

Bootstrapping Reflective Systems: The Case of Pharo

G. Polito^{1,2,*}, S. Ducasse¹, L. Fabresse², N. Bouraqadi², B. van Ryseghem¹

Abstract

Bootstrapping is a technique commonly known by its usage in language definition by the introduction of a compiler written in the same language it compiles. This process is important to understand and modify the definition of a given language using the same language, taking benefit of the abstractions and expression power it provides. A bootstrap, then, supports the evolution of a language. However, the infrastructure of reflective systems like Smalltalk includes, in addition to a compiler, an environment with several self-references. A reflective system bootstrap should consider all its infrastructural components. In this paper, we propose a definition of bootstrap for object-oriented reflective systems, we describe the architecture and components it should contain and we analyze the challenges it has to overcome. Finally, we present a reference bootstrap process for a reflective system and Hazelnut, its implementation for bootstrapping the Pharo Smalltalk-inspired system.

Keywords: Object-Oriented Programming and Design, Bootstrap, Smalltalk, Software Evolution

1. Introduction

Reflective systems are those that reason about and act upon themselves [21]. A causal connection exists between the program and its representation inside the program itself as a meta-program [16]. This reflective architecture introduces self-references: an object-oriented system is composed by objects, which are instances of classes, which are also objects, and so on. These self-references, also known as meta-circularities [6], allow the manipulation of several meta-levels on one infrastructure.

Reflective systems traditionally modify their self-representation to evolve and define new abstractions. However, the self-modification approach of evolution has many drawbacks, such as making difficult the self-surgery operations [5] or the loss of the reproducibility of the system. On the other hand, non-reflective systems develop an evolution approach by recreation. Whenever a change has to be made to the system, a new system is created with the new changes applied. This approach solves many of the drawbacks of the reflective approach.

When the system in construction is introduced into its own recreation process, this process is named a *bootstrap*. The most prominent example is the C compiler which was initially developed in another language, and later rewritten in the C language itself and used to compile itself. In an informal way, a bootstrap is the process to use the system being developed as early as possible to define this exact system. The key idea is to be able to use the new abstractions provided by the new system as early as possible and also to support the modification of the system. This way, the system can benefit from its expressive power and abstractions to evolve.

Bootstrapping a system provides a deterministic, explicit and self-describing process for generating an autonomous reflective object-oriented system. Its direct benefit is the generation of a complete self-description of the system, which can be manipulated by the system itself. The self-description eases the understanding of the system by using the abstractions it provides to define itself. A key advantage is the description of its meta-circularities, the initialization of the system and how they are solved.

*Corresponding author

Email addresses: guillermo.polito@mines-douai.fr (G. Polito), stephane.ducasse@inria.fr (S. Ducasse), luc.fabresse@mines-douai.fr (L. Fabresse), noury.bouraqadi@mines-douai.fr (N. Bouraqadi), benjamin.vanryseghem@gmail.com (B. van Ryseghem)

¹RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1.

²Université Lille Nord de France, Ecole des Mines de Douai.

Besides understanding, bootstrapping supports the evolution of a system. Once bootstrapped, the specification of the system can be easily modified to make possible deep changes to the system. For example, a system may change its object model or layout of objects in memory.

In this paper we explore the bootstrap of object-oriented reflective systems, its challenges, mechanism and benefits. We present a model for bootstrapping a reflective system and a concrete implementation to bootstrap Pharo, a Smalltalk-inspired system. In a previous article, we presented a first attempt to perform such a bootstrap [4]: the current article describes a new approach and a complete rewrite compared to the previous article.

The contributions of this paper are the definition of a bootstrap process for a reflective object-oriented system and a description of its challenges and benefits; and the definition of a bootstrap process model and the description of its implementation when bootstrapping the Pharo system.

Outline. The rest of the paper is structured as follow. In Section 2, we present how a reflective system evolves and the problems this evolutive approach presents. Section 3 describes how non-reflective systems evolve, introducing the case of bootstrapping, its challenges and the properties it provides to a system. Section 4 presents the idea of bootstrapping a reflective object-oriented system and proposes a conceptual model to achieve it. Section 5 presents a concrete implementation of the model we propose stating its challenges and how to overcome them. Section 6 presents the results of our experiments and validations. The subsequent Sections discuss some key points in the model and implementation of Hazelnut. Finally, the paper presents related work and concludes.

2. Problem: Evolving a Reflective System

Reflective systems are those that reason about and act upon themselves [21]. They embed their structure and behavior in themselves in a causally connected way. Causal connections enable a reflective system to modify itself and reflect their changes immediately. The ability to change itself is called *self-modification*.

A system evolves by self-modification when it applies *deltas* directly on itself to alter its representation. A delta is an atomic and indivisible change of the system, such as the introduction of a class or a method, or the assignment of a variable. Deltas can both extend and shrink the state and behavior of the objects in the system. The extension of the system is achieved by adding instance variables, methods and classes, while shrinking is achieved with their removal.

Each delta applied to the system triggers a migration of the entities in the system, which automatically reflect the change. Thus, the system is at any moment during its life synchronized with its internal representation. For example, adding or removing an instance variable from a class definition migrates all its instances by adding them the new extra slot.

To evolve a system to a desired state, a *stream* of deltas in a specific order must be applied to it. Figure 1 shows how a stream of changes $\{\Delta_1, \Delta_2\}$ is applied to version 1 of a reflective system to obtain, in order, version 2 and version 3.

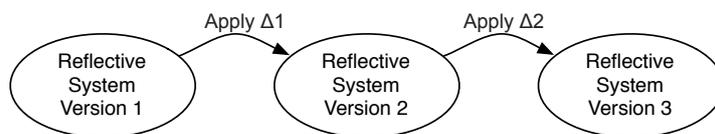


Figure 1: Evolution of a reflective system by self-modification

Pharo [2] and Squeak [1] are object-oriented systems evolving traditionally by self-modification. These systems store their state and all their objects in an *image*. A snapshot of the image can be stored at any time, creating a *backup* of the state of the system. When the image is restarted, its state is the same as it was at the last snapshot. These systems evolve by applying deltas on themselves and making a snapshot of the new state of the system.

Based on our experience maintaining and evolving them during several years [8], we list some problems appearing in these evolutive approach:

Evolving requires sequences of side effects. A stream of ordered and compatible delta have to be applied to the system to reach a particular version of it. A delta could depend on the deltas applied before it, and enable the application of its subsequent ones. It may be difficult to order the deltas of big changes to get the system to a specific state. Consider for example a Smalltalk system with an Array class defining an iterator method do:. This iterator method is used in critical parts of the system. A refactor to move the iteration method to the superclass of the Array class consists basically in the removal of the method from the Array class and the introduction into its superclass. However, removing first the method from the Array class leads to an irrecoverable system crash. To perform this refactor safely, the order in which the actions are realized must be reverted: first the iterator method must be introduced into the superclass. So, when removing the method in the Array class, the method is looked up correctly. Evolution by self-modification lacks a way to apply several changes atomically to a system.

