Bootstrap-Based Language Development: Turning an existing VM into a polyglot VM

Carolina Hernández Phillips

Numéro d'ordre : XXX

THÈSE

présentée en vue d'obtenir le grade de

Docteur

Discipline: Informatique

par

Carolina Hernández Phillips

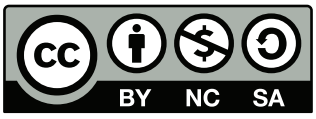
Doctorat de l'Université de Lille
Délivré par IMT Nord Europe

Bootstrap-Based Language Development: Turning an existing VM into a polyglot VM

Soutenue le 5 Novembre 2021 devant le jury d'examen :

Rapporteurs :	Alain PLANTEC	Professeur – Université de Bretagne Occidentale
	Wolfgang DE MEUTER	Professeur – Vrije Universiteit Brussel
Examinateurs :	Elisa GONZALEZ BOIX	Professeur – Vrije Universiteit Brussel
	Michael ENGEL	Professeur assistant – NTNU Norwegian University of Science and Technology
Directeur :	Luc FABRESSE	Professeur – IMT Nord Europe
Encadrant :	Guillermo POLITO	Docteur – Univ. Lille, CNRS, Inria, CRIStAL
Invités :	Stéphane DUCASSE	Directeur de Recherche – INRIA Lille
	Noury BOURAQADI	Professeur – IMT Nord Europe

Ecole Doctorale SPI 072 (Lille I, Lille III, Artois, ULCO, UVHC, Centrale Lille, IMT Nord Europe)



Acknowledgments

This work is not the product of my efforts alone. It is the result of the combined efforts of many people who have generously given me their support during the course of my PhD.

First, I would like to express my gratitude to Professor Luc Fabresse, my thesis director. Luc, thank you for your wise and kind guidance during these three years. Thank you for listening to my ideas with an open mind and heart, for improving them with your expertise, for opening doors by asking the right questions. Thank you for your good intentions and for your trust.

I would like to thank Guillermo Polito, my thesis *encadrant*. Guille, thank you for sharing your expertise and motivation with me during these three years. Your advices, high expectations, and example made me strive to improve, prioritizing what is important, making ideas concrete. Thank you especially for approaching me and offering me your help when I was feeling lost in my work but I did not dare to say. You believed in me and for that I am grateful.

Professor Stéphane Ducasse, leader of my research team RMoD, thank you for your trust in making me part of the team. Thank you sharing your long experience as a researcher with me, offering me career advices and supporting my professional growth. Thank you for making me feel that my work is recognized and important.

Pablo Tesone, my friend and colleague, thank you for giving me your selfless and unconditional support since the first day of my PhD. Thank your for sharing your technical expertise with me, especially at the beginning, when I needed it the most. But most importantly, thank you for your good will, time, and attention. Thank you for making me feel that, even during hard times, I am never alone because I can count on you.

Professor Noury Bouraqadi, thank you for attentively listening to my ideas and for carefully reading my work. Your comments have often unveiled perspectives that I would not have been able to discover on my own. I thank you not only for offering me your ideas, but also for the kindness with which you gave them to me.

Maximilian Willembrinck, thank you for your uninterested help, without which this work would not be the same. You have witnessed the darkest passages of this path and, instead of looking away, you have decided to reach me and share your courage with me. Thank you for caring so much, I will forever be in debt to you.

I would like to thank my friends in France, who made me never feel alone even though my country is so far away. Thank you for the shared happiness and sadness. You have given me the courage to finish this thesis and never give up.

Finally I would like to thank my family in Chile. Lissete, you are my sister and the light of my life, thank you for all the love you give to me, which is so strong that the distance cannot fade it away, not even a bit. Saif, my nephew, you may be too young to realize this, but your phone calls and our Fortnite sessions have lit my days. Your young soul has filled mine with strength and peace especially when I have needed it the most. Doris Phillips, my mother, this thesis would not exist if it were not for your efforts. You gave me all the tools and reasons I needed to arrive here. Your loving and deeply wise education made me the person I am today. This work is yours.

Abstract

Programming languages need to evolve as software requirements change, but their prototyping and extension comes at the cost of great development efforts. Reflective languages which are able to modify their own semantics provide a high-level approach to language implementation. While easy to use, the reflective approach has a limited range of languages it can produce, because modifications to core system elements can break circular references and leave the system in an inconsistent state. To overcome this limitation the system's kernel architecture must be re-built in a bootstrap process. Bootstrap has been shown effective to produce a family of similar languages. However, adapting an existing bootstrap implementation to generate new languages is challenging. This difficulty is caused by a lack of proper abstractions for language specification and, similarly to other techniques, a late manifestation of errors together with abstraction leaps during debugging tasks.

In this dissertation we study the design of a bootstrap based language development technique that supports the generation of multiple languages with low efforts. For this we introduce *MetaL*, a bootstrapping framework where language specification is based on metamodels and high-level reflective initialization instructions. A Meta Object Protocol (MOP) that is aware of VM constraints keeps the abstraction level high, providing operations to build the model of the language and then the kernel. Our MOP ensures model correctness by construction and ensures kernel health, detecting corruption early during the generation process.

To validate our approach, we report on the successful generation of seven object-oriented language kernels, plus an experiment by an external user. These experiments show that *MetaL*: (i) removes the need for expertise with virtual machine,(ii) reduces the abstraction gap in debugging tasks, (iii) it is positively perceived as meeting the need for easy-to-use bootstrapping solutions, and (iv) it is applicable in real world scenarios. Nevertheless, this experience also shows that using the metamodel for language specification hinders code reuse since a semantic feature implementation is spread across multiple metamodel classes. To solve the previous limitation we propose using Aspect-Oriented Programming (AOP) to bootstrap. For this, we implemented *AspectMetaL*, an aspect-oriented layer on top of *MetaL* that allows language specification at the level of semantic features. To validate *AspectMetaL*, we generated 3 kernels by combining both predefined and custom semantic features, learning that AOP is effective for improving code reuse and rising the abstraction level for language specification in bootstrap.

Keywords: bootstrapping, programming language design, reflective languages, language runtime initialization.

Résumé

Les langages de programmation doivent évoluer au fur et à mesure que les exigences des logiciels changent, mais leur prototypage et leur extension se font au prix de grands efforts de développement. Les langages réflectifs, qui sont capables de modifier leur propre sémantique, offrent une approche de haut niveau pour l'implémentation des langages. Bien que facile à utiliser, l'approche réflexive ne peut produire qu'une gamme limitée de langages, car les modifications apportées aux éléments centraux du système peuvent briser les références circulaires et laisser le système dans un état incohérent. Pour surmonter cette limitation, l'architecture du noyau du système doit être reconstruite avec un processus d'amorçage (bootstrap). Cette technique s'est avérée efficace pour produire une famille de langages similaires. Cependant, l'adaptation d'une implémentation de bootstrap existante pour générer de nouveaux langages est un défi. Cette difficulté est due à un manque d'abstractions appropriées pour la spécification des langages et, comme pour d'autres techniques, à une manifestation tardive des erreurs avec des sauts d'abstraction pendant les tâches de débogage.

Dans cette thèse, nous étudions la conception d'une technique de développement de langages basée sur le bootstrap qui permet de générer plusieurs langages avec peu d'efforts. Pour cela, nous introduisons *MetaL*, un cadriciel d'amorçage où la spécification du langage est basée sur des métamodèles et des instructions d'initialisation réflectives de haut niveau. Un Meta-Object Protocol (MOP) réifie les contraintes de la machine virtuelle (VM) et assure un niveau d'abstraction élevé, fournissant des opérations pour construire un modèle de langage et ensuite un noyau. Notre MOP garantit la correction du modèle par construction et assure la cohérence du noyau, en détectant la corruption au début du processus de génération.

Pour valider notre approche, nous rapportons la génération réussie de sept noyaux de langages orientés objet, ainsi qu'une expérience par un utilisateur externe. Ces expériences montrent que *MetaL*: (i) supprime le besoin d'expertise sur la VM (ii) réduit le fossé d'abstraction dans les tâches de débogage, (iii) est perçu positivement comme répondant au besoin de solutions d'amorçage faciles à utiliser, (iv) est applicable dans des scénarios du monde réel. Néanmoins, cette expérience montre également que l'utilisation du métamodèle pour la spécification du langage entrave la réutilisation du code, car l'implémentation d'une fonctionnalité sémantique est répartie sur plusieurs classes du métamodèle. Pour résoudre cette limitation, nous proposons *AspectMetaL*, une couche orientée aspect (POA) au dessus de *MetaL* qui permet de spécifier un langage au niveau des fonctionnalités sémantiques. Pour valider *AspectMetaL*, nous avons généré 3 noyaux en combinant des caractéristiques sémantiques prédéfinies et personnalisées. Nous avons ainsi pu montrer que la POA est efficace pour améliorer la réutilisation du code et augmenter le niveau d'abstraction pour la spécification du langage.

Mots-clés: bootstrapping, conception de langages de programmation, langages reflectifs, initialisation de lexecution d'un langage.

Contents

List of Figures

List of Tables

Introduction

Chapter 1

Contents

Programming languages are a central topic in computer science. They evolve as software requirements change. Their evolution to integrate new features, constructs or paradigms, or to revise their internal semantics should be done efficiently. Nevertheless, prototyping and extending programming languages comes at the cost of great development effort.

Language implementation techniques (LIT) are software reuse techniques that reduce efforts of language implementation providing solutions that can be used to generate different languages with little change [Biggerstaff 1992]. Among existing kinds of LIT we find *Self-Surgery* Techniques, which are offered by systems providing a reflective language that has control over its own semantics. To produce new languages, developers write programs that modify the semantics of the base language.

Self-surgery has a limited range of produced languages because modifications to core elements of the base language architecture can break internal references leaving the system in an inconsistent state. However, it is possible to overcome this limitation by rebuilding the system's runtime in a bootstrap process. This runtime, also named *kernel*, contains the system's new architecture, in which the semantics of the new language is implemented.

Bootstrap is a high-level approach for kernel generation. It is specially used in the generation of reflective languages. The idea is to use as early as possible the capacities of the new language for building itself. Until now, Bootstrap supports the evolution of a single programming language or a family of similar languages [Polito 2015].

Even though the bootstrap process is written in a high-level language, modifying an existing bootstrap implementation to generate new languages imposes a high cognitive distance on developers. Small changes introduced by the developer can produce failures that manifest during late stages of the kernel generation process or as late as during the execution of the kernel. The larger the distance between error causes and symptoms, the harder it is to track the infection chain [Zeller 2005]. Furthermore, debugging these errors often requires debugging VM or bootstrapper code, where the abstraction level is lower than that of the code written by the developer. This situation causes abstraction leaps, where developers mentally map concepts between different levels of abstractions.

In this dissertation we study how Bootstrap can be adapted to become a LIT. For this we must reduce the cognitive distance and increase the variety of languages it can produce. To achieve cognitive distance reduction we must find proper abstractions to specify new languages and mechanisms to support debugging errors occurring during the kernel generation process, preventing abstraction leaps.

1.1 Abstractions for Language Definition

Abstraction is the essential feature in any reuse technique [Wegner 1983]. The most important abstractions in LIT are are those representing the language syntax and semantics.

According to Krueger [Krueger 1992] abstractions have two levels: *specification* is the highest of the two levels and only express what is relevant for the developer (*e.g.* objects in object-oriented languages are a high-level representation of data in memory), and *realization* which contains detailed information about the abstraction that is not relevant for the developer (*e.g.* object's layout in memory).

The language definition process in LIT occurs in three steps as illustrated in Figure 1.1. During *Conceptualization* developers reason informally about the syntax and semantics of the new language. Then, during *Specification* they express syntax and semantics in terms of specification abstractions. Finally during *Specification To Realization Mapping* the system automatically transforms specifications into realizations.

Krueger states that the success of a reusable software technique depends on the effectiveness of its abstractions, which can be evaluated in terms of *cognitive distance*. Cognitive distance is an informal and intuitive measure of the amount of intellectual effort that takes using a technique.

For the creator of a software reuse technique, the goal is to minimize cognitive distance by providing specification abstractions that are succinct, expressive and close to abstractions used for conceptualization, and using automated mappings from abstrac-

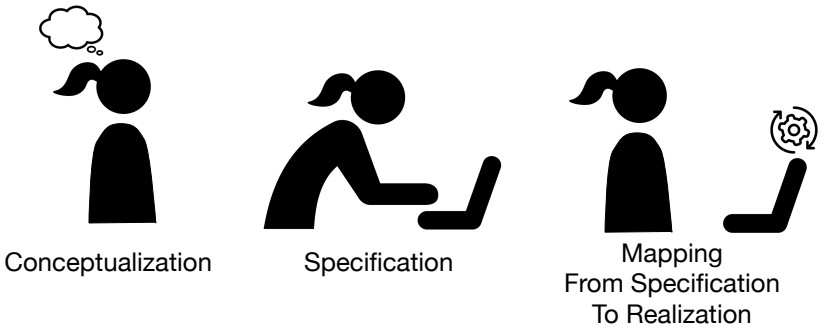


Figure 1.1: Steps in language elaboration using LIT

tion specification to abstraction realization.

1.2 Reflective Languages

Reflection is the ability of a system to observe and (possibly) change its own execution [Smith 1982]. There are multiple kinds of reflection, being the most powerful one behavioral intercession reflection. An object-oriented programming language providing this kind of reflection reifies its own semantics in metaobjects whose modification changes the semantics of the language.

Meta-Object Protocols(MOP [Kiczales 1991]) are protocols to modify metaobjects. They give users the ability to incrementally modify the language metaobjects through operations that can be applied at run-time.

Self-surgery is a LIT where developers implement new languages by manipulating the metaobjects of an existing reflective language. This technique is applied for both implementing internal DSLs and prototyping new languages. In self-surgery, the specification of a new language is a program written in the reflective language provided by the technique. This program uses MOP operations to modify the semantics of the reflective language to obtain the new language.

As a consequence of the previous, abstraction leaps during debugging tasks never occur because language *specification* and *realization* are at the same level of abstraction. Moreover, debugging tools from the original system are useful to debug language specifications because they are written in the same language the system originally provides. Also, interaction with low-level code never happens, since most low-level language concepts are represented by high-level metaobjects.

1.3 Bootstrap

Bootstrap is a high-level approach for language runtime initialization that takes the definition of a new language as input and produces its language runtime (also named *kernel*) as output [Polito 2015]. It is specially applied to generate systems that define reflective languages, since they are able to modify themselves.

The main idea of Bootstrap is to use as early as possible the reflective capabilities of the resulting language to build the kernel. In consequence, great part of the system building process written in the same language that is being produced. This makes Bootstrap is a circular process through which reflective languages generate their own implementation (*i.e.* kernel). Considering the previous, Bootstrap takes advantage of the reflective features of the new language in the same way than self-surgery techniques do.

Bootstrapping a new language is done by modifying both the definition and the application that generate the kernel of an existing language. This is challenging because building reflective systems is a delicate process. Also, if the kernel is built to be executed by an existing VM, spread in the language definition and bootstrapper application there are critical sections of code in charge of creating low-level structures that the target VM expects to find in the kernel. These critical sections are unidentifiable a priori. Unfulfilled VM expectations produce failures either during kernel generation or as late as during kernel execution. Debugging these errors requires dealing with bootstrapper and VM code that is external to the user.

When targeting an existing VM, the kernel is an array of bytes during its construction. Contrary to self-surgery, in Bootstrap the abstraction level of specifications and realizations is different. While reflective language code is used for the specification, the realization are bytes in the array of bytes. During debugging tasks, realizations are accessed through mirrors providing generic tools for their manipulation as objects. This causes abstraction gaps when debugging the generation process.

Taking the previous into consideration, Bootstrap does not have access to the same benefits as self-surgery regarding immediate feedback and debugging support.

1.4 Problem Statement

Even though Bootstrap has been used to generate similar programming languages families [Polito 2015], it is not yet a language implementation technique for two reasons: first, because current implementations are limited to a single language or to a narrow variety of similar languages; and second, because it is difficult to modify the bootstrap imple-

mentation to generate a new language. This difficulty is caused by a lack of proper abstractions for language specification, late manifestation of errors, and abstraction leaps during debugging tasks.

To address these problems we ask the following research questions.

RQ1: What abstractions for language specification reduce the cognitive distance in a bootstrap-based LIT?

Krueger [Krueger 1992] states that effective specification abstractions are succinct, expressive and close to abstractions used for conceptualization. We must consider that specification of the language semantics alone is not enough, because a kernel contains the language runtime which includes core system's classes, methods, global state, etc. It is important to identify suitable abstractions for kernel specification in Bootstrap, which take into account different kinds of failures occurring during the generation process. This leads to our second research question.

RQ2: What mechanisms support the automatic mapping from specification to realization in a bootstrap-based LIT?

In Bootstrap failures can manifest late and often require debugging external code, not written by developers, such as VM, compiler, and/or bootstrap process code. To address these problems we need to analyze the different causes of failures in Bootstrap to prevent them whenever is possible. For those who cannot be prevented, we must design strategies to force their manifestation as early as possible during the bootstrap process.

Manifestation of failures often occurs at a lower abstraction level than that of their original cause (*e.g.* a segmentation fault in the VM, caused by the missing definition of one element in the language definition). Traditional debugging tools do not provide appropriate support for this scenario [Chis 2015]. It is important to study abstraction gaps in Bootstrap to provide suitable debugging tools for each situation.

1.5 Hypothesis

Considering the research questions stated in Section 1.4, we propose the following hypothesis:

H1: It is possible to apply self-surgery concepts about abstractions, immediate feedback, and debugging mechanisms to a bootstrap-based LIT to reduce the cognitive distance.

Contrary to self-surgery, in Bootstrap there is an abstraction gap between specifications and realizations during debugging tasks. If we can improve bootstrap to offer specifications that resemble metaobjects and, at the same time, we improve mirrors to also resemble metaobjects, we would significantly reduce the abstraction gap between specifications and realizations. This is a first step towards cognitive distance reduction in bootstrap.

In self-surgery, when MOP operations to modify base language semantics fail to execute, most of the time they are debugged using the host system's debugger. In this way, debugging tasks are kept at the abstraction level of specifications because MOP operations are directly debugged with the host debugger. On the other hand, in Bootstrap, when the generation process fails, it is often necessary to debug low-level code from the VM or code form the bootstrap process which was not defined by the user. Thus, debugging tasks in bootstrap are not kept at the level of specifications, falling to the level of realizations and forcing developers to mentally map concepts between different abstraction levels. If we can design debugging tools in the style of self-surgery, we would reduce mental operations needed from developers, reducing the cognitive distance.

In self-surgery, the system modifies itself (possibly) at run-time. This provides immediate feedback to developers. If we can implement immediate feedback in Bootstrap, alerting developers about corruption and providing live interaction with objects (*i.e.* see the effects of kernel modifications immediately), we would shorten the distance between defects and failures. We know from [Zeller 2005] that the larger this distance, the harder it is to track the infection chain.

1.6 Results

We present a summary of the main results obtained in this work, answering our research questions and testing our hypothesis.

1.6.1 Research Question 1

To answer RQ1 we studied the abstractions provided by state of the art LIT and then we designed *MetaL*, a framework for bootstrapping kernels that provides a kernel specifica-

tion based on metamodels. To test *MetaL*, we use it to generate seven example kernels. These examples were selected such as they have important semantic differences. In this way we test the metamodel flexibility. In our experience generating these kernels, we never needed to become aware about VM implementation details.

MetaL was used by an external user in his project of building Smalltalk environments that use the browser as a view/frontend. From his experience we concluded that (i) *MetaL* is positively perceived as meeting the need for easy-to-use bootstrapping solutions and that (ii) *MetaL* is applicable in real world scenarios.

In *AspectMetaL*, aspects are the main abstraction for kernel specification: one aspect represents one semantic feature. Kernels in *AspectMetaL* are specified declaratively, providing the list of aspects that define it. We performed experiments to test the capacity of aspects to be reused in the definition of different kernels, obtaining positive results.

Our answer to RQ1 is that both *MetaL* and *AspectMetaL* abstractions reduce cognitive distance when defining kernels in a bootstrap-based LIT.

1.6.2 Research Question 2 and Hypothesis 1

To answer RQ2 and test H1, we undertook a comprehensive analysis of failures occurring in Bootstrap which resulted in a taxonomy of errors. From it we conceived a set of desirable features of a bootstrap-based LIT where each feature solved one category of error.

To evaluate our strategy and test H1 we rely on our experience generating kernels. We acknowledge that our own experience is not enough to validate the effectivity of our approach in a general context. However, we confirm its effectivity to reduce cognitive distance in the generation of kernels presented in this dissertation.

We can answer RQ2: combining early detection of corruption through automatic tests, a debugger for reflective code, and smart-mirrors which raise the abstraction level to interact with kernel contents, provides effective support for automatic mapping from specification to realization when the approach is applied to the example kernels presented in this work.

Finally, we accept H1: the concepts from self-surgery applied to *MetaL* have shown effective on reducing the cognitive distance on developers in all tested cases.

1.7 Contributions

The contributions of this thesis are summarized below:

- ***MetaL*:** A framework for bootstrapping kernels that reduces cognitive distance. *MetaL* is based on metamodels for kernel specification and a VM constraints aware MOP which ensures compatibility of the kernel with the target VM.
- **Kernel Experiments:** The definition of seven object-oriented kernels which are compatible with the Pharo VM but present important semantics differences with respect to Pharo.
- ***AspectMetaL*:** An aspect-oriented layer on top of *MetaL* that raises the abstraction level of kernel specifications to the level of semantic features. *AspectMetaL* allows users to generate kernels by combining semantic features and never modifying the metamodel.

1.8 Thesis Outline

- **Chapter 2: Language implementation Techniques**, presents the state of art in language implementation techniques, evaluating their methods for cognitive distance reduction.
- **Chapter 3: Challenges Bootstrapping Reflective Kernels**, presents a taxonomy of errors in the bootstrap process and a presentation of the bootstrap limitations through a case of study.
- **Chapter 4: Bootstrap-Based Language Implementation: *MetaL***, presents *MetaL*, our bootstrap framework to achieve cognitive distance reduction in bootstrap.
- **Chapter 5: *MetaL* Evaluation: Bootstrapping Kernels**, shows the implementation in *MetaL* of 8 kernels with different semantics and one experiment done by an external user.
- **Chapter 6: Aspect-Oriented Bootstrap: *AspectMetaL***, presents our implementation of a bootstrap-based LIT using aspect-oriented programming to raise the level of abstraction for language specification.
- **Chapter 7: Conclusions**, exposes the main results of this work and concludes this dissertation.

- **Appendix A: Associated Publications**, presents the list of publications produced by this research work.

Chapter 2

Language Implementation Techniques

Contents

In this dissertation we study the elaboration of a novel language implementation technique based on Bootstrap. Language implementation techniques (LIT) are software reuse techniques that decrease efforts of language implementation providing solutions that are used to generate multiple languages with little change [Biggerstaff 1992]. Different techniques implement different approaches to cognitive distance reduction, providing different the abstractions and debugging support. This chapter focuses on studying these mechanisms to guide our design of a bootstrap-based LIT.

We start by giving a brief introduction to the concept of cognitive distance in Section 2.1. A classification of LITs and the evaluation criteria we use to compare the presented LIT is offered in Section 2.2. Sections 2.3 and 2.4 describe solutions belonging to each LIT category. These solutions are then compared in the evaluation presented in Section 2.5. Before presenting Bootstrap, we explain the concept of language runtime initialization in Section 2.6. Following, Section 2.7 presents some Bootstrap examples. Our final remarks about the chapter are given in Section 2.8.

2.1 Cognitive Distance

We use the ideas of Krueger [Krueger 1992] about cognitive distance to evaluate existing LIT solutions later in this chapter.

According to Krueger, LIT success depends on the effectiveness of its abstractions, which can be evaluated in terms of cognitive distance. He defines *cognitive distance* as an informal and intuitive measure of the amount of intellectual effort required to accomplish software development tasks. For the creator of a software reuse technique, the goal is to minimize cognitive distance by using abstractions that are both succinct and expressive, and by using automated mappings from abstraction specification to abstraction realization.

Analyzing the automatic mapping process requires taking into consideration possible errors produced during the process. An *error* is a combination of a *defect* (the cause) and a *failure* (the symptom). We define these concepts as follows.

- *Defect.* Also named bug, is an unintended mistake introduced by the programmer in the source representation of a program [Beizer 1990].
- *Failure.* Is an unwanted executional behavior that is externally observable by a developer while a process is running [Spinellis 2018]. A *critical failure* [Zeller 2005] is a failure that interrupts the execution of the process.

Developers reason about and formulate questions using concepts and abstractions from their specification domain. Not offering a one-to-one mapping between developer questions and debugging support forces developers to refine their high-level questions into low-level ones and mentally piece together information from various sources. [Sillito 2008].

2.2 Language Implementation Techniques (LIT)

LIT are software reuse techniques that reduce the efforts of implementing new languages providing solutions to produce different languages with little input from developers [Biggerstaff 1992], improving developers productivity [Frakes 2005].

