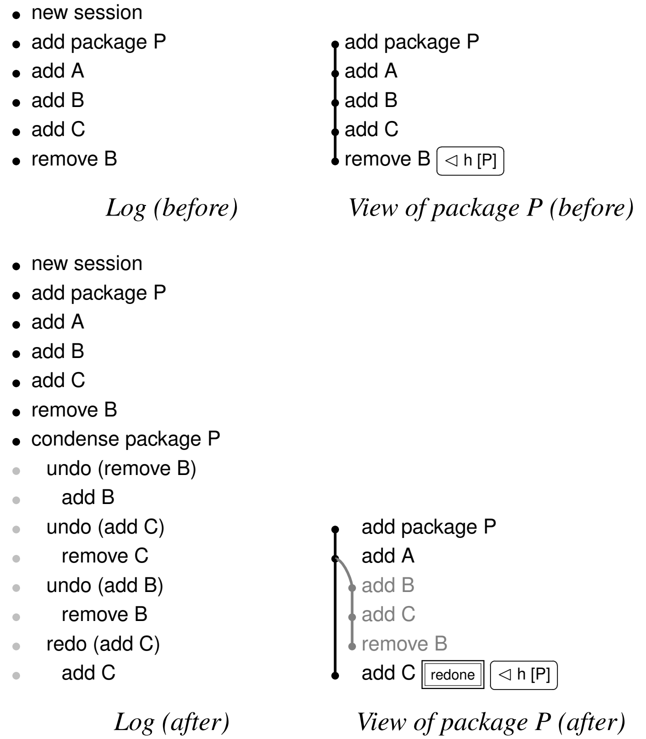


Figure 6. Condense operation.

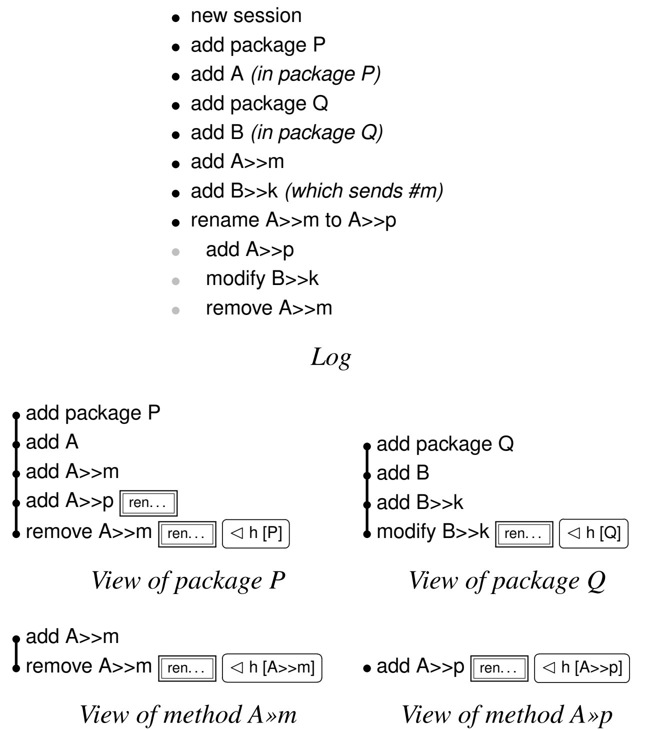


Figure 7. Rename A»m to A»p. The method B»k uses it so it is modified by the refactoring.

Replaying a concrete event. When two projects are forks from each other, events of one fork can be replayed in the other.

Replaying the intent of a refactoring. When a library developer updates an API (*e.g.,* by renaming a method), he can provide high-level events which can be replayed by library users so that they can update their client code automatically.

4. Scenarios: an Analysis

We analysed several existing code change representations: ChangeSets, RingC [7], Cheops [2], NewChangeSystem,[3] and DeltaStreams.[4] From previous work and the scenarios presented above we define the following requirements.

4.1 Requirements

1. Replay and undo operations. Starting from the same or similar system, the information in the log should be enough for reconstructing the state of the system at any point of the log.
2. Log must be immediately persisted out of the volatile memory so information survives IDE crashes.
3. Log entries can have tags, *i.e.,* meta-information. A tag can reference another entry. Tags can be added after the entry has been persisted.
4. Events should be represented as first-class entities.
5. The change model should support modelling many different types of changes: structural elementary changes (method definitions), composed ones (refactorings), and system changes such as expression evaluation, redo, and branch creation.

5. Epicea

We implemented Epicea, an early prototype of the log and the event model. It was developed in Pharo [1]. Epicea model started as a branch of NewChangeSystem project and then was deeply modified and extended.

5.1 Event Model

In Figure 8 we show the class hierarchy of *events* we implemented in Epicea. The most important sub-hierarchy is the one of CodeChange, which represents the operation that made the code change, such as class creation, method modification, etc. Code changes hold enough information about the operation performed for either reverting the change or redoing it. Epicea uses Ring definitions to take snapshots of the involved code units.

We need to record information about the situation in which events are logged. That is the timestamp when it was done, the author who did it, the potential event that triggered it (for example, undoing a method addition triggers a method removal). This meta-information of the event is stored in log entries, as explained below.

[3] http://smalltalkhub.com/#!/~EzequielLamonica/NewChangeSystem

[4] http://wiki.squeak.org/squeak/6001

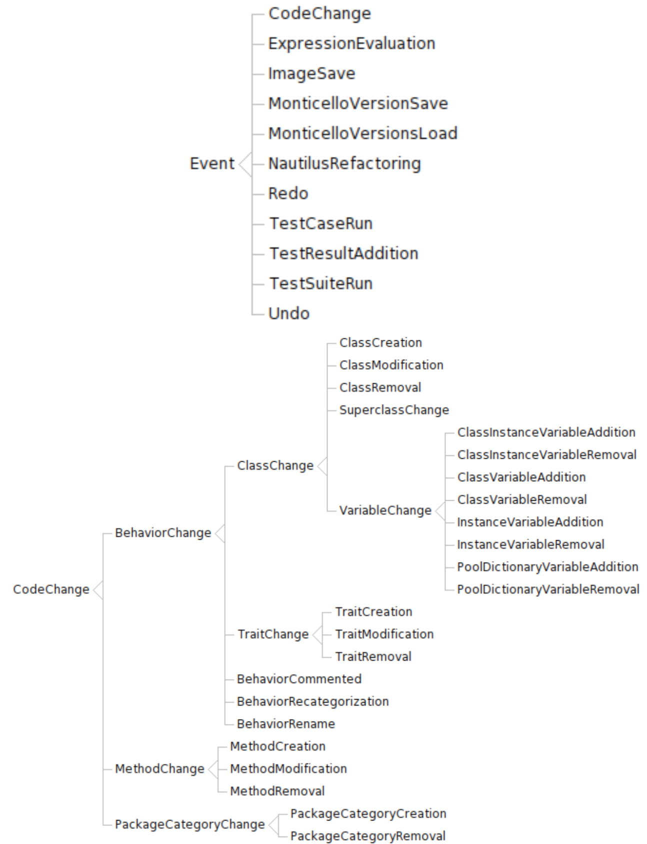


Figure 8. The hierarchies of Event and CodeChange used in our prototype.

5.2 Log Model

In Figure 9 an object diagram shows how a log is represented in the prototype. A log has a head pointing to the entry where the upcoming entry will be attached. Each entry points to a parent entry and the content event. In Figure 10 we show the design we implemented for Epicea. An entry has a dictionary of tags that allows attaching meta-information (author and timestamp). In the case of an event that triggers other events, each of these events has a tag pointing to the triggering event.

6. Revisiting the Scenarios

In Figure 11 an expression was evaluated. It triggered the load of the package named ConfigurationOfFuel. In turn, the load triggered many elemental code changes (package, class and method additions). In Figure 12 we show the log of an undo operation. The class A has been added in package P; then two methods have been added (A»m and A»k). Following, the undo of the addition of the method A»m

- ...
- load P version 1
  - add package P
  - add A

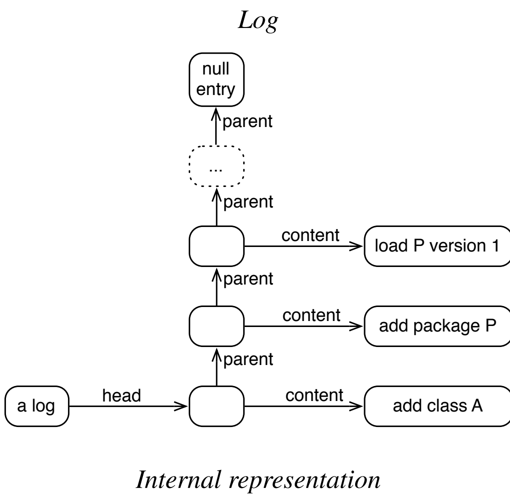


Figure 9. Object diagram of an Epicea log.

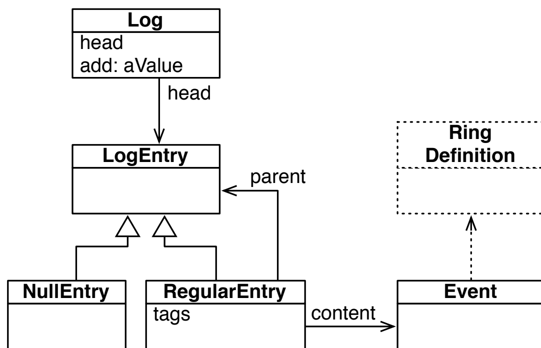


Figure 10. Design of Epicea logs.

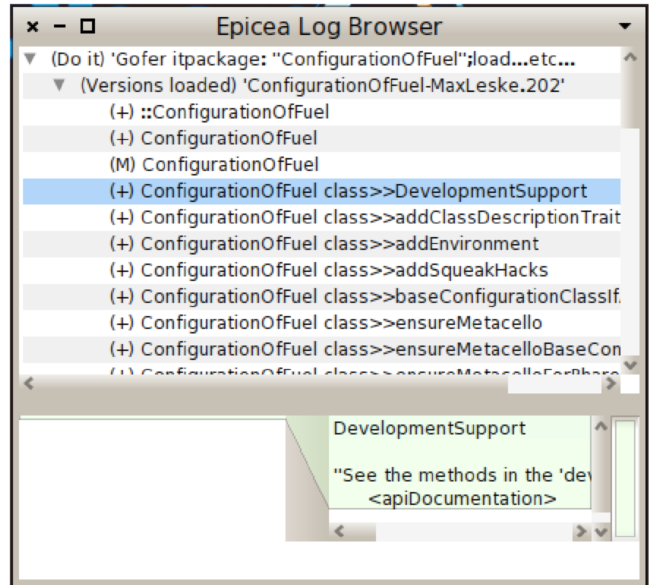


Figure 11. Epicea log browser screenshot: an expression was evaluated. It triggered the load of the package named ConfigurationOfFuel. In turn, the load triggered many elemental code changes (package, class and method additions).

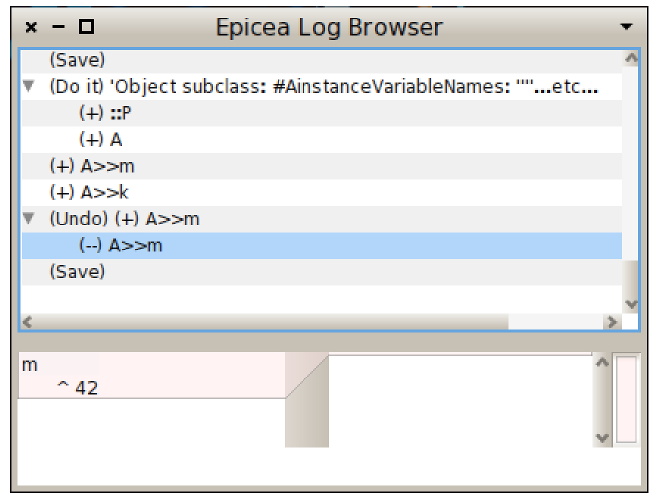


Figure 12. Epicea log browser screenshot: Undo.

triggered the removal of such method. In Figure 13 we show a class rename refactoring as it is logged by Epicea.