Impossibility to recover from system crashes. When doing self brain surgery in the system [5] (modifying parts of the system that are used to performing the changes themselves), a delta in a given stream may leave the system in a broken state *i.e.*, the removal of a core method in use such as new, or removing a core instance variable required by virtual machine contract such as the format instance variable in Class. This leads to the loss of the already applied deltas. The system relies on its backups or snapshots to recover to a previous state, making difficult and tedious the process to evolve the system to versions involving many critical deltas.

Unmaintained code. The initialization methods of certain classes are not systematically exercised. This makes the code of these initializations get easily broken or obsolete. For example, the character table in the system can be modified by altering it directly without updating its initialization code. Additionally, methods can refer to nonexistent classes or send messages that are not understood any more. Such a situation presents a problem when these parts of the system have to be re-initialized again.

Lack of support for building up the system. The system is represented by a monolithic image containing lots of packages not needed for every usage, such as the UI, networking, debugger or compiler. This monolithic image, due to its size, represents a problem when deploying it on a resource constrained environment such as embedded devices. Building a system with only the necessary parts is nowadays only feasible by image shrinking with self-modification, making the system unstable during its modification. The image has been a big monolithic unit since years, leading to hidden and hard to break dependencies in it, making the shrinking process tedious. Additionally, the dynamic nature of the system and the use of reflective features breaks static analysis when trying to understand the hidden dependencies [15].

3. System Evolution And Bootstrapping

3.1. Evolution of Programming Systems by Recreation

Opposed to reflective systems, non-reflective programming systems evolve by recreation. This means they evolve by the generation from scratch of a new system representing a new version of it. These systems are generated by a system *builder* which takes a *specification* of the system and builds the new system (Figure 2). The specification of the system must include the elements and behavior of the new system in addition of the order in which the system should be built. For example, to build a C compiler another compiler (the system builder) will take the source of the new C compiler (the specification) as input.

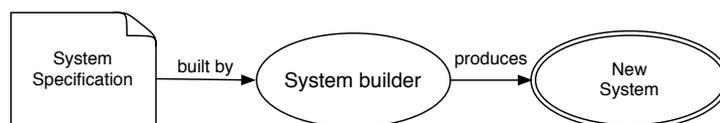


Figure 2: Process to generate a new system.

Figure 3 depicts how a system such as a C compiler can evolve using this approach. At the first stage, the compiler and its source code represent version 1 of the system. A change is introduced in the source code of the compiler, to depict version 2 of the source code. The version 1 of the compiler, which conforms to the version 1 of the source code, is now *out of synchronization* with the new source code or specification. The source code version 2 is used to generate the compiler version 2. Finally, the compiler version 2 conforms to the source code in version 2.

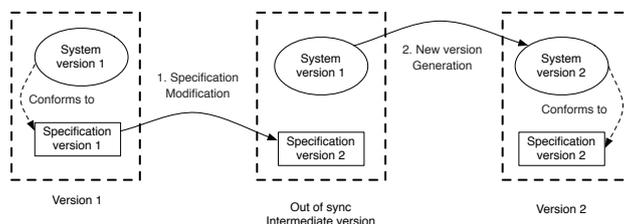


Figure 3: Evolution of a non causally connected system by recreation after each modification of the system specification

This approach provides a deterministic and explicit process to build a new version of the system from scratch. Multiple changes can be applied to the specification while it is out of synchronization with the system. These changes will be applied atomically in the new system when it is generated. On the other hand, the main drawback of this approach is the slow feedback loop in the development process. The specification and the system can stay out of synchronization during a large period of time without providing information about errors and mistakes. Thus, it is easier to reach non-working stages.

A particular case of system recreation is the introduction of the newly generated system into the generation process. The newly generated system can be used as a system builder in the process of generating a new system. This case, commonly known by its usage in compilers, is normally referred as *bootstrapping*.

3.2. Bootstrapped Systems

The concept of *bootstrap* is well known in the context of compilers. For example, a bootstrapped C compiler is a compiler that, by using its own source code, can produce another compiler with its same behavior. We can generalize the process to obtain a bootstrapped system as the following.

Definition 1 (Bootstrap Process). A *bootstrap* is a process that defines a system S using S itself, by providing a self-describing specification of its own structure and behavior. Alternatively, we can say a *bootstrap* is a process that defines a system S using a specification that can be fully processed by S itself.

Once bootstrapped, a system relies only on itself and its specification to re-generate its own infrastructure. Improvements such as bugfixing, speed optimizations and mutations of the language are performed by modifying the specification and recreating the system. We can thus define a bootstrapped system as the following.

Definition 2 (Bootstrapped System). A *bootstrapped system* is a system that supports its own evolution.

3.3. Why bootstrapping is important?

Bootstrapping a system may be perceived as an academic exercise but it has strong software engineering positive properties:

Enforces a self description. A bootstrapped system describes and manipulates itself. Then, the system takes advantage of its own abstractions and concepts to describe its own elements and the process to build them.

Provides an agile and explicit process. A bootstrap provides an explicit description of the system including its elements and building process. It also allows the introduction of agile techniques such as continuous integration, building and testing for the complete system, including its own construction. First, it ensures that the system can always be built from the ground. Second, it ensures that the initialization of key parts of the system is exercised each time the system is built, limiting broken and outdated code and showing up hidden dependencies. Finally, it supports the idea of software construction by selecting and assembling elements.

Warranties initial state. A system built from the ground by using the same specification and builder should have a deterministic behavior and initial state.

Supports explicit malleability and evolution. Having an explicit and operational machine executable process to build a kernel is also important to support a system's evolution [5]. The evolution can be achieved by defining new entities and their relationships with existing ones inside the specification describing the system. There is no need to build transition paths from an existing system to the next version. This is particularly important for radical changes where the migration would be too complex.

3.4. Challenges of Bootstrapping a Reflective Object-Oriented System

Meta-circularities make the process of building a new reflective system more complex than building a system without them. The building process should solve these meta-circularities and provide a working system. Once solved, we say the meta-circularities are *closed*. Meta-circularities in a reflective system can be illustrated with several examples:

Reflective core are expressed in themselves. The core of many reflective and dynamic languages such as Ruby, Smalltalk and even Java are self-referencing. For example, the Metaclass class is an instance of the Metaclass metaclass and vice versa, as shown in Figure 4. This leads to a recursive dilemma: how can each of these two entities be initially built without its defining pair. Due to its importance in the system structure this meta-circularity should be mandatory solved during the first steps of the bootstrap. We call this, the *basic structural meta-circularity* of the system.