2.2.1 LIT Evaluation Criteria

On the basis of Krueger [Krueger 1992] ideas, we propose the following criteria to evaluate cognitive distance in LIT:

C1. Specification abstractions provided by the technique must be succinct, expressive, and must be close to abstractions used for conceptualization of language syntax and semantics.

C2. The mapping between abstraction specification and realization must be automatic. In the case the mapping process fails, the technique should provide debugging support that answers developer questions formulated at the level of their abstraction specifications.

2.2.2 LIT Classification

LIT can be divided according to their approach for language specification and subsequent mapping to realization in two groups: *generation techniques* and *self-surgery techniques*. Our classification is consistent with the classification for software reuse techniques proposed by Biggerstaff et al. [Biggerstaff 1987], which is based on the nature of the software components being reused.

2.3 Generation Techniques

In Generation LIT the abstraction level of specification is higher than that of realization[1]. The logics to transform specifications into realizations is inside the generator program. These logics are the technique's reusable components and, in principle, they are kept hidden from the user.

As an example, we present in Listing 2.1 the specification for the semantics of the boolean operator 'not' in DynSem [Vergu 2015], a DSL for specification of dynamic semantics which is part of the language workbench Spoofax [Metaborg]. The semantics of 'not' is specified as two reduction rules written in DSL code. DynSem's interpreter generator is an application written in Java that takes these rules as input and generates the Java code that implements the specified semantics in an interpreter.

```
" DSL rules for 'not' operator "
rules
  true −not−> false
  false −not−> true

" Generated Java pseudo−code "
class Interpreter {
   public Object evaluate(Operator operator, Arguments <Node> arguments) {
      ...
      if (operator.symbol() == 'not') {
         if (arguments[1].value()) {
            return false; }
         else {
            return true; }
      }
   }
}
```

Listing 2.1: Specification and realization of the semantics for the boolean operator 'not' in Generation LIT.

We represent in Figure 2.1 the mapping process from specification to realization for the previous example. We follow the style of T-diagrams, commonly used in compilers to represent language transformation processes.

We call *host-language* to the language in which the generator application is written.

[1]This category corresponds with Generation Technologies proposed by Biggerstaff et al. [Biggerstaff 1987]. The reused components are the patterns in the generator program to transform specifications into realizations. Reuse is less a composition matter than it is an execution matter of component generators.

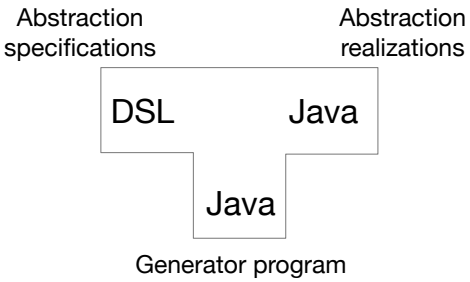


Figure 2.1: Mapping from abstractions specification to realization in Generation LIT

Depending on the technique, the host language can be the same as the specification language or not. We call *guest-language* to the language that is generated.

2.3.1 Language Workbenches

Language workbenches [Fowler 2005] are integrated environments to develop domain-specific languages. They provide a set of DSLs to specify language syntax and semantics. The set of specifications is used to generate software artifacts which integrate with existing ones to provide parsing/compilation and code edition tools for the guest-language. The generation is a multi-stage process, as depicted in Figure 2.2 using Xtext [Xtext] as an example.

Popular examples are Xtext, Spoofax [Metaborg], Jet-Brains Meta Programming System (MPS) [JetBrains], MetaEdit+ [Metacase], Rascal [Rascal].

Although generation of general purpose programming languages (GPL) is possible in workbenches such as Xtext, it requires a considerable amount of code [Willink 2011], much of which is not DSLs code, but Java code.

C1. Abstractions for language definition. We split our description in two accordingly to the language concepts represented by abstractions.

Syntax and static semantics. DSLs for syntax and static semantics specification resemble formal attribute grammars notation: a grammar is defined by a set of rules expressed in a declarative way. The abstraction level of these DSLs is close to that of syntax conceptualization, *e.g.* Xtext DSL [Xtext], SDF3 [SDF3]. Dedicated DSLs for binding analysis and scoping rules work as a complement of attribute grammar DSLs, *e.g.* NaBL2 [Konat 2012]. In this way, DSLs work together in a tool-chain. The resulting realizations are parsers and compiler extensions. Customizations which are not

supported by the DSLs must be expressed in the language of the workbench, *i.e.* the host-language.

Dynamic semantics. Dynamic semantics is the aspect with less support among current solutions. DynSem [Vergu 2015] and XSemantics [XSemantics] are DSLs for specification of the dynamic semantics of a language as a set of conditional term reduction rules, resembling operational semantics formal notation. Specifications are mapped to interpreters and validators used during compilation.

Dynamic semantics can also be expressed through model-to-model transformations. For example, in Stratego [Visser 2004, Bravenboer 2008], developers define rules from which guest-language code is transformed into Java code. Model-to-model transformations are not close to semantics conceptualization. Dynamic semantics specification often requires customization code written in the language of the workbench.

C2. Mapping from specification to realization. Language workbenches offer code editors for their DSLs. These editors can mark defects due to wrong syntax in DSL code and even suggest possible solutions. However, other kinds of defect are not so easy to capture.

DSL code with the right syntax but the wrong meaning maps into realizations that behave unexpectedly. To find the defect back in DSL code, the realization code (*e.g.* a parser) is generated as traced code to find back the defect in DSL code (*e.g.* a grammar). In this way they reduce the abstraction gap between specification and realization.

Code written in the host-language can make the generation process fail. These failures are solved by debugging framework code. Moreover, since DSLs work in a toolchain, defects introduced early in the process can generate corrupt realizations which produce failures that manifest late, either interrupting the mapping or producing a language that behaves unexpectedly.

2.3.2 Meta-Compilation Techniques

Meta-compilers are a special kind of compiler used to construct compilers of different programming languages. They focus on language performance and support a wide variety of general purpose languages. They integrate to a self-optimizing interpreter defined by the user and infer a just-in-time compiler from it. RPython [Bolz 2009] and GraalVM [GraalVM , Wimmer 2012, Würthinger 2017, Wimmer 2019] are popular state of the art meta-compilers.

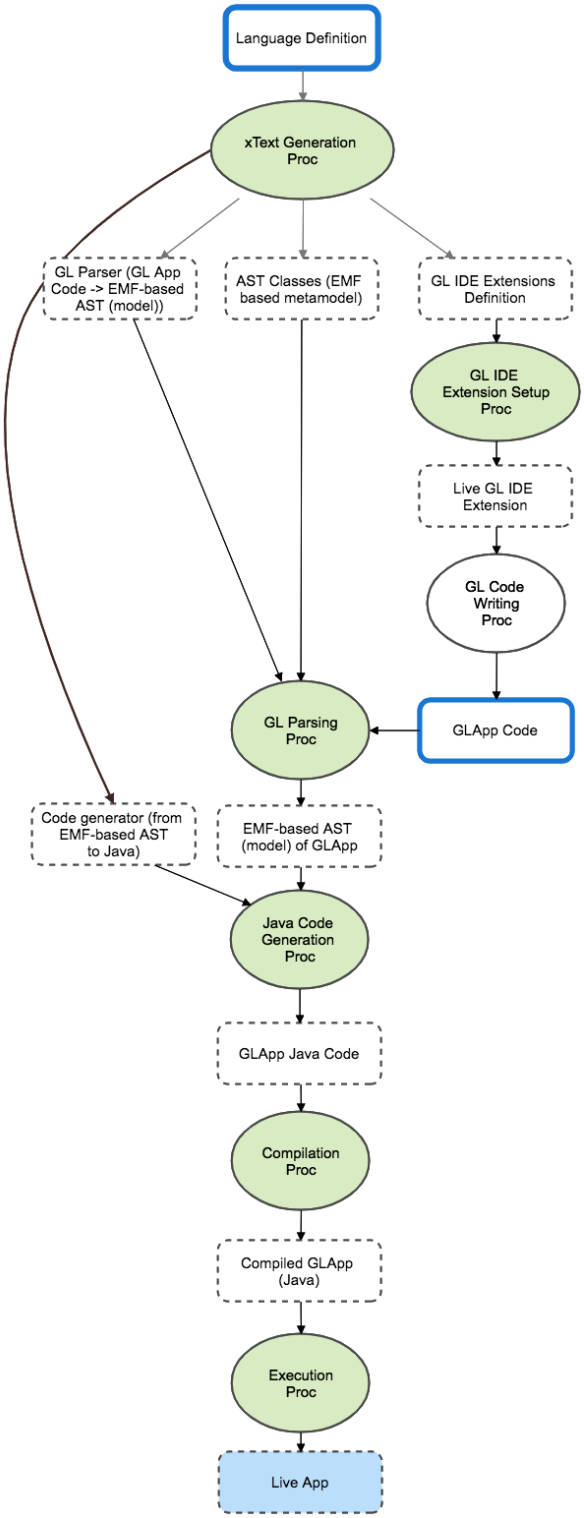


Figure 2.2: Xtext artifacts generation multi-stage process.

C1. Abstractions for dynamic semantics. The semantics of the new language is defined in an interpreter. The interpreter can be generated from DSL code using for example DynSem, or using a Java API such as the one offered by Truffle [Wimmer 2012]. To enable JIT compilation, the interpreter must be capable of self optimization. This kind of interpreters can be generated using a framework such as Truffle, where node specialization operations are expressed using Java annotations. While the interpreter and node specialization operations are used as specification, the realization corresponds to the resulting JIT compiler.

C2. Mapping from specification to realization. Defects in the original interpreter produce failures in the generated code and must be debugged at that level, losing the causality with the original source-code. Failures in this stage prevent the generation of code from node rewriting instructions, and the posterior compilation of the full interpreter. *E.g.* defects in node rewriting operations are debugged using tools to analyze compilation graphs. These tools allow to navigate guest-language source code from AST nodes. There is an abstraction gap because developers must find back the causes of specific compilation steps in their node specialization operations and interpreter code. Failures due to bugs in the AST interpreter manifest in Java, and are debugged using the Java debugger.

2.4 Self-Surgery Techniques

This category comprises LIT where abstraction specifications and abstraction realizations are the same. In consequence, the mapping between specification and realization is not performed by a generator program[2]. Instead there is only one system where the specifications are defined and applied.

This kind of systems provide a base language that has control over its own semantics and implementation. This is possible for languages that provide behavioral intercession reflection (see Section 2.4.1). Object-oriented reflective languages of this kind reify their own semantics as well as the data used to execute a program in metaobjects.

Metaobjects represent language components like methods, classes, and metaclasses; and even execution components like contexts, messages, etc. In self-surgery, developers implement new languages by manipulation of these metaobjects, never interacting with low-level code, as most low-level language concepts are represented by high-level

[2]This category corresponds with Composition Technologies proposed by Biggerstaff et al. [Biggerstaff 1987]. The reused components are the metaobjects of a language. Language specification is a matter of modifying an existing language manipulating its metaobjects using well-defined protocols. These protocols are called Meta Object Protocols (MOP) and are explained in Section 2.4.1.

metaobjects. Metaobject manipulation can even occur at run-time, while the system is executing [Malenfant 1996].

Specifications are programs that modify the semantics of its own language, transforming it into the new language. The realization is the same system where the specifications are executed, but after the execution of specifications. Examples of these systems offering self-surgery are Smalltalk and CLOS, where self-surgery has been applied for both implementing internal DSLs[3] and prototyping new languages.

As an example, below we present the specification for the semantics of the boolean operator 'not' in Smalltalk. The meaning of 'not' is specified in the method not of classes True and False. These methods are written in Smalltalk code. Modifying these methods modifies the semantics of Smalltalk.

```
" 'not' operator definition in metaobjects "
True >> not
  ^ false
False >> not
  ^ true
```

2.4.1 Concepts About Reflection

Reflection is the ability of a system to observe and (possibly) change its own execution [Smith 1982]. A programming language is said to be reflective when it provides reflective features, *i.e.* programs written in these languages are able to reflect on their own execution and structure. A reflective system is a system providing a reflective language and which is used to execute instructions written in this language.

Reflective Systems Classification. Considering what can be done with the self-representation of a system, reflection can be classified as [Kiczales 1991]:

- *Introspection:* The system can dynamically observe itself but not modify itself. This is possible when concepts of the runtime system are reified as ordinary objects which can be queried and inspected. An example is to access the class of an object.

```
'hello' class "returns the class String"
```

- *Intercession:* The system can dynamically modify itself, including the customization of its own interpretation or meaning. This is possible when modifications to

[3]Internal DSLs are a particular form of API in a host general purpose language, often referred to as a fluent interface [Fowler 2010].

these reified objects reflect back to the runtime system. Dynamically adding and removing object fields and methods at runtime is a typical example of intercession.

```
1 Car addInstVarNamed: #driver
```

Another criterion to categorize reflective systems is considering what can be reflected. According to this condition, two levels of reflection are identified:

- *Structural reflection:* The information reflected is the structure of the program, offered to the programmer as data. In case the program structure is modified, changes will be reflected at runtime. The capability of reading class structures (*e.g.* Java) and modifying them (*e.g.* Python) are two examples of structural reflection.

- *Behavioral reflection:* The ability to access system semantics. In case the semantics is modified, it will involve a customization of the runtime behavior of programs. Meta-Object Protocols (MOPs) are a common way to implement this level of reflection.

A reflective system capable of intercession incorporates structures representing (aspects of) itself, and this representation is *causally connected* to the system it represents. Then we can say that the system has an accurate representation of itself, and also the status and computation of the system are always in compliance with this representation.

Metaobject protocols (MOP). In an object-oriented reflective language, objects that define functionalities are called *base objects* while objects defining execution mechanisms are called *metaobjects*. Protocols to manipulate meta-objects are called *metaobject protocols* ([Kiczales 1991]). Therefore, MOPs give users the ability to incrementally modify the languages behavior and implementation.

Examples of reflective languages. The level of support for reflection varies across languages. Java supports introspection, but not intercession. Python and Ruby support all operations except for behavioral intercession (*i.e.* intercession operations cannot modify the system's behavior). Smalltalk and CLOS support all kinds of reflection [Maes 1987, Smith 1984]: Performing self-surgery requires capacity for behavioral intercession. In our analysis we include only Smalltalk and CLOS.

2.4.2 Smalltalk Self-Surgery

Smalltalk is a dynamic reflective object-oriented programming language. Smalltalk reflection offers both introspection and intercession, and for which reflexivity is both structural and behavioral.

C1. Abstractions for dynamic semantics specification. Smalltalk reflective capabilities are offered in the form of metaobjects implementing protocols for their modification (MOPs). Smalltalk metaobjects include first-class entities for most of the elements of the language such as classes, methods, method dictionaries, etc. But also for dynamic execution elements such as context, message, etc.

Developers can add reflective facilities by examining the system metaobjects and inserting specific behavior. Some examples are the following.

- Variable read and write: If variable accesses are implemented using messages, as in Self [Ungar 1987]then reflective mechanisms for messages will also work for variables. Another alternative is to introduce active variables [Messick 1985]. However, this approach requires modifying the compiler to convert variable accesses into message sends to ActiveVariable objects, which in turn regulated access to their contents.

- Sending a message: Messages sent by an object can be intercepted, but this requires modifying the Smalltalk-80 compiler to wrap code around each send operation.

- Receiving a message: When a message sent to an object is not found in the inheritance hierarchy of that object's class, the Smalltalk virtual machine sends the object the message doesNotUnderstand:. The original message selector and message arguments are bundled together in a Message object and passed as the argument to doesNotUnderstand:. Users can implement forwarding mechanisms, no need for compiler modifications.

- Returns: If an object can intercept messages sent to it before dispatching them, its dispatching routine can inspect results returned to it before returning them itself. Thus, redefining returns falls out of redefining message dispatching.

Smalltalk has been extensively used for implementing and experimenting with new languages. Some examples, classified in three families, are:

Semantic Extensions. Define new features from within the language itself:
Garf [Garbinato 1995], Distributed Smalltalk [Bennett 1987] or [McCullough 1987] introduce object distribution in a transparent manner. Language features like multiple inheritance [Borning 1982], backtracking facilities [LaLonde 1988], instance-based programming [Beck 1993b, Beck 1993a], explicit metaclasses [Bouraqadi 2000], prototypes [Bergel 2004], or inter-objects connections [Ducasse 1995] have been introduced. Futures [Pascoe 1986, LaLonde 1991] or atomic messages [Foote 1989] are also based on message passing control capabilities.

Definition of new object models. Introducing concurrent aspects such as active objects ([Briot 1989]) and synchronization between asynchronous messages (Concurrent Smalltalk [Yokote 1987]). Other works propose new object models like the composition filter model [Aksit 1992]. or CodA [McAffer 1995] that is a metaobject protocol that controls all the activities of distributed objects.

Internal DSLs. In [Ducasse 2006], Smalltalk is referred as "A Reflective Executable Meta-Language", suitable to define DSLs in a similar way than object-oriented meta-languages (such as MOF or EMOF), but with the additional capacity of allowing description of operational semantics. PetitParser [PetitParser]: a framework for building modular parsers using Smalltalk code, applies this concept. This framework allows to dynamically reuse, compose, transform and extend grammars. (*i.e.* the BNF `ID ::= letter letter | digit ;` is expressed in PetitParser as `id := #letter asParser , (#letter asParser / #digit asParser) star`).

2.4.3 CLOS Self-Surgery

CLOS, the Common Lisp Object System [DeMichiel 1987, Bobrow 1988, Keene 1989, Gabriel 1991], is a class-based object-oriented layer, implemented on top of Common Lisp. Like Smalltalk, CLOS is a reflective language, offering both introspection and intercession, and for which reflexivity is both structural and behavioral. Unlike Smalltalk, in CLOS a class can have more than one superclass, and methods can be instance specific [Bobrow 1988].

C1. Abstractions for dynamic semantics specification. The CLOS MOP [Kiczales 1991] allows programmers to extend or modify both the syntax and the semantics of the language itself.

In CLOS the message passing concept is replaced by the generic function.[4] In consequence, the MOP allows users to control all the aspects of the generic function application, as message passing control is taken an entry point by the MOP [Kiczales 1991].

Developers can, for example, adjust aspects of the implementation strategy such as instance representation, or aspects of the language semantics such as multiple inheritance behavior [Kiczales 1993].

One limitation when compared to Smalltalk is that CLOS does not provide first-class representations for execution concepts such as contexts.

CLOS has been used to prototype experimental object-oriented paradigms such as context orientation [Hirschfeld 2008], filtered [Costanza 2008], and predicate dispatch [Ernst 998, Ucko 2001].

2.4.4 Self-Surgery Limitations

Limitations to this approach exist due to the difficulty of changing core parts of the system, breaking causal connections and meta-circularities at run-time [Chiba 1996], leaving the system in an inconsistent state and reducing the range of supported guest-languages.

The *metastability problem* is a well-known example of this. It occurs when meta-level code (*i.e.* code affecting metaobjects) triggers the execution of its own code, producing an infinite meta-call recursion and turning the system unusable [Kiczales 1991] This problem occurs often when modifying core language features such as Integers, Floats, and Arrays.

Denker et al. [Denker 2008] propose a solution to this issue by making reflection "context-aware", meaning that the system knows when meta-level code is being executed, and avoids meta-call recursions by copying problematic code and splitting execution in different levels. However, when the introduced change affects a great number of system components (*i.e.* changing the boolean operator 'not'), this solution is not enough. To implement this sort of modifications the system must be rebuilt while having them into consideration. By rebuilding the system we mean initializing its language runtime, as described in Section 2.6.

[4] A generic function is a group of methods. During the application of a generic function, methods from that group are selected to constitute an effective method application. This is the method that is executed.

2.5 LIT Evaluation

After describing modern techniques for language implementation, we compare them according to the evaluation criteria introduced in Section 2.2.1.

2.5.1 C1. Abstractions

Abstractions offered by LIT are compared regarding their expressiveness and their specifications abstraction level. A visual representation of the comparison is illustrated in Figure 2.3.

The most expressive way to specify syntax and static semantics is by coding a compiler from scratch or extending an existing one, but the abstraction level of specification is the lowest. Extensible compilers are the predecessors of DSLs for attribute grammars. They proposed a compiler whose architecture could be extended by specifying new AST nodes together with AST transformations. Polyglot [Caballero 2007], and JaCo [Zenger 2001] are examples of this approach. At the other extreme of the chart we find PetitParser, an example of using an internal DSL to define changes in the host-language compiler. The chart is consistent with the popularity of nowadays solutions for syntax and static semantics specification: DSLs for attribute grammars combined with DSLs for advanced type checking and name binding are a good option that balances expressiveness and level of abstraction.

The scenario for dynamic semantics specification is different. The most expressive option is the interpreter, but as with compilers, its abstraction level is low. DSLs for model transformations by themselves do not position well in the chart, however these solutions are often applied in combination with other techniques, such as operational semantics DSLs. DSLs for operational semantics have a good level of expressiveness, however their abstraction level is lower than that of MOP and metaobjects. Abstractions offered by self-surgery solutions are the most high-level, and their expressiveness remains high. CLOS MOP is more expressive than Smalltalk's metaobjects because it provides a more flexible object-model, which makes it also more complex. One limitation in CLOS is that it does not reify execution concepts, while Smalltalk does.

2.5.2 C2. Mapping From Specifications To Realizations

In generation techniques specifications and realizations are at different levels of abstraction. The logics to perform the mapping from one level to the next is found in the generator program. DSLs for syntax and static semantics specification allow defects

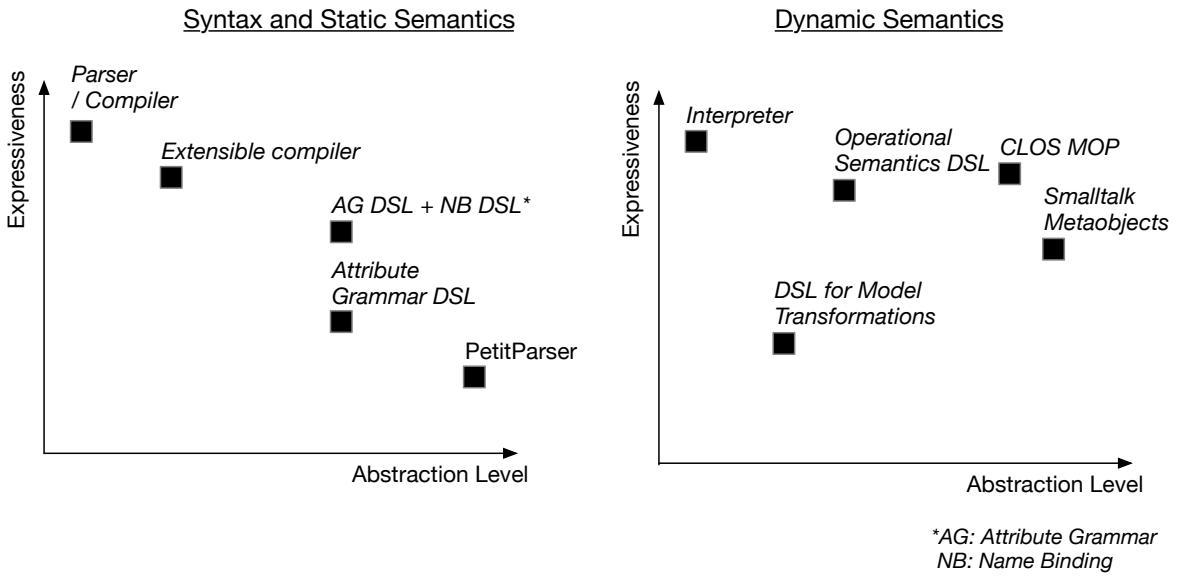


Figure 2.3: Comparison of abstractions for language specification in state of the art LIT.

to be are captured statically and debugged at the level of their specification code. By contrast, defects in interpreters and DSLs for dynamic semantics often cannot be captured statically. Some of these defects make the generator program fail or produce realizations that do not behave as expected. To find the cause of these failures, users must debug either generator code or the realizations code. This situation forces developers to mentally track back failures manifested at a lower abstraction level to their specifications.

Generation LIT implement different strategies to reduce the cognitive load required to debug these failures: generation of annotated code, multi-level debuggers, and compilation graphs are some examples. However, the abstraction gap in debugging tasks is not a problem of implementation, but rather it arises as a natural result of placing the logics for construction of language implementation in the generator program.

On the other hand, self-surgery LIT do not face the abstraction gap in debugging tasks problem. The host debugger is useful to debug both specifications and realizations. However, preventing metastability issues requires from users deep understanding about the system's object-model. Metastability issues manifest as critical failures which are hard debug because they make the system freeze or crash. Solutions to solve meta-stability issues [Denker 2008] effectively reduce the efforts to debug these failures. However, modifications to the semantics of the language that require to rebuild the full object-model are still not well handled.