7. Related Work

SpyWare [5] captures and stores the code changes in a centralised repository in a extremely fine granularity. SpyWare records detailed changes such as a line added in a method, as well as more high-level changes like refactorings. The authors aim at post-mortem comprehension of developer work, while we focused on helping developers for their day-to-day work.

CoExist [6] is a Squeak/Smalltalk extension that preserves intermediate development states and provides immediate access to source code and run-time information of previous development states. CoExist allows for back-in time easily, automatic forks, inter-branch operations (such as re-

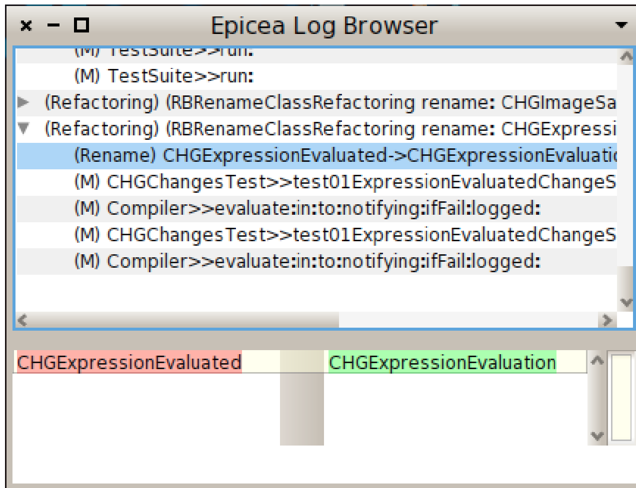


Figure 13. Epicea log browser screenshot: Class rename refactoring.

base and cherry-pick). However, the authors do not talk of a persistence mechanism for the captured code changes. CoExist is not meant to be used to share code between images and projects. Still, CoExist is a source of inspiration for the Epicea model.

JET [7] allows analysing the dependencies between VCS versions. The authors extend the Ring meta-model [8] to perform the computations. Epicea uses Ring as well. It would be interesting to apply JET dependency analysis to logs to get fine-grained results.

8. Conclusion

Modern tools for sharing code lose extra information from IDE. We want to work on a new generation of tools that use such information to help understanding the intention behind code changes. In this paper we have presented our initial steps working in this direction. We have first described a series of scenarios that help discovering main requirements of our approach. Then, we have analyzed the problems found in current Smalltalk systems, focusing on the case of ChangeSets. Finally, we have presented our early prototype with an overview of the design, as well as some screenshots that show it in action.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council.

References

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

[2] Peter Ebraert. First-class change objects for feature-oriented programming. In *Proceedings of the 15th Working Conference on Reverse Engineering*, WCRE'08, pages 319–322. IEEE Computer Society, 2008.

[3] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999. ordered but not received.

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

[5] Romain Robbes and Michele Lanza. SpyWare: a change-aware development toolset. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 847–850, New York, NY, USA, 2008. ACM.

[6] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. CoExist: Overcoming aversion to change - preserving immediate access to source code and run-time information of previous development states. In *DLS'12: Proceedings of the 8th Dynamic Languages Symposium*, DLS '12, pages 107–118, New York, NY, USA, 2012. ACM.

[7] Verónica Uquillas Gómez. *Supporting Integration Activities in Object-Oriented Applications*. PhD thesis, Vrije Universiteit Brussel - Belgium & Université Lille 1 - France, October 2012.

[8] Verónica Uquillas Gómez, Stéphane Ducasse, and Theo D'Hondt. Ring: a unifying meta-model and infrastructure for Smalltalk source code analysis tools. *Journal of Computer Languages, Systems and Structures*, 38(1):44–60, April 2012.

Language-side Foreign Function Interfaces with NativeBoost

Camillo Bruni Stéphane Ducasse
Igor Stasenko

RMoD, INRIA Lille - Nord Europe, France
http://rmod.lille.inria.fr

Luc Fabresse

Mines Telecom Institute, Mines Douai, France
http://car.mines-douai.fr

Abstract

Foreign-Function-Interfaces (FFIs) are a prerequisite for close system integration of a high-level language. With FFIs the high-level environment interacts with low-level functions allowing for a unique combination of features. This need to interconnect high-level (Objects) and low-level (C functions) has a strong impact on the implementation of a FFI: it has to be flexible and fast at the same time.

We propose NativeBoost a language-side approach to FFIs that only requires minimal changes to the VM. NativeBoost directly creates specific native code at language-side and thus combines the flexibility of a language-side library with the performance of a native plugin.

Categories and Subject Descriptors D.3.3 [*Programming Language*]: Language Constructs and Features; D.3.2 [*Programming Language*]: Language Classifications—Very high-level languages

Keywords system-programming, reflection, managed runtime extensions, dynamic native code generation

1. Introduction

Currently, more and more code is produced and available through reusable libraries such as OpenGL[1] or Cairo[2]. While working on your own projects using dynamic languages, it is crucial to be able to use such existing libraries with little effort. Multiple solutions exist to achieve access to an external library from dynamic languages that are executed on the top of a virtual machine (VM) such as Pharo[3], Lua[4] or Python[5]. Figure 1 depicts four possibilities of dealing with new or external libraries in a high-level language.

Language-side Library. One solution is to re-implement a library completely at language-side (cf. Figure 1.a). Even though this is the most flexible solution, this is often not an option, neither from the technical point of view (performance penalty), nor from the economic point of view (development time and costs).

VM Extension. The second one (1.b) is to do a *VM extension* providing new primitives that the high-level language uses to access the native external library. This solution is generally efficient since the external library may be statically compiled within the VM. However a tight integration into the VM also means more dependencies and a different development environment than the final product at language-side.

VM Plugin. The third solution (1.c) is similar to the previous one but the extension is factored out of the VM as a *plugin.* This solution implies again a lot of low-level development at VM-level that must be done for each external library we want to use. Additionally we have to adapt the plugin for all platforms on which the VM is supposed to run on.

FFI. A higher-level solution is to define *Foreign Function Interfaces* (FFIs) (cf. Figure 1.d). The main advantage of this approach is that once a VM is FFI-enabled, only a language extension (no VM-level code) is needed to provide access to new native libraries. From the portability point of view, only the generic FFI VM-plugin has to be implemented on all platforms.

Implementing an FFI library is a challenging task because of its antagonist goals:

[1] http://www.opengl.org/

[2] http://cairographics.org/

[3] http://pharo.org/

[4] http://lua.org/

[5] http://python.org/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © ACM [to be supplied]. . . $10.00

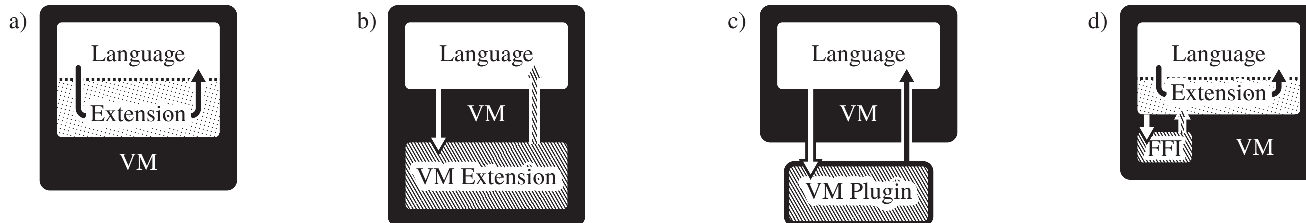


Figure 1: Comparing different extension mechanisms: a) library implemented completely at language-side running on a standard VM, b) language using features from a VM extension, c) language using features from a VM plugin, d) language-side implementation of an extension.

- it must be flexible enough to easily bind to external libraries and also express complex foreign calls regarding the memory management or the type conversions (marshalling);
- it must be well integrated with the language (objects, reflection, garbage collector);
- it must be efficient.

Existing FFI libraries of dynamic languages all have different designs and implementations because of the trade-offs they made regarding these goals and challenges. Typical choices are resorting purely to the VM-level and thus sacrificing flexibility. The inverse of this approach exists as well: FFIs can be implemented almost completely at language-side but at a significant performance loss. Both these pitfalls are presented in more detail in Section 3.

This paper presents NativeBoost-FFI[6] an FFI library at language-side for Pharo that supports callouts and callbacks, which we present in Section 2. There are at least two other existing FFI libraries in Pharo worth mentioning: C-FFI and Alien. Nevertheless, they both present shortcomings. C-FFI is fast because it is mostly implemented at VM-level, however it is limited when it comes to do complex calls that involve non-primitive types or when we want to define new data types. On the opposite, Alien FFI is flexible enough to define any kind of data conversion or new types directly at language-side but it is slower than C-FFI because it is mostly implemented at language-side. In essence, NativeBoost-FFI combines the flexibility and extensibility of Alien that uses language-side definition for marshalling and the speed of C-FFI which is implemented at VM-level. The main originalities of NativeBoost-FFI are:

Extensibility. NativeBoost-FFI relies on as few VM primitives as possible (5 primitives), essentially to call native code. Therefore, most of the implementation resides at language-side, even low-level mechanisms. That makes NativeBoost-FFI easily extensible because its implementation can be changed at any time, without needing to update the runtime (VM). It also presents a noticeable philosophical shift, how we want to extend our language in future. A traditional approach is to implement most low-level features at VM-side and provide interfaces to the language-side. But that comes at cost of less flexibility and longer development and release cycles. On the opposite, we argue that extending language features, even low-level ones, should be done at language-side instead. This results in higher flexibility and without incurring high runtime costs which usually happen when using high-level languages such as Smalltalk.

Language-side extension. Accessing a new external library using NativeBoost-FFI involves a reduced amount of work since it is only a matter of writing a language-side extension.

Performance. Despite the fact it is implemented mostly at language-side, NativeBoost-FFI achieves superior performance compared to other FFI implementations running Pharo. This is essentially because it uses automatic and transparent native code generation at language-side for marshalling.

[6] `http://code.google.com/p/nativeboost`

2. NativeBoost-FFI: an Introduction

This section gives an overview of the code that should be written at language-side to enable interactions with external libraries.

2.1 Simple Callout

Listing 1 shows the code of a regular Smalltalk method named `ticksSinceStart` that defines a callout to the `clock` function of the `libc`. NativeBoost imposes no constraint on the class in which such a binding should be defined. However, this method must be annotated with a specific pragma (such as `<primitive:module:>`) which specifies that a native call should be performed using the NativeBoost plugin.

```
ticksSinceStart
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin>
```

```
^ self
    nbCall: #(uint clock ())
    module: NativeBoost CLibrary
```

Code 1: NativeBoost-FFI example of callout declaration to the `clock` function of the `libc`

The external function call is then described using the `nbCall:module:` message. The first parameter (`#nbCall:`) is an array that describes the signature of C function to callout. Basically, this array contains the description of a C function prototype, which is very close to normal C syntax. The return type is first described (`uint` in this example[7]), then the name of the function (`clock`) and finally the list of parameters (an empty array in this example since `clock` does not have any). The second argument, `#module:` is the module name, its full path or its handle if already loaded, where to look up the given function. This example uses a convenience method of NativeBoost named `CLibrary` to obtain a handle to the standard C library.