Finding the order of construction is complex. A class must receive the new message to create a new object. A method with the same selector is looked up in the class' metaclass hierarchy. Since methods are objects, they have to be instantiated also. The meta-circularity resides in the fact a first new method should exist to define all methods.

Such self-references are also present in other languages with reflective capabilities: *e.g.*, in Java the ClassLoader is a class itself; Ruby presents an object-model very similar to Smalltalk extended with mixins.

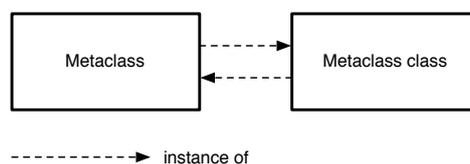


Figure 4: Smalltalk Metaclass circularity.

4. Hazelnut Model For Bootstrapping Reflective Systems

4.1. Bootstrapping a Reflective Object-Oriented System

A reflective object-oriented system is a reflective system where its entities are represented by objects. These objects are causally connected so they are able of reasoning about and acting upon themselves. The causal connections create a peculiar relation between reflective systems and their specification: a reflective object-oriented system contains a *reified* version of its own structure and behavior. For example, they contain meta-objects representing entities such as classes, instance variables, packages or methods. Meta-objects insert the idea of meta-circularity in the reflective system: a meta-object is an object, and it is also described by a meta-object. Introspection and self-modification in reflective object-oriented systems are the result of respectively querying and mutating these meta-objects.

Though reflective object-oriented systems reflect their own structure and behavior, they do not necessarily reflect the process to build and initialize themselves. A bootstrap for an object-oriented reflective system must ensure the meta-circularities of the system are closed and introduce this process in the system. Thus, we can define the bootstrap of an object-oriented reflective system as the following.

Definition 3 (Bootstrap Process for an Object-Oriented System). A bootstrap of an object-oriented reflective system is a process that defines an object-oriented system S using S itself, by reifying the structure and behavior of the system in itself.

Definition 4 (Bootstrapped Object-Oriented Reflective System). A bootstrapped object-oriented reflective system is the object-oriented system resulting from a bootstrap process which is capable of reasoning about and acting upon itself in order to evolve.

In particular, the specification describing an object-oriented reflective system must contain:

The Structure of the system. The entities and connections conforming the structure of the system and their behavior. For example, classes, metaclasses and methods are described to define the entities of the system. Additionally, their relations such as superclass and subclass are described since they are needed to assemble the system.

The initialization order of the system. The specification must express how to initialize the system. Some initializations are coupled, so their order must be specified. For example, an object A must be necessarily initialized before another object B , because B uses A .

4.2. Overview of Hazelnut

Hazelnut is a *system builder* that performs the bootstrap process of an object-oriented reflective system out of an existing reflective system: Pharo. We will refer to Pharo as the *source system* from where the *new system* will be created. Hazelnut takes advantage of the reflective capabilities of the source system to perform tasks such as create classes, compile methods and modify the system structure.

Hazelnut defines both a builder and a specification model that can be used to generate object-oriented reflective systems. The specification describes both the structure and behavior of the bootstrapped system, as well as parts of the process to build it.

The structure and behavior of the system is described as class and method declarations in files. For example, [Figure 5](#) exemplifies the structural specification with an extract of the Point declarations. The Point class is described specifying its superclass, instance variables, and package. The source code of the class' methods are also included in this part of the specification.

```
Class
  name: #Point;
  superclass: #Object;
  instanceVariables: #(x y);
  package: #'Kernel-BasicObjects'.

Class Point >> corner: aPoint
[
  ^ Rectangle origin: self corner: aPoint
]

Class Point >> dist: aPoint
[
  | dx dy |
  dx := aPoint x - x.
  dy := aPoint y - y.
  ^ (dx squared + dy squared) sqrt
]
```

Figure 5: Extract of Point class definition in the structural part of the bootstrap specification

On the other hand, the initialization process of the system is described as a *specification object*. The Hazelnut builder delegates the particular parts of the initialization process of the new system to this specification object. The resulting new system is a reflective object-oriented system, reifying its own structure and behavior in a causal connected way. It will contain all the objects needed to run independently of the source system. The resulting reflective system, again, does not mandatory describe the process to rebuild itself. Hazelnut achieves the creation of the new system by creating a new namespace in the source system. This new space represents the new system. It co-exists side by side with the source system, on top of the same Virtual Machine. Hazelnut takes the specification of the new system and builds all the corresponding objects: each class and object defined in the specification will have an alive counterpart residing in the new system.

The key benefit of having two systems side by side is that objects of the new system are able to receive messages during the construction. Encapsulation and polymorphism ease their manipulation. Without this capability, objects should be manipulated as raw data structures without behavior. This supposes a problem when modifying complex objects such as a Dictionary because all the code to maintain coherent its internal structure should be rewritten.

Regarding the meta-circularities, during the first stages of the construction of the new system, objects in the new system reference *temporary* objects from the source system. Temporary objects are polymorphic with objects in the new system, so the new objects can delegate to them perceiving no difference. Thus, they allow objects in the new system to perform some tasks while still partially initialized. When the new system is ready, references to temporary objects are replaced by references to objects in the new system. The new system is independent of the source system and we consider it bootstrapped. For example, the ObjVLisp [7] bootstrap solves the meta-circularity problems by creating a first temporary version of the class Class (not using inheritance) using low level API. The class Object is created as an instance of the first class Class. Finally, Class is reimplemented using the first one and this time inheriting from Object.

4.3. Bootstrap process of Hazelnut

Hazelnut defines the following process to bootstrap a reflective system, as illustrated in Figure 6. Each of the steps will be fully explained while presenting the Pharo bootstrap in the next section.

- **Step 1: Load specification.** The specification describing the system is read by Hazelnut. Each entity appearing in the specification is reified into Hazelnut as a *specification object*. For example, a class described in the specification will be transformed into a class definition in hazelnut.
- **Step 2: Solve Basic Structural Meta-circularity.** A reflective system meta-level is created by causally connected objects ending up in meta-circularities. This step is responsible for solving the basic meta-circularity of the system, which is the minimal structural part of it enabling the construction of the rest of the system. For example, in order to create a class for a Smalltalk system, the class Metaclass and its class should exist. Thus, this step involves the creation of the first Metaclass, which in the upcoming steps will be used to define new classes. This first structural objects will refer to temporary objects of the source system after this step.
- **Step 3: Build Class Shells.** In this step the skeleton of all classes of the new system are created. These skeletons act as *class shells* since they have the structure of a class, but do not yet contain all the information they need to work. The creation of class shells is delegated to the hazelnut class definitions loaded during the step 1 of the process. The class shells created in this step are partially initialized with temporary objects from the source system and do not have methods installed in them yet. Relations between the class shells –such as superclass/subclass– are not set in this step, so it can be performed independently of any order. Class shells will be filled in the following steps to become fully functional classes.
- **Step 4: Install Methods.** Once all class shells are built, methods are created and installed into their respective owners. Depending on the format of the specification, method building relies on compiling source code or loading pre-compiled binary code. In case of building a system with a different semantics or byte-codes, a cross-compiler inside the source system can be used to compile the methods source code.
- **Step 5: Complete Initialization of the new System.** In this step, the initialization of the new system is completed. The class shells are filled, and the relation between them are set. Other state of the system is

initialized. For example, the objects representing the packages of the system could be created. Initializations of this step are performed by the collaboration of the source and the new system, until the new system is completely independent and isolated from the source system.