2.5.3 Conclusion

Syntax and static semantics are in general well handled by current LIT so we leave them out of the context of this work. The alternatives for dynamic semantics specification are multiple, which makes them more interesting to study. Operational semantics DSL face the problem of hard-to-solve failures during the automatic mapping process. On the other hand, self-surgery approaches do not face this problem. Additionally, the abstraction level of metaobjects and CLOS is higher than that of operational semantics DSL. For these reasons we decide to focus our research on self-surgery techniques, studying how to overcome their limitations.

2.6 Overcoming Self-Surgery Limitations: Language Runtime Initialization

Language runtime initialization is required in high-level VM-based languages like Java, Ruby and Smalltalk, however they perform it in two different ways: *start-up initialization* and *ahead-of-time initialization*.

In start-up initialization, the language runtime is initialized each time the system starts. The process is defined as part of their VM initialization routines, which makes it hard to isolate and modify. Examples of this are: Ruby [Koichi] and Java (*e.g.* as in Java's bootstrap class loader [Oracle]).

In ahead-of-time initialization, the language runtime is initialized only once and then saved as a memory dump in disk. Initialization code is implemented in a process that is external to the Virtual Machine. The Bootstrap technique uses ahead-of-time initialization. Two languages using Bootstrap to initialize their language runtime are Pharo and CLOS.

2.6.1 Ruby and Python Runtime-Initialization

Ruby and Python bootstraps are very similar, therefore we explain only the case of Ruby. Ruby's bootstrap is implemented as part of its VM, which is written in C. The implementation of this process [5] mixes Ruby and C code.

At the beginning of the process, stub objects[6] (*e.g.* nil, true, the symbol table, etc) and stub classes are created, as shown in Listing 2.2, where classes are created using

[5] Version Yarv-MJIT [Kokubun]

[6] Stubs are temporary substitutes of elements in the runtime, which are necessary to build the rest of elements in the runtime.

C instructions. The result is the class hierarchy shown in Figure 2.4. Then, multiple modules containing classes are initialized, getting their definition from Ruby code. Once the modules are initialized, the virtual machine is operational and able to execute code.

Since Ruby's runtime initialization is implemented as part of the VM, there is no clear separation between language runtime and VM. Modifying this process to alter Ruby's semantics is difficult due to the high coupling between C and Ruby code, and because in case of failure, debugging C code is necessary. In this scenario, Ruby's abstractions (classes, objects, etc) are not available during the debugging process, as only low-level tools (GCC debugger) are available to manipulate and inspect Ruby objects.

```
1  void Init_class_hierarchy(void)
2  {
3      rb_cBasicObject = boot_defclass("BasicObject", 0);
4      rb_cObject = boot_defclass("Object", rb_cBasicObject);
5      rb_gc_register_mark_object(rb_cObject);
6
7      rb_cModule = boot_defclass("Module", rb_cObject);
8      rb_cClass =  boot_defclass("Class",  rb_cModule);
9  }
10 static VALUE
11 boot_defclass(const char *name, VALUE super)
12 {
13     VALUE obj = rb_class_boot(super);
14     ID id = rb_intern(name);
15
16     rb_name_class(obj, id);
17     rb_const_set((rb_cObject ? rb_cObject : obj), id, obj);
18     return obj;
19 }
```

Listing 2.2: Excerpt of Ruby's language runtime initialization process.

2.6.2 Discussion

When language-runtime initialization is implemented as part of the VM, as in Ruby and Python, the high-coupling between VM and language runtime makes the process hard to modify. Therefore, we focus our research in ahead-of-time initialization techniques.

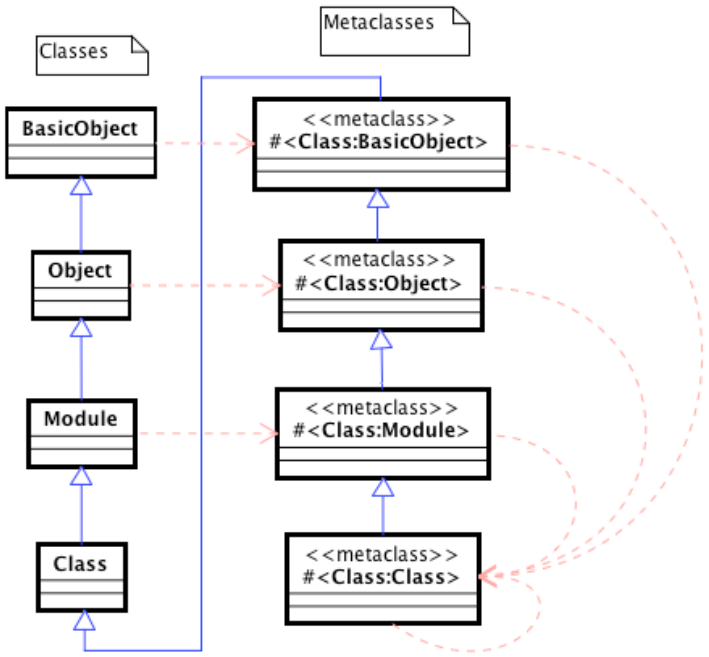


Figure 2.4: Ruby initial class hierarchy.

2.7 Bootstrap

Bootstrap is a high-level technique to implement ahead-of-time language runtime initialization, applied by reflective languages who make a clear distinction between language runtime (also named *kernel*) and VM, we represented in Figure 2.5. Bootstrap is not yet a language implementation technique, because it is traditionally used to produce a single language or a family of similar languages.

Bootstrapping a kernel is the process that builds the minimal structure of a language that is reusable to define this language itself. The idea is to use as early as possible the benefits of the resulting language by implementing a minimal core whose goal is to be able to build the full system. Few languages implement ahead-of-time language-runtime initialization in bootstrap. Below we present two existing systems implementing this solution.

2.7.1 Common Lisp Bootstrap

Common Lisp, like Smalltalk, has the concept of image apart from VM. Taking ideas from [Rhodes 2008], Durand et al. [Durand 2019] bootstrap Common Lisp from a previous functional version of Common Lisp. They create a Common Lisp system by building

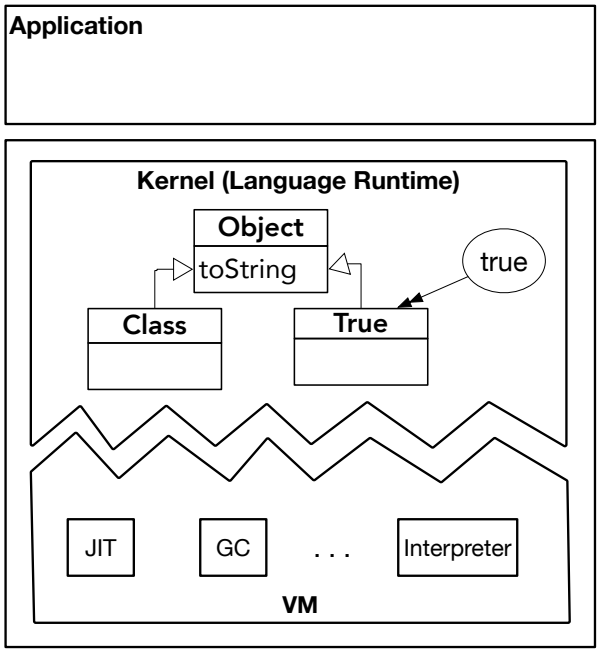


Figure 2.5: VM and language runtime.

it from its associated source code (which is written in Common Lisp). They use various tools and language processors to transform that source code into an executable file (reusing the compiler in the host system), which is finally dumped to disk and loaded by the VM.

Their approach is original because they build the CLOS MOP classes and generic functions first, allowing them to use the CLOS machinery for building many other parts of the system, thereby decreasing the amount of special-purpose code, improving maintainability of the system, and increasing the level of abstraction.

Their technique represents great advantages to maintenance of the bootstrapped system. Contrary to Ruby, there are no dependencies between CLOS code and other code that require duplication of information that must be kept synchronized when some code is modified.

Although their implementation takes advantage of abstractions provided by the MOP, it is language specific. Introducing modifications requires experience with the bootstrap process, with the resulting object-graph structure constraints, and with the target VM.

2.7.2 Smalltalk Bootstrap

We have selected the programming language Pharo [Kiczales 1991], a Smalltalk inspired language, to analyze Bootstrap in Smalltalk. This decision derives from the maturity level of the Pharo bootstrap: Pharo's official image is currently bootstrapped from its sources.

The Pharo bootstrap generates kernels containing the language runtime that are executed in the Pharo VM. Polito et al. [Polito 2015] have proven this technique effective to generate *Pharo*-like *languages* (*e.g.* MetaTalk and Candle) while always targeting the Pharo VM. Pharo-like languages are languages with an execution semantics close to Pharo, but with a different class-model (*e.g.* Traits in Pharo[7] are implemented in classes, removing these classes from the kernel also removes Traits from the language). Chapter 3 presents the Pharo bootstrap technique in detail.

The Pharo Bootstrap process occurs in stages as represented in Figure 2.6. An generator application, running in a previous functional version of Smalltalk, takes as input a language specification and generates a kernel. The language definition is highly coupled to both the bootstrap process and the target VM. Modifying this definition or the generator application demands familiarity with the bootstrap process and VM implementation.

Failures manifesting at different stages provide information about domains other than language definition domains. The worst case scenario occurs when a failure manifests during the kernel execution in the VM (*e.g.* a defect in the language definition produces a segmentation fault in the VM). Debugging failures at a high abstraction level is in general nos supported.

GLApp code (application code, written in guest language) is compiled into target VM bytecode using the compiler in the host Smalltalk. The produced bytecode is loaded into the kernel. Finally, the kernel is executed by the target VM.

The variety of languages that can be bootstrapped while targeting a single VM has not been explored.

2.7.3 Discussion

We think that Bootstrap has the potential to become a LIT, because it applies the same principles as self-surgery, being able to benefit from the same advantages for cognitive distance reduction. We believe that it is possible to adapt bootstrap so that definition and generation of multiple languages is possible for developers with no previous

[7]Traits are a composition mechanism alternative to multiple or mixin inheritance.

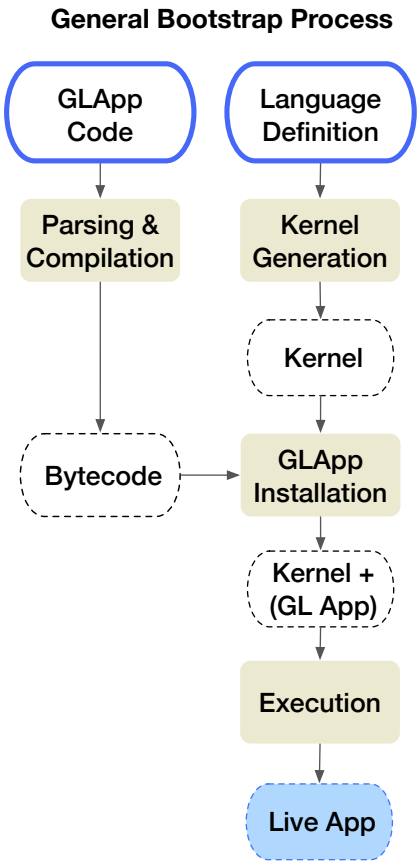


Figure 2.6: Bootstrap multi-stage process.

experience to bootstrap nor to virtual machine implementation.

Both Common Lisp and Smalltalk Bootstrap are high-level implementations to language-runtime initialization. While Common Lisp bootstrap is not in a mature state, Smalltalk implementations, such as Pharo, have a mature bootstrap process. Moreover, previous work for bootstrapping Pharo-like languages exists [Polito 2015]. Additionally, CLOS MOP, although expressive, remains complex. We take Pharo's Bootstrap as the starting point in our way to explore whether bootstrap can be transformed into a LIT or not.

2.8 Conclusions

In this Chapter we have presented the state of the art in language implementation techniques, paying particular attention to the cognitive burden they impose on developers using them. We have seen that the specification of syntax and static semantics is well

supported by current solutions, but this is not true for dynamic semantics since DSLs usually require customizations introduced in the base language. The self-surgery approach to dynamic semantics specification has the advantage of preventing debugging challenges associated to mapping between specifications and realizations. Also, the level of abstraction of dynamic semantics specification is the highest among all studied techniques. However, the variety of produced languages is limited by metastability issues. This issues can be solved with the help of Bootstrap. We think that applying concepts from self-surgery to Bootstrap reduces the cognitive distance, bringing the technique one step forward to become a LIT. We have decided to take Pharo's Bootstrap as the starting point in our research, because it has a mature implementation and it has been already used to define Pharo-like languages.

Chapter 3

Challenges Bootstrapping Reflective Kernels

Contents

In this Chapter we analyze Bootstrap extensively, paying special attention to the reasons for increasing cognitive distance. Section 3.1 uses Pharo [Ducasse 2017], a Smalltalk based reflective language, as an example to present an in-deep explanation of the bootstrap process. Section 3.2 analyzes the causes for failures appearing during bootstraps and proposes a taxonomy where defects, failures, and underlying causes are interrelated. Section 3.3 describes the process of bootstrapping a language different from Pharo to concretely show challenges faced by developers. Section 3.4 analyzes the reasons for increased cognitive distance in Bootstrap. Finally, Section 3.6 closes this chapter with our final observations.

During this chapter, and the rest of this dissertation, we use the concepts about cognitive distance, defects, and failures presented in Section 2.1. To the previous concepts we add the ones below.

- *Kernel:* is the realization[1] of the language-runtime to be generated. The kernel is an array of bytes in host memory during its generation, and a binary file in disk after its exportation.
- *Bootstrapper:* application that generates the new kernel.
- *Host:* system where the bootstrapper runs.
- *Host-language:* language of the host system.
- *Guest-language:* language whose kernel is generated.

3.1 Pharo Bootstrap

We use Pharo to present a deep explanation of the Bootstrap technique, and to analyze causes of cognitive increment.

[1]Software abstractions, as explained in Section 1.1 and proposed by Krueger [Krueger 1992], have two levels: specification and realization. Specification is at the abstraction's highest-level of detail and realization at the lowest.

3.1.1 Pharo Bootstrap in a Nutshell

As presented in Figure 3.1, kernels are generated by a standard Pharo application, named *bootstrapper*, which runs in a full Pharo system that serves as *host*. The process starts when the bootstrapper reads the language sources ((A), see Section 3.1.2) and produces a language *model* accordingly to the sources. Then, it applies *kernel generation instructions* ((B, C), see Section 3.1.3)defined in the bootstrapper to generate the kernel. During kernel generation, the host compiler is used to compile guest-language code into bytecode which is then installed in the kernel.

Before exporting the kernel, the bootstrapper installs the guest-language application (D) in the kernel. Even though language kernels are independent from applications, in our analysis we must consider the application code because bootstrap late failures manifest during the execution of application code. After this step, the kernel is serialized and written as a file to disk, which ends the generation stage.

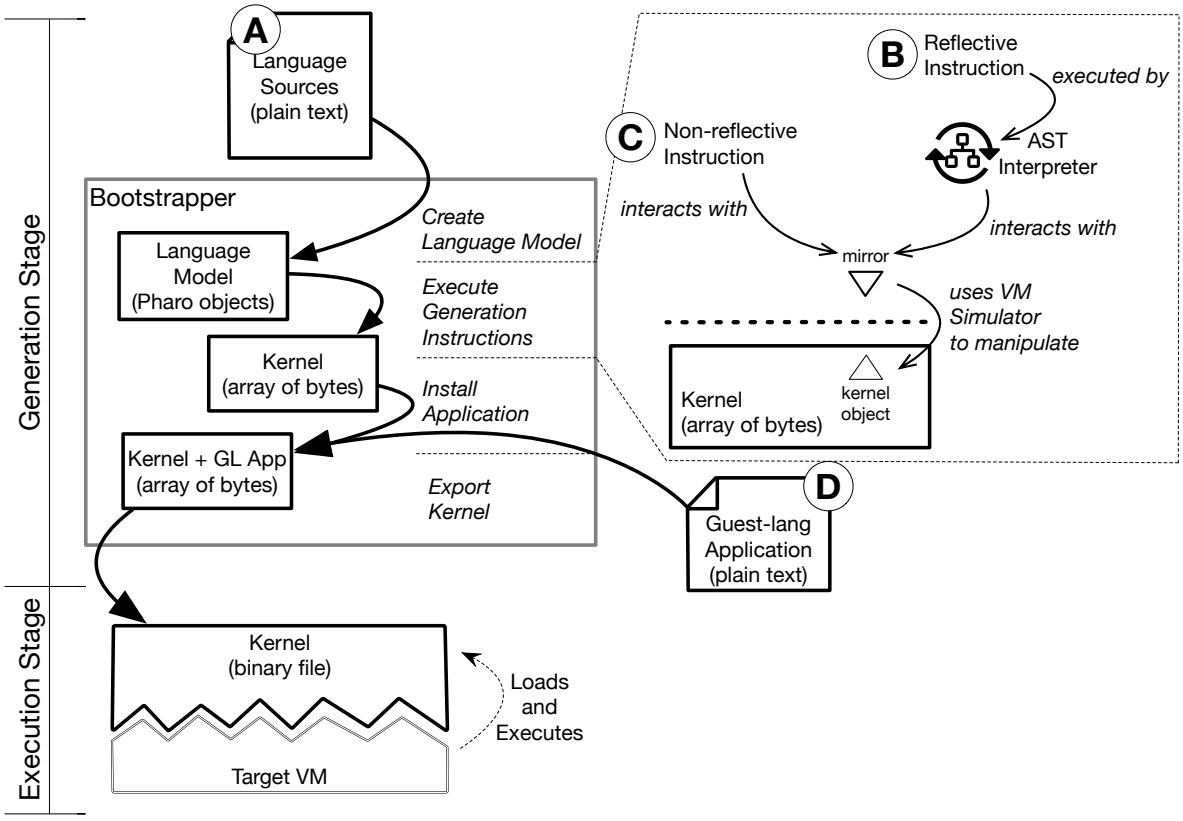


Figure 3.1: Pharo bootstrap overview.

A kernel in host memory is, at the low-level, an array of bytes. Objects in the kernel, named *kernel-objects* are manipulated through *mirrors* [Papoulias 2011]. Mirrors are

objects in the host used to encapsulate the physical representation of kernel-objects. Mirrors use the *VM Simulator Library* to manipulate the kernel. This library implements low-level operations to build and modify kernel-objects at the level of bits. The VM Simulator Library is installed in the host, and it is specific for the Pharo VM.

The execution stage, or run-time, begins when the Pharo VM loads the kernel file, and starts reading specific sections in it. At one point, the it retrieves the kernel's entry point, and starts bytecode interpretation. Kernels must fulfil the *VM constraints* (Section 3.2.1), otherwise the VM fails.

Because to create a new language developers must modify language sources and generation instructions, the *language definition* comprises all of these elements. The rest of this section describes the language definition by parts.

3.1.2 (A) Language Sources and Language Model (declarative)

Language sources are plain text files defining the structure of classes and methods to be installed in the kernel. They are used by the Bootstrapper to build the language model. The next code is an excerpt of Pharo's sources, showing the definition of the class ProcessorScheduler (used to coordinate processes in the system) The definition of its method activeProcess is also presented.

```
1  "(A) language sources"
2  class {
3    name: #ProcessorScheduler
4    superclass: #Object
5    instVarsNames: 'processList activeProcess'
6    classVarsNames: ''
7  }
8  ProcessorScheduler >> activeProcess
9    "Answer the currently running Process."
10   ^ activeProcess
```

To avoid interaction with plain text, the Bootstrapper creates the *language model* by creating sources into objects in the host system. These objects are named *object-models*. They are instances of classes in the host system. The language *metamodel* is the set of classes in the host-system whose instances are object-models. The metamodel is an existent component of the host, used to model the host language itself. We call *class-models* to object-models representing classes, and *method-models* to those representing methods. The Bootstrapper creates object-models according to specifications in language sources.

3.1.3 (B, C) Generation Instructions (imperative)

Generation instructions are imperative instructions executed to create kernel-objects based on the contents of the model. Each object-model is installed in the kernel *e.g.* the kernel-object ProcessorScheduler is based on the object-model with the same name, it has Object as superclass and two instance variables named processList and activeProcess (Section 3.1.2).

Some kernel-objects are not represented in the model. However they still depend on it, as they are created by instantiating kernel-classes defined by class-models. (*e.g.* the object Processor is created by instantiating the kernel-class ProcessorScheduler).

Generation instructions can be written using both guest-language and host language code. Depending on its language, we classify instructions as follows:

3.1.3.1 (B) Reflective instructions

Written in guest-language. They are reflective intercession operations through which the guest-language modifies itself. The next code is an extract of Pharo's Bootstrapper, showing reflective instructions to create and initialize the lists of processes in the kernel. In these instructions, the object Processor is created as an instance of ProcessorScheduler and stored as an entry inside Smalltalk. A list of processes is stored in Processor's first instance variable.

```
1 "(B) reflective instruction initializing the system's process scheduler. The following is guest language code"
2 Smalltalk
3   at: #Processor
4   put: (ProcessorScheduler basicNew).
5 (Smalltalk at: #Processor)
6   instVarAt: 1
7   put: ((1 to: 80) collect: [ :i | ProcessList new ])
```

Reflective instructions make sense in the kernel context, not in the host. Meaning that classes, methods and variables appearing in their code refer to objects in the kernel. For this reason, they need to be evaluated in a special way. The Bootstrapper uses a custom AST interpreter to evaluate reflective operations. Pharo's parser is used to pass from guest-language code to AST. The AST is then given to the AST interpreter. This interpreter visits the AST nodes performing evaluation. Variable bindings are resolved in the kernel context *e.g.* class names refer to classes installed in the kernel. Even though message evaluation is applied on kernel-objects, the lookup of methods is done in class-models of the language model. Method's source code is obtained from method-models found in these class-models.

The result of the evaluation is a mirror pointing to the returned kernel-object. Therefore, our previous code example, is actually implemented as follows.

```
1  Bootstrapper >> initializeProcessScheduler
2    | mirror |
3    "(B) + AST interp"
4    self interpreter evaluate: '
5      Smalltalk
6        at: #Processor
7        put: (ProcessorScheduler basicNew).
8      (Smalltalk at: #Processor)
9        instVarAt: 1
10       put: ((1 to: 80) collect: [ :i | ProcessList new ])'.
11   mirror := self interpreter evaluate: '
12     Smalltalk globals associationAt: #Processor.'.
13   ...
```

3.1.3.2 (C) Non-reflective instructions

These are instructions written in host language. Objects in the kernel are directly manipulated using their corresponding mirrors. The following code shows the continuation of the method initializeProcessScheduler. The object Processor must be referenced from an association stored in a specific position inside a special kernel-object named special objects array. The Bootstrapper uses the method kernel to get an object representing the kernel in memory. The message specialObjectsArray sent to kernel returns a mirror. The message at:put: is sent to this mirror to store the corresponding association.

The mirror directly manipulates its representation in the kernel. There is no need to pass through guest-language code. Non-reflective operations are provided by mirrors. They include basic reflective operations such as getting and setting values for instance variables, getting the class of an object, getting and setting elements in collections, and instantiation operations.

```
1  Bootstrapper >> initializeProcessScheduler
2    ...
3    "(C) non−reflective instruction"
4    self kernel specialObjectsArray at: 4 put: mirrorToProcessorAssoc
```

3.2 Causes of Bootstrap Failures

One of the main causes of increased cognitive distance in Bootstrap is the difficulty to solve errors occurring during the generation of the kernel. In this Section, we analyze

the reasons behind these failures and their relationship to the language specification.

3.2.1 VM Constraints on the Kernel Structure

VMs manipulate data in memory while executing applications. Kernels structure is partially fixed by the VM, as it contains data to be accesses and manipulated by the VM. Data in the kernel must respect specific layouts in memory (at level of bits), and certain data structures cannot be missing (*e.g.* basic elements representing the booleans values true and false). We call *VM constraints* to the set of restrictions the VM imposes on the structure of kernels.

In object-oriented languages, objects are the unit of data encapsulation. In Pharo, everything is an object and even primitive types are instance of a class. Therefore, Pharo VM restrictions concretely mean that the kernel must contain specific objects and classes, and that their representation in memory must follow specific layouts. For example, the Pharo VM expects that an object representing undefined values, namely nil, is stored in a specific position in the kernel, and that its layout has a fixed size of zero (*i.e.* nil cannot store data internally). Additionally, as in Pharo every object must be instance of a class, the class UndefinedObject must also exist. This class must respect the layout constraints imposed for its instances: UndefinedObject must declare exactly zero instance variables.

Not all VM constraints are low-level. Some are related to VM assumptions on relationships between language objects, and partially fix the language semantics. For example, the Pharo VM assumes single inheritance for method lookup, forcing that each class in the kernel has at most one superclass.

Bootstrapping a kernel that complies with VM constraints requires that every part of the language definition (*i.e.* the bootstrapper, the kernel source code, and the language model) from which the kernel is generated respect VM constraints. We say that definitions and kernels are *valid* when they fulfill VM constraints, otherwise they are *corrupt*.