2.2 Callout with Parameters

Figure 2 presents the general syntax of NativeBoost-FFI through an example of a callout to the `abs` function of the `libc`. The `abs:` method has one argument named `anInteger` (cf. ❶). This method uses the pragma `<primitive:module:error:>` which indicates that the `#primitiveNativeCall` of the `#NativeBoost-Plugin` should be called when this method is executed (cf. ❷). An `errorCode` is returned by this primitive if it fails and the regular Smalltalk code below is executed (cf. ❸). The main difference with the previous example is that the `abs` function takes one integer parameter. In this example, the array `#(uint abs(int anInteger))` passed as argument to `#nbCall:` contains two important information (cf. ❹). First, the types annotations such as the return type (`uint` in both examples) and arguments type (`int` in this example). These types annotations are then used by NativeBoost-FFI to automatically do the marshalling between C and Pharo values as illustrated by the next example. Second, the values to be passed when calling out. In this example, `anInteger` refers to the argument of the `abs` method, meaning that the value of this variable should be passed to the `abs` C function. Finally, this `abs` function is looked up in the `libc` whose an handle is passed in the `module:` parameter (cf. ❺).

[7] The return type of the `clock` function is `clock_t`, but we deliberately used `uint` in this first example for the sake of simplicity even if it is possible to define a constant type in NativeBoost.

```
abs: anInteger ❶
    <primitive: #primitiveNativeCall ❷
     module: #NativeBoostPlugin
     error: errorCode> ❸
    ^ self
        nbCall: #(uint abs(int anInteger)) ❹
        module: NativeBoost CLibrary ❺
```

Figure 2: Example of the general NativeBoost-FFI callout syntax

2.3 Automatic Marshalling of Known Types

Listing 2 shows a callout declaration to the `getenv` function that takes one parameter.

```
getenv: name
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin>

    ^ self
        nbCall: #(String getenv(String name)
        module: NativeBoost CLibrary
```

Code 2: Example of callout to `getenv`

In this example, the NativeBoost type specified for the parameter is `String` instead of `char*` as specified by the standard `libc` documentation. This is on purpose because strings in C are sequences of characters (`char*`) but they must be terminated with the special character: \0. Specifying `String` in the `#nbCall:` array will make NativeBoost to automatically do the arguments conversion from Smalltalk strings to C strings (\0 terminated `char*`). It means that the string passed will be put in an external C `char` array and a \0 character will be added to it at the end. This array will be automatically released after the call returned. This is an example of automatic memory management of NativeBoost that can also be controlled if needed. Obviously, the opposite conversion happens for the returned value and the method returns a Smalltalk String. This example shows that NativeBoost-FFI accepts literals, local and instance variable names in callout declarations and it uses their type annotation to achieve the appropriate data conversion. Table 1 shows the default and automatic data conversions achieved by NativeBoost-FFI.

Listing 3 shows another example to callout the `setenv` function. The return value will be converted to a Smalltalk `Boolean`. The two first parameters are specified as `String` and will be automatically transformed in `char*` with an ending \0 character. The last parameter is `1`, a Smalltalk literal value without any type specification and NativeBoost translates it as an `int` by default.

```
setenv: name value: value
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin>

    ^ self
```

Primitive Type	Smalltalk Type
uint	Integer
int	Integer
String	ByteString
bool	Boolean
float	Float
char	Character
oop	Object

Table 1: Default NativeBoost-FFI mappings between C/primitive types and high-level types. Note that `oop` is not a real primitive type as no marshalling is applied and the raw pointer is directly exposed to Pharo.

```
nbCall: #(Boolean setenv(String name,
                         String value,
                         1)
module: NativeBoost CLibrary
```

Code 3: Example of callout to `setenv`

Another interesting example of automatic marshalling is to define the `abs` method (cf. Figure 2) in the `SmallInteger` class and passing `self` as argument in the callout. In such case, NativeBoost automatically converts `self` (which is a SmallInteger) into an `int`. This list of mapping is not exhaustive and NativeBoost also supports the definition of new data types and new conversions into more complex C types such as structures (cf. Section 4).

2.4 Supporting new types

The strength of language-side FFIs appears when it comes to do callouts with new data types involved. NativeBoost-FFI supports different possibilities to interact with new types.

Declaring structures. For example, the Cairo library[8] provides complex structures such as `cairo_surface_t` and functions to manipulate this data type. Listing 4 shows how to write a regular Smalltalk class to wrap a C structure. NativeBoost only requires a class-side method named `asNBExternalType:` that describes how to marshall this type back and forth from native code. In this example, we use existing marshalling mechanism defined in `NBExternalObjectType` that just copies the structure's pointer and stores it in an instance variable named `handle`.

```
AthensSurface subclass: #AthensCairoSurface
    instanceVariableNames: 'handle'.

AthensCairoSurface class>>asNBExternalType: gen
    "handle iv holds my address (cairo_surface_t)"
    ^ NBExternalObjectType objectClass: self
```

Code 4: Example of C structure wrapping in NativeBoost

Callout with structures. Listing 5 shows a callout definition to the `cairo_image_surface_create` function that returns a `cairo_surface_t*` data type. In this code example, the return type is `AthensCairoSurface` directly (not a pointer). When returning from this callout, NativeBoost creates an instance of `AthensCairoSurface` and the marshalling mechanism stores the returned address in the `handle` instance variable of this object.

```
primImage: aFormat width: aWidth height: aHeight
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin
     error: errorCode>

    ^self nbCall: #(AthensCairoSurface
        cairo_image_surface_create (int aFormat,
                                    int aWidth,
                                    int aHeight) )
```

Code 5: Example of returning a structure by reference

Conversely, passing an `AthensCairoSurface` object as a parameter in a callout makes its pointer stored in its `handle` iv (cf. Listing 6) to be passed. Since the parameter type is `AthensCairoSurface` in the callout definition, NativeBoost also ensures that the passed object is really an instance of this class. If it is not, the callout fails before executing the external function because passing it an address on a non-expected data could lead to unpredicted behavior.

```
primCreate: cairoSurface
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin>

    ^self nbCall: #(
        AthensCairoCanvas cairo_create (
                AthensCairoSurface cairoSurface))
```

Code 6: Example of passing a structure by reference

Accessing structure fields. In NativeBoost, one can directly access the fields of a structure if needed, even if it is not a good practice from the data encapsulation point of view. Nevertheless, it may be mandatory to interact with some native libraries that do not provide all the necessary functions to manipulate the structure. Listing 7 shows an example of a C struct type definition for `cairo_matrix_t`.

```
typedef struct {
   double xx; double yx;
   double xy; double yy;
   double x0; double y0;
```

[8] http://cairographics.org

	Memory	Address	Marshalling	Constraint
C-managed struct	C heap	fixed	passed by reference	must be freed
Pharo-managed struct	Object memory	variable	passed by reference or passed by copy	may move costly

Table 2: Wrapping structures possibilities in NativeBoost

```
} cairo_matrix_t;
```

Code 7: Example external type to convert back and forth with the Cairo library

Listing 8 shows that the `NBExternalStructure` of NativeBoost-FFI can be subclassed to define new types such as `AthensCairoMatrix`. The description of the fields (types and names) of this structure is provided by the `fieldsDesc` method on the class side. Given this description, NativeBoost lazily generates field accessors on the instance side using the field names.

```
NBExternalStructure
    variableByteSubclass: #AthensCairoMatrix.

AthensCairoMatrix class>>fieldsDesc
    ^ #(  double sx; double shx;
          double shy; double sy;
          double x; double y;  )
```

Code 8: Example of NativeBoost-FFI definition of an `ExternalStructure`

Listing 9 shows a callout definition to the `cairo_matrix_multiply` function passing `self` as argument with the type `AthensCairoMatrix*`. NativeBoost handles the marshalling of this object to a struct as defined in the `fieldsDesc`.

```
AthensCairoMatrix>>primMultiplyBy: m
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin
     error: errorCode>

"C signature"
"void cairo_matrix_multiply (
                    cairo_matrix_t *result,
                    const cairo_matrix_t *a,
                    const cairo_matrix_t *b );"
    ^self nbCall: #(void   cairo_matrix_multiply
        (AthensCairoMatrix * self,
        AthensCairoMatrix * m ,
        AthensCairoMatrix * self ) )
```

Code 9: Example of callouts using `cairo_matrix_t`

Memory management of structures. Table 2 shows a comparison between C-managed and Pharo-managed structures. The first ones are allocated in the C heap. Their addresses are fixed and they are passed by reference during a callout. But the programmer must free them by hand when they are not needed. The second ones are allocated in the Pharo object-memory. Their addresses are variable since their enclosing object may be moved by the garbage collector. They can either passed by copy which is costly or by reference. Passing a reference may lead to problems is the C function stores the address and try to access it later on since the address may changed.

2.5 Callbacks

NativeBoost supports callbacks from native code. This means it is possible for a C-function to call back into the Pharo runtime and activate code. We will use the simple `qsort` C-function to illustrate this use-case. `qsort` sorts a given array according to the results of a compare function. Instead of using a C-function to compare the elements we will use a callback to invoke a Pharo block which will compare the two arguments.

```
bytes := #[ 120 12 1 15 ].
callback := QSortCallback on: [ :a :b |
                (a byteAt: 0) - (b byteAt: 0) ].

self ffiQSort: bytes
     length: bytes size
     compareWith: callback
```

Code 10: Example of callout passing a callback for `qsort`

Code 10 shows the primary Pharo method for invoking `qsort` with a `QSortCallback` instance for the compare function. In this example `qsort` will invoke run the Pharo code inside the callback block to compare the elements in the `bytes` array.

To define a callback in NativeBoost we have to create a specific subclasses for each callback with different argument types.

```
NBFFICallback
    subclass: #QSortCallback.

NBFFICallback class>>signature
    ^#(int (NBExternalAddress a, NBExternalAddress b))
```

Code 11: Example of callback definition

Code 11 shows `QSortCallback` which takes two generic external addresses as arguments. These are the argument types that are being passed to the sort block in Example 10.

```
ffiQSort: base len: size compare: qsortCallback
    <primitive: #primitiveNativeCall
     module: #NativeBoostPlugin>

    "C qsort signature"
    "void qsort(
        void *base,
        size_t nel,
        size_t width,
        int (*compar)(const void *, const void *));"

    ^ self
        options: #( optMayGC )
        nbCall: #(void qsort (
                    NBExternalAddress array,
                    ulong size,
                    1, "sizeof an element"
                    QSortCallback qsortCallback))
        module: NativeBoost CLibrary
```

Code 12: Example of callout passing a callback

The last missing piece in this example is the callout definition shown in Code 12. The NativeBoost callout specifies the callback arguments by using `QSortCallback`.

Callback lifetime. Each time a new callback is instantiated it reserves a small amount of external memory which is freed once the callback is no longer used. This is done automatically using object finalization hooks..

2.6 Overview of NativeBoost-FFI Internals

This section provides an overview of the internal machinery of NativeBoost-FFI though it is not mandatory to know it in order to use it as demonstrated by previous examples.

General Architecture. Figure 3 describes the general architecture of NativeBoost. Most code resides at language-side, nevertheless some generic extensions (primitives) to the VM are necessary to activate native code. At language-side, callouts are declared with NativeBoost-FFI which processes them and dynamically generates x86 native code using the `AsmJit` library. This native code is responsible of the marshalling and calling the external function. NativeBoost then uses a primitive to activate this native code.