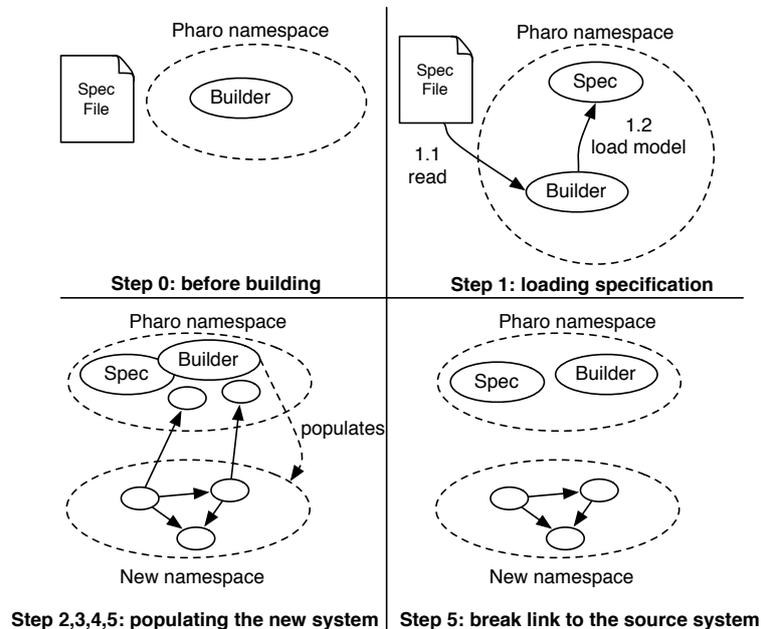


Figure 6: Overview of Hazelnut process to bootstrap a reflective system

5. Hazelnut in Action: Bootstrapping Pharo

We implemented Hazelnut to bootstrap Pharo 2.0. Pharo is a class-based object-oriented environment with the addition of stateless traits [20]. The stateless traits are used to define part of the kernel of the system. This way, Pharo adds more meta-circularities to the ones found in a typical Smalltalk system.

In the following subsections we present the concrete implementation of Hazelnut to bootstrap Pharo 2.0. Prior to building, we need to create a specification for the targeted Pharo system, since Pharo was obtained by successive self-modifications (See section 2). We explain first the process to obtain an explicit textual specification for a Pharo system. Then we describe how Hazelnut implements the bootstrap process steps, emphasizing on how it overcomes the identified challenges.

The presented code snippets are extracted from the Hazelnut sources available under MIT license at <http://rmod.lille.inria.fr/web/pier/software/Seed>. For the sake of clarity, we intentionally skipped some details and changed some names to focus on the most relevant concepts.

5.1. Infrastructure Challenges and Limitations

Besides the meta-circularity challenges appearing when bootstrapping a reflective system, new challenges emerge from the Pharo system. These implementation challenges derive from assumptions hardwired into some Pharo infrastructural components such as the Virtual Machine and the compiler:

Immediate Objects. Immediate objects, such as small integers, are objects that are encoded in references by the Virtual Machine. They are not represented as separate object data structures, and thus, they don't have a class pointer. The Pharo Virtual Machine has an explicit reference to the classes it has to use for immediate objects. These well-known classes are present in the special objects array *i.e.*, an array containing the objects the Virtual

Machine uses to run. For example, the Pharo Virtual Machine hardwires that in the 6th position of the special objects array it will find the `SmallInteger` class. Because of this, the Virtual Machine allows only to have one class for these immediate objects at the same time, and thus, immediate objects from the bootstrapped system cannot be properly manipulated with their corresponding classes.

Well Known Objects. Objects such as `nil`, `false` and `true` are unique and well known by the system. These objects are used by the Virtual Machine to run checks and optimized byte codes for code such as `isNil` and `ifTrue:ifFalse:`. In such cases, the Virtual Machine does not recognize other instances of them and forbids the usage of the equivalent instances of these objects meant for the bootstrapped system.

Compact Objects and Classes. For memory saving purposes, classes with large amount of instances such as `Array`, `CompiledMethod` or `Association` are optimized as *compact classes* in the Pharo Virtual Machine. A compact class is a class whose instances have one header less than normal objects. The Virtual Machine references an array of compact classes. Instead of a direct reference to their class, the objects include inside their base header an index to the corresponding position of their class in the compact classes array. The Virtual Machine only accepts one copy of the compact classes array, and cannot properly treat objects of the bootstrapped system as compact objects.

Assumed Object Structures. The Pharo Virtual Machine hardwires knowledge about the structure of important objects. For example, it assumes that the first three instance variables of classes are the superclass, method dictionary and format, or that a the system process scheduler has an array of linked lists. This assumptions are not easily configurable and limit the existence of different models and the modification of the existing one.

5.2. Step 0: Extracting the new System Specification

The goal of this bootstrap is to generate a reproducible *core* Pharo distribution, including a subset of its original packages. This core distribution aims to support the modularization of the system. Since the focus of the bootstrap process is not to determine the smallest core for a viable system, but just to build it automatically, we understand the performed selection of packages may not be minimal and can be subject of work for dependency analysis tools.

Creating a specification for the new core of Pharo involves figuring out which entities should be part of the system. In addition, it involves extracting their structure and behavior as well as the order and procedure needed to initialize the system. For the core entity selection we used a previous work done by Pavel Krivanek whose objective is to produce a smaller core for Pharo (<https://ci.lille.inria.fr/pharo/view/Pharo-Kernel%202.0/>). It contains only some core packages which are mandatory for a running system. The original Pharo distribution contains 2892 classes and traits which includes a big set of extensions in addition to the core part of the system. Figure 7 depicts an overview of the final selection which contained 461 classes and traits from the original system. Note that the new core does not contain the compiler but loads classes in a binary format using Fuel [9].

<i>Package Name</i>	<i>Number of Classes & Traits</i>
Announcements	8
Collections	79
Files	62
FuelSerializer	66
Kernel	103
Multilingual	38
Traits	17
Others	88

Figure 7: Overview of the selection of the specification

The extraction of the entities was eased by the introspective capabilities of the Pharo system. Pharo, as a Smalltalk-inspired system, provides full access to the internal structure of classes, metaclasses, traits, and methods. This way,

the specification of the system could be extracted into source code files with contents such as the one already shown in [Figure 5](#).