3.2.2 Classification of Defects and Failures

Defects in the language definition produce failures at different moments of bootstrap, even at run-time when the VM interprets application code. Defects in the VM are beyond the scope of this research. We analyze and classify defects and their consequent failures in bootstrap to finally propose the taxonomy of defects and failures presented in Section 3.2.3.

3.2.2.1 Classification of Defects

We classify defects according to their location in the definition of the language.

Structural (declarative). Structural defects are found in language sources, which are used to build the the model. They are related to missing or corrupt definition of classes. They always produce *VM Constraints Failures* (Section 3.2.2.2). In *ObjVLisp* bootstrap example, fully described in Section 3.3, failures presented in Sections 3.3.2, 3.3.3, and 3.3.6 have their origin in structural defects.

Non-Reflective (imperative). Non-Reflective defects are found in Bootstrapper code, which is written in host language. They are related to instructions that corrupt the kernel structure. They produce *VM Constraints Failures* (Section 3.2.2.2). We do not analyze non-reflective defects producing failures unrelated to kernel corruption (*e.g.* sending an inexistent message), because they are debugged using the host debugger, just like normal host applications. Example in Section 3.3.7, corresponds to this kind of defect.

Reflective (imperative). Reflective defects are found in reflective instructions, which are written in guest-language. Some reflective defects produce kernel corruption, manifesting as *VM Constraints Failures* at any stage of bootstrap and run-time (Section 3.2.2.2). Some reflective defects do not produce kernel corruption, but they make the AST interpreter fail, preventing their own execution, and manifesting as *Guest-Language Code Failures* (Section 3.2.2.2) during kernel generation. Section 3.3.5 presents presents an example where kernel corruption is produced.

Application (imperative). They are found in application code that is executed at run-time. This code is written in guest-language. They do not relate to kernel corruption, but to problems in the application. They produce *Guest-Language Code Failures* (Section 3.2.2.2). They are are independent of VM constraints. Section 3.3.8 presents an example. Application defects produce failures at run-time, never before. Information about the produced failure depends on debugging features implemented in the kernel.

3.2.2.2 Classification of Failures

We classify failures according to their root cause.

VM Constraints Fails. Produced when either class-models or the generated kernel do not fulfill VM constraints. VM constraints are explained in Section 3.2.1. VM constraints failures arise at any stage of bootstrap, and manifest as either a handled exception, (*i.e.* showing information about the error), or as an unhandled exception (*i.e.* showing no further information, such as in segmentation faults).

Guest-Language Code Fails. Produced when guest-language code cannot be executed due to invalid operations (*e.g.* sending an inexistent message), or when its execution has an unexpected result. Failures of this kind are independent of VM constraints. They manifest as either as an exception during the AST interpretation process, as a VM exception during at run-time, or as an unexpected result, also at run-time. Debugging requires debugging guest-language code, but no tools are available do do this.

3.2.3 Taxonomy of Defects and Failures

Based on the previous classification, we build a taxonomy where defects, failures, and their underlying causes are interrelated. Figure 3.2 presents our taxonomy of defects and failures arising during the bootstrap process. Each cell relates type of defect, with stage of failure manifestation, with type of failure. From this table we conclude that:

- Structural and Non-Reflective defects are always related to VM Constraints, manifesting as failures at any stage of the process.
- Application defects produce failures manifesting only at run-time. They are unrelated to VM constraints, having their cause in errors in guest-language code.
- Reflective defects producing failures manifested during the generation stage, are caused by both: VM constraints and Semantic errors in guest-language code. In the second case, the kernel generation process is always interrupted. In the first case, the generation process is either interrupted due to kernel corruption, or a corrupt kernel is fully generated. Therefore, failures due to this kind of defects, manifesting at run-time, are always related to VM requirements.

3.3 Challenges Bootstrapping ObjVLisp

To present challenges associated to the bootstrap technique, in this Section we play the role of a developer modifying Pharo's bootstrap to generate ObjVLisp [Cointe 1987]. ObjVLisp is a minimal class-based language with explicit metaclasses (unlike Pharo,

		Failure Stage	
		Generation	Execution
Defect Type	Structural	VM Constraint	VM Constraint
	Non-Reflective	VM Constraint	VM Constraint
	Reflective	VM Constraint / Guest-Lang Code	VM Constraint
	Application	n/a	Guest-Lang Code

Figure 3.2: Taxonomy of Defects and Failures in Bootstrap.

which implements implicit metaclasses). ObjVLisp defines only two classes: OLObject and OLClass (see Figure 3.3). We use the prefix OL to name classes to explicitly distinguish them from classes in the host.

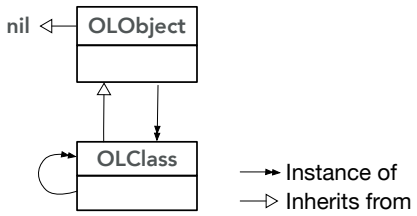


Figure 3.3: OLObject and OLClass are the only classes defined in ObjVLisp.

We create our language definition source code from scratch, editing plain text files defining classes and methods in Tonel format. We define only two class-models: OLObject and OLClass. To provide a way to access methods installed in classes from language code, we declare the instance variable methods in OLClass, where each class will store its dictionary of methods.

```
1 class {
2   name: #OLObject
3   superclass: nil
4 }
5 class {
6   name: #OLClass
7   superclass: #OLObject
8   instVarsNames: 'methods'
9 }
```

3.3.1 Pharo's Metamodel Does Not Support Explicit Metaclasses

The first challenge we face is that Pharo's metamodel (Ring [Ring2]) is designed for implicit metaclasses. Therefore, it automatically creates one metaclass for each class it finds in the Tonel repository. Overcoming this challenge requires multiple modifications to Ring. We do not explain here how this was done because it is just a technical matter, but make clear that Pharo's metamodel is not flexible enough to support modelling languages too different from Pharo.

3.3.2 Structural Def. causes VM Const. Fail at Generation (easy)

Undefined class-model produces failure during kernel generation Nevertheless we continue with the Bootstrapper execution, leaving the resolution of this problem for later. The following failure is produced from method createStubForClassNamed:.

Error

Using the host debugger to understand this failure, we realize that it is produced during the creations of stubs and that the names of 20 classes are accessed by their name. We add these classes to our structural definitions and modify their names in the Bootstrapper adding the prefix OL.

```
1 class {
2   name : #OLArray
3   superclass : #OLObject
4   instVarsNames: 'methods'
5 }
6 ...
```

3.3.3 Structural Def. causes VM Const. Fail at Generation (hard)

Class-Model defining wrong format produces failure during kernel generation.

We execute again the Bootstrapper, obtaining a new failure as shown below.

Error: # '\\' was sent to nil.

Figure 3.4 presents a screenshot of the host debugger at the moment this failure manifests.

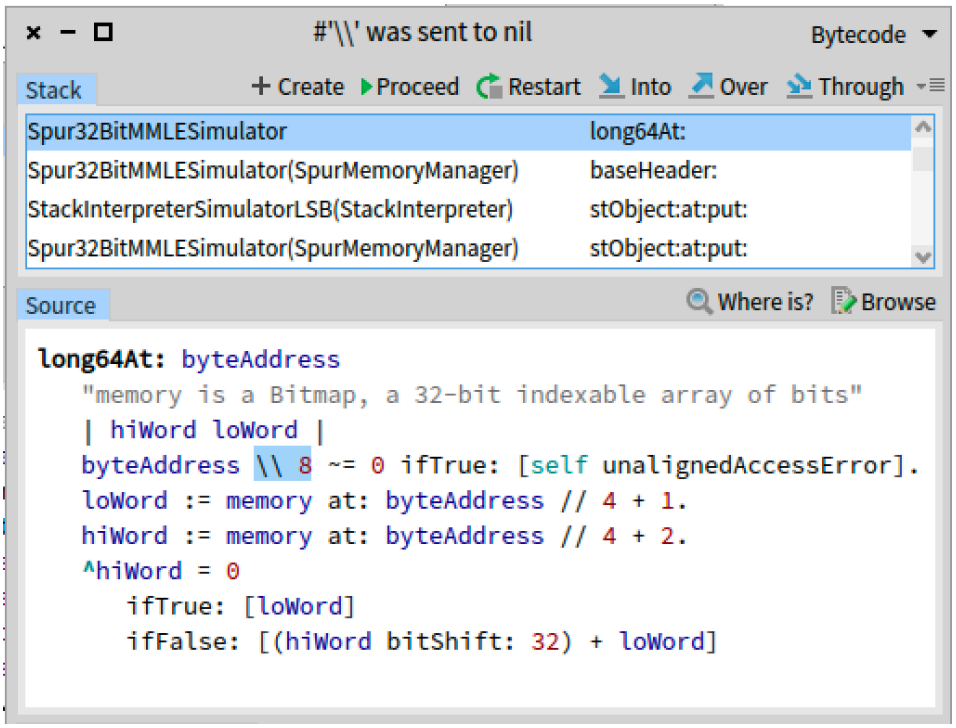


Figure 3.4: Host debugger showing the error presented in Section 3.3.3

Debugging this failure, we realize that the problem occurs because the special objects array mirror is not referencing any object in the kernel, but instead it points to nil (the object nil in the host). Therefore, when the Boostrapper tries to store an object for the first time in the special objects array, the method long64At:, called by stObject:at:put: (both methods belonging to the VM Simulator Library) fails with a non-intuitive failure (see the call stack in Figure 3.4).

To find the way to fix this error, we must debug the code that creates the special objects array. Eventually we arrive again to VM Simulator code, and find the cause in the method below.

```
1  VMSimulator >> newBootstrapInstanceFromClassFormat: classFormat indexableSize: nElements classIndex:
       classIndex
2    "Allocate an instance of a variable class, except CompiledMethod."
3    | instSpec numSlots newObj fillValue |
4    instSpec := self instSpecOfClassFormat: classFormat.
5    fillValue := 0.
6    instSpec caseOf: {
7      [self arrayFormat]  ->
8        [numSlots := nElements.
9         fillValue := nilObj].
10     ...
11     ...
12     }
13     otherwise: ["some Squeak images include funky fixed subclasses of abstract variable
14   superclasses. e.g. DirectoryEntry as a subclass of ArrayedCollection.
15   Allow fixed classes to be instantiated here iff nElements = 0"
16             (nElements ~= 0 or: [instSpec > self lastPointerFormat]) ifTrue:
17               [^nil].
18             numSlots := self fixedFieldsOfClassFormat: classFormat].
19   ...
```

The previous method returns nil (line 17) because the variable instSpec, which depends on classFormat (line 4) given as parameter, does not match the expected value for arrayFormat (line 7). The argument classFormat is read from the language model. Eventually we realize that the structural definition of OLArray is missing the specification for its type, which should be variable. The correct definition should be:

```
1  class {
2    name : #OLArray
3    superclass : #OLObject
4    type: #variable
5  }
```

To prevent this problem from happening again, we decide to select and copy structural definitions from Pharo. However, Pharo's kernel is too large and cherry picking structural definitions is not an easy task.

Therefore, we decide base our structural definitions on a minimal hand-crafted Pharo-like language named *Candle*. Candle defines only 49 classes, and implements basic support for basic type operations and basic IO support through files. Since Candle implements implicit metaclasses we modify it to have explicit metaclasses like $Ovlisp_L$.

3.3.4 Reflective Def. causes Guest-Lang. Code Fail at Generation

Reflective instruction accessing undeclared variable produces failure during kernel generation.

Pharo Bootstrap uses a class installer defined as part of its structural definitions to install the definitive version of classes in the kernel. In the same way, Candle implements its own class installer. We modify it to fit ObjVLisp needs. ObjVLisp's class installer is implemented as a class and its methods in the guest-language structural definitions. ObjVLisp's class installer is the class OLClassBuilder.

As shown in the code extract below, the Bootstrapper executes guest-language code that creates an instance of the class OLClassBuilder (line 5). The instance of OLClassBuilder is configured according to information extracted from the class-model: superclass, class name, instance variables and type (lines 6 to 9). By sending the message build to this instance (line 11), the class defined by the class-model received as parameter is installed in the kernel.

The Bootstrapper uses its custom AST interpreter to execute reflective generation instructions (line 13), which are written in guest-language code. The result of the evaluation is expected to be a mirror to the new class installed in the kernel.

```
1  Bootstrapper >> basicInstallClass: classModel
2    | type code mirror |
3    type := self typeFor: classModel.
4    code := '| newClass |
5      newClass := (OLClassBuilder new
6          superclass: ', classModel superclass name ,';
7          name: ', classModel name ,';
8          instVarNames: ', classModel instVarNames ,';
9          type: ', type ,';
10         yourself)
11       build.
12     newClass'.
13   mirror := self interpreter evaluate: code.
14   ^ mirror
```

The next piece of code shows the structural definition of class OLClassBuilder, its method build, and the method instSize are sent by build (line 11 in previous code extract). We created this method by modifying the build method found in Candle, as follows:

```
1  OLClassBuilder
2    superclass: #OLObject
3    instVarsNames: 'name superclass instVarNames formats'
4
5  OLClassBuilder >> build
6    | theClass |
7    theClass := OLClass new.
8    theClass superclass: superclass.
9    theClass setFormat: ((self instSpec bitShift: 16) bitOr: self instSize).
10   theClass name: name.
11   ^ theClass.
12
13 OLClassBuilder >> instSize
14   ^ (superclass ifNil: [ 0 ] ifNotNil: [ superclass instSize ])
15     + instVarNames size
```

We execute the Boostrapper, but execution never concludes. We interrupt execution opening a host debugger window, presented in Figure 3.5. The AST interpreter used to execute reflective instructions has fallen into an infinite loop. Debugging guest-language code by debugging the AST interpreter is not an easy task. Objects in the kernel are accessed through mirrors that only display their address in memory when inspected. This is the case for the object receiver. To have hints about which code is being executed, it is necessary to inspect the AST nodes, as shown for the case of the message node.

Our intuition leads us to suspect that the receiver is actually the object nil. To confirm our hypothesis we compare the address of receiver with the address of nil, as shown in the lower right corner of the host debugger in Figure 3.5. We confirm that receiver in fact nil. This happened because we did not initialize the value for instance variable names when the building class, preventing its instance size from being calculated inside the method instSize. We fix this defect by introducing a line of code that sends the message instVarNames: with the set of instance variables before line 9.

3.3.5 Reflective Def. causes VM Constraint Fail at Generation

Reflective instruction corrupting the kernel produces failure during kernel generation.

After fixing our code as described above, we execute the Bootstrapper once again, but the following failure is produced.

'Instance of EPPrimitiveFailed did not understand #bytecodes:'

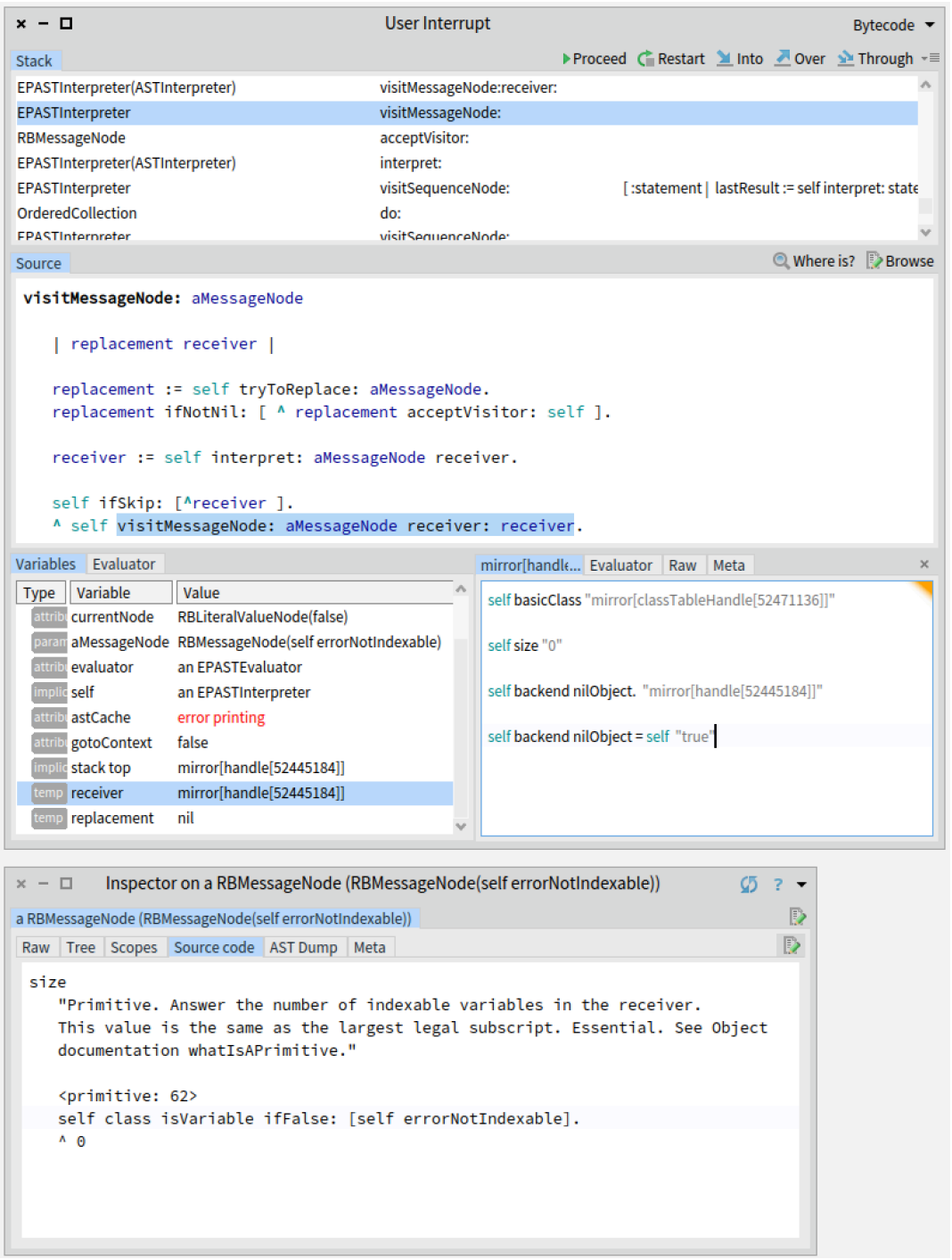


Figure 3.5: Debugging guest-language code execution by using the host debugger to debug AST interpreter execution.

The message bytecodes: is sent to an instance of EPPrimitiveFailed. Debugging Bootstrapper code we realize that the Bootstrapper expected an instance of the class OLCompiledMethod.

We debug Bootstrapper code, eventually arriving to VM simulator code where an instance of OLCompiledMethod is created by executing a primitive (see Figure 3.6a).

We realize that the VM expects the value of instSpec for the class OLCompiled-Method to be 24. We learn this value by evaluating the message firstCompiledMethod-Format (line 8).

We check our structural definition for the class OLCompiledMethod and find its type to be correctly set as compiledMethod. Consequentially, its class-model instSpec is correctly set in 24. However, for some reason, the value of instSpec for the class in the kernel does not match the one specified in its structural definition.

To discover why this is happening, we must debug the process used to install classes. However, this is a complex task, because this code is written in guest-language, and it is interpreted by an AST interpreter. The only way to debug this code is debugging the interpreter's execution as it interprets guest-language code, by traversing its AST nodes, as shown in Figure 3.6b.

We finally discover that the mistake is in the method OLClassInstaller » instSpec, which returned the value 12 for the OLCompiledMethod class, instead of 24. Old versions of Pharo VM used the value 12 for the instSpec of CompiledMethod. Candle's class builder code was outdated.

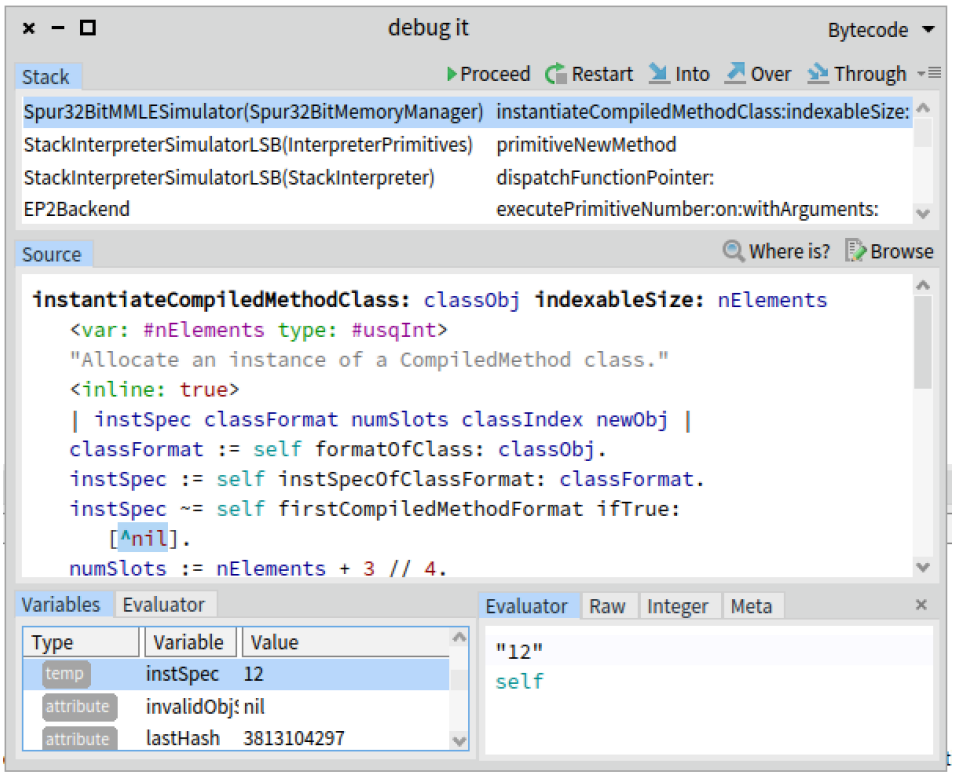
Finding this defect back in our reflective instructions took big efforts, since the host debugger is not suitable to debug guest-language code.

3.3.6 Structural Def. causes VM Constraint Fail at Execution

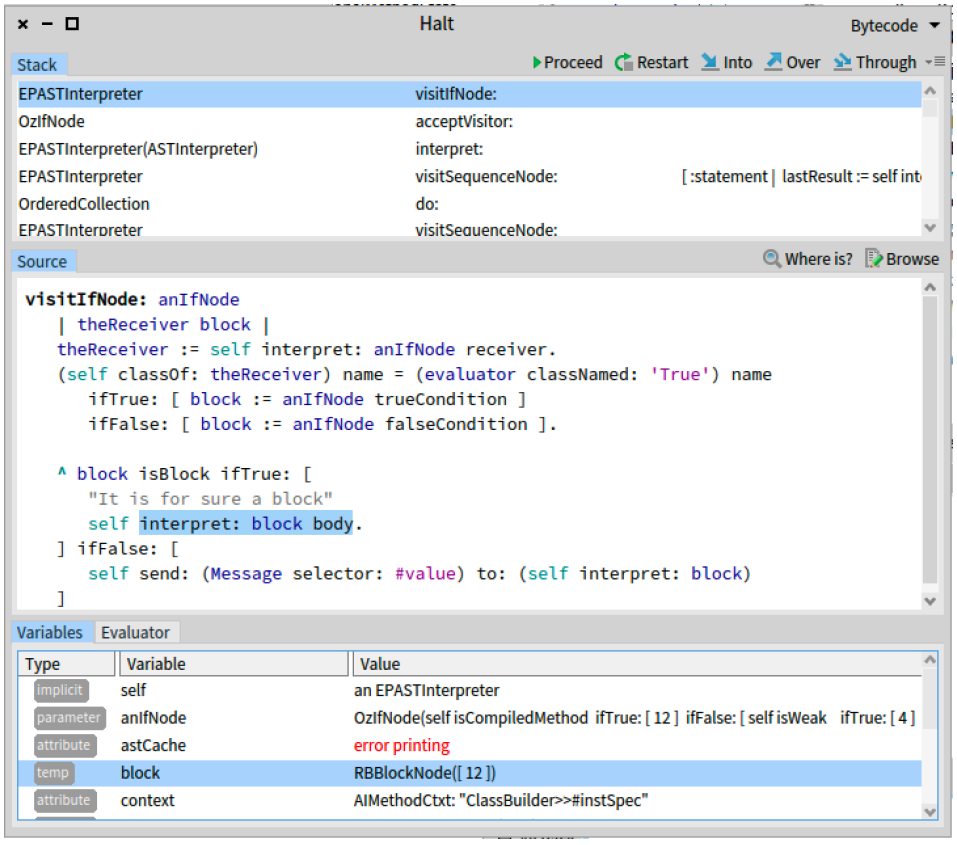
Class-Model defining instance variables in wrong order produces failure during kernel execution.

Candle's structural definition for its equivalent to OLClass defines the next 3 instance variables: superclass, methods, and format. We add the missing instance variables superclass and format to OLClass:

```
1 class {
2   name : #OLClass
3   superclass : #OLObject
4   instVarsNames: 'methods superclass format'
5 }
```



(a) Debugging VM simulator code to understand Primitive Failure error shown in Section 3.3.5.