Callout propagation. Figure 4 shows a comparison of the resolution of a FFI call both in NativeBoost-FFI and a plugin-based FFI. At step 1, a FFI call is emitted. The NativeBoost-FFI call is mostly processed at language-side and it is only during step 4 that a primitive is called and the VM effectively does the external call by executing the native code. On the opposite, a plugin-based FFI call already crossed the low-level frontier in step 2 resulting that part of the

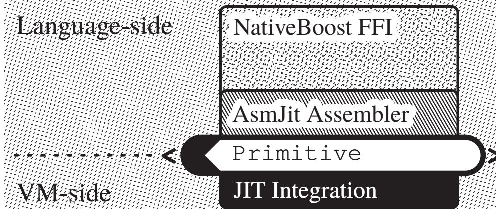


Figure 3: NativeBoost main components that major part of the code resides at language-side.

type conversion process (marshalling) is already done in the VM code. In NativeBoost-FFI, doing most of the FFI call processing at language-side makes easier to keep control, redefine or adapt it if needed.

3. NativeBoost-FFI Evaluation

In this section we compare NativeBoost with other FFI implementations.

Alien FFI: An FFI implementation for Squeak/Pharo that focuses on the language-side. All marshalling happens transparently at language-side.

C-FFI: A C based FFI implementation for Squeak/Pharo that performs all marshalling operations at VM-side.

LuaJIT: A fast Lua implementation that has a close FFI integration with JIT interaction.

Choice of FFI Implementations. To evaluate NativeBoost we explicitly target FFI implementations running on the same platform, hence we can rule out additional performance differences. Alien and C-FFI run in the same Pharo image as NativeBoost allowing a much closer comparison.

Alien FFI is implemented almost completely at language-side, much like NativeBoost. However, as the following benchmarks will stress, it also suffers from performance loss.

On the other end there is C-FFI which is faster than Alien but by far not as flexible. For instance only primitive types are handled directly.

As the third implementation we chose Lua. Lua is widely used as scripting language in game development. Hence much care has been taken to closely integrate Lua into C and C++ environments. LuaJIT integrates an FFI library that generates the native code for marshalling and directly inlines C functions callout in the JIT-compiled code.

Evaluation Procedure. To compare the different FFI approaches we measure 100 times the accumulative time spent to perform 1′000′000 callouts of the given function. From the 100 probes we show the average and the standard deviation for a 68% confidence

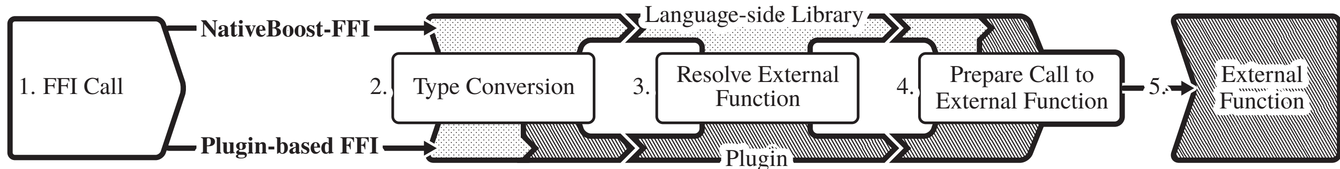


Figure 4: Comparison of FFI calls propagation in NativeBoost-FFI and a typical VM plugin-based implementation. NativeBoost resorts to VM-level only for the native-code activation, whereas typical implementations cross this barrier much earlier.

interval in a gaussian distribution. To exclude the calling and loop overhead we subtract from each evaluation the time spent in the same setup, but without the FFI call. The final deviation displayed is the arithmetic average of the measured deviation of the base and the callout measurement.

The three Smalltalk FFI solutions (NativeBoost, Alien, C-FFI) are evaluated on the very same Pharo 1.4 (version 14458) image on a Pharo VM (version of May 5. 2013). For the Lua benchmarks we use LuaJIT 2.0.1. The benchmarks are performed under the constant conditions on a MacBook Pro. Even though a standalone machine could improve the performance we are only interested in the relative performance of each implementation.

Choice of Callouts. We chose a set of representative C functions to stress different aspects of an FFI implementation. We start with simple functions that require little marshalling efforts and thus mainly focus on the activation performance and callout overhead. Later we measure more complex C functions that return complex types and thus stress the marshalling infrastructure.

3.1 Callout Overhead

The first set of FFI callouts show mainly the overhead of the FFI infrastructure to perform the callout.

For the first FFI evaluation we measure the execution time for a `clock()` callout. The C function takes no argument and returns an integer thus guaranteeing a minimal overhead for marshalling and performing the callout.

	Call Time	Relative Time
NativeBoost	492.13 ± 0.73 ms	1.0×
Alien	606.6 ± 1.9 ms	≈ 1.2×
C-FFI	541.77 ± 0.88 ms	≈ 1.1×
LuaJIT	343.0 ± 1.2 ms	≈ 0.7×

Table 3: Speed comparison of an `uint clock(void)` FFI call (see Code 1).

`abs` is a about the same complexity as the `clock` function, however accepting a single integer as argument.

	Call Time	Relative Time
NativeBoost	65.34 ± 0.23 ms	1.00×
Alien	175.77 ± 0.31 ms	≈ 2.69×
C-FFI	148.77 ± 0.21 ms	≈ 2.27×
LuaJIT[9]	2.035 ± 0.015 ms	≈ 0.03×

Table 4: Speed comparison of an `int abs(int i)` FFI call (see Figure 2).

Evaluation. For measuring the calling overhead we chose the `abs` FFI callout. This C function is completed in a couple of instructions which in comparison to the conversion and activation effort of the FFI callout is negligible. In Table 4 we see that NativeBoost is at least a factor two faster than the other Smalltalk implementation. Yet LuaJIT outperform NativeBoost by an impressive factor 30. LuaJIT has a really close integration with the JIT and this is what makes the impressive FFI callout results possible.

3.2 Marshalling Overhead for Primitive Types

The third example calls `getenv('PWD')` expecting a string as result: the path of the current working directory. Both argument and result have to be converted from high-level strings to C-level zero-terminated strings.

	Call Time	Relative Time
NativeBoost	105.29 ± 0.24 ms	1.0×
Alien	1058.7 ± 2.0 ms	≈ 10.1×
C-FFI	282.94 ± 0.24 ms	≈ 2.7×
LuaJIT[10]	97.3 ± 5.1 ms	≈ 0.9×

Table 5: Speed comparison of an `char * getenv(char *name)` FFI call (see Code 2).

As a last evaluation of simple C functions with NativeBoost, we call `printf` with a string and two integers as argument. The marshalling overhead is less than for

[9] Downsampled from increased loop size by a factor 100 to guarantee accuracy.

[10] Downsampled from increased loop size by a factor 10 to guarantee accuracy.

the previous `getenv` example. However, `printf` is a more complex C function which requires more time to complete: it has to parse the format string, format the given arguments and pipe the results to standard out. Hence the relative overhead of an FFI call is reduced.

	Call Time	Relative Time
NativeBoost	371.03 ± 0.51 ms	1×
Alien	1412.37 ± 0.79 ms	≈ 3.8×
C-FFI	605.02 ± 0.23 ms	≈ 1.6×
LuaJIT	202.4 ± 2.1 ms	≈ 0.6×

Table 6: Speed comparison of an `int printf(char *name, int num1, int num2)` FFI call

Evaluation. Table 3 and Table 4 call C functions that return integers for which the conversion overhead is comparably low. However we see that Alien compares worse in the case of more complex Strings. Table 5 and Table 6 show this behavior. For the `getenv` a comparably long string is returned which causes a factor 10 conversion overhead for Alien.

3.3 Using Complex Structures

To evaluate the impact of marshalling complex types, we measure the execution time for a callout to `cairo_matrix_multiply`. In all cases, the allocation time of the structs is not included in the measurement nor their field assignments. Table 7 shows the results.

	Call Time	Relative Time
NativeBoost	79.00 ± 0.27 ms	1.0×
Alien	753.82 ± 0.51 ms	≈ 9.5×
C-FFI	380.8 ± 2.7 ms	≈ 3.6×
LuaJIT	5.66 ± 0.15 ms	≈ 0.07×

Table 7: Speed comparison of an `cairo_matrix_multiply` FFI call (cf. Listing 9)

Evaluation. In Table 7 shows that NativeBoost outperforms the two other Smalltalk implementations.

3.4 Callbacks

Table 8 shows a comparison of `qsort` callouts passing callbacks. Callbacks are usually much more slower than callouts.

	Call Time	Rel. Time
NativeBoost	2300.0 ± 1.1 ms	1.0×
Alien	600.83 ± 0.35 ms	≈ 0.26×
C-FFI	NA	NA
LuaJIT	46.13 ± 0.62 ms	≈ 0.02×
NativeBoost with Native Callbacks	4.98 ± 0.21 ms	≈ 0.002×

Table 8: Speed comparison of a `qsort` FFI call (cf. Listing 10)

Evaluation. The results show that NativeBoost callbacks are currently slower than Alien's ones. This is because Alien relies on specific VM support for callbacks making their activation faster (context creation and stack pages integration). On the opposite, NativeBoost currently uses small support from the VM side and even do part of the work at image side. This `qsort` demonstrates the worst case because it implies a lot of activations of the callback. For each of these calls, NativeBoost creates a context and make the VM switch to it. To really demonstrate that these context switches are the bottleneck, Table 8 also shows the result of doing the same benchmark in NativeBoost but using a native callback i.e. containing native code. We do not argue here that callbacks should be implemented in native code but that NativeBoost support for callback can be optimized to reach Alien's performance at least.

4. NativeBoost-FFI Implementation Details

The following subsections will first focus on the high-level, language-side aspects of NativeBoost, such as native code generation and marshalling. As a second part we describe implementation details of the low-level extensions, such as the NativeBoost primitives and the JIT interaction.

4.1 Generating Native Code

In NativeBoost all code generation happens transparently at language-side. The various examples shown in Section 2 show how an FFI callout is defined in a standard method. Upon first activation the NativeBoost primitive will fail and by default continues to evaluate the following method body. This is the point where NativeBoost generates native code and attaches it to the compiled method. NativeBoost then reflectively resends the original message with the original arguments (for instance `abs:` in the example Figure 2). On the second activation, the native code is present and thus the primitive will no fail but run the native code. Section 4.2.1 will give more internal details about the code activation and triggering of code generation.

4.1.1 Generating Assembler Instructions

Figure 3 shows that NativeBoost relies on AsmJit[11], a language-side assembler. AsmJit emerged from an existing C++ implementation[12] and currently supports the x86 instruction set.

In fact it is even possible to inline custom assembler instructions in Pharo when using NativeBoost. This way it is possible to meet critical performance requirements. Typically Smalltalk does not excel at algorithmic code since such code does not benefit from dynamic message sends.

4.1.2 Reflective Symbiosis

NativeBoost lives in symbiosis with the Pharo programming environment. As shown in the examples in Section 2 and in more detail in Figure 2 NativeBoost detects which method arguments correspond to which argument in the FFI callout. To achieve this, NativeBoost inspects the activation context when generating native code. Through reflective access to the execution context we can retrieve the method's source code and thus the argument names and positions.

4.1.3 Memory Management

NativeBoost supports external heap management with explicit allocation and freeing of memory regions. There are interfaces for `allocate` and `free` as well as for `memcopy`:

```
memory := NativeBoost allocate: 4.
bytes  := #[1 2 3 4].
"Fill the external memory"
NativeBoost memCopy: bytes to: memory size: 4.

"FFI call to fill the external object"
self fillExternalMemory: memory.

"Copy back bytes from the external object"
NativeBoost memCopy: memory to: bytes size: 4.
NativeBoost free: memory.
```

Code 13: Example of external heap management in NativeBoost

Using the external heap management it is possible to prepare binary blobs and structures for FFI calls.