Finally we extracted the initialization of the system. Pharo contains part of its initialization code as initialize class methods. On the other side, since not exercised by years, some of these initialization methods are out of sync with the system. Thus, we rewrote the majority of the initialization code of the system and its order of evaluation.

5.3. Step 1: Loading the Specification

The specification is reified, so it is materialized as a graph of objects. Each entity in the structural part of the specification is represented by its corresponding definition object following the model in [Figure 8](#). For example, a class definition will be reified as a `HzClassDefinition` and a trait definition as a `HzTraitDefinition`. These definition objects are in charge of building their respective pairs in the new system and collaborate in their initialization. In particular, method definition objects know whether they build the real methods by compilation or by assembling bytecode.

In addition to creating definition objects, loading the specification results into a specification object, instance of `HzSpecification`. The `HzSpecification` object acts as a façade of the rest of the definitions. The main responsibility of the specification object is to perform the basic initialization of the new system in the correct order.

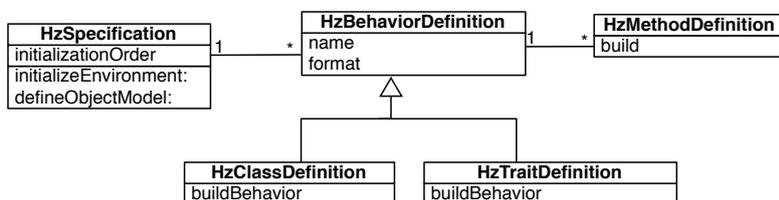


Figure 8: Hazelnut Specification Model

5.4. Step 2: Solving Basic Structural Meta-Circularity

Bootstrapping Pharo involves in this step the creation of the first class `MetaClass` and the first class `Trait`. These two first classes will then enable the creation of the rest of the system. [Figure 9](#) illustrates the creation of the first class `MetaClass`. First, the message `basicNew` is sent to the `MetaClass` in the source system to create the first meta-class shell in the new system (Step 2.1). The superclass, method dictionary and format of this meta-class shell are temporary objects from the source system. Later, the message `basicNew` is sent to that new meta-class shell to obtain the class shell of `MetaClass` corresponding to the new system (Step 2.2). This class is again partially initialized with temporary objects by sending it the message `instVarAt:put:`. Temporary objects provide to these two meta-classes the minimal information to be instantiated and be able to lookup methods. Afterwards, the meta-class meta-circularity already shown in [Figure 4](#) is closed. This is done by using the method `adoptInstance:` that changes the class of an object (Step 2.3).

In addition of the small integers and method dictionaries from the source system, the new class and meta-class also inherit from classes from the source system. This inheritance relationship lets these objects benefit from the already defined behavior. This leads to class shells which can still be used to define the rest of the system. The Pharo specific bootstrap includes in addition the creation of the class shells of `Trait` and `ClassTrait`. Their creation is achieved in a similar way as `MetaClass` is created. [Figure 10](#) shows the state of the system after this step is performed. Note that the class and meta-class shells in the new system do not have methods, but they understand messages such as `basicNew`, that are inherited from classes of the source system.

5.5. Step 3: Build Class Shells

Class and trait shells of the new system are instantiated by their respective definition objects. The definition objects send the `basicNew` message to the new class `MetaClass` to create the class shell meta-class. Then, the class shells are created by sending the `basicNew` message to the corresponding meta-class shells. In the case of traits, the definition

```

HzSpecification>>solveStructuralMetaclassMetacircularity
| metaclassClass metaclass |

"Step 2.1"
"We rely here on Metaclass from the source system
This dependency will be removed during the system initialization."
metaclassClass := Metaclass new.
metaclassClass
  superclass: Metaclass class
  methodDictionary: MethodDictionary new
  format: classFormat.

"Step 2.2"
metaclass := metaclassClass basicNew.
metaclass instVarAt: 1 put: Metaclass.
metaclass instVarAt: 2 put: MethodDictionary new.
metaclass instVarAt: 3 put: metaFormat.

"Step 2.3"
metaclass adoptInstance: metaclassClass.

```

Figure 9: Defining metaclass circularity

objects send the `basicNew` message to the new `Trait` and `ClassTrait` class shells. We set the instance variables of the new classes to refer to temporary objects of the source system by sending the `instVarAt:put:` message.

Figure 11 shows the state of the system after the creation of a `Point` class shell in the new system. The `Point` class and metaclass shells belong to the new system and are correctly related between them. They also refer to temporary objects from the source system as happened in the previous step with the first shells. Note that the `Point` class shell is a subclass of `Object` of the source system to inherit methods such as `instVarAt:put:` and `instVarAt:`. The metaclass `Point` class inherits from the class `Class` from the source system because it defines all the necessary behavior to instantiate it and define new objects.

5.6. Step 4: Install Methods

In this step, methods are built and installed into their respective owners either by compiling source code or assembling already compiled bytecode. A key point during this step is the resolution of literals appearing in methods. Method literals may refer to well known instances such as `nil`, `true`, `false`, numbers, strings, symbols and arrays. Due to the Pharo infrastructure limitations (See subsection 5.1), the literals of a method are kept as temporary objects of the source system. To circumvent these limitations we have an extra step at the end the bootstrap process that consists in swapping objects while serializing the system *i.e.*, saving the image of the new system (See subsection 5.8).

Literals can also be global variables which often refer to classes. They are represented by `Association` instances pointing to the name of the global variable and its value. The associations are considered special objects to the VM because of their compact nature and the assumptions on their variable offsets. Thus, we treat them as temporary objects. However, the value of these associations objects refer to the correct class shell in the new system, since all class shells exist.

Figure 12 shows a simplified example of the literal resolution for the method `Point>>corner:`. The built method is an instance of the `CompiledMethod` of the source system because the Virtual Machine does not support multiple versions of this class. Literal objects such as the `corner:` `ByteSymbol` representing the method selector is a `ByteSymbol` from the source system. Literal representing bindings, such as the one referencing the `Rectangle` class, are represented by `Association` instances of the source system. These associations reference to the objects they are binding in the new system, such as the new system `Rectangle` class. The last serialization step of the bootstrap process will ensure that references to the source system are replaced by references to the new bootstrapped system (See subsection 5.8).