(b) Debugging AST Interpreter while executing reflective instructions shown in Listing 3.1.

Figure 3.6: Debugging primitive failure when instantiating OLCompiledMethod.

```
1  OLClassBuilder >> instSpec
2    ^ self isImmediate
3      ifTrue: [ 7 ]
4      ifFalse: [ self isCompiledMethod
5        ifTrue: [ 12 ]
6        ifFalse: [ self isWeak
7            ifTrue: [ 4 ]
8            ifFalse: [ self isPointers
9                ifTrue: [ self isVariable
10                   ifTrue: [ self instSize > 0
11                       ifTrue: [ 3 ]
12                       ifFalse: [ 2 ] ]
13                   ifFalse: [ self instSize > 0
14                       ifTrue: [ 1 ]
15                       ifFalse: [ 0 ] ] ]
16               ifFalse: [ self isWords
17                   ifTrue: [ 10 ]
18                   ifFalse: [ 16 ] ] ] ] ]
```

Listing 3.1: OLClassBuilder » instSpec, copied from Candle, has a semantic defect in line 5: instSpec for CompiledMethod should be 24 instead of 12 according to VM constraints.

We execute the Bootstrapper and successfully generate the kernel. When we load the kernel in the Pharo VM. A failure occurs manifesting with the uninformative message shown below.

```
Abort trap: 6
```

We debug VM code to find the cause of this error and realize that classes must store their methods in their second instance variable, and their superclass in the first instance variable. Therefore, the first two instance variables declared by OLClass must be superclass and methods.

Listing 3.2 shows information shown by the VM debugger to find the cause of this failure.

The first time a message is sent, the VM searches for it in the method dictionary of the receiver's class. This is is effectively a class object. The VM assumes that the method dictionary is stored in its second instance variable. The VM finds instead, the superclass of the current class in that position. It handles the error occurring when trying to access a corrupt method dictionary, and tries to continue the lookup process fetching the current class' superclass. But in that position there is no class, instead there is a method dictionary.

The current class is now a method dictionary. The VM tries again to find the

```
1  [(lldb) bt
2  * thread #1, queue = 'com.apple.main-thread', stop reason = EXC_BAD_ACCESS (
       code=1, address=0x37)
3    frame #0: 0x0002994a Pharo`longAtPointer(ptr="") at sqMemoryAccess.h: 141
4    frame #1: 0x000296bf Pharo`longAt(oop=55) at sqMemoryAccess.h:164
5  * frame #2: 0x0002d2b5 Pharo`lookupMethodInClass(class=235013248) at gcc3x-interp.
       c:54877
6    frame #3: 0x0000e352 Pharo`interpret at gcc3x-interp.c:5651
7    frame #4: 0x000295b8 Pharo`enterSmalltalkExecutiveImplementation at gcc3x-interp.
       c:51866
8    frame #5: 0x00002b91 Pharo`interpret at gcc3x-interp.c:2452
9    ...
10 [(lldb) frame select 2
11  frame #2: 0x0002d2b5 Pharo`lookupMethodInClass(class=235013248) at gcc3x-interp.
       c:54877
12    54874    currentClass = class;
13    54875    while (currentClass != GIV(nilObj)) {
14    54876        /* begin followObjField:ofObject: */
15 -> 54877        objOop = longAt ((currentClass + BaseHeaderSize) + (((int)((usqInt)
       (MethodDictionaryIndex) << (shiftForWord())))));
16    54878        assert(isNonImmediate(objOop));
17    54879        if (((longAt(objOop)) & ((classIndexMask()) - (
       isForwardedObjectClassIndexPun())))) == 0) {
18    54880            objOop = fixFollowedFieldofObjectwithInitialValue(
       MethodDictionaryIndex, currentClass, objOop);
19 [(lldb) p currentClass
20  (sqInt) $5 = 43
```

Listing 3.2: Debugging VM code to find the cause of the failure shown in Listing 3.5. The defect is that the instance variables superclass and method dictionary declared in the class Class are reversed in order.

searched method in the method dictionary of the current class, failing but handling again the error. The VM tries to continue the lookup process by fetching the current class' superclass. The first instance variable of a method dictionary is an immediate, containing the number of methods it stores, in this case it is 43.

The current class is now the immediate value 43. When the VM tries to fetch the method dictionary of the current class, a segmentation fail occurs, because the current class is an immediate (line 54877).

To prevent future failures of this kind, we check that all our structural definitions for classes define instance variables in the same order they are defined in Candle and in Pharo. However, we cannot know if all of them are required by the VM or not.

After applying these corrections, we generate the kernel again. We define the application code as follows:

```
1 OLClass log: 'Hello from ObjVLisp'.
2 OLClass quit.
```

This code is executed by the VM, when it VM loads and executes the kernel. This application is installed in the kernel at the end of the generation process.

The message log uses File IO support inherited from Candle definitions. It creates a file containing the message received as argument. The message quit calls a VM primitive to finish the kernel execution.

We execute the kernel and it is a success. The file is created and it contains the message as we expected.

Now we would like to test our kernel with more interesting applications.

3.3.7 Non-Reflective Def. causes VM-Constraint Fail at Execution

Class-Model With Wrong Name Produces Failure During Kernel Execution.

Pharo is a reflective system. In a nutshell, this means that programs are able to reflect on their own execution and structure [Kiczales 1991]. An interesting example is that it provides access to the run-time stack, through the pseudo-variable thisContext. Whenever thisContext is referred to in a running method, the entire run-time context of that method is reified and made available to the language as a series of chained Context objects. The sender of a Context is the Context immediately before in the run-time stack (*e.g.* the Context that was active at the moment the first Context was created).

To test that our kernel's reflective capabilities integrate well with the Pharo VM, we decide to install the next application, which logs the sender of the current context.

```
1 OLClass log: 'thisContext sender: ', thisContext sender asString.
2 OLClass log: 'Good bye'.
3 OLClass quit
```

Failure manifestation. The kernel is generated again, it is then loaded and executed by the Pharo VM. The execution is not interrupted by a critical failure. Instead, a non-critical failure occurres during the execution. The message logged in the output file (generated by the kernel as a product of its execution) is not as expected, but it is the following.

```
'thisContext sender: -537223584
Good bye'
```

The first line of the previous message was expected to be something like: 'thisContext sender: OLClass » entryPoint', since Contexts are printed showing the method selector and class for which they were created.

Apparently, the message OLContext sender is returning a pointer to a space in memory that does not contain an object, but contains something that can be interpreted as an integer and it is printed as such.

Checking structural definitions. We check the structural definition of class OLContext and method OLContext » sender, finding the definitions below.

```
1 class {
2   name : #OLContext
3   superclass : #OLObject
4   instVarsNames: 'sender pc stackp method closureOrNil receiver'
5   type: #variable
6 }
7
8 OLContext >> sender
9   ^ sender
```

Not having found evident defects in structural definitions, we proceed to debug VM code.

Inspecting guest-language objects using the VM debugger.
The VM debugger provides low-level tools to read objects in the kernel. Listing 3.3 shows an intermediate step during the VM execution, where the method OLContext » sender was found and assigned to the variable objOop3 (line 54972).

The VM implementation provides the basic function printOop(address) to print objects in the kernel, taking a memory address as argument. The object objOop3 has a compiled method structure. The first variables are literals and the rest are bytecodes.

Debugging VM code.
We continue to debug and realize that the first bytecode in the method OLContext » sender (bytecode 139) instructs the VM to execute a primitive. This primitive puts at the top of the execution stack the object found right after the object's header (see Listing 3.4). The result of this operation, stored in the variable aValue, should be a context but that is not the case.

```
Process 14084 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x0002d8d8 Pharo` lookupMethodInClass(class=252536384) at gcc3x-
    interp.c:54974
   54971     && (((longAt(obj0o0p3)) & ((classIndexMask()) - (
    isForwarded0bjectClassIndexPun()))) == 0)) {
   54972         obj0op3 = fixFollowedFieldof0bjectwithInitialValue(index -
    SelectorStart, methodArray, obj0op3);
   54973     }
-> 54974     GIV(newMethod) = obj0op3;
   54975     found = 1;
   54976     goto 118;
   54977 }
Target 0: (Pharo) stopped.
[(lldb) p print0op(obj00p3)
0xf0e5500: a(n) bad class class nbytes 21
    0x20005 0xf0dd258 0xf0ede20
0x0f0e5514: 8b/139 08/8 01/1 00/0 7c/124 00/0 00/0  00/0
0x0f0e551c: 00/0
(sqInt) $15 = 0
[(lldb) p print0op(0xf0dd258)
  0xf0dd258: a(n) bad class class nbytes 6
sender
(sqInt) $16 = 1
```

Listing 3.3: Inspecting the object OLContext » sender (a compiled method) using the VM debugger (LLDB) low level tools.

Debugging Bootstrapper code.
We debug Bootstrapper code to find the reason causing a wrong compilation of the method OLContext » sender. As shown in Figure 3.7, methods defined in the Context class must be compiled using a special option called optionLongIvarAccessBytecodes. However, the name of the class Context is hardcoded in the Bootstrapper code. This produces a failure since our class is named OLContext.

3.3.8 Application Def. causes Guest-Lang. Code Fail at Execution

Application Code Sending Message With Argument Of Incorrect Type Produces Failure During Kernel Execution

To test our kernel performing mathematical operations, we generate it again, but this time we install the next application code:

```
Process 13763 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = step over
    frame #0: 0x0000e4f5 Pharo` interpret at gcc3x-interp.c:5670
   5667 if (localPrimIndex >= 264) {
   5668     /* begin internalStackTopPut: x*/
   5669     aValue = longAt(((LlongAtPointer(localSP)) + BaseHeaderSize) + (((sqInt) ((
    usqInt) ((localPrimIndex - 264)) << (shiftForWord()))))
 ->5670     longAtPointerput(localSP, aValue);
   5671     goto 1859;
   5672 }
   5673 if (localPrimIndex == 256) {
Target 0: (Pharo) stopped.
(lldb) p printOop(aValue)
0xbfff88b1=-536886184
(sqInt) $7 = 0
```

Listing 3.4: Debugging VM code to find the cause of failure shown in Section 3.3.7

```
OLClass log: '10 ln =', 10 ln.
OLClass quit.
```

We execute it in the VM, the execution finishes correctly, but the log file generated by the kernel does not contain what we expected.

```
'Error: Strings only store Characters

CallStack:
*10 ln =*   OLByteString(OLObject) >> error:
*10 ln =*   OLByteString(OLString) >> at:put:
*10 ln =*   OLByteString(OLSequenceableCollection) >> replaceFrom:to:with:startingAt:
*10 ln =*   OLByteString(String) >> replaceFrom:to:with:startingAt:
*10 ln =*   OLByteString(OLSequenceableCollection) >> copyReplaceFrom:to:with:
*10 ln =*   OLByteString(OLSequenceableCollection) >> ,
*Class*     OLClass(OLClass) >> start
*Class*     OLClass(OLClass) >> entryPoint
*nil*       OLUndefinedObject(nil) >> noMethod'
```

Listing 3.5: Error message logged in text file by the kernel when execution failure arises due to defect in guest-language code

Our application code fails when the VM executes it. The logic to handle this failure and log information is implemented inside the kernel, through methods contained in our language structural definitions.

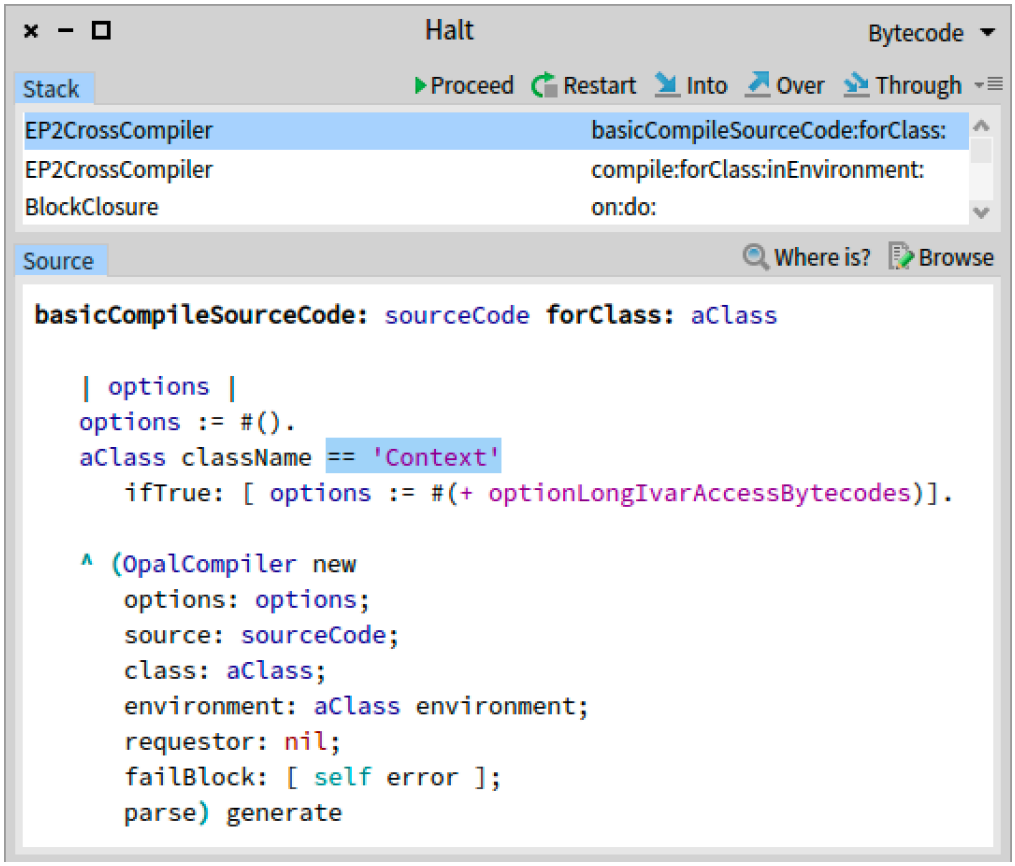


Figure 3.7: Debugging Bootstrap code using the host debugger to find the cause of failure shown in Section 3.3.7

Since our kernel does not provide advanced debugging tools, we must infer the error back in our application code, by manually checking source files containing structural definitions of methods involved in the error.

We realize that the problem is that we are trying to concatenate a string and a number using " , " (the comma message). We fix our application code by adding the message asString at the end of the first line, as follows:

```
1 OLClass log: '10 In =', 10 In asString.
2 OLClass quit.
```

3.4 Analysis Of Cognitive Distance In Bootstrap

Designing a solution that overcomes bootstrap challenges requires understanding the causes for cognitive distance increase in bootstrap. To make the concept of cognitive distance more concrete, we propose representations presented in Section 3.4.1. We analyze Bootstrap's causes of high cognitive distance in Section 3.4.2. And finally, we define the set of desirable features that a technique for bootstrapping different languages should implement in Section 3.5.

3.4.1 Introduction to Cognitive Distance Representations

To provide a more concrete notion of cognitive distance we propose *process diagrams* and *defect backtracking support tables*, and apply them to Bootstrap's cognitive distance representation (see Figures 3.8 and 3.9).

Process diagrams. Diagrams of language implementation processes that show the kernel generation process in stages, along with input/output data flow represented by solid arrows (see Figure 3.8). Failures are represented by saw-shaped labels on top of the process where they manifest. Failures provide a symptom of their cause (this can be an informative message, a system freeze, a system crash, etc). The domain to which the symptom belongs is encoded in an acronym and defined at the bottom of the diagram. Round-shaped labels on top of user-defined inputs represent the domains of expertise required to write the input. Finally, red-dotted arrows represent the flow of mental defect backtracking operations that developers must carry out to trace back the cause of a failure. When defect backtracking is supported, no red arrow is displayed.

Defect backtracking support table. Defect backtracking support refers to the delivery of tools to debug the cause of failures at the abstraction level of their corresponding defects in the language specification. Defect backtracking support tables summarize defects, failures, and defect backtracking support (see Figure 3.9). Rows define inputs written by developers. Columns represent failures occurring during a specific stage. Domain labels in each cell represent defects producing the kind of failure defined by the cell column, and located in the user input defined by the cell row. Green check marks indicate that defect backtracking is supported, while red cross marks indicate the opposite.

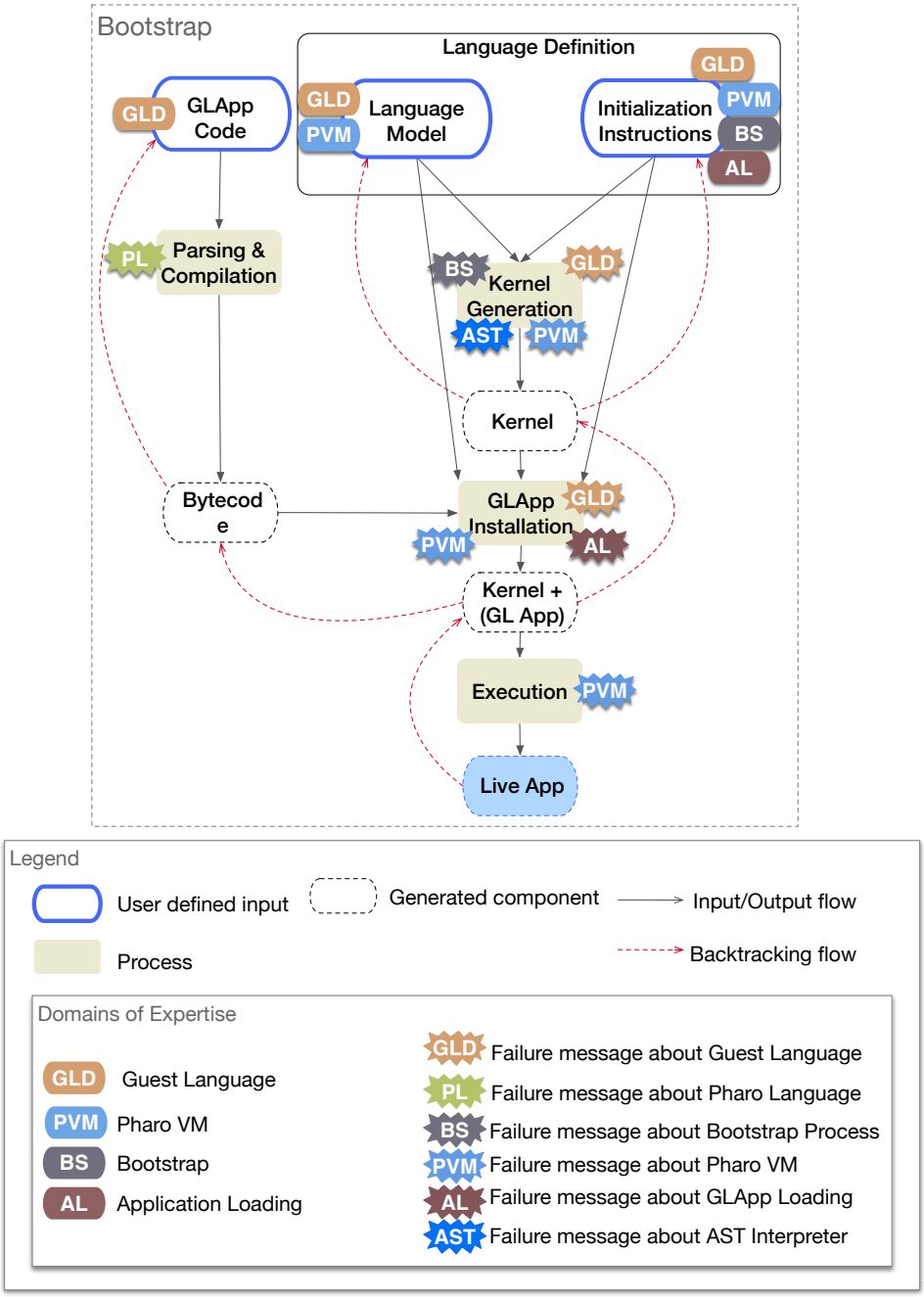


Figure 3.8: Cognitive Distance in Bootstrap.

3.4.2 Causes Of Large Cognitive Distance In Bootstrap

In Bootstrap, the language definition comprises both the definition of language elements (language model) and the process that generates the kernel (initialization instruc-

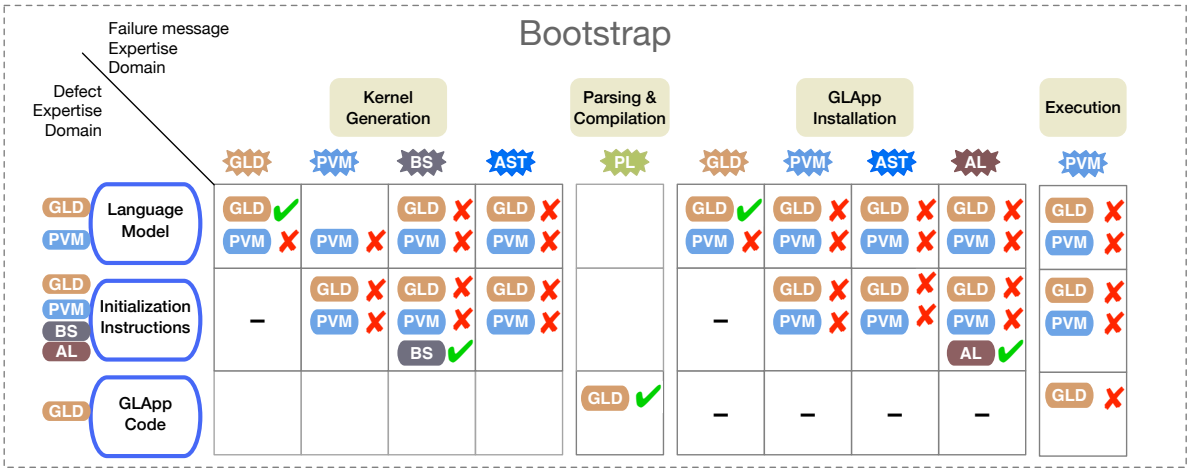


Figure 3.9: Bootstrap defect backtracking required domains

tions). The model is specified in plain text. The abstraction level for this specification is high (Tonel format[2]), but manual edition of text files is prone to errors. Initialization instructions are specified in both Pharo code and guest-language code. The abstraction level for this specification is high, but identifying the right place to apply modifications requires dealing with bootstrapper code, including low-level operations.

Language model creation and modification requires knowledge about the target VM implementation. Developers unfamiliar with the VM are prone to corrupt the model, because sections fulfilling VM constraints are unidentifiable a priori. Definition of initialization instructions additionally requires knowledge about the bootstrapping process, because the code to generate the kernel is mostly language specific and its modification relies on user expertise. Defective initialization instructions can corrupt the kernel, making application loading fail. Initialization instructions definition also requires knowledge about the process of application loading.

The most important cause for cognitive distance increase is the abstraction gap and temporal distance between defects and failures. Failures manifest at a different abstraction level than the abstraction level of their original cause in the language definition (*e.g.* a segmentation fault produced by the VM, and caused by the missing definition of one element in the language definition). Traditional debugging tools do not provide appropriate support when failures and defects belong to different abstraction levels [Chis 2015]. Additionally, late manifestation of errors force developers to debug code not written by them, demanding familiarity with domains out of the language definition, such as VMs, compilers and the bootstrap process itself. The greater the distance between defects and failures, the more difficult it is to track the infection

[2]https://github.com/pharo-vcs/tonel

chain [Zeller 2005].

Even for small kernels, bootstrapping takes considerable time. For example, bootstrapping $Ovlisp_L$ takes in average 32,2 seconds in a machine with the following characteristics. Processor: 2,9GHz quad-core Intel Core i7-7700HQ, cache 6MB. Memory: 16 GB 2133MHz LPDD3. Storage: PCIe SSD 512GB.

User feedback arrives late, because the effects of modifying the language definition are visible only at run-time, unless a failure occurs during generation in which case the kernel is not generated. Even small modifications require the generation process to be executed from start to finish. The same situation occurs for application code. The kernel must be generated each time the application code is changed. The only way to have proper debugging tools for the application is to install them in the kernel, incrementing its size and complexity.

3.5 Desirable Features Of A Bootstrap-Based LIT

According to Krueger [Krueger 1992] proper abstractions together with automatic generation of code are the key aspects to implement efficient software reuse techniques. We have applied his ideas in the elaboration of the evaluation criteria for LIT presented in Section 2.2.1 and summarized as follows:

C1. Specification abstractions must be expressive and similar to abstractions used for conceptualization of language syntax and semantics.

C2. The mapping between abstraction specification and realization must be automatic, providing proper debugging support where appropriate.

The previous criteria combined with our analysis of cognitive distance in Bootstrap presented in Section 3.4 has resulted in the desirable features of a bootstrap-based LIT that we propose below.

3.5.1 Requirement 1

"Specification abstractions provided by the technique must be succinct, expressive, close to abstractions used for conceptualization of language syntax and semantics."