In the previous example Code 13 the `memory` variable holds a wrapper for the static address of the allocated memory. Hence accessing it from low-level code is straight forward. However in certain situations it is required to access a high-level object from assembler. Pharo has a moving garbage collector which means that you can not refer directly to a high-level object by a fixed address.

[11] http://smalltalkhub.com/#!/~Pharo/AsmJit

[12] https://code.google.com/p/asmjit/

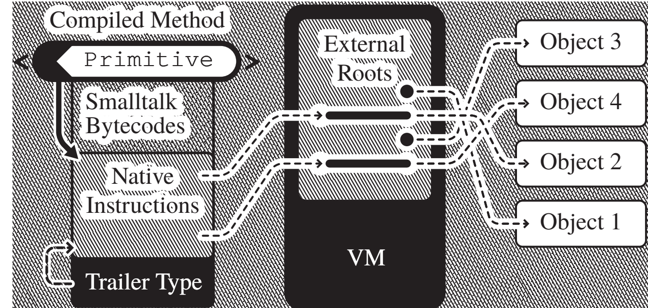


Figure 5: Pointers in a CompiledMethod to objects registered as external roots are pinpointed at fixed offset in global VM-level object.

To deal with this problem the VM has a special array at a known address that contains pointers to high-level objects. The garbage collector keeps this external roots array up to date. Hence it is possible to statically refer to a Pharo object using a double indirection over the external roots. Figure 5 visualizes how native code directly accesses Pharo objects through this indirection.

4.2 Activating Native Code

In this section we present the VM-level interaction of NativeBoost. Even though NativeBoost handles most tasks directly at language-side it requires certain changes on VM level:

- executable memory,
- activation primitives for native code.

Since NativeBoost manages native code at language-side there is no special structure or memory region where native code is stored. Native instructions are appended to compiled methods which reside on the heap. Hence the heap has to be executable in order to jump to the native instructions.

4.2.1 The NativeBoost activation Primitive

In Section 4.1 we explained how NativeBoost creates FFI callouts at language-side. However, so far we left out the part on how the generated native code is activated.

The examples in Section 2, especially Figure 2 show that each NativeBoost FFI callout requires a special primitive. Figure 6 shows how a NativeBoost method is activated.

- In the first step (cf. ❶) the NativeBoost callout primitive is activated. The primitive checks if the compiled method actually contains native code.
- On the first activation there is no native code available yet. Hence the primitive will fail and the Smalltalk body (cf. ❷) of the NativeBoost method

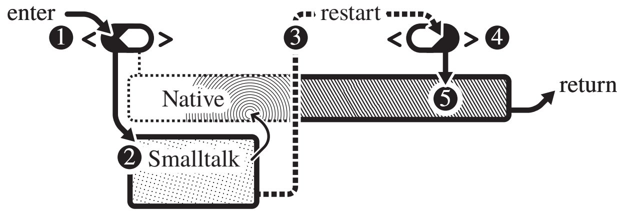


Figure 6: Native code activation. The first call triggers the code generation. Then the method is restarted and the native code executed.

gets evaluated. This is where NativeBoost prepares the native code for the FFI callout.

- After installing the native code in the method trailer, the NativeBoost method is reactivated with the original arguments (cf. ❸).
- Again we end up in the NativeBoost activation primitive (cf. ❹). However, this time there is native code (cf. ❺) available and thus the primitive jumps to the native code instead.

5. Related Work

Typical Smalltalk system are isolated from the low-level world and provide only limited interoperability with C libraries. However there are notable exceptions: Étoilé and Smalltalk/X.

Chisnall presents the Pragmatic Smalltalk Compiler [3], part of the Étoilé project, which focuses on close interaction with the C world. The main goal of this work is to reuse existing libraries and thus reduce duplicated effort. The author highlights the expressiveness of Smalltalk to support this goal. In this Smalltalk implementation multiple languages can be mixed efficiently. It is possible to mix Objective-C, Smalltalk code. All these operations can be performed dynamically at runtime. Unlike our approach, Étoilé aims at a complete new style of runtime environment without a VM. Compared to that, NativeBoost is a very lightweight solution.

Other dynamic high-level languages such as Lua leverage FFI performance by using a close interaction with the JIT. LuaJIT [1] for instance is an efficient Lua implementation that inlines FFI calls directly into the JIT compiled code. Similar to NativeBoost this allows one to minimize the constant overhead by generating custom-made native code. The LuaJIT runtime is mainly written in C which has clearly different semantics than Lua itself.

On a more abstract level, high-level low-level programming [4] encourage to use high-level languages for system programming. Frampton et al. present a low-level framework which is used as system interface for Jikes, an experimental Java VM. However their approach focuses on a static solution. Methods have to be annotated to use low-level functionality. Additionally the strong separation between low-level code and runtime does not allow for reflective extensions of the runtime. Finally, they do not support the execution and not even generation of custom assembly code on the fly.

QUICKTALK [2] follows a similar approach as NativeBoost. However Ballard et al. focus mostly on the development of a complex compiler for a new Smalltalk dialect. Using type annotations QUICKTALK allows for statically typing methods. By inlining methods and eliminating the bytecode dispatch overhead by generating native code QUICKTALK outperforms interpreted bytecode methods. Compared to Waterfall, QUICKTALK does not allow to leave the language-side environment and interact closely with the VM.

Kell and Irwin [5] take a different look at interacting with external libraries. They advocate a Python VM that allows for dynamically shared objects with external libraries. It uses the low-level DWARF debugging information present in the external libraries to gather enough metadata to automatically generate FFIs.

6. Future Work

Even though NativeBoost shows good overall performance when it comes to callbacks it does not keep up with other Smalltalk-based solutions. In the current development phase not much attention was payed to callback performance as it is not a common use case for FFI callouts. Fast callbacks require close interaction and specific modifications at VM-level. However, initially NativeBoost kept the modifications to the VM at a minimum. We assume that we can reach the same performance as Alien relying on the same low-level implementation.

As a second issue we would like to address the callout overhead by using an already existing JIT integration of NativeBoost. Currently the VM has to leave from JIT-mode to standard interpretation mode when it activates an NativeBoost method. This context switch introduces an unnecessary overhead for an FFI callout. A current prototype directly inlines the native code of a NativeBoost method in the JIT. Hence the cost for the context switch plus the cost of activating the NativeBoost callout primitive can be avoided.

7. Conclusion

In this paper we presented NativeBoost a novel approach to foreign function interfaces. Our approach relies only on a very generic extension of the VM to allow for language-side code to directly call native instructions.

Using a in depth evaluation of NativeBoost comparing against two other Smalltalk FFI implementations

and LuaJIT we showed in Section 3 that our language-side approach is competitive. NativeBoost reduces the callout overhead by more than a factor two compared to the two closest Smalltalk solutions.

Compared to LuaJIT there is still space for improvements. We measured a factor 30 lower calling overhead due to a close JIT integration. However for typical FFI calls the absolute time difference between NativeBoost and Lua is roughly 30%. With a partial solution ready to integrate NativeBoost closer with the JIT we expect to come close to Lua's performance.

Furthermore we showed that NativeBoost essentially combines VM-level performance with language-side flexibility when it comes to marshal complex types. New structures are defined practically at language-side and conversion optimizations are added transparently.

Acknowledgments

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011-IRSES.

References

[1] LuaJIT FFI Library. `http://luajit.org/ext_ffi.html`.

[2] M. B. Ballard, D. Maier, and A. W. Brock. QUICKTALK: a smalltalk-80 dialect for defining primitive methods. *SIGPLAN Not.*, 21(11):140–150, June 1986.

[3] D. Chisnall. Smalltalk in a C world. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '12, New York, NY, USA, 2012. ACM.

[4] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, Eliot, and S. I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 81–90, New York, NY, USA, 2009. ACM.

[5] S. Kell and C. Irwin. Virtual machines should be invisible. In *VMIL '11: Proceedings of the 5th workshop on Virtual machines and intermediate languages for emerging modularization mechanisms*, page 6. ACM, 2011.

Pragmatic Visualizations for Roassal: a Florilegium

Mathieu Dehouck[1] Usman Bhatti[1] Alexandre Bergel[2]
Stéphane Ducasse[1]

[1]RMoD, INRIA Lille Nord Europe, France

[2]Department of Computer Science (DCC), University of Chile, Santiago, Chile

[3]Synectique, Lille, France

ABSTRACT

Software analysis and in particular reverse engineering often involves a large amount of structured data. This data should be presented in a meaningful form so that it can be used to improve software artefacts. The software analysis community has produced numerous visual tools to help understand different software elements. However, most of the visualization techniques, when applied to software elements, produce results that are difficult to interpret and comprehend.

This paper presents five graph layouts that are both expressive for polymetric views and agnostic to the visualization engine. These layouts favor spatial space reduction while emphasizing on clarity. Our layouts have been implemented in the Roassal visualization engine and are available under the MIT License.

1. INTRODUCTION

Software analysis and reverse engineering large software systems are known to be difficult [DDN02]. Visualizing software eases analysis by using cognitive abilities to understand software and identify anomalies [Die07]. Visualizing software elements as a graph is a natural visual representation commonly employed:

- Graphs are relatively cheap and easy to visualize due to the amount of available dedicated libraries (*e.g.,* D3[1], Raphael[2]).
- Graphs are a structure effective to represent many different aspects of a software, including control flow and dependencies between structural elements.

Visualization techniques are known to be effective at analyzing package dependencies, correlating metric values, package connectivity and cycles, package evolution or the common usage of package classes (*e.g.,* [DLP05, LDDB09, vLKS+11]). A large body of existing work on software understanding is based on visualization approaches [HMM00, SvG05], in particular, on node-link visualizations [SM95, CIK03, KD03, HSSW06]. On one hand, some researchers explored matrix-based representation of graphs [HFM07, AvH04] or of software [MFM03] and its evolution [VTvW05]. On another hand, important progress has been made to support navigation over large graphs and to propose scalable and sophisticated node-link visualizations for visualizing the connectivity graph of software entities [GFC05, HSSW06, Hol09].

Roassal[3] [BCDL13] is a visualization engine for the Pharo language[4] [BDN+09]. The question here was not to invent a new way of representing information, but to find relevant existing layouts and to implement them in Roassal, alongside Roassal layout such as grid, circle, tree. The novelty of our approach is that even when the nodes do not have same size they are drawn correctly.

We have thus proposed five new graph layouts, each one focusing on a particular aspect: the *radial-tree* focuses on representing hierarchies, while with regular trees the root is repulsed to the top, radial-tree keeps the root at the center of the visualization. *Force-based* layout allows one to represent cyclic graphs such as dependency graphs. The *compact tree* family is just another implementation of trees using the same algorithm as radial-tree so that it saves space for large hierarchies. The *reversed radial tree* layout is another way of representing hierarchies where the position of an element does not depend on its depth but on its distance from the bottom of the graph. The *rectangle-packing* layout is an implementation of a rectangle packing algorithm to allow representing a lot of elements of different sizes in a reasonably restricted space.

To avoid confusion, we define terms used in this paper. A *layout* is an algorithm that determinate position of the graphical elements contained in a visualization following some particular constraints. A *node* or *vertex* is a basic element of a graph, typically in software analysis a package, a class or a method. An *edge* or a *link* represents a relation between two nodes, typically inheritance, composition or call. A *tree* is an acyclic directed graph, for example a simple object hierarchy. A *root* node is an entry point from which all the nodes are reachable by transitivity, typically the superclass of a class hierarchy.