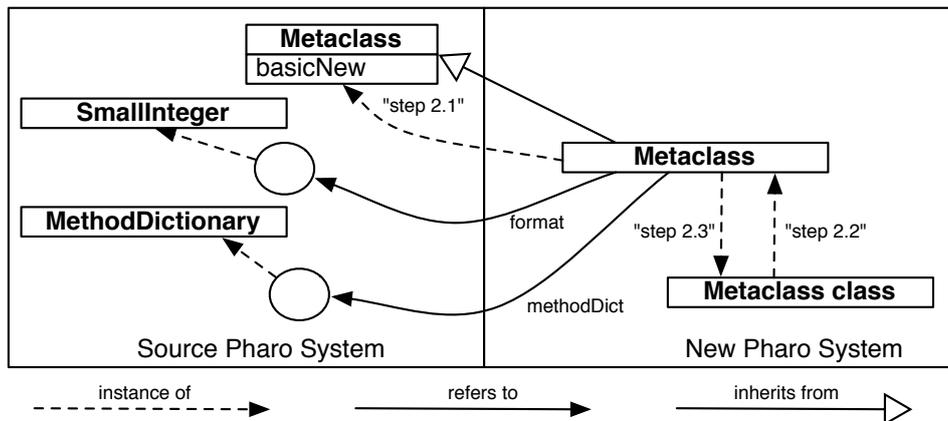


Figure 10: State of the new system after step 2

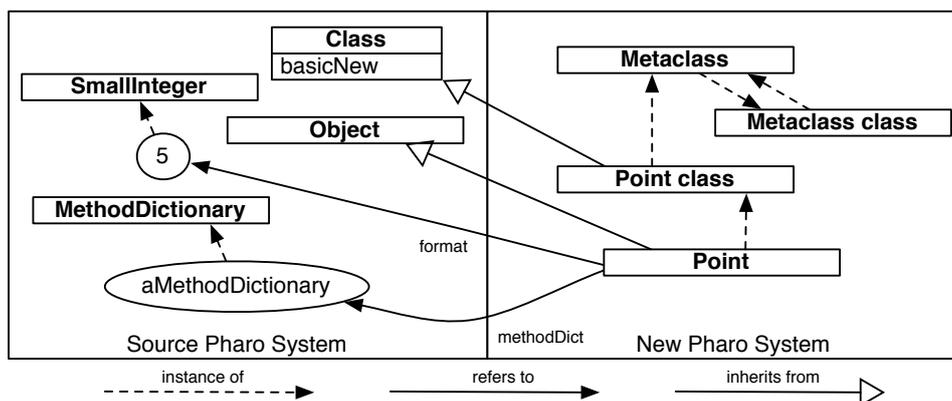


Figure 11: State of the created classes after step 3

5.7. Step 5: Initialize new System

The system is initialized following the next substeps:

1. *Class hierarchy set up.* The structural relations between classes –such as superclass/subclass or uses– are set up. This leads to a half-alive system where objects can receive and answer messages following their inheritance relationships;
2. *System basic initialization.* The HzSpecification object collaborates with the newly defined classes to initialize the remaining system state. For example character tables and package objects are initialized;
3. *Delayed initialization.* In addition, an initialization process is created in the new system. This process will be run during the first start of the system and perform two main tasks. First, it initializes the state that couldn't be initialized during the bootstrap itself *i.e.*, it initializes the *symbol table* with the symbols remaining alive in the new system after serialization. Second, it is responsible for the creation of all the processes needed at runtime. Because the current infrastructure does not allow one to run two complete systems side by side, it is not possible to execute and create all the active processes needed for the new kernel. Therefore we create a one

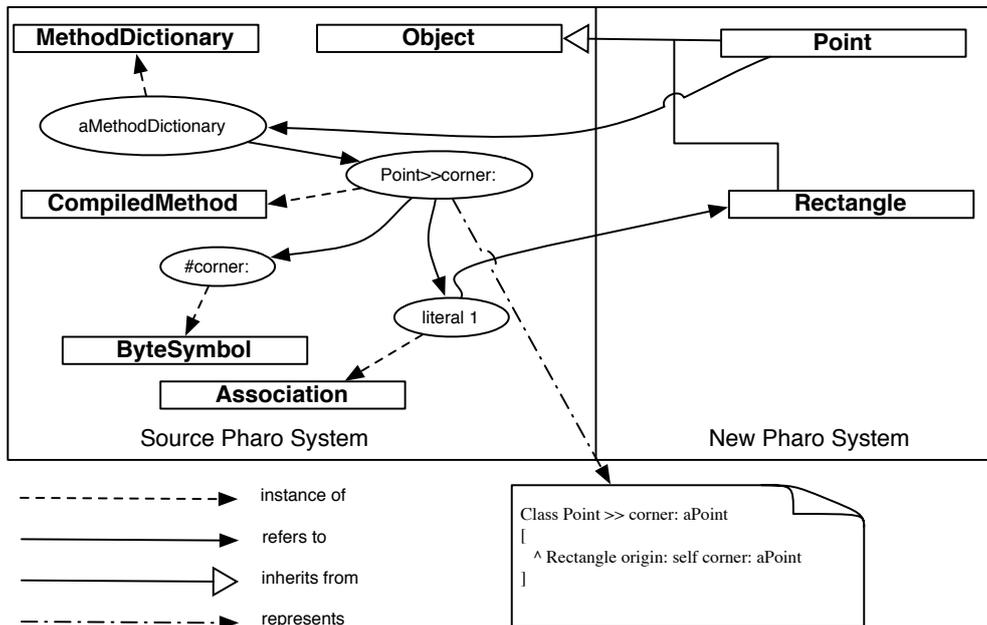


Figure 12: State of method literal resolution after step 4

shot startup process in a suspended state which will spawn all the processes that the system needs (such as UI, Delay, duration timers...) when it is first run. The execution will not happen in the current image but during the first startup of the resulting system.

At this point, as shown in Figure 13, the new system structure is created, initialized and almost isolated from the source system. The remaining exceptions are only well-known objects that cannot be replaced or duplicated because of the infrastructure limitations such as symbols, as described previously in subsection 5.1.

5.8. Step 6: Serialize Image

This last step is mandatory to finish the bootstrap because of implementation limitations. On a less constrained Virtual Machine this step should not have been necessary.

After the initialization step, the new system still keeps references to objects in the source system. The new system must be independent of the source system to finish the building process. To achieve that, the new system object graph is serialized into a standalone image file. A serializer traverses the graph to write the file in a binary format the Pharo Virtual Machine specifies. During this step references to objects in the source environment are replaced to references to their newly defined pairs. The specification façade object expresses the correspondence of object between the two systems.

The final result of this final step is an image file containing all objects correspondent to the specification with their according initialization. When the new system is started, the delayed process will spawn all other processes needed by the system leading to a working system.

6. Results

In this section we present two results: First the resulting Pharo bootstrapped system and second an alternative kernel having a reduced specification. These results show that the process can produce different system when fed with different specifications. The resulting bootstrapped systems are available at <http://car.mines-douai.fr/ci/view/Seed%20Tests/>.