In reflective languages, metaobjects represent the semantics of the new language. In a bootstrap-based LIT, abstractions for language specification must be at the same

level as language metaobjects or above. It is important to keep the kernel generation process at that abstraction level from the user's point of view.

The solution must hide the non-customizable aspects of the bootstrap process and provide specific extension points for customization. No expertise about VM implementation or the bootstrap process should be required.

3.5.2 Requirement 2

"The mapping between abstraction specification and realization must be automatic and efficient."

The technique must infer as much information as possible from user defined code, limiting code redundancy. The generation process should execute without user intervention unless failures occur. The process should be efficient, and immediate feedback as well as live interaction with kernel-objects should exist.

3.5.3 Requirement 3

"In the case the mapping from specification to realization fails, the technique should provide debugging support that answers developer questions formulated at the level of their abstraction specifications."

The technique must provide defect backtracking support. According to C2 (Section 3.5), the mapping from specifications to realizations must be automatic. Effective automation demands support for debugging when the process fails. The debugger must display meaningful and immediate information, emulating the style of live self-surgery. Keeping the debugging process at the abstraction level of language specification is essential for closing abstraction gaps and limiting the domains of expertise required from developers.

We propose a set of tools for solving each category of failure presented in our taxonomy. These tools are illustrated in Figure 3.10 and explained below.

Ahead Of Time Model Validations. Automatic detection of defects in declarative definitions must occur before kernel generation, interrupting the process if corruption is detected. This measure prevents all failures caused by Structural Defects[3] arising during the generation and execution stages.

[3]A quick reminder: structural defects have their cause only in VM constraints

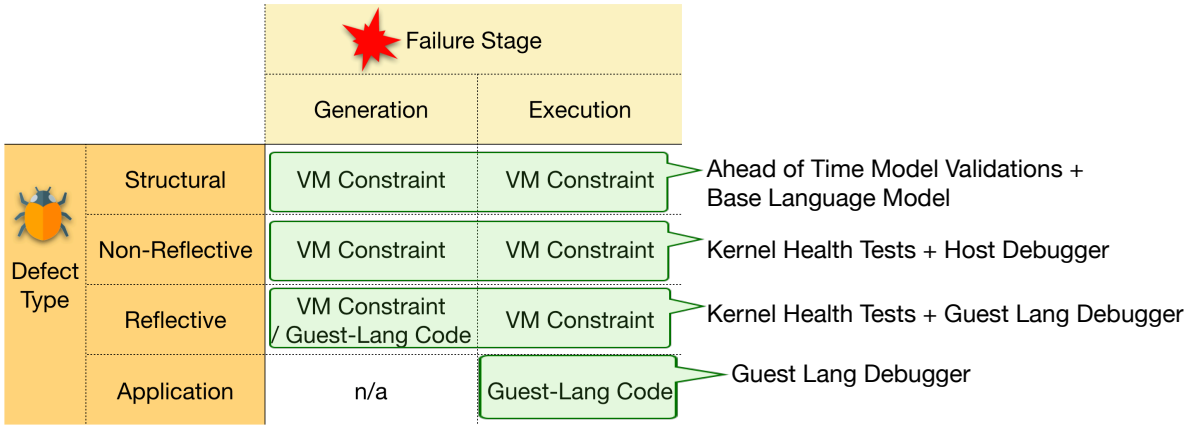


Figure 3.10: Solutions to Bootstrap Defects and Failures.

Kernel Health Tests. Reflective and non-reflective definitions are imperative and cannot be tested before their execution. Kernel health must be constantly checked to stop the generation process as soon as corruption occurs, and a debugger must be automatically opened, displaying the instruction that caused the failure.

Guest Language Debugger. The host debugger is not suitable for debugging guest-language, because it does not allow controlling the execution flow (stepping) of code, and because it hinders access to guest-language variables and abstractions, since they must be accessed from interpreter abstractions such as the interpreter's execution stack and opaque inspectors for mirrors. Custom debugging tools for reflective instructions and application code are needed, as well as inspectors for kernel-objects. These inspectors must display information at least at the same abstraction level than the kernel's object-model.

If a defect is found in a reflective instruction, there are two abstraction levels of execution: bootstrapper code and reflective code (*e.g.* in Section 3.3.4 example, the method `basicInstallClass:` from the bootstrapper executes a reflective instruction that sends the message `build` to create new classes). The custom debugger must provide information about both levels of execution, allowing users to understand the failure. This technique reduces the efforts to fix errors that cause kernel-corruption. Combining this technique with model validations and kernel tests, the health of the kernel is ensured before writing to disk. Failures during execution due to VM constraints are prevented this way.

3.6 Conclusions

This chapter presented an in-depth analysis of the Bootstrap technique paying special attention to the causes of increased cognitive distance. We have provided a taxonomy of defects and explained their origins and the resulting failures. We have shown the limitations of the existing bootstrap process through a case of study. To make our analysis of cognitive distance more concrete, we have proposed visual representations where defect backtracking operations and the need for expertise in different domains are displayed. Based on this analysis, we have proposed a set of desirable features that a bootstrap-based language generation technique should provide in order to minimize cognitive burden on developers.

The next chapter puts our recommendations and gives a first design of a bootstrap-based LIT.

Chapter 4

Bootstrap-Based Language Implementation: *MetaL*

Contents

The previous chapter presented a thorough analysis of the bootstrap technique stressing the causes for cognitive distance increase whose analysis lead to the elaboration of the required features of a bootstrap-based LIT. This chapter presents *MetaL*,

an implementation of our approach for a bootstrap-based LIT that aims at cognitive distance reduction. *MetaL* is an open source project published in a public repository[1].

We begin giving a general explanation of the solution in Section 4.1. This is followed by a more detailed explanation of the technique in Section 4.2 using the case study from the previous chapter. Next, in Section 4.3 we introduce defects from the previous chapter into our case study and we show how they are solved in *MetaL*. Then, we elaborate about *MetaL*'s validations in Section 4.4. Finally, Section 4.5 offers a summarizes the most important ideas of this chapter.

4.1 *MetaL* in a Nutshell

MetaL is our framework for bootstrapping reflective kernels. *MetaL* focuses on minimizing the cognitive distance for developers by bringing the advantages of self-surgery (*i.e.* immediate feedback and same abstraction level of specifications and realizations) into the Bootstrap technique, keeping language definition and debugging tasks at a high abstraction level.

MetaL kernel generation process is represented in Figure 4.1. Users provide a metamodel of the new language, which comprisess a set of classes in the host. These classes represent on the one hand the structure of language metaobjects, and on the other hand operations for these metaobjects. The semantics the the new language is defined in its metaobjects. Syntax definition is out of *MetaL*'s scope; all generated languages have the same syntax as Pharo.

The language reified model is generated by instantiating metamodel classes. It is composed of objects in the host that call *model-objects*. These objects are related among each other in an object-graph. Users specify the way to construct the model through MOP operations to create and transform model-objects.

Generating a kernel requires initializing the environment of the new system, *e.g.* initialization of globals, creation of a system dictionary, etc. Initialization instructions are specified by users in two possible ways: reflective operations written in guest-language code, or MOP operations to manipulate objects in the kernel through smart-mirrors, an improved version of Bootstrap mirrors. Finally, the kernel is dumped in disk and executed by the target VM.

MetaL provides the definition of the model and metamodel of an abstract minimal reflective language used by developers as *base-language*. Users define new metamodels by extending classes in base-language metamodel, and defining extension points for model

[1] https://github.com/carolahp/MetaL

transformations and, optionally, custom kernel initialization instructions and custom variable bindings.

To ensure compatibility with the target VM, *MetaL* reifies VM requirements in a set of *roles* which are associated to object-models and metamodel classes. These roles are used to validate both model and kernel during their construction, ensuring compliance with VM constraints.

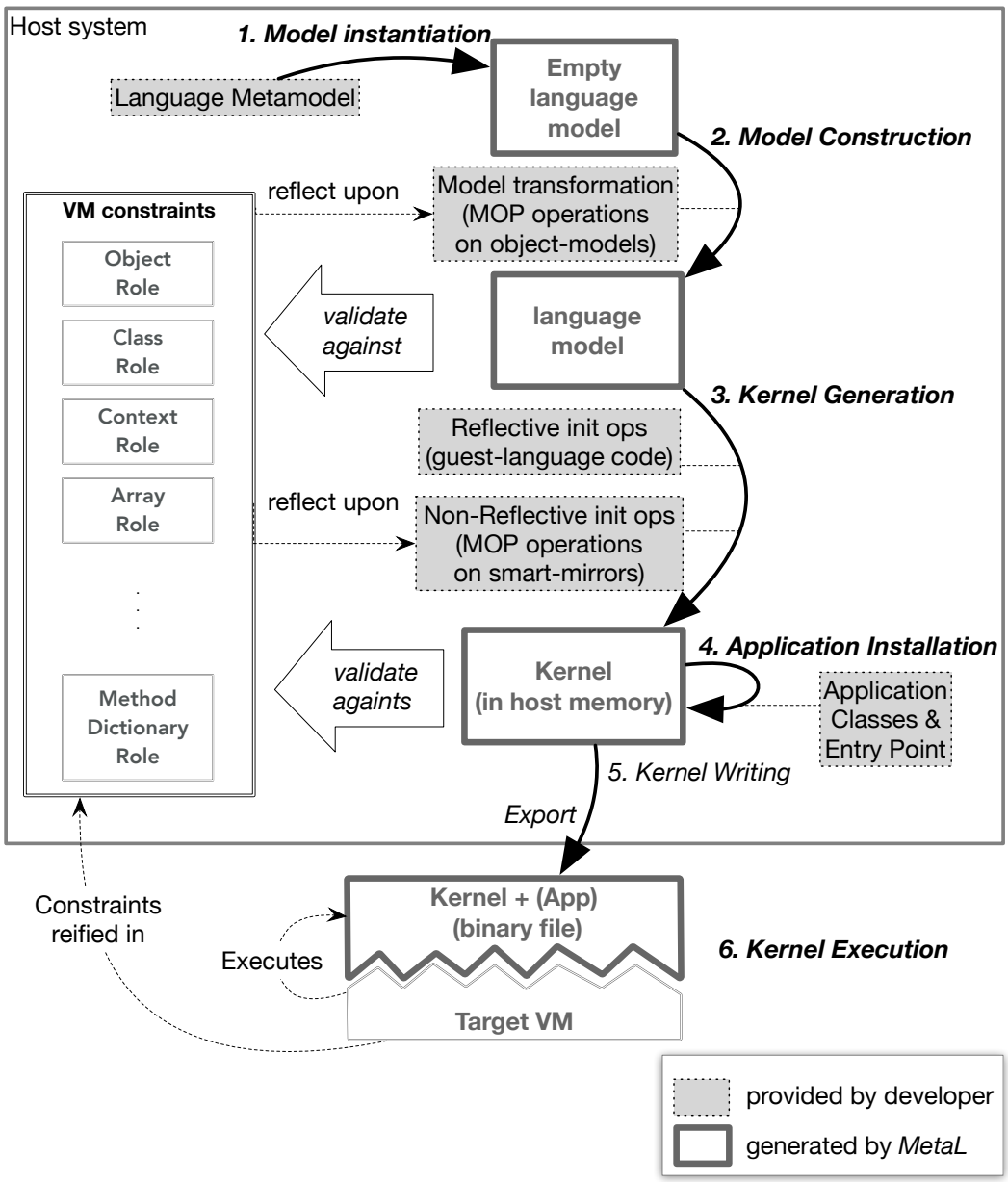


Figure 4.1: *MetaL* bootstrap process.

4.2 *MetaL* by Example: Generating $Ovlisp_L$

To introduce *MetaL*, we describe the generation of $Ovlisp_L$, a simplified version of ObjVLisp [Cointe 1987]. ObjVLisp is a minimal class-based language where classes are first-class objects. Like ObjVLisp, $Ovlisp_L$ has explicit metaclasses and defines only two classes: Object and Class. Having explicit metaclasses means that each class is instance of one metaclass, and that one metaclass can have multiple instances. Therefore, unlike Pharo, there is no parallel hierarchy between classes and metaclasses. Class is the first metaclass and it is instance of itself. Unlike ObjVLisp, $Ovlisp_L$ implements single inheritance. This is to keep the example simple. Further examples include an implementation of ObjVLisp with multiple inheritance (Section 5.2).

4.2.1 General Bootstrap Process

Kernel implementation in *MetaL* starts with the definition of the language metamodel. Defining a metamodel is done by extending classes from *MetaL*'s base metamodel presented in Figure 4.2. Section 4.2.2 offers an explanation of metamodel definition. $Ovlisp_L$'s metamodel is presented in the left section of Figure 4.3. It contains only two classes: OvlispLanguageModel to represent the language, and OvlispClassModel to represent classes. Once the metamodel is ready the bootstrap process can start.

The bootstrap process occurs in three steps: language model construction, kernel generation in memory, and kernel writing to disk. These steps are illustrated in Listing 4.1, where the instructions to bootstrap $Ovlisp_L$ are presented. First, an instance of OvlispLanguageModel is created providing a name for the kernel and an application entry point that is the code to be executed by the VM when it loads the generated kernel. We say that the model is empty because it consists in only one object representing the language. The full model is constructed by sending the message build. This means that multiple object-models are created and added to the model. Model construction is described in Section 4.2.4. Then, the kernel is generated in memory, and finally it is written to disk. Executing the kernel in the target VM is done by sending the message executeInVM, the result of the execution is displayed in Pharo's Terminal. Kernel generation, writing, and execution are described in Section 4.2.5.

4.2.2 Metamodel Definition

To bootstrap $Ovlisp_L$, we must first define its metamodel by extending classes from *MetaL*'s base metamodel. The base metamodel is composed of several classes, which

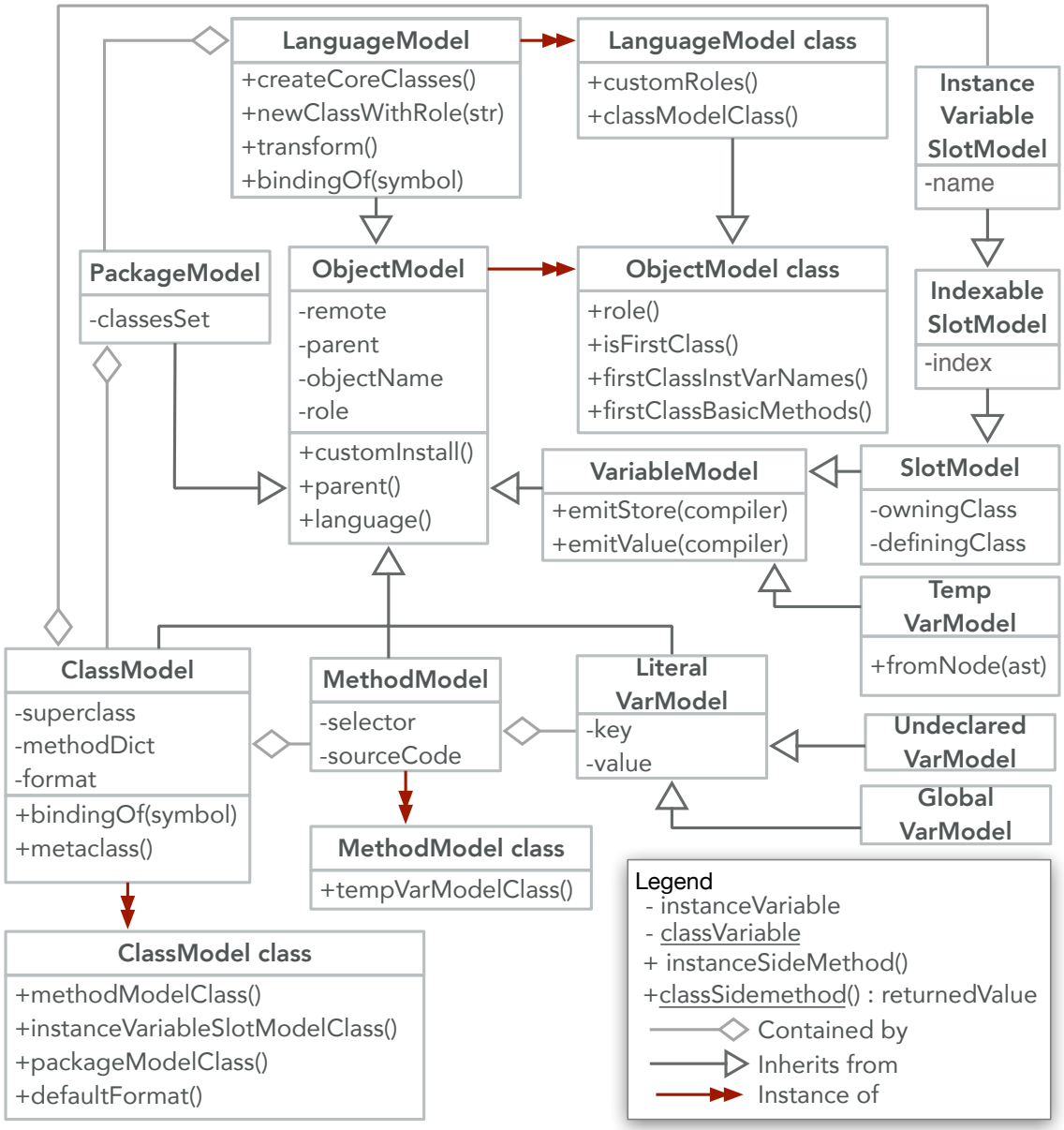


Figure 4.2: *MetaL* Base Language Metamodel.

are installed in the host-system (Pharo). *MetaL* builds ObjVLisp's model by creating instances of these classes. The resulting model is a graph of objects representing ObjVLisp language elements to be installed in the generated kernel. This graph constitutes the model of the language and it is composed by instances of classes in the metamodel, we call these objects *object-models*. $Ovlisp_L$'s model and metamodel are illustrated in Figure 4.3. Boxes in the *metamodel* section represent classes installed in the host sys-

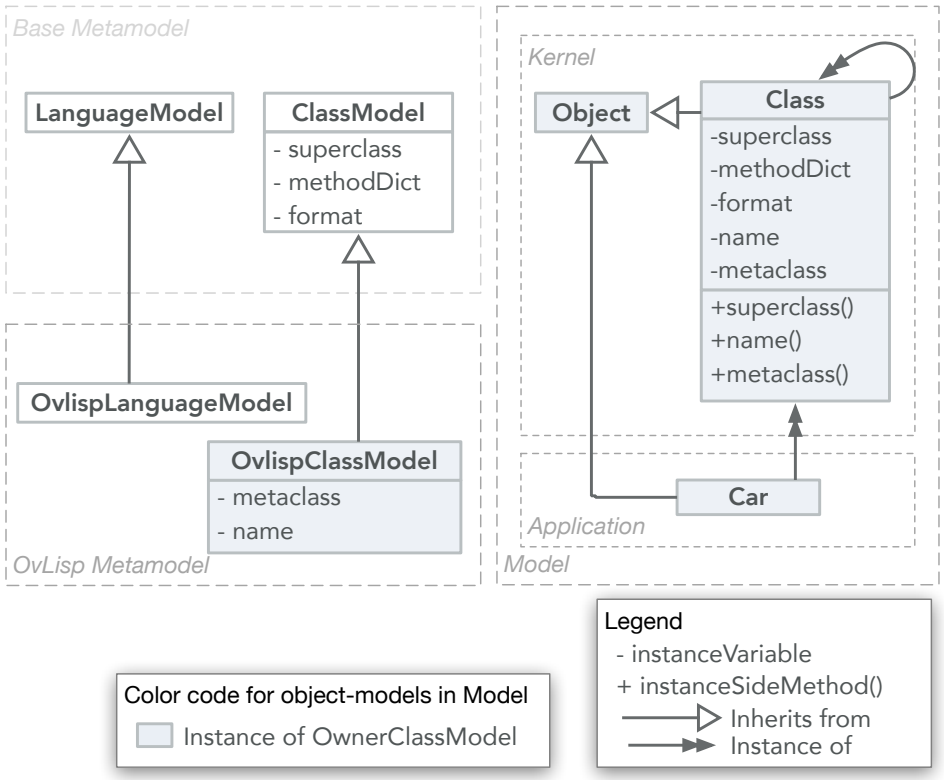


Figure 4.3: $Ovlisp_L$ metamodel and model.

```
model := OvlispLanguageModel
  newWithName: 'Ovlisp'
  withEntryPoint:
    'System log: Car metaclass name.              " prints Class "
  System log: Car metaclass metaclass name.    " prints Class "
  System log: Car superclass name.           " prints Object "
  System log: Car superclass superclass asString. " prints nil "
  System quit'.
model build.
model generateKernel.
model writeKernel.
model kernelInDisk executeInVM
```

Listing 4.1: Instructions to bootstrap $Ovlisp_L$.

tem, while boxes in the *model* section are instances of these classes, *i.e.* object-models. Colors relate object-models with their class in the metamodel. $Ovlisp_L$'s model only comprises instances of OvlispClassModel, except for the object-model representing the language. In model diagrams we make the distinction between *kernel* and *application*. Classes belonging to the application are installed in the kernel during bootstrap, as our kernels do not support code compilation at run-time.

Figure 4.2 illustrates *MetaL*'s base metamodel. All class names contain the suffix 'Model' to express that their instances are object-models. Several language elements, such as packages (PackageModel), methods (MethodModel), and slots (SlotModel) have a representation in this model. To define $Ovlisp_L$ we only need to focus on the next three classes: LanguageModel, ClassModel, and ObjectModel.

Since $Ovlisp_L$ implements explicit metaclasses, each class must know its metaclass. Therefore, object-models representing classes (we call them *class-models*) must store a reference to the class-model representing their metaclass. However, ClassModel only declares three instance variables: superclass, methodDict, and format. Consequently, Ovlisp's metamodel needs its own version of ClassModel. For this reason, we subclass ClassModel creating the class OvlispClassModel, which declares two extra instance variables: metaclass and name (see Listing 5.3). Name is included to easily identify classes once they are installed in the kernel. Additionally the user must provide accessor methods for each extra instance variable it declares (this requirement is explained in Section 4.2.5).

Among other kinds of relationships, object-models relate to each other through "containment" relationships (*e.g.* a method belongs to a class, a class belongs to a language, etc), see Figure 4.2. The instance variable parent, declared in ObjectModel, stores this information. The class LanguageModel represents the new language itself. A language

```
ClassModel subclass: #OvlispClassModel
  instanceVariableNames: 'metaclass name'

OvlispClassModel >> metaclass
  ^ metaclass

OvlispClassModel >> metaclass: aClassModel
  metaclass := aClassModel

OvlispClassModel >> name
  ^ name

OvlispClassModel >> name: aString
  name := aString

LanguageModel subclass: #OvlispLanguageModel
  instanceVariableNames: ''

OvlispLanguageModel class >> classModelClass
  ^ OvlispClassModel
```

Listing 4.2: $Ovlisp_L$ metamodel definition

model contains a single instance of this class, which acts as root of the membership graph, *i.e.* every object-model eventually belongs to the language. LanguageModel is an abstract class, thus subclassing it is mandatory when defining a new language. The method classModelClass must be defined by the developer to indicate *MetaL* the class used by default to instantiate class-models.

4.2.3 Definition of Roles

MetaL's bootstrapping process is agnostic to the language model given as input. However, some class-models play special roles during the generation process. Role assignation allows *MetaL* accessing class-models generically by their role rather than by their name, helping to keep the bootstrapper agnostic to the model (more about roles in Section 4.4). Roles also allow automatic modifications to the model to make it compliant with VM constraints.

MetaL defines 28 roles (the complete list is given in Appendix C). Users defining class-models that have a role must specify it as illustrated in Listing 4.3. Users do not need to define one class-model for every existing role, since *MetaL* is able to complete the model automatically. To define $Ovlisp_L$ we care about only two roles: #Class and #ProtoObject. The way to define them is shown in Listing 4.3: the class named 'Object' has role #ProtoObject, and the class named 'Class' has the role #Class.

```
1 OvlispLanguageModel >> customClassRoles
2   ^ { #ProtoObject -> 'Object' .
3       #Class -> 'Class' }
```

Listing 4.3: $Ovlisp_L$ definition of class-model roles

The class-model with role #ProtoObject is by default the root of the inheritance hierarchy, *i.e.* it inherits from nil, and all classes in the model inherit from it (unless otherwise stated by the user).

Understanding the role #Class requires a short explanation about the relationship between metamodel and classes installed in the kernel (we name them *kernel-classes*). Class-models represent classes to be installed in the kernel. In $Ovlisp_L$, class-models are instance of the class OvlispClassModel, which inherits three instance variables from ClassModel and declares one extra instance variable to store a reference to the class-model's metaclass (*MetaL* ignores instance variables declared in ObjectModel, as they are only used for bootstrapping purposes).