In Section 2 we will introduce the problem and then in Section 3 describe the different layouts we have implemented, explaining for each the intention we had, the problems we encountered, the way we solved them and the limits of our solution.

[1] http://d3js.org

[2] http://raphaeljs.com

[3] http://objectprofile.com/#/pages/products/roassal/overview.html

[4] http://www.pharo-project.org

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

2. PROBLEM DESCRIPTION

In reverse engineering we deal with old and complex systems that are not understood easily. Moose provides powerful tools to analyse these pieces of software and we end up with a large amount of data and metrics. But it is hardly more understandable, thus we need a way of having a quick and smart overview of the relevant information.

The solution is to map the most important textual information to graphical features, and to organize them to be easily readable. The aim of the layouts is to organize this visual information. We may have to represent various kinds of data and we may want to focus on different features, it is therefore necessary to have several layouts which will organize data.

The main constraint is computing duration, hence the choice of a pragmatic answer. For example, we give to the rectangle packing layout a desired size for the resulting rectangle and do not really compute the optimal arrangement which would have minimized the surface because it is time consuming.

3. A FLORILEGIUM OF VISUALIZATIONS

In this section we present some algorithms we added to Roassal. In particular we present the general intention of the algorithm, the main challenges it poses and the solutions we chose.

For each algorithm proposed here, we show the resultant layout with the Collection class hierarchy of Pharo: there are 131 classes in this hierarchy.

3.1 Radial Tree

Intention.

When dealing with inheritance it is natural to have large trees, and the problem with regular tree representation is that the root is repulsed to the top of the visualisation. Sometimes we want to avoid that, and to keep the root amidst the visualisation. This is the aim of the radial tree.

Difficulties.

There are several difficulties when drawing a radial tree.

- **Parent position node.** Firstly we had to choose if the position of a parent node would influence its children nodes position, or if the parent node position would be influenced by its children nodes position.
- **Supporting interaction.** Another constraint was that in Roassal we do not just want to represent data, but we also want to interact with them, so it was important to have an airy representation. This was the problem encountered with the old implementation, the representation was so compact that it was not possible to interact properly with the nodes.
- **Algorithm selection.** The last problem and maybe the most important one, is what kind of algorithm should we use to compute nodes position. If we choose to compute directly radial position for each node, as a circle has a finite perimeter then we take the risk of having to displace each node several times, that gives a complexity in $O(n^2)$, and we can do better for such layout.

Solutions.

We propose a solution inspired by the modified version of Reingold-Tilford algorithm [BJL02]. We compute node position beginning at the leaves and then we ascend the tree to the root, displacing subtrees when they overlap. We do this in a Cartesian coordinate system, with some minor modifications to nodes position so that the radial tree looks nice at the end: typically the space between nodes depends on the layer they belong to. And then we transform our regular tree into a radial one wrapping the layers around the root (Figure 1).

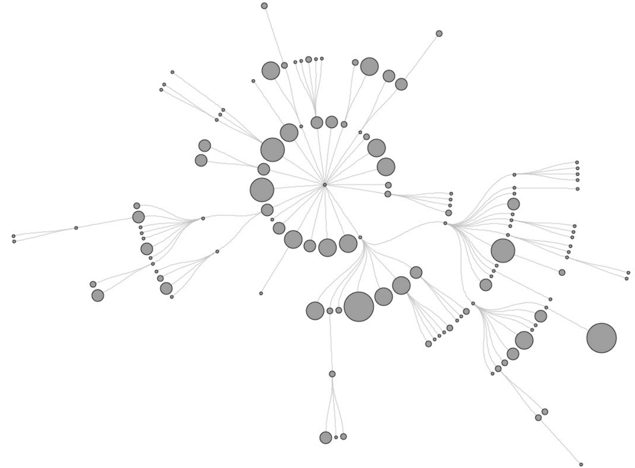


Figure 1: Radial-Tree Layout

Limits.

This layout is interesting for visualizing hierarchies, since we want to interact with the nodes, there must be enough space between them, and so when there are many nodes on a layer, then the tree has an enormous diameter and the root remains all alone in the middle of the visualization, far from its children. This is the main drawback of the radial-tree layout.

3.2 Force Based

Intention.

When dealing with methods invocations or module dependencies, trees are seldom encountered due to cyclic connections. And sometimes it does not make sense to give more importance to a node in particular, so a tree layout is not always appropriate. The force based layout considers nodes as repulsive charges and links as springs, then we have a representation which respect nodes connectivity.

Difficulties.

The main problem of force-based layout algorithms is their temporal complexity which is considered to be $O(n^3)$ for the most trivial implementations, as each iteration has a quadratic complexity (we compute force action for each pair of nodes) and we must iterate enough times (which is thought to be of the same order as the number of nodes) to reach a local minimum. And since the goal is to represent big graphs, it is necessary to have a less time consuming algorithm.

Solutions.

Our solution is inspired by D3 Javascript library implementation and the FADE algorithm [QE01]. Quadtrees reduce the number of

calculi at each step and thus give a *O(n log(n))* complexity. It is also possible to specify charge for a particular node, strength of a link, gravity center. Our force based layout is highly parametrisable, so that it is possible to focus on different aspects of a system (see Figure 2).

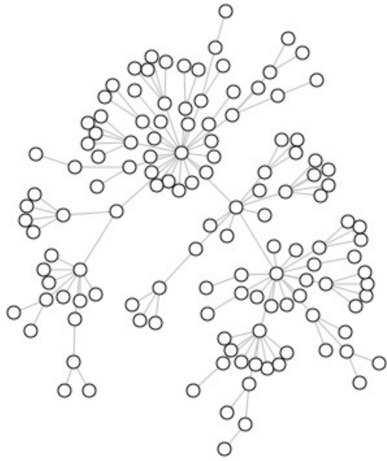


Figure 2: Force-Based Layout

Limits.
Here the limit is the running time. Even with a complexity in *O(n log(n))*, large graphs takes much longer, and then it may be difficult to use it in live.

3.3 Compact Tree

Intention.
There were already tree layouts in Roassal, but they make large graphs since they only keep track of the biggest abscissa where a node has been set. Thus our goal here was to have a less space consuming algorithm, which permits us to draw condensed tree when there is not much space.

Difficulties.

- **Vacant position.** A trap in this kind of algorithm is that we need to know for each layer the abscissa where we can set nodes, this can be done multiple ways, but the trivial way consists of checking all the previously set nodes, and then you have a complexity in $O(n^2)$, which is a loss of time in this case since trees can be drawn with a smaller complexity.
- **Node shifting.** Then as this algorithm is recursive, when setting a child node we do not know where the parent node will be set, and then when setting the parent node, sometimes it occurs that we have to move children nodes, and here the trivial solution has also a complexity in $O(n^2)$.

Solutions.
Here we also use a Reingold-Tilford like algorithm with some improvements such as pointers for left-most and right-most children of a node. This is done so that we do not look at all the previously set nodes when we need to know where we can put a node. When placing a new node, we just skim the contour of the graph (the right-most and the left-most nodes of each layer) and it is less time consuming. In the same way, when we have to move children nodes to correspond to their parent node position, instead of moving them each time, the parent keeps a "modification" value, that spreads to the children when they are drawn, once again it saves time (see Figure 3).

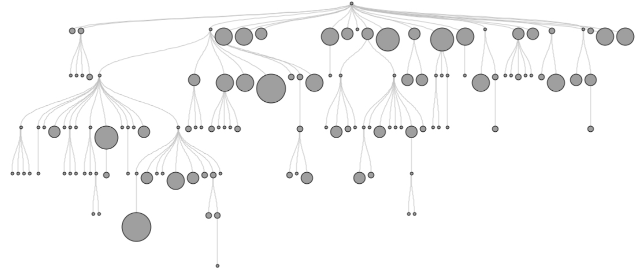


Figure 3: Vertical Compact Tree Layout

Limits.
Our solution is pragmatic thereby computed tree is not the narrowest since even when children order is not important the tree is drawn as if the children were ordered. But then it would be necessary to go through the hierarchy several times to sort the nodes in order to have the narrowest tree, and it would have a high complexity.

3.4 Reversed Radial Tree

Intention.
The reversed radial tree layout is another tree layout for hierarchy representation, but when most of the trees focus on the distance between the nodes and the root, the reversed radial tree layout focuses on the position of nodes compared to the whole tree, thus leaves are on the border regardless of their distance from the root.

Difficulties.
There are no real difficulties for the reversed radial tree layout. It is just important to avoid useless route in the graph.

Solutions.
We skim the tree from the leaves to the root, recording for each node the maximum distance to the leaves in the subtree induced by the node. And then as we have the list of leaves, we compute nodes position from the leaves to the root (see Figure 4).

Limits.
Here we have the same kind of problems as with the radial tree. As leaves are all on the border of the visualization, with many leaves, the diameter of the visualization may be large and the visualization may be almost empty, we will have lots of nodes on the border (the leaves) and very few nodes in the circle, with long edges between them.

3.5 Rectangle Packing

Intention.
Sometimes we want to represent a lot of elements of different

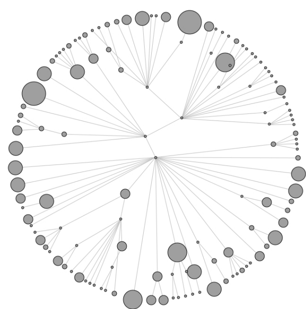


Figure 4: Reversed radial tree Layout

sizes and a grid layout is not always a good choice as it does not use the visual space efficiently. The goal of the rectangle packing layout is to show many elements of various sizes in the available restricted visual space.

Difficulties.
The problem of rectangle packing is NP-hard, that means that we cannot find a solution in polynomial time but we cannot afford excessive computation time.

Solutions.
Here our solution is very pragmatic: instead of looking for the arrangement that will minimize the surface occupied by the elements, we provide the layout a ratio (2/3 by default), which corresponds to the width divided by the height of the rectangle we want to fill with our elements (Figure 5). Then the layout starts placing the elements and resizes the containing rectangle until it has succeed in placing every element.

Limits.
There are two limits:

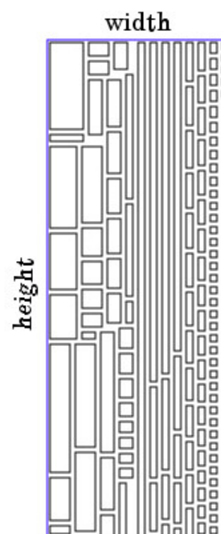
- **Running time.** Even without looking for the best arrangement, it is time consuming. Thus it is difficult to apply it on a large number of elements.

- **Biggest elements.** We are dependent of the biggest elements (typically the longest and the widest). Sometimes we have little elements and a few big ones, then if we ask for a shape oriented in the other direction as the big ones, we will not have it. In our example, we provided the ratio 1/1 since we wanted to arrange elements in a square, but as there is a very long and thin one, we do not have a square at all, but a thin rectangle.

  Here we may raise the question of node resizing which is a touchy one, since we may break the sense originally provided by node size. And then, how do we resize nodes? Do we resize all the nodes, or only the biggest ones?