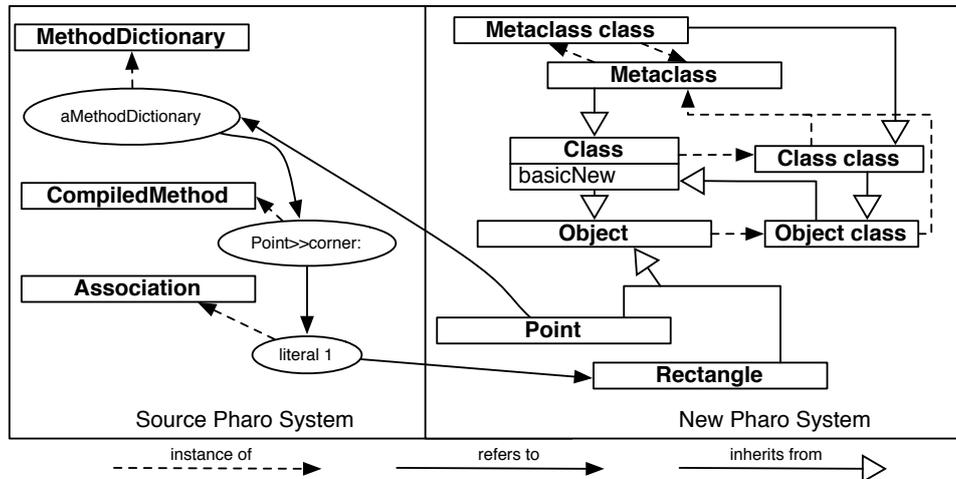


Figure 13: State of classes and special objects after step 5

6.1. Pharo bootstrap: results

The built Pharo core system includes reflective capabilities, process scheduling and binary loading by only including the 19% of the classes and traits of the original system. The specification used by the system can be found in <https://github.com/guillep/PharoKernel>. The size in disk of the resulting image is 2.2MB, contrasting its 22MB original counterpart.

Regarding its health, the bootstrapped kernel can be tested using the SUnit testing framework. Unit tests of the kernel itself are loaded using the binary loader and run in the new system. Using this same mechanism, core packages like the compiler are able to be tested isolated from other libraries.

A peculiarity of this system is that it is capable of bootstrapping a copy of itself. This is achieved by loading the binary packages of hazelnut and using its own specification in the building process.

Hazelnut is planned to be introduced into the infrastructure of Pharo, enhancing the building process of the upcoming versions. Regarding its size, which is certainly not minimal, the result shows that the specification should be refined to create a cleaner version of the system.

6.2. Second case of study: a micro kernel

As a second validation, we build a new and different system. Since a desirable evolution path for a Pharo system is its introduction into resource-constrained environments like portable phones or embedded devices. The second case of study is the building of minimal a Smalltalk environment based on Pharo and MicroSqueak [17]. We create a specification inspired on MicroSqueak for a system we called *PharoCandle*. The specification to generate the Pharo Candle system is available at <https://github.com/guillep/PharoCandle>.

PharoCandle is defined by a total of 49 classes with a reduced set of methods, providing only the core entities and basic file IO support. Also, PharoCandle metaclasses differ from the Pharo ones in their definition: they contain a different set of instance variables, and they do not support Traits.

The kernel entities included only core classes and their methods. Figure 14 shows the list of classes included in the PharoCandle specification.

The built micro-kernel is a working environment with a size of 80KB. This environment contains a very limited set of features because of its reduced specification and it is characterized by its lack of a loader or compiler library. This absence removes the ability to install new code, thus, keeping it as a fixed kernel with a particular responsibility. For example, a mp3 player application without the need of a display could be built. The corresponding playing classes should be introduced in the specification and the system should be bootstrapped.

PCBehavior	PCClass	PCMetaclass	PCArrayedCollection
PCCollection	PCLinkedList	PCAssociation	PCProcessList
PCSequenceableCollection	PCArray	PCByteArray	PCInterval
PCOrderedCollection	PCString	PCSymbol	PCValueLink
PCDictionary	PCSet	PCBlock	PCCompiledMethod
PCContext	PCInteger	PCSmallInteger	PCFile
PCMessage	PCMethodContext	PCMethodDictionary	PCCharacter
PCLargeNegativeInteger	PCLargePositiveInteger	PCMagnitude	PCNumber
PCFalse	PCObject	PCTrue	PCUndefinedObject
PCFloat	PCIdentityDictionary	PCIdentitySet	PCPoint
PCBitBlt	PCForm	PCWordArray	PCProcess
PCProcessorScheduler	PCReadStream	PCWriteStream	PCSystem
PCSemaphore			

Figure 14: Overview of the selection of the specification

7. Discussion and Future Work

Hazelnut provides the studied system with the ability to evolve by generation in addition to the evolution by self-modification. This ability solves all of the problems stated in [section 2](#). All changes made in the specification are applied atomically to the system when recreated, *avoiding the unstable intermediate steps* a self-modification approach often needs. System crashes are no longer a problem since *the process to build the system is reproducible*. The formerly unmaintained code is now exercised each time the system is built *preventing it from becoming obsolete or rotten* again. Additionally, Hazelnut stands as a *first step on the modularization of the monolithic Pharo system*, by supporting the control of dependencies in the system and providing a way to generate a new stable system.

Bootstrapping an alive system. Hazelnut achieves the bootstrap of a new system by creating objects from the first time. Objects in the bootstrapped system are alive in the sense that they can respond and send messages as any other object in the system. The bootstrap process can use these objects even not in a completely consistent state. Once the new system reaches the initialization step, some responsibilities can be delegated to the newly created objects. The process can take advantage of the encapsulation of their internal representation. This ability is exploited mainly during the initialization step, when the class initializations are delegated to the classes themselves.

An alternative approach would have been the manipulation of objects as raw memory structures. In this approach, `ByteArray` objects could have been used to represent the objects of the new system in a serialized state. On one side, this approach could have reduced the limitations imposed by the VM since the new system would have not been running. No messages should have been sent to objects in the new system until the bootstrap process was finished. Additionally, no need to install temporary objects in the new system should have been needed. However, this approach should perform all operations on the new system as array modifications and byte manipulations. It could have introduced difficulties while initializing and manipulating objects.

A mixed strategy could have used a mirrors implementation [3] to manipulate classes on their initial stage. The inheritance from classes of the source system, needed to reuse behavior such as `instVarAt:` or `basicNew` could have been avoided.

Virtual Machine implications. Hazelnut is completely implemented on the language side of Pharo. It does not need additional support on the Virtual Machine.

The Virtual Machine provides the basic reflective features needed by the bootstrap process:

- `basicNew` to create new instances;
- `instVarAt:` and `instVarAt:put:` to modify the internal state of objects before they are fully functional;
- `adoptInstance:` to allow the construction of the Metaclass loop by changing an object's class.