Kernel-classes must define the same instance variables as their corresponding class-model [2], *i.e.* superclass, methodDict, format, and metaclass. At the same time, the VM expects from the kernel that, as in Pharo, kernel-classes are normal objects in the kernel, and therefore that they are instance of a kernel-class. In consequence, there must be one kernel-class that declares the four instance variables superclass, methodDict, format, and metaclass, and whose instances are kernel-classes. The class-model defining this special kernel-class is the one with role #Class.

In $Ovlisp_L$, OvlispClassModel is the only class in the metamodel mapped to a class-model. However, as shown in further examples, this mapping is also possible for other classes in the metamodel, such as Package, Slot, and Method. The mapping between classes in the metamodel and class-models is based on roles. Classes in the metamodel can also define a role. As shown in Figure 4.2, the class ObjectModel class defines the method role. By default, this method returns nil. However, this behavior is overridden in some subclasses of ObjectModel class (such as in PackageModel class, MethodModel class, Slot class, and ClassModel class), to return the name of the role associated to that class in the metamodel. For example, ClassModel class the method role returns the value #Class. Except for roles #Class and #Method, roles defined by these classes are optional, and it is up to the user whether to include them in the model or not. Further examples show how to use this feature.

[2] We remark the distinction between *declaring* and *defining* an instance variable. The class Point *declares* two instance variables: x and y. Instances of this class *define* these two instance variables: they store values for each one of them. The confusion arises because in Pharo classes are also objects and therefore they are instance of a class. In consequence they *declare* instance variables for their instances, and also *define* values for their own instance variables, which are *declared* in their metaclass.

4.2.4 Model Construction

Now that $Ovlisp_L$ metamodel is defined, its model can be created. The following code excerpt shows how to do this.

```
1 model := OvlispLanguageModel
2   newWithName: 'Ovlisp'
3   withEntryPoint:
4     'System log: Car metaclass name.
5   ...
6   System quit'.
7 model build.
8 ...
```

When instantiating the class OvlispLanguageModel, users specify the name for the generated kernel file in disk, and the entry point code. The entry point contains the instructions to be interpreted by the VM when the kernel is loaded and executed. Our entry point logs the name of Class's metaclass and then quits, terminating the execution (like exit(0) in C). The entry point uses the class System, defined in *MetaL*'s base reflective kernel definition (see Section 4.2.4.2).

The model is empty. Creating the rest of object-models is done by sending the message build. The method build populates the model in three steps: creating user-defined core class-models, automatically completing the model to make it valid (*i.e.* VM compliant), and applying user-defined transformations.

4.2.4.1 Core Class-Models

Core class-models are class-models necessary to initialize other class-models, and their creation is specified in the method createCoreClasses. *MetaL* provides a default implementation of this method, as shown in Listing 4.4. Users must override or extend this method when their language introduces additional core class-models.

The need for core-classes to be installed first follows the next ideas. As explained in Section 4.2.3, the class-model with role #ProtoObject (this is Object in $Ovlisp_L$) is by default the superclass of class-models. When a class-model is created, Object must already exist. The same situation occurs with Class, as it is the metaclass by default. For this reason, Object and Class are core and must be created in a special way. The problem is evident when checking the method initialize[3] defined in ClassModel, and specialized in OvlispClassModel (see Listing 4.4). This method assumes the existence of core class-models.

[3]Object instantiation in Pharo occurs two steps: allocation and initialization. Initialization instructions are defined in the method initialize of the object's class

In $Ovlisp_L$, Class and Object are the only two core class-models. Users overriding this method create and manipulate (core) class-models using methods from *MetaL*'s MOP. A description of the MOP is presented in Appendix B. As shown in Listing 4.4, core class-models are created in three steps:

- Basic instantiation (lines 3, 4), sending the message basicNewClassWithRole:, which creates a new class-model but does not initialize it.
- Addition to the language-model (lines 6, 7), the class-model is added to the language.
- Initialization (lines 9, 10), performed only after adding all core class-models to the language

```
1  LanguageModel >> createCoreClasses
2    | objectModel classModel |
3    objectModel := self basicNewClassWithRole: #ProtoObject.
4    classModel := self basicNewClassWithRole: #Class.
5
6    self addClass: objectModel.
7    self addClass: classModel.
8
9    objectModel initialize.
10   classModel initialize.
11
12 ClassModel >> initialize
13   self isProtoObject
14     ifFalse: [ superclass := self language classWithRole: #ProtoObject ].
15   methodDict := IdentityDictionary new.
16   format := self defaultFormat.
17   role ifNotNil: [ role transform: self ]
18
19 OvlispClassModel >> initialize
20   super initialize. "execute the method initialize defined in ClassModel"
21   metaclass := self parent classWithRole: #Class.
22   name := objectName
```

Listing 4.4: Definition of core class-models in $Ovlisp_L$

4.2.4.2 Automatic Model Completion

Up to this point the model is still not valid, as it does not fulfill VM constraints (see Section 3.2.1). *MetaL* applies automatic transformations according to VM constraints to

make it valid. Then, it transforms the model according to the classes in the metamodel. Finally, it completes the model loading source code files.

Completion According to VM Constraints. *MetaL* uses *roles* to automatically transform the model, making it compliant with VM constraints. A full list of the 28 roles defined by *MetaL* is presented in Appendix C. Roles in *MetaL* are objects that reify VM constraints on kernel-classes and class-models. They encapsulate low-level information about them, such as index in class table, index in special object array, and format for class-models. Information about format includes restrictions on instance variables declared in the class-model.

For each role having no class-model that defines it in the model, a class-model is automatically created according to the role specifications. For example, the class Array is created, its superclass is set as Object, its layout is set as 'variable', its role is set as #Array. The object role #Array defines the value 51 for the index in the class table, 8 for the index in special objects array, 2 for the instance specification, 'variable' for the layout, and forbids the definition of instance variables in Array.

User-defined class-models having a role are also transformed, making them compliant with specifications of their role. These transformations are performed in the method initialize of ClassModel as shown in Listing 4.4. The role transforms the class according to VM constraints. For example, the instance variables superclass, methodDict, and format are added to the class-model Class. Roles are explained in more depth in Section 4.4.1

Completion According to the Metamodel. Classes in the metamodel with a role are used to transform class-models with the same role. For example, in $Ovlisp_L$, the class-model Class has the role #Class, same as OvlispClassModel. Class is transformed such as it declares the same instance variables as OvlispClassModel. *MetaL* automatically adds the instance variables metaclass and name to Class. *MetaL* also creates by default accessor methods for these variables automatically and adds them the class-model. This option can be disabled. These methods are also object-models (we name them *method-models*). They are instances of the class MethodModel. An extract of $Ovlisp_L$ model after completion is presented in Figure 4.4, the full model contains 28 classes, one for each role.

Kernel Source Code Loading. By default, *MetaL* completes the model loading the source code of a provided base reflective kernel[4]. Alternatively, users can provide their

[4]https://github.com/carolahp/base-reflective-kernel

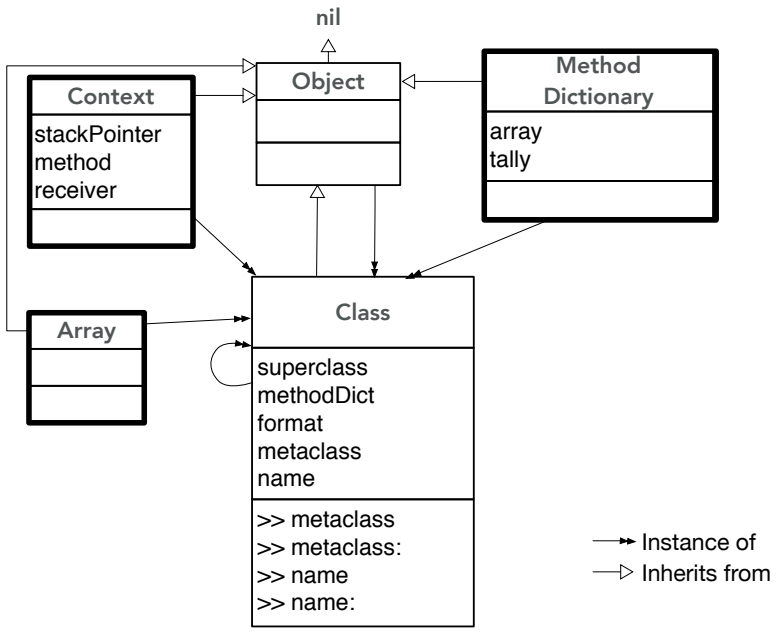


Figure 4.4: Extract of $Ovlisp_L$ model after automatic completion by *MetaL*. Automatically created classes are in thick lines, their structure is as defined by corresponding roles. Class instance variables and methods have been automatically created.

own source code, in which case they must execute the following code before sending the message build to the model. Sources must be specified in Tonel format[5].

```
...
model sourceCodePath: '/path/to/sources'.
model build
...
```

We implement $Ovlisp_L$ using *MetaL*'s base reflective kernel sources provided by default. The base kernel defines a class-based language with single inheritance, which is extended by *MetaL* according to $Ovlisp_L$ definition. Classes representing primitive types (UndefinedObject, True, False, Array, Float etc), language elements (*e.g.* Object, Class, MethodDictionary, etc), and execution elements (*e.g.* Context, Process, Message, etc) are included, along with classes implementing IO operations (System, File, Stream, etc). There are 67 classes in total. These classes implement a methods to support operations for primitive types, basic reflective operations (instVarAt:, class, ...), and basic IO operations based on files.

4.2.4.3 User-Defined Model Transformations

Users are able to define custom transformations to the model, that are applied after automatic transformations and source code loading. Transformations must be defined in the method transform of the class representing the language. In this case, we remove the method-model metaclass: from the class-model Class. This is shown in the following code extract.

```
OvlispLanguageModel >> transform
  | classModel |
  classModel := self classNamed: #Class.
  classModel removeLocalMethod: (classModel localMethodNamed: #metaclass:)
```

4.2.5 Kernel Generation, Writing, and Execution

After the model is built, users generate the kernel sending the message generateKernel to the model as presented in Listing 4.1. This method assembles the kernel in host memory. A kernel in memory is, at the low-level, an array of bytes. It starts empty and by the end of the generation process it contains the language runtime of the new language.

Custom kernel initialization operations are not necessary to generate $Ovlisp_L$. After

[5]https://github.com/pharo-vcs/tonel

defining the model, a user can simply execute the commands to generate, write and execute the kernel, as *MetaL* is able to infer information from metamodel and roles to automatize the rest of the process. The following paragraphs explain some key concepts helping *MetaL* to maximize automation.

Automatic installation of classes. *MetaL* is able to automatically infer which classes must be installed first in the kernel (*e.g.* Class in $Ovlisp_L$, Metaclass and Metaclass class in Pharo) by detecting circular references in the class metaclass chain of class-models (even if class-models in a language do not store their metaclass in a field, overriding the method metaclass in ClassModel is mandatory).

Even though $Ovlisp_L$ defines its own representation for classes (*e.g.* the fields metaclass and name are not defined in the base metamodel), special instructions to install classes are not needed.

The method installCustom completes the basic installation of classes in the kernel. It can be overridden or extended when necessary. The set of missing instance variable names is obtained from the metamodel as shown in line 3 of the following code excerpt. In line 8 the value defined in class-model for each field is get. In line 9 that value is set to the corresponding kernel-object. perform: is a reflective method in Pharo to send messages using the name of the message selector. Values defined by the class-model for metaclass and name are host objects, but they used to set an instance variable of a kernel-object. Next we explain how this is done.

```
1  ClassModel >> installCustom
2    | missing |
3    missing := self class allModelInstVarNames
4      difference: ClassModel allModelInstVarNames.
5    missing
6      do: [ :each |
7        | value |
8        value := self perform: each.
9        self remote perform: each , ':' with: value ].
10     self test
11
12 ObjectModel class >> allModelInstVarNames
13   ^ self allInstVarNames
14     difference: ObjectModel allInstVarNames
```

Smart Mirrors to interact with kernel-objects. The automatic class installation is possible thanks to *MetaL*'s *Smart Mirrors*, which are host objects wrapping and raising the level of abstraction of traditional mirrors used in bootstrap to manipulate kernel-objects. Smart mirrors combine information from the metamodel and roles to

provide inspectors and high-level methods. In particular, smart mirrors provide accessor methods to modify the instance variables of a kernel-object, even when they do not define such methods. This feature uses reflective features in Pharo to intercept message sends for undefined methods. Smart mirrors provide automatic transformation of basic host objects (numbers, arrays, dictionaries, etc) and object-models into kernel-objects. Smart mirrors together with roles and the metamodel, expand the possibilities for automation raising the level of abstractions for users.

Kernel writing and execution. After generation, the kernel is written to disk, saved as a file. Users have two options to execute the kernel file. Download a Pharo VM and execute it giving the kernel file as argument. Or alternatively, send the message executeInVM, which does the same as before but automatically, and displays the result of the execution in Pharo's Terminal. The following piece of code illustrates all previous steps.

```
1 ...
2 model generateKernel. "generate kernel in memory"
3 model writeKernel.
4 model kernelInDisk executeInVM
```

4.3 Debugging $Ovlisp_L$ Bootstrap in *MetaL*

To illustrate *MetaL*'s defect backtracking support, we present a set of examples of defects in the definition of $Ovlisp_L$. Presented defects are selected from Section 3.3. We compare how *MetaL* overcomes challenges studied in Chapter 3.

4.3.1 Solving Structural Defects

Structural defects, explained in Section 3.2.2.1, are those producing model corruption. For example, removing VM required class-models or applying forbidden transformation operations produces model corruption. This is automatically detected by *MetaL*, always before kernel generation starts.

In the example from Section 3.3.2, the class Array is not defined. In traditional Bootstrap the failure is manifested during the generation and it solving requires debugging the bootstrapper code. In *MetaL*, Array is loaded by default as it has a defined role. However, removing this class from the model produces the following failure before kernel generation, when the model is tested.

```
ClassModelNotFound: required class with role #Array not found.
```

The example from Section 3.3.3 sets the wrong format value for the class Array. In traditional Bootstrap, this defect produces a failure during kernel generation and solving requires debugging low-level code from the VM simulator. In *MetaL*, operations setting the wrong format for a class-model with role fail to execute. This is for both MOP operations, and wrong definitions in kernel source code (in Tonel format). The following error message is displayed:

```
Incompatible type 'format' for class #Array (Role #Array).
Correct type is 'variable'
```

In the example from Section 3.3.6, instance variables of Class are defined in the wrong order. In traditional Bootstrap, the defect produces a failure in the VM during kernel execution. To find the failure's cause, debugging VM code is necessary. The same defect in *MetaL* produces a failure during model build, when testing before generation.

4.3.2 Solving Reflective Defects

Reflective defects are found in reflective generation instructions and therefore they are written in guest-language code, as explained in Section 3.2.2.1.

Reflective instruction fails. The example from Section 3.3.4 presents a defective reflective instruction that sends an invalid message to an undeclared variable. In traditional Bootstrap, solving requires to debug the code of the AST interpreter used to evaluate reflective instructions.

The bootstrap interpreter is compatible with *MetaL* and it is used by default for reflective instruction evaluation. But stepping operations necessary to implement a debugger are not supported. *MetaL* implements its own interpreter to debug and also evaluate reflective instructions. This interpreter enables stepping operations for guest-language code and better integration with the metamodel.

Continuing the example, we load in $Ovlisp_L$ model the source code file for ClassBuilder used in the original example. Introducing the desired defect requires editing the source of the method build in ClassBuilder. *MetaL* provides integration with code edition tools provided by Pharo. The code browser in Figure 4.5 displays the code of build after edition.

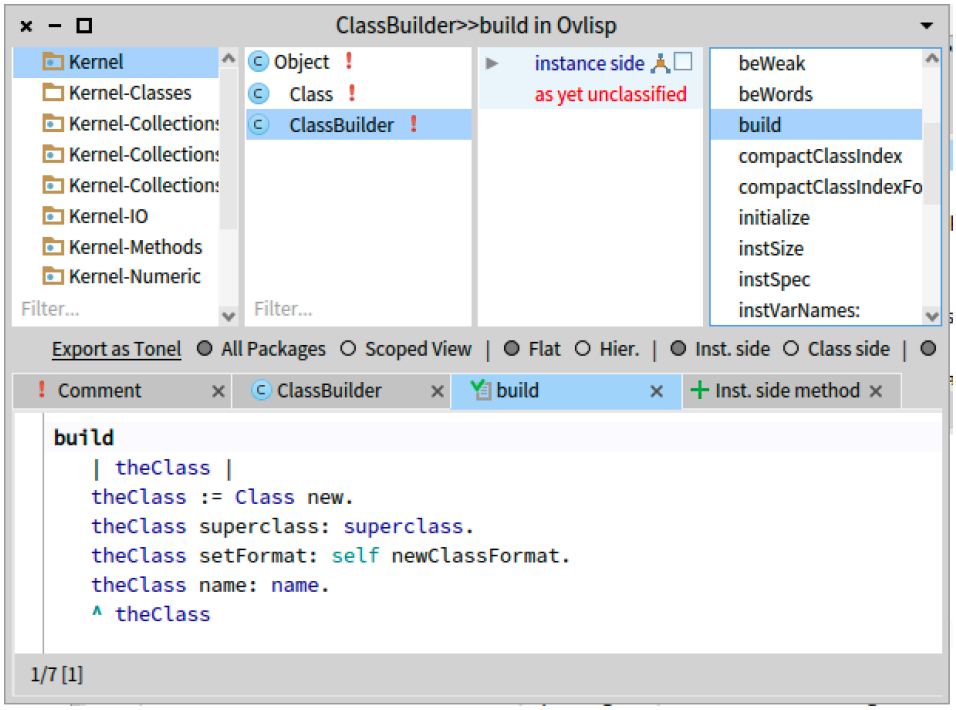


Figure 4.5: Editing $Ovlisp_L$ code using Pharo's code browser.

Even though to generate $Ovlisp_L$ in *MetaL*, a class builder is not required because automatic installation of classes is enough, we override installCustom and define class installation through reflective code.

```
1  OvlispClassModel >> installCustom
2    | type code |
3    type := self type.
4    code := '| newClass |
5      newClass := (ClassBuilder new
6          superclass: ', self superclass name ,';
7          name: ', self name ,';
8          type: ', type ,';
9          yourself)
10       build.
11     newClass'.
12   remote := self evaluateCode: code.
13   ^ remote
```

The execution of reflective code fails opening two debuggers as shown in Figure 4.6. In the back there is the host debugger, the one in front is *MetaL*'s debugger for guest-language code. This debugger provides basic debugging operations. To the left there is the execution stack, in the middle the list of variables in that page. Seeing that

instVarNames value is nil is straightforward.

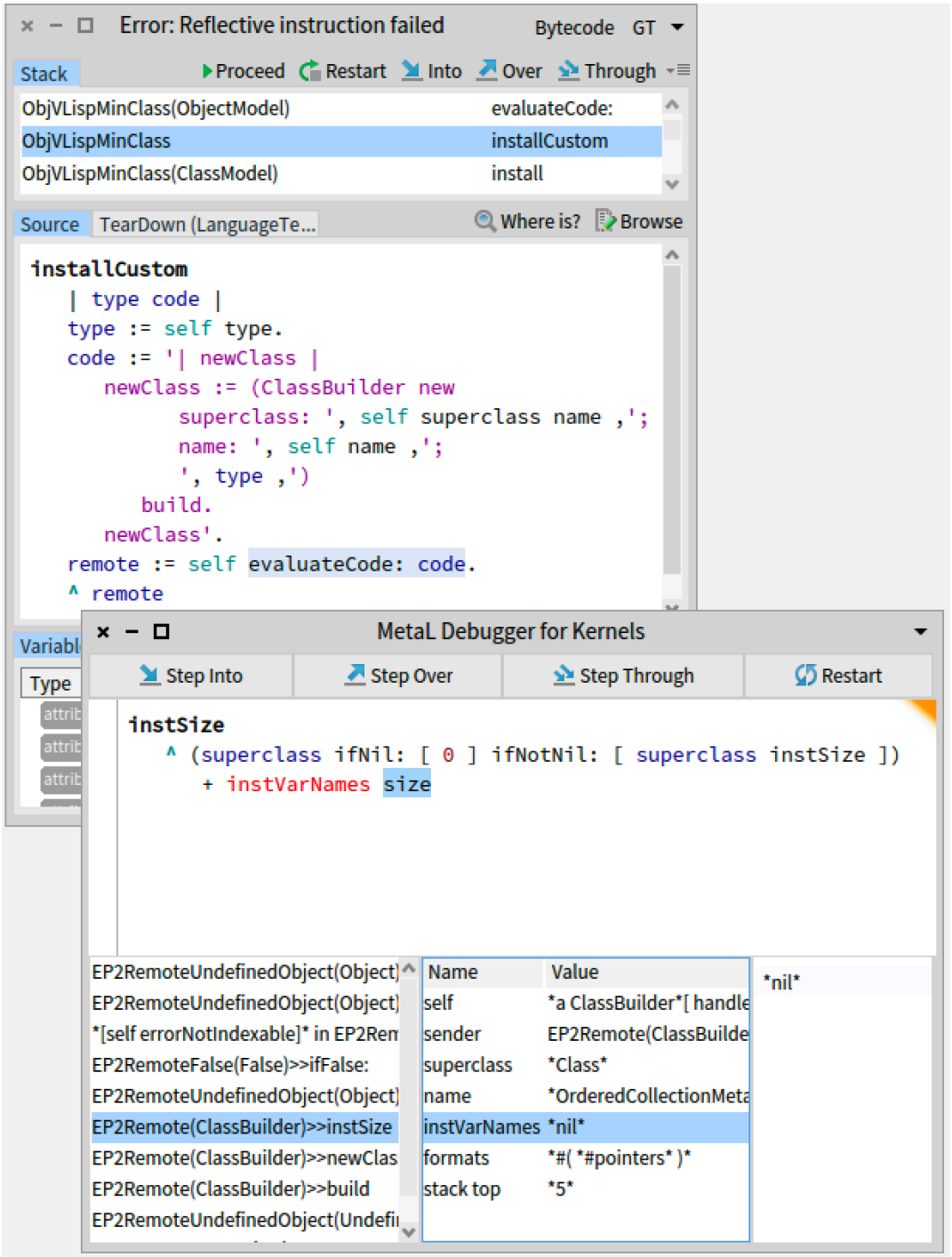


Figure 4.6: Debugging $Ovlisp_L$ code execution using *MetaL*'s Kernel Debugger.

Reflective instruction corrupts the kernel. The example from Section 3.3.5 is produced because reflective code sets a wrong format to the class CompiledMethod. In Bootstrap, the kernel generation fails with a non descriptive message. VM simulator and bootstrapper code must be debugged.

In *MetaL* the failure manifests during generation, specifically during kernel health testing. The host debugger opens displaying the following information.

> Wrong InstSpec in remote class #CompiledMethod. It does not match model.
> Expected: 24 but found: 12

The host debugger, presented in Figure 4.7, presents the point in execution where the test fails. Since kernel tests are applied by *MetaL* automatically in different steps of the execution process, the defect causing corruption should be close. In the execution stack, the message sent immediately before test is installCustom. Reflective code debugging is done using *MetaL*'s special debugger.

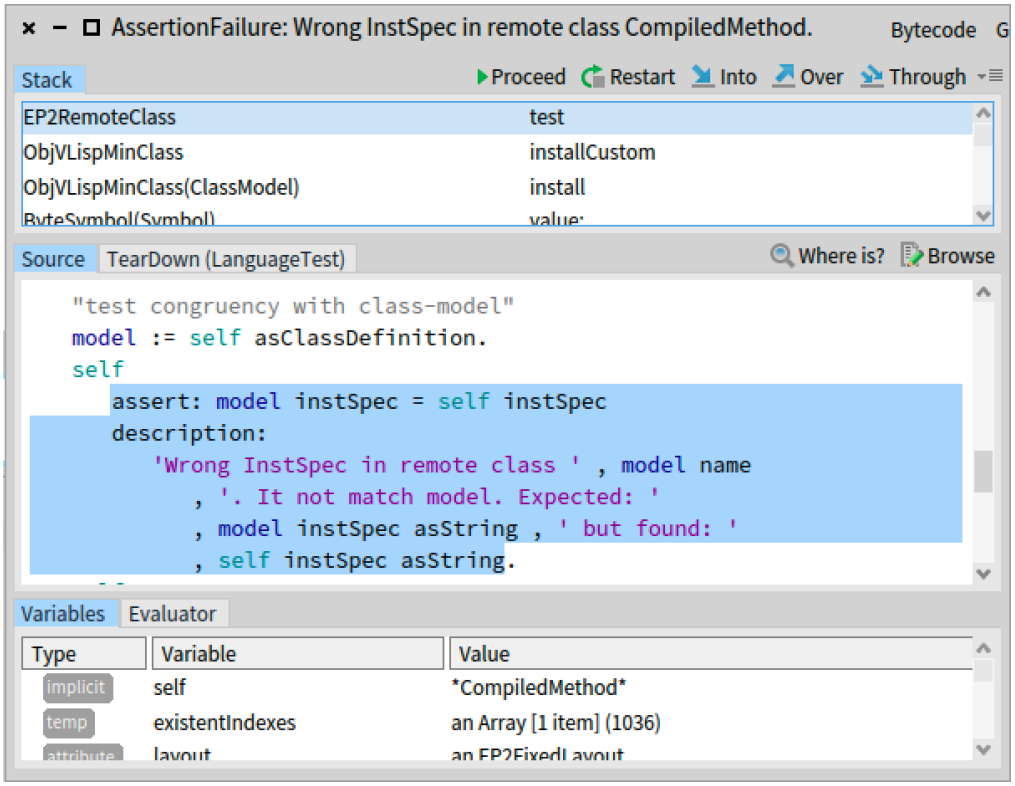


Figure 4.7: Kernel health-test fails during kernel generation.