4. CONCLUSION

For large amounts of data, Roassal and similar visualization engines need to find a pertinent representation so that data are presented in a meaningful form and understood by the end users. In this paper, we have presented five graph layouts that are both expressive for polymetric views and agnostic to the visualization engine. The layouts favor spatial space reduction while emphasizing



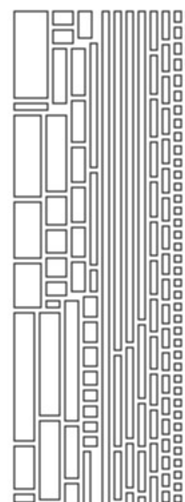


Figure 5: Rectangle Packing Layout

on clarity. Our solution is tractable and diverse, as the variety of layouts allows to analyze data in various forms. It should be noted that even with good layouts, if the amount of information is too big then it is hardly understandable, and the user has to himself select the most relevant information to be shown. We can make our layouts even more customisable, by for example proposing nodes staggering which can sometimes be a good way of saving even more space.

Acknowledgements.
This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011- IRSES MEALS.

This work has been partially funded by Program U-INICIA 11/06 VID 2011, grant U -INICIA 11/06, University of Chile, and FONDECYT project 1120094. We thank the support from the Plomo INRIA associated Team.

5. REFERENCES

[AvH04] James Abello and Frank van Ham. Matrix zoom: A visual interface to semi-external graphs. In *10th IEEE Symposium on Information Visualization (InfoVis 2004), 10-12 October 2004, Austin, TX, USA*, pages 183–190. IEEE Computer Society, 2004.

[BCDL13] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[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.

[BJL02] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. Improving walker's algorithm to run in linear time. In *Revised Papers from the 10th International Symposium on Graph Drawing*, GD '02, pages 344–353, London, UK, UK, 2002. Springer-Verlag.

[CIK03] Neville Churcher, Warwick Irwin, and Ron Kriz. Visualising class cohesion with virtual worlds. In *APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation*, pages 89–97, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[Die07] Stephan Diehl. *Software Visualization*. Springer-Verlag, Berlin Heidelberg, 2007.

[DLP05] Stéphane Ducasse, Michele Lanza, and Laura Ponisio. Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 70–77. IEEE Computer Society, 2005.

[GFC05] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135, 2005.

[HFM07] Nathalie Henry, Jean-Daniel Fekete, and Michael J. McGuffin. Nodetrix: a hybrid visualization of social networks. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1302–1309, 2007.

[HMM00] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[Hol09] Danny Holten. *Visualization of Graphs and Trees for Software Analysis*. PhD thesis, Computer science department, 2009. ISBN 978-90-386-1882-1.

[HSSW06] Holt, Schürr, Sim, and Winter. Gxl: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, April 2006.

[KD03] Said Karouach and Bernard Dousset. Visualisation de relations par des graphes interactifs de grande taille. *Journal of ISDM (Information Sciences for Decision Making)*, 6(57):12, March 2003.

[LDDB09] Jannik Laval, Simon Denier, Stéphane Ducasse, and Alexandre Bergel. Identifying cycle causes with enriched dependency structural matrix. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Lille, France, 2009.

[MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–ff. IEEE, 2003.

[QE01] Aaron Quigley and Peter Eades. Fade: Graph drawing, clustering, and visual abstraction. In *Proceedings of the 8th International Symposium on Graph Drawing*, GD '00, pages 197–210, London, UK, 2001. Springer-Verlag.

[SM95] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275–284. IEEE Computer Society Press, 1995.

[SvG05] Margaret-Anne D. Storey, Davor Čubranić, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis'05: Proceedings of the 2005 ACM symposium on software visualization*, pages 193–202. ACM Press, 2005.

[vLKS+11] Tatiana von Landesberger, Arjan Kuijper, Tobias Schreck, Jörn Kohlhammer, Jarke J. van Wijk, Jean-Daniel Fekete, and Dieter W. Fellner. Visual analysis of large graphs: State-of-the-art and future research challenges. *Comput. Graph. Forum*, 30(6):1719–1749, 2011.

[VTvW05] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. Cvsscan: visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2005. ACM.

Part II
Short papers

The goal of the workshop is to create a forum around advances or experience in Smalltalk and to trigger discussions and exchanges of ideas. Participants are invited to submit research articles.
Short papers are position papers describing emerging ideas or ongoing works at an early stage.

Identifying Equivalent Objects to Reduce Memory Consumption

Alejandro Infante, Juan Pablo Sandoval, Alexandre Bergel

Department of Computer Science (DCC)
University of Chile, Santiago, Chile

ABSTRACT

Executing an application may trigger the creation of a large amount of objects. For many applications, a large portion of these objects are unnecessary and their creation could simply be avoided.

We describe a lightweight profiling technique to identity "equivalent" objects. Such equivalent objects are simply redundant and may be shared or reused to reduce the memory footprint. We propose *object-centric execution blueprint*, a visual representation to help practitioners identify cases where objects may be reused instead of being redundant.

1. INTRODUCTION

Garbage collection alleviates the programming activity by delegating the burden of memory deallocation to the virtual machine. The advances of memory models and garbage collectors have significantly reduced the cost of managing memory. Benefits of garbage collection are tremendous: software programs are easier to write and are likely to have less memory-related problems than when written in plain C or C++. However, an excessive use of garbage collection may have a significant impact on the application performance. Creating, initializing and destroying an object consume execution time and memory space.

Current object-oriented programming languages "make it too easy" to create objects. Consider the following code example, inspired by one of our case studies:

```
"Version 1 of Builder"

Builder>>createNode
	^ GraphicalElement new color: self defaultColor.

Builder>>defaultColor
	"Gray color"
	^ Color r: 0.5 g: 0.5 b: 0.5
```

Pharo is an object-oriented language and environment for the classic Smalltalk-80 programming language [1].

[1] http://pharobyexample.org

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

The class `Builder` creates a new colored graphical element when receiving the message `createNode`. The method `defaultColor` creates an instance of the class `Color`. In Pharo, the class `Color` is defined as immutable: `Color` does not provide mutators for its instance variables and any attempt to modify it raises an error. Once instantiated, the value of a color cannot be modified.

The version of the class `Builder` given above is clearly suboptimal since a new color object is associated to each element and all these color objects are equals. In Pharo, a color object weighs 36 bytes.

We found that each ∼18,000 color object creation initiates a garbage collection, thus incurring the inconvenience of garbage collecting the memory (*e.g.,* pause in the program execution, lack of reactiveness in case of CPU intensive activity).

A possible improvement of the `Builder` class may be (difference is indicated in **bold**):

```
"Version 2 of Builder"

Builder>>createNode
	^ GraphicalElement new color: self defaultColor.

Builder>>defaultColor
	"Gray color"
	defaultColor ifNil: [ defaultColor := Color r: 0.5 g:
	0.5 b: 0.5 ].
	^ defaultColor
```

In this Version 2, `Builder` is augmented with a new instance variable called `defaultColor`. This variable acts as a memorization cache to keep a unique reference of the default color.

Identifying the necessary changes to move from Version 1 to Version 2 of `Builder` does not present any significant challenge on this contrived example. However, typical object-oriented programs create and destroy a large number of objects. Identifying the objects that are unnecessarily created or destroyed too early presents some challenges[**?**]. Identifying places of redundant object creations is not trivial in many cases. It often requires a deep knowledge of the program intent and implementation. Unfortunately, traditional memory profiling tools do not give any indication about whether objects are redundant or not. As discussed in the related work section, traditional memory analyzers are limited to providing metrics about the heap consumption.

This paper is about a profiling technique to help software engineers identify situations for which reusing or sharing an object is beneficial.

This paper presents a lightweight profiling technique that

identifies equivalent objects, intended to be shared to reduce the memory footprint. Our profiler identifies for a given program execution objects that are both non-mutable and are structurally equals. Equality is verified by comparing object snapshots, a kind of hash value that does not rely on the object identity.

Our profiler is accompanied with *Object-Centric Execution Blueprint*, a visual representation of the memory consumption to help practitioners identify sets of equivalent immutable objects that may safely be replaced by one representative shared object. We have successfully used the blueprint to detect and remove a number of redundant objects in a Pharo real world application.

The paper is structured as follows. Section 2 presents our memory profiling in a nutshell. Section 3 describes a case study we have carried out on the Roassal application. Section 4 presents the visual support given to the practitioner to identify critical situations. Section 5 briefly describes the implementation of our profiler. Section 6 gives an overview of the related work. Section 7 concludes and presents future work.

2. IDENTIFYING EQUIVALENT OBJECTS

We propose to optimize applications by identifying groups of equivalent objects. Once identified, a group of equivalent objects may be merged into a unique sharable and reusable object. A definition of object equivalence is provided (Section 2.1) and how such equivalence is measured in an application in the Pharo programming language (Section 2.2).

2.1 Object equivalence

Two objects o_1 and o_2 are said to be *equivalent* if all objects pointing to o_1 may instead point to o_2 without affecting the program semantics and execution. We say that o_1 and o_2 are equivalent if:

(a) **o_1 and o_2 are instances of the same class** – This requirement implies that two objects being from different classes are not interchangeable. This requirement is not strictly necessary, meaning that two objects may be inter-changeable even if they have different classes as long as their interface and contract is similar. However, this requirement significantly simplifies our profiling technique.

(b) **both o_1 and o_2 have identical state** – This requirement implies that each pair of corresponding field values in both objects are either a pair of identical values or a pair of references to objects which are themselves equivalents. For instance, if o_1 := `Point x:5 y:4` and o_2 := `Point x:5 y:4` then o_1 and o_2 have identical state, because their field values in both objects are identical.

(c) **both o_1 and o_2 do not mutate once their construction has completed** – *i.e.,* after the control flow has left the `initialize` method. This implies that side effects are permitted up to the point the object is initialized. If an object changes its state after its creation, such object cannot be equivalent to any other object. In practice, an object is initialized within a factory method located on the metaclass. Examples of such factory methods are `new` and `new:`. Sending the message `new` returns an object supposedly initialized (the method `new` invokes `initialize`). We designate a factory method as a class method returning an instance of the class.

(d) **neither o_1 nor o_2 receive the `identityHash` and `==` message** – It forbids any attempt to access the identity of an object. Receiving a message `identityHash` or `==` makes the object receiver not equivalent to any other object. In Pharo, the identity hash value is a value that reflects an internal number in the virtual machine. The reference equality compares two memory locations.

(e) **Neither the creation of o_1 nor o_2 perform any side effect on the executing context** – It implies that during the creation of an object, side effects are allowed only on the object under creation. Any side effect on another object carried out before exiting a factory method makes the object not equivalent to any other object. Our motivation behind this requirement is that if creating an object performs a side effect, then this creation cannot be avoided else the application behavior is not preserving, even if the object is redundant.

The proposed definition of object equivalence is conservative, meaning that (i) if two objects are equivalent, then one of them is redundant and (ii) two redundant objects are not necessarily equivalent.

This definition is similar to the definition of "mergeability" given by Marinov and O'Callahan [2]. Section 6 detail difference and motivate the need for another definition.

2.2 Profiling

We have built a profiler that identifies groups of equivalent objects. During an execution, our profiler stores in a global table recorded information for each object created in the profiled application. The profiler knows for each object its bit of "history" to determine after the program execution whether that object is equivalent to other objects.

More specifically, our profiler records for each object (i) the number of times it has mutated after having left a factory method and (ii) whether it has received the message `identifyHash`.

In Pharo, everything is an object, and everything happens by sending messages. Nevertheless, certain messages are byte coded by the compiler and no lookup is performed. This is the case of the `==` method. Because of this, detecting when an object receives the message `==` is difficult and, in fact, it is unsolved issue.