However, the Pharo Virtual Machine introduces several limitations to the bootstrap process (See [subsection 5.1](#)). Hazelnut solves these limitations by the introduction of the extra bootstrap step of serializing the image. This step provides the possibility to replace references during the graph traversal and modify object formats. However, using the system before the serialization may lead to undesired side effects because of the references to the source system. This presents as main drawback the impossibility to use and test completely the bootstrapped system from the source system.

For future work, we plan to explore the implementation of Hazelnut in a system providing a more flexible Virtual Machine with support of several name spaces. For example, the introduction of object spaces [5] could let us validate our model by being able to replace the temporary objects without the extra serialization step.

8. Related Work

Hazelnut uses exhaustively the reflective capabilities of the Pharo system to define the a new reflective system. On this topic, several articles cover the reflective facilities offered by Smalltalk [11, 19, 10] and CLOS [14], which Pharo is close to. The Pharo bootstrap also introduces traits [20] to define the system itself, which must be considered in the bootstrap process.

Within the approaches for bootstrapping a reflective environment we can find a similar one in the Common Lisp bootstrap [18]. Lisp, like Smalltalk, has the concept of image and new images are created by migrating the old ones. They describe their approach for generating a new virtual machine and image:

1. A cross compiler is installed inside the source environment.
2. The cross compiler generates lisp object files by using a special namespace, isolated from the source.
3. Those files are then loaded into a byte stream representing the memory layout of a Lisp image.
4. Once the image is built, the virtual machine loads and initializes it.

However, in his article Rhodes does not focus on the challenges a bootstrap process for a reflective system may overcome, nor many of the problems it solves besides the self-description of the system.

Gybels *et al.*, discuss the issues of reflective operations to enable interlanguage symbiosis [13]. However, it does not address the bootstrapping process of a reflective language as a meta-programming operation between two systems (the original one and the one that is being created). Similarly, Wuyts discuss the symbiotic relationship between languages (Prolog and Smalltalk) where one language can access the other during its execution and the reflective operations required for such situation [22].

Some projects such as Dart, Amber Smalltalk or CoffeeScript use interlanguage symbiosis to define a new language on top of an existing high level language (JavaScript in their case). This approach benefit from the idea of manipulating live objects on a high expressive language, such as Hazelnut does. This way, they use the abstractions provided by their source language to define new abstractions. In the case of Amber and CoffeeScript, for example, they provide class based languages on top of the Prototype based JavaScript, offering also a more consistent and safe model for programming on a Web browser.

Opposed to Hazelnut, projects like GNU Smalltalk [12], the Python CPython implementation, Ruby and Spider-Monkey JavaScript refer to *bootstrapping* as the initialization of their systems. Their initialization is performed in low-level languages such as C or C++. This supposes the manipulation of entities like objects and classes as raw memory structures. Objects are not able to receive messages during construction, thus, not taking benefit from their power. This is helpful in the initial state of the system but leads to an increase in the complexity in the process which must include code to manipulate complex objects. Additionally, this low level approach leads to mix the initialization of the object-system as well as the VM initialization and the specification into the building process. These solutions are often complex and not easily modifiable.

9. Conclusion

Bootstrapping is commonly known by its usage on language definition, more precisely on compiler building. It can be generalized to the introduction of any software system to its own building process. A bootstrapped software system can evolve by recreating itself. A bootstrap process enables a system to modify its own representation and reproduce its own construction.

This paper explores the idea of bootstrapping an object-oriented reflective system. Bootstrapping provides a reflective system with the ability to evolve by recreation in addition to self-modification. We present Hazelnut, a model for bootstrapping a reflective system, which we implemented to bootstrap the Pharo system. Hazelnut bootstraps an object-oriented reflective by populating a new namespace with alive objects. By being alive, Hazelnut can use these objects during the bootstrap process. The bootstrap process can benefit from the power of object-oriented programming (e.g. encapsulation and polymorphism) and reflection instead of handling bytes.

Hazelnut solves the meta-circularity problems of a reflective system by first resolving the main structural meta-circularity. Objects created on the new system reference to temporary objects from the source system on a first stage. Later, once all the classes are created and initialized, the temporary objects are replaced by their new pairs. The system is considered bootstrapped when the system is closed in itself and no references to the source system remain.

We validated Hazelnut by providing an implementation that successfully bootstrapped the Pharo 2.0 system and a micro-kernel of Smalltalk of 80KB. Each one of these systems was automatically built from a provided specification processed by our Hazelnut implementation. The bootstrapped systems are stable and fully operational. This is why we plan to introduce Hazelnut in the building process of the Pharo system.

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. We would also like to thank Pavel Krivanek and John Maloney for the code they provided and we used to extract the specification of the PharoKernel and PharoCandle respectively.

References

- [1] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Squeak by Example*. Square Bracket Associates, 2007.
- [2] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.
- [3] 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.
- [4] Gwenaël 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.
- [5] 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.
- [6] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding confusion in metacircularity: The meta-helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.
- [7] Pierre Cointe. Metaclasses are first class: the ObjVlisp model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 156–167, December 1987.
- [8] Marcus Denker and Stéphane Ducasse. Software evolution from the field: an experience report from the Squeak maintainers. In *Proceedings of the ERCIM Working Group on Software Evolution (2006)*, volume 166 of *Electronic Notes in Theoretical Computer Science*, pages 81–91. Elsevier, January 2007.
- [9] Martin Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. Fuel: A fast general purpose object graph serializer. *Journal of Software: Practice and Experience*, 2012.
- [10] Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.
- [11] Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, October 1989.
- [12] Cano Gokel. *Computer Programming using GNU Smalltalk*. http://www.canol.info/books/computer_programming_using_gnu_smalltalk/, 2010.
- [13] Kris Gybels, Roel Wuyts, Stéphane Ducasse, and Maja D'Hondt. Inter-language reflection — a conceptual model and its implementation. *Journal of Computer Languages, Systems and Structures*, 32(2-3):109–124, July 2006.
- [14] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

- [15] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of Asian Symposium on Programming Languages and Systems*, 2005.
- [16] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.
- [17] John Maloney. Microsqueak. <http://web.media.mit.edu/~jmaloney/microsqueak/>.
- [18] Christophe Rhodes. Sbcl: A sanely-bootstrappable common lisp. In *International Workshop on Self Sustainable Systems (S3)*, pages 74–86, 2008.
- [19] Fred Rivard. Reflective Facilities in Smalltalk. *Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février 1996*, February 1996.
- [20] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. Technical Report IAM-02-005, Institut für Informatik, Universität Bern, Switzerland, November 2002. Also available as Technical Report CSE-02-014, OGI School of Science & Engineering, Beaverton, Oregon, USA.
- [21] Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of POPL '84*, pages 23–3, 1984.
- [22] Roel Wuyts and Stéphane Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *ECOOP 2001 International Workshop on MultiParadigm Programming with Object-Oriented Languages*, 2001.