After a profiling, it compares all objects and categorizes them in:

- *groups of equivalent objects* – All objects in these groups have the same final state and they did not mutated during the execution after their creation, it means that they had the same state during the execution.
- *groups of near-to-be equivalent objects* – All objects in these groups partially meet our object equivalent definition. We consider that two object are near-to-be equivalent if they do not meet some requirements, for instance, without meeting requirement (c) and (d).

3. CASE STUDY

We have carried out an analysis of the Roassal application and identified a number of situations in which objects have been unnecessarily created.

Roassal. We have analyzed Roassal, an agile visualization engine[2]. Roassal allows one to build sophisticated visualizations, pluggable for any arbitrary domain model. Many objects are involved in a typical Roassal visualization. Each visual element comes with a web of interconnected objects to offer support for interaction and representation.

Excessive use of memory is a barrier from making Roassal scalable: visualizations get slower and less responsive. In addition, by being realized within the virtual machine, the garbage collection overhead does not explicitly appear in a profiling report.

Equivalent objects. Roassal comes with a large amount of tests. The test coverage of Roassal is about 80%, giving us confidence that a fair portion of Roassal features are exercised by unit tests. We have profiled the execution of Roassal unit tests and extracted the following information.

Running Roassal unit tests produces 112,513 objects, instances of Roassal classes. Our profiler has identified that 10.97% of these objects are redundant with the remaining 89.03% of the objects. These 10.97% represents the portion of objects that are unnecessary, and thus the possible gain of the reduction of the object construction.

The largest group of equivalent of objects we have identified is made of instances of the class `RONullShape`. Running the tests of Roassal instantiates this class 13,343 times for which 11,777 objects are equivalent between them. This result means that 11,776 objects are simply unnecessary.

The second largest group is made of all instances of the class `RODraggable`. The 3,027 instances of this class are all equivalent, indicating the need of a singleton pattern.

Improvement of Roassal. We went through some of the group of equivalent objects mentioned in the previous section and refactored Roassal accordingly. We have reduced the amount of created objects by 5.1%. The total amount of objects went from 112,513 to 106,806. This 5.1% of reduction represents a gain of 45Kb approximately, leading to a reduction of 1.4% of the memory consumption.

We have refactored Roassal by implementing singleton patterns on various classes. The class `RODraggable` has been refactored as follows:

```
RODraggable class>>elementToBeAdded
   instance ifNil: [ instance := self new ].
   ^ instance
```

The singleton pattern is implemented in **bold**.

4. VISUAL SUPPORT

Object-centric execution blueprint is a visual aid to identify groups of equivalent (and therefore redundant) objects. We use a polymetric view [1] for that purpose, since we relate different metrics for each structural visual element.

Our blueprint is made up of colored boxes and inner boxes and links (Figure 1). Such visual representation of the program execution is obtained after the completion of the execution.

Nesting outer boxes represents classes. Inner boxes represent groups of objects that are either equivalent or near-to-be equivalent (*i.e.,* without meeting requirement (c) and (d), about the mutation).

[2]`http://objectprofile.com/#/pages/products/roassal/overview.html`

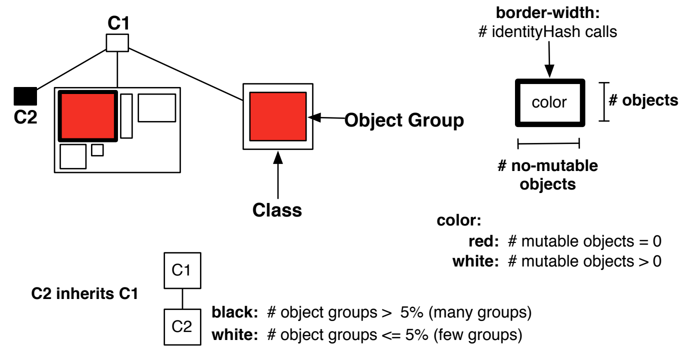


Figure 1: Object-Centric Execution Blueprint

Each object group is characterized with two metrics and two properties:

- height is the amount of (mutable and immutable) objects that belong to the group, using a logarithmic scale;
- width is the amount of immutable objects belonging to the group, using a logarithmic scale;
- presence of a bold border indicates `identityHash` has been invoked on any object of the object group;
- color can be red or white, red if all the objects in the object group do not mutate and white if at least one of them mutate.

An object group painted in red visually indicates that all the objects belonging to this group are equivalent. A white group indicates that the objects are near-to-be equivalent. Such groups may require some further action from the software engineer to make these objects equivalent.

It frequently happens that instances of a class are heterogeneous, which may result in many different groups. Such situations are discarded from the visualization by using a threshold number of groups. In our experiment, we consider a threshold of 5, meaning that groups of a class are shown if at most 5 groups for 100 instances. The purpose of this arbitrary heuristic is to reduce the amount of data that would be difficult to improve.

Figure 1 shows that `C1` has three subclasses. Each of them tells a different story and we need to deal with them in different ways.

From left to right, class `C2` is black filled, meaning that instances of `C2` cannot be grouped into equivalent objects.

The class in the middle has 5 object groups inside, which means that in this experiment the instances of this class can only have 5 possible states. Furthermore, one of these groups is colored red, indicating no mutation occurs for that group. Objects belonging to that group would have been equivalent if they had a non bolded border: the message `identityHash` is sent to the objects of this group.

The last class of the figure shows a single red object group, so all the instances of this class do not mutate and have the same state across all the execution, also nobody called `identityHash` on them. This is an excellent opportunity to use the singleton pattern and reuse a single object to fulfill the job of all the previously used objects.

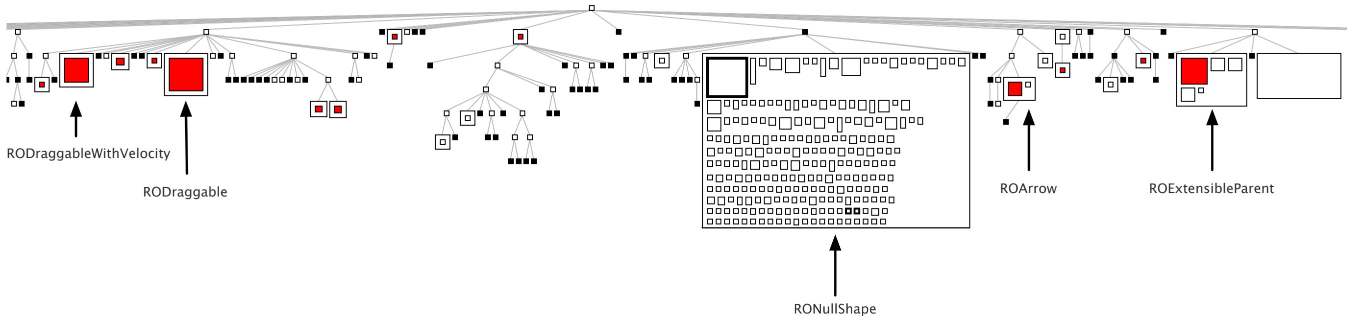


Figure 2: Object-Centric Execution Blueprint before optimization

Also the visualization provides some interactions to provide the user the opportunity go deep into the experiment. Just moving the mouse over a class displays as a tooltip its class name, the number of instances and the memory consumed by its instances. Moving over an object group displays the number of objects, the amount of mutable and non-mutable objects. Furthermore the tooltip displays whether any object in the group caused a side effect on its creation, a requirement settled before reusing the objects.

Clicking the shapes also allows the user to inspect some objects or browse some classes.

Figure 2 shows an excerpt of the visualization for the unit test execution of Roassal.

5. IMPLEMENTATION

We briefly describe the two key ingredients to implement our approach. Our Memory profiler is available under the MIT License[3].

5.1 Gadget profiling

Gadget Profiler[4] is a framework for method instrumentation. It allows the programmer to inject code before and after every method of an automatically set of selected classes. Also it provides some essential information about the execution to be used by the injected code, like the receiver and the arguments of the message.

Memory Profiler is built using Gadget Profiler. Also when a method is called we check if the method is the `identityHash`, performs a mutation or causes an external side effect. Finally, it groups the objects as we described before.

5.2 Snapshotting objects

Keeping track of the side effects may be done in a number of fashions (*e.g.,* keeping track of the write bytecodes, modifying the abstract syntax tree [3]). We employ here a technique based on object snapshotting. We define an *object snapshot* as an integer that represents the complete state of an object. This integer is computed using a *bitXor* operation between the identity hash of attributes and the identity hash of object class.

```
1 Object>>snapshotAsInteger
2   | index value |
3   index := self class instSize.
4   value := self class gadgetIdentityHash.
5   [index > 0]
6     whileTrue:
7       [ value := (value bitShift: 1) bitXor: (self instVarAt:
      index) gadgetIdentityHash.
8         index := index − 1].
9   ^ value
```

An object snapshot is useful to compare objects states. Comparing objects states we can detect: (i) objects that have the equivalent state, and (ii) if an object has a different state after a method execution. Both features are essential to detect equivalent objects.

6. RELATED WORK

Marinov *et al.* presented Object Equality Profiling (OEP), a profiling technique to discover opportunities for replacing a set of equivalent object instances with a single representative object [2].

Their tool performs a dynamic analysis that records all the objects created during a particular program run. The tool partitions the objects into equivalence classes, and uses collected detail timing information to determine when elements of an equivalence class could have been safely collapsed into a single object. They use an instrumentation byte code technique to record fine-grained information. They insert instrumentation at the following program points: allocation sited for objects and arrays, field writes, array element writes, field reads, array element reads among others. Adding a considerable overhead.

Our object snapshot technique takes snapshots before and after a method execution, and saves the last state of the objects (the last snapshot) causing a significantly lower execution overhead. And having a trade-off between overhead and accuracy. We also propose *object-centric execution blueprint* as a visual aid to detect, understand, and delete redundant object.

7. CONCLUSION & FUTURE WORK

Currently, the large majority of code profilers and debuggers use inadequate abstractions in their analysis. We believe this is a critical situation and hope the tool and ideas presented in this paper will contribute to addressing it.

Thanks to the analysis above, we eliminated more than 5.1% of the identified unnecessary objects refactoring code using singleton pattern, but working on the analysis and the abstraction we expect to categorize possible source code

[3]`http://smalltalkhub.com/#!/~ainfante/MemoryProfiler`
[4]`http://smalltalkhub.com/#!/~ainfante/GadgetProfiler`

refactoring to eliminate the totality of redundant objects.

Acknowledgements

Juan Pablo Sandoval Alcocer is supported by a Ph.D. scholarship from CONICYT and AGCI, Chile. CONICYT-PCHA/ Doctorado Nacional/2013-63130199. This work has been partially funded by Program U-INICIA 11/06 VID 2011, grant U-INICIA 11/06, University of Chile, and FONDECYT project 1120094.

8. REFERENCES

[1] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[2] Darko Marinov and Robert O'Callahan. Object equality profiling. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 313–325, New York, NY, USA, 2003. ACM.

[3] Jorge Ressia. *Object-Centric Reflection*. Phd thesis, University of Bern, October 2012.

Part III
Author Index

Alcocer, Juan Pablo Sandoval 45
Bhatti, Usman 65
Bergel, Alexandre 73, 65
Bouraqadi, Noury 11
Bera, Clement 27
Bruni, Camillo 53
Cassou, Damien 45
Dehouck, Mathieu 65
Denker, Marcus 27
Dias, Martín 45
Drey, Zoé 37
Ducasse, Stéphane 11, 45, 53, 65
Fabresse, Luc 11, 53
Infante, Alejandro 73
Le Lann, Jean-Christophe 37
Polito, Guillermo 11
Sandovar, Juan Pablo 73
Schneider, Jean-Philippe 37
Stasenko, Igor 53