4.4 Kernel and Model Validations

The reification of VM constraints distinguishes *MetaL* from other language implementation approaches, where the requirements of target components are hidden in their code, resulting in the need to debug generated or code external to the user when failures occur.

In the following two sections we present validations provided by *MetaL*.

4.4.1 Model Validations: Roles

Models are automatically tested on every step of its construction, ensuring they are valid before the kernel generation process starts. *MetaL* provides default tests in the base metamodel, but users are able to extend tests according to their needs.

The method test in ObjectModel subclasses is executed by *MetaL* for testing by default. The following example shows test in ClassModel. If the class-model has role, the role test its structure. When testing a model, every object-model in composition with the model is tested, for example method-models are tested in line 5. The tests below check for structural flaws that apply to every class-model, *e.g.* cycles in the inheritance chain, like a class inheriting from itself, produce errors in VM's method lookup and are forbidden.

```
1 ClassModel >> test
2   self hasRole ifTrue: [ self role test: self ].
3
4   self assert: self methodDict isDictionary.
5   self methodDict do: [ :method | method test ].
6
7   self testInheritanceChainCycles.
8   self assert: self metaclass isNotNil.
9   ...
```

The method role returns a host object that is a reification of the VM requirements for the class with that specific role.

4.4.2 Roles and Smart Mirrors

Roles obtain information about VM requirements from a set of classes defined by *MetaL* and installed in the host system. These classes provide methods through which VM constraints are accessed. Their names are prefixed with 'Remote' (*e.g.* RemoteArray, RemoteClass, etc), and all of them inherit from the class Remote, also defined by

MetaL in the host system. Subclasses of Remote are mapped to roles: each subclass of Remote implements the method role where the name of the associated role is returned. While the class side of Remote subclasses provides static information to validate class-models implementing that corresponding role, the instance side is used to instantiate smart-mirrors, providing high-level operations, tests, and inspectors for kernel-objects.

To make the previous ideas more concrete we introduce an example. In Pharo, the class MethodDictionary defines a special kind of dictionary where classes store their methods. This class, and its instances, play a fundamental role since the VM obtains methods from them in method lookup routines. Therefore, *MetaL* defines the mandatory role #MethodDictionary. RemoteMethodDictionary is a class in the host system defined by *MetaL*. An instance of this class is a smart-mirror pointing to an object in the kernel that is instance of the class MethodDictionary, also installed in the kernel (this is a VM required class, therefore it must exists). The class side of RemoteMethodDictionary encodes VM constraints for the class with role #MethodDictionary. The instance side provides high-level messages to access and manipulate method dictionaries in the kernel. As an example, the method at:ifAbsent: presented below, searches for the value of key in the kernel-object pointed by the smart-mirror. The result is an object that also lives in the kernel. Therefore the returned value is a mirror pointing to the corresponding kernel-object.

```
1  RemoteMethodDictionary class >> role
2    ^ #MethodDictionary
3  RemoteMethodDictionary class >> modelInstSpec
4    ^ 3
5  RemoteMethodDictionary class >> modelInstVarNames
6    ^ #(#tally #array)
7  RemoteMethodDictionary >> at: key ifAbsent: aBlock
8    | index assoc |
9    index := self findElementOrNil: key.
10   assoc := self array at: index.
11   assoc isNilObject
12     ifTrue: [ ^ aBlock value ].
13   ^ assoc asRemoteAssociation value
```

Smart-mirrors raise the level of abstraction for kernel manipulation. Their methods extend *MetaL*'s MOP, giving users high-level operations to specify non-reflective instructions.

Class-Model Automatic Transformations Class-models with a Role are automatically transformed by *MetaL* to fulfil VM requirements. Automatic transformations are performed before user-defined transformations. The following code excerpt shows the method used by roles to transform the class-model received as argument. The value

for the class layout is set. Missing instance variables are added, respecting the definition order set by the Role. Finally, if the role defines methods for the class-model, method-models are created from their source code and then added to the class-model.

```
Role >> transform: aClassModel
  aClassModel layout: (self layoutClass on: aClassModel).
  roleInstVars := self modelInstVarNames.
  roleInstVars
    doWithIndex: [ :ivname :index | self addSlot: ivname index: index ].
  role modelDefaultMethodsSrc do: [ :assoc |
    self addLocalMethodFromSource: assoc value selector: assoc key asSymbol
  ].
```

4.4.3 Kernel Validations: Smart Mirrors

Kernel corruption due to defective imperative instructions is possible even when the language model is structurally correct. As each object-model knows its corresponding remote object, validations are performed by asking object-models to validate their corresponding remote. Each remote is a smart mirror, and therefore instance of a Remote subclass. These classes have a corresponding test method. These methods use a high-level reflective MOP on remotes. These validations are implemented using the role pattern used in static language model validations. To extend these validations the process is the same as extending static validations.

To illustrate the previous ideas we propose the following example: Consider objects in the kernel representing compiled methods. These objects are accessed by the VM during execution. Defects in their structure make the VM fail. Subclasses of Remote implement the method test, where the kernel-object pointed by that smart mirror is checked to ensure its structural soundness with VM requirements. As an example, the following code shows the method test in the class RemoteCompiledMethod.

```
RemoteCompiledMethod >> test
  self assert: self basicClass instSpec = RemoteCompiledMethod modelInstSpec
       description: 'Incorrect format of compiled method: Wrong instSpec of its class'.
  self localBytecodes do: [ :bc | bc isInteger ].
  self literals do: [ :lit | lit test ]
```

4.4.4 Extending validations for a new VM

We provide 28 different templates, one for each role required to use the full features of the target VM. In the case a new feature of the target VM, or a change in the target

VM, it is possible to extend the set of templates. To do this the user must define a new template or modify an existing template and register them in the method ClassModel » templateForRole:. Template subclasses define whether a role is mandatory or not.

MetaL's design makes it possible to change the target VM, since VM requirements are encapsulated in classes.

4.5 Conclusions

This chapter has introduced *MetaL* through a study case, showing that its development style resembles self-surgery. The main abstraction for specification is the metamodel. This metamodel is extended by users to define new languages. Low-level requirements about the VM and most of kernel building logics are hidden from users. The bootstrapper application is generic and users do not need to modify it. The selected study case is the same study case used in the previous chapter to illustrate bootstrap limitations.

We illustrate cognitive distance reduction and debugging capabilities in *MetaL* introducing bugs to the definition of our example kernel and solving them with the help of *MetaL*'s tools working in combination (*i.e.* *MetaL*'s MOP, validations, automatic code generation, smart-mirrors, and debugger).

In the following Chapter we test our approach challenging its ability to generate languages other than $Ovlisp_L$ while maintaining the cognitive distance low. We analyze weaknesses and their causes, proposing possible solutions for improvement.

MetaL Evaluation: Bootstrapping Kernels

Chapter 5

Contents

In the previous chapter we presented *MetaL*, an implementation of our bootstrap-based LIT approach. In this chapter we describe our experience for validating our technique and answering RQ2. Using *MetaL*, we generate seven kernels with important semantic differences. Taking into account available abstractions, capacity for automation, and defect backtracking support, we evaluate the effectiveness of the technique for keeping a low cognitive burden on developers. In addition to being a validation of *MetaL*, our experiments are a contribution in the exploration of the range of different kernels supported by the Pharo VM.

We generate kernels by extending the definition of $Ovlisp_L$[1] or the base language definition. The family of kernels generated by us is illustrated in Figure 5.1. These kernels are the following.

- $Ovlisp_L{}^{slot}$ introduces a first-class representation of instance variables using slots, showing that extending kernels with pre-defined features provided by *MetaL* is simple (Section 5.1).

- *ObjVLisp* implements multiple-inheritance for classes, showing that reflective features from the base kernel definition are powerful and easy to use (Section 5.2).

[1] $Ovlisp_L$ is a minimal class-based language with single inheritance and explicit metaclasses presented in Section 4.2)

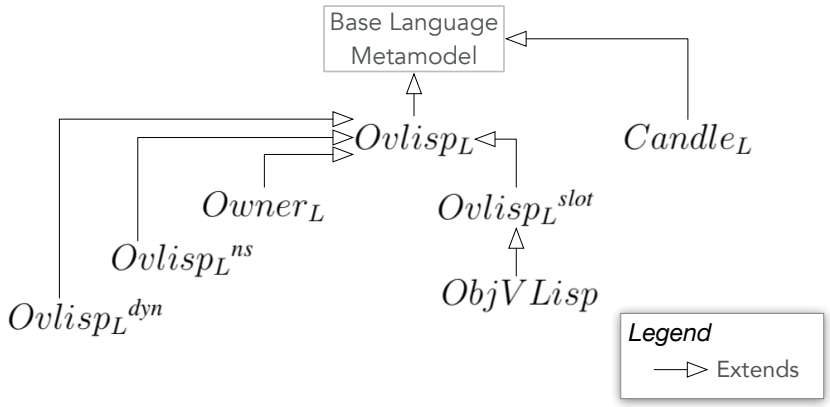


Figure 5.1: Tree of kernels generated to validate *MetaL*.

- $Ovlisp_L{}^{ns}$ introduces a basic implementation of namespaces, showing that variable binding of object fields is specified at a high-level of abstraction, as methods in the metamodel (Section 5.3).
- $Candle_L$ extends *MetaL*'s base language introducing implicit metaclasses *à la Pharo* and class variables, showing that *MetaL* is not limited to $Ovlisp_L$ like languages (Section 5.4).
- $Owner_L$ introduces a control-policy to the execution of reflective operations, showing that the process of designing and implementing a model that challenges VM constraints is kept at the abstraction level of the language specification (Section 5.5).
- $Ovlisp_L{}^{dyn}$ introduces dynamic binding for temporary variables, showing that customizations to bytecode compilation are specified at a high-level of abstraction (Section 5.6).

Then we present an additional experiment done by an external user in Section 5.7). This experiment not only shows *MetaL*'s ability to produce different kernels, but also its ease of use. Finally, we close with a discussion about *MetaL*'s limitations and possible improvements in Section 5.9.4 followed by the main conclusions of the chapter in Section 5.10.

MetaL Base Metamodel

As a help for the reader, we present *MetaL*'s base metamodel introduced for the first time in Section 4.2.1. Kernels generated by us extend this metamodel of $Ovlisp_L$.

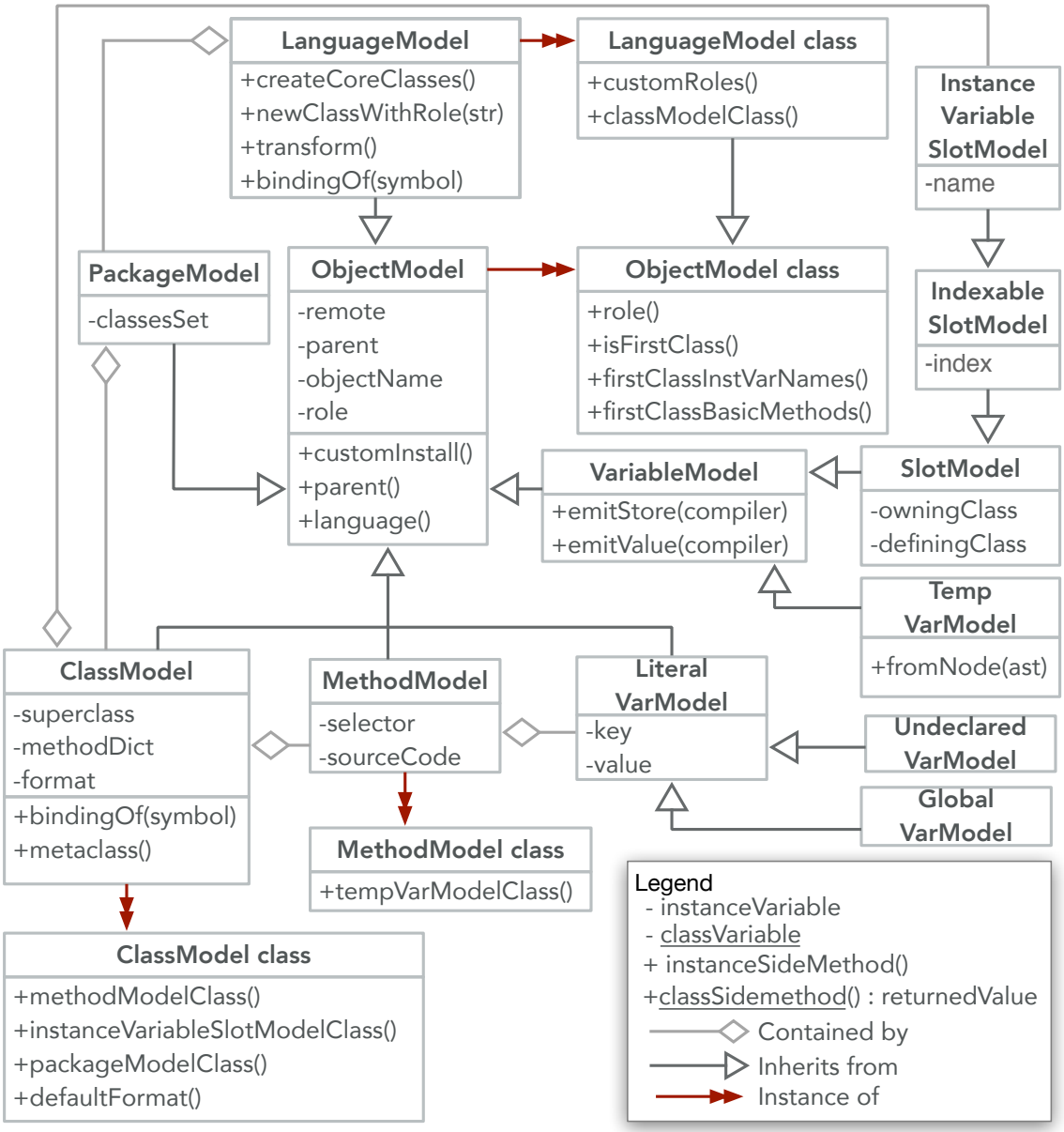


Figure 5.2: *MetaL* Base Language Metamodel (Identical to Figure 4.2).

5.1 $Ovlisp_L{}^{slot}$

$Ovlisp_L{}^{slot}$ is an extension to $Ovlisp_L$ that implements instance variables as first-class objects named slots. Slots are a useful tool to control the semantics of variable evaluation in a language. For this reason, *MetaL* provides special support for their implementation as described below.

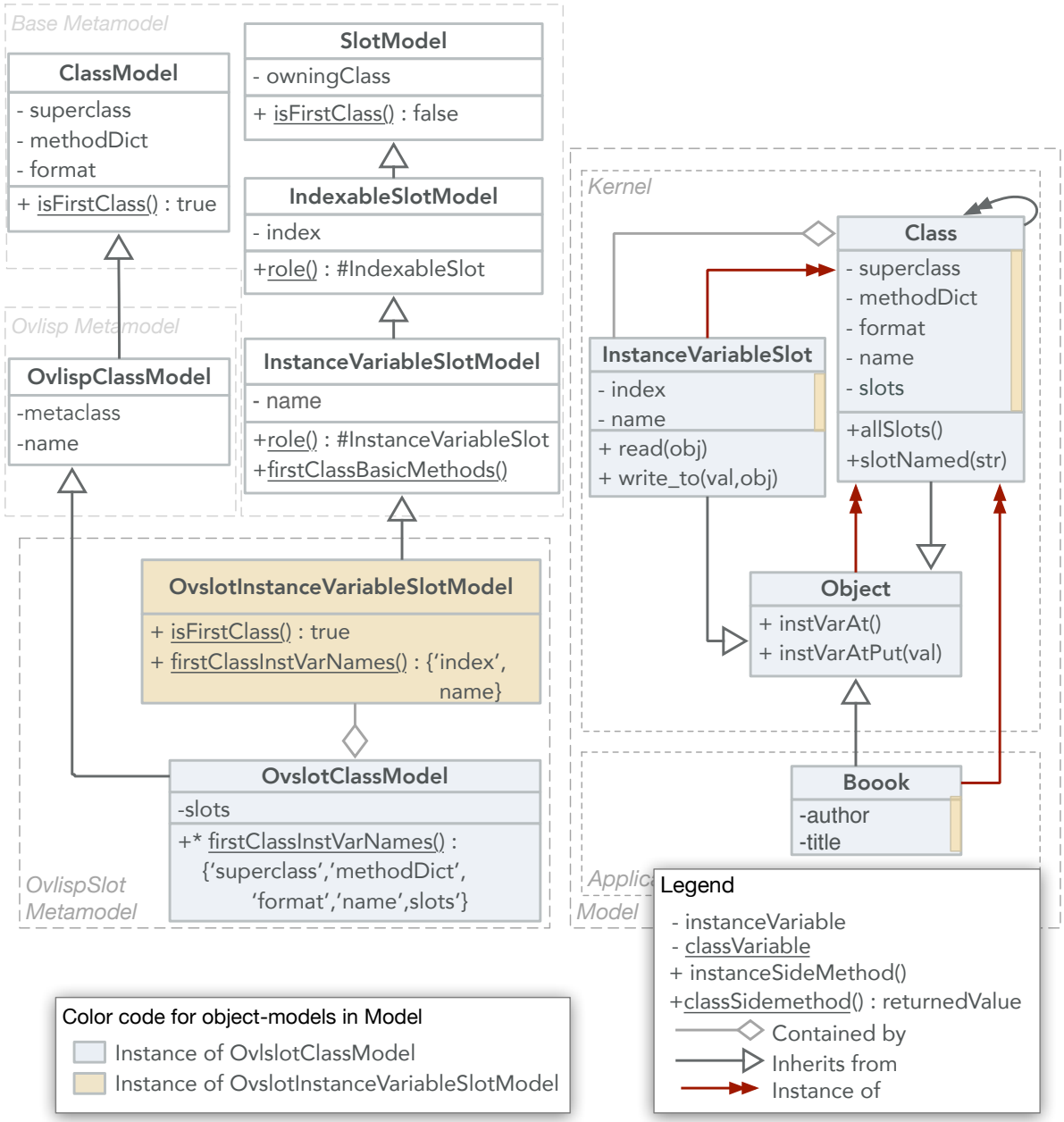


Figure 5.3: $Ovlisp_L{}^{slot}$ metamodel and model.

5.1.1 Application

The code below is the entry point installed in the kernel. In this code, slot objects are used to read the instance variables in an object. The expected output is "Ernest Hemingway The Old Man and the Sea". Methods read: and write:to: define the mechanism used by slots to access variables. Book is not part of the kernel but of the application.

```
1 "entry point code"
2 | book |
3 book := Book new.
4 (Book slotNamed: #author) write: 'Ernest Hemingway' to: book.
5 (Book slotNamed: #title) write: 'The Old Man and the Sea' to: book.
6 Book allSlots do: [:slot |
7   System log: (slot read: book) asString ]
```

5.1.2 Metamodel

$Ovlisp_L{}^{slot}$'s metamodel and model are presented in Figure 5.3. Colors reveal of which metamodel class is an object-model instance. Users do not need to create a class to represent slots because *MetaL* already uses slots to represent instances variables, but their representation as objects is not installed in the kernel by default.

Class-models are divided in two groups: *kernel* and *application*. Classes belonging to the application are installed in the kernel during bootstrap, as our kernel does not contain a compiler and cannot install classes at run-time.

Listing 5.1 shows the code to create the metamodel. The metamodel is an extension of $Ovlisp_L$'s, but it has one extra class to represent slots. The class OvlispClassModel declares one extra field named slots where class-models store their instance variables as a collection of slots. To specify "containment" relationships between classes in the metamodel, the methods classModelClass and instanceVariableSlotModelClass are defined.

Slots as first-class objects. The class OvslotSlotModel overrides the method isFirstClass to indicate that it should be mapped to a class-model with the same role, #InstanceVariableSlot. For each class in the metamodel declared as "first class", *i.e.* returning true from isFirstClass, *MetaL* automatically creates a class-model with the same role, adding it to the model.

```
OvlispLanguage subclass: #OvslotLanguageModel
  instanceVariableNames: ''

OvlispClass subclass: #OvslotClassModel
  instanceVariableNames: 'slots'

InstanceVariableSlotModel subclass: #OvslotSlotModel
  instanceVariableNames: ''

OvslotLanguageModel class >> classModelClass
  ^ OvslotClass

OlslotClassModel class >> instanceVariableSlotModelClass
  ^ OvslotSlot

OvslotSlotModel class >> isFirstClass
  ^ true
```

Listing 5.1: $Ovlisp_L^{slot}$ metamodel definition

MetaL provides a default implementation for them, returned by the method firstClassBasicMethods from the class side of InstanceVariableSlotModel. The default implementation is used to create the methods read: and write:to:, adding them to the class-model with role #InstanceVariableSlot. This mechanism is not unique to #InstanceVariableSlot. Class-models mapped to a "first class" metamodel class, are automatically transformed to declare the same instance variables as that class and to include methods defined in firstClassBasicMethods.

5.1.3 Model

Our entry point sends the message allSlots to the class Point. We need to create this method and add it to the class-model Class[2] as shown in Listing 5.2.

No further customizations are necessary. Users do not need to initialize the variable slots because ClassModel in *MetaL* already use slots to represent their instance variables, however these slots are not installed in the kernel by default. Declaring an extra field named slot in a subclass of ClassModel and making InstanceVariableSlot "first class" is all that it takes to create a kernel where instance variables are first-class objects. Now, we can build the model and generate the kernel to execute it with the target VM.

[2]When a message is sent to an object in Pharo, the method lookup starts in the class of that object and, if not found, it continues up in the inheritance hierarchy. In $Ovlisp_L$ Point is instance of Class, therefore the message allSlots is first searched in Class.

```
" provided by MetaL "
InstanceVariableSlot >> read: anObject
  ^ thisContext object: anObject instVarAt: index

InstanceVariableSlot >> write: aValue to: anObject
  ^ thisContext object: anObject instVarAt: index put: aValue

" defined by the user "
Class >> allSlots
  self superclass
    ifNil: [ ^ slots asOrderedCollection ].
  ^ self superclass allSlots asOrderedCollection
    addAll: slots;
    asArray

Class >> slotNamed: aSymbol
    ^ (self allSlots
        select: [:slot | slot name = aSymbol asSymbol ]) anyOne
```

Listing 5.2: $Ovlisp_L{}^{slot}$ important model definitions

5.1.4 Discussion

Creating a language that uses one of *MetaL*'s pre-defined features is done in a few steps. *MetaL* is able to automatize the generation of the class-model associated to the implemented feature (*e.g.* the class-model InstanceVariableSlot including its methods), but integration with other classes must be done manually: methods for querying slots in Class were created by hand. Ideally, we would like users to be able to integrate language features modularly.

5.2 *ObjVLisp*

ObjVLisp [Cointe 1987] is a minimal class-based language with explicit metaclasses and multiple-inheritance of methods and instance variables. In this example, reflective capabilities in *MetaL* base kernel are used to implement multiple inheritance.

5.2.1 Application

To test multiple inheritance of methods and instance variables, we create two classes: Animal and Pet, and a third class named Dog that inherits from both previous classes. This is illustrated in Figure 5.4, in the application section. The following code creates a dog, setting values for its instance variables: both inherited and declared by itself.

The message fullData access all instance variables in dog to return a string for display. This method does not use accessors to get the value of variables, instead, the name of variables is directly put in the code. The expected result is "Spike, black, mixed".

```
" entry point "
  | dog |
  dog := Dog new
    name: 'Spike'
    color: 'black';
    breed: 'mixed'.
  System log: dog fullData asString.
  System quit

Dog >> fullData
  ^ name, ', ', color, ', ', breed
```

5.2.2 Metamodel

ObjV Lisp metamodel and model are presented in Figure 5.4. We have extended $Ovlisp_L{}^{slot}$ metamodel, inheriting its implementation of slots. We define two extra classes: ObjVLispMultiClassModel and ObjVLispMultiSlotModel to represent classes with multiple-inheritance and their instance variables. Class-models in the *Kernel* section are instance of ObjVLispClassModel, while those in the *Application* are instance of ObjVLispMultiClassModel.

ObjVLispMultiClassModel is key in our design. Listing 5.3 presents its implementation. This class inherits the variable superclass where class-models store the reference to one of its superclasses. For the rest, it declares one extra instance variable named superclasses where references to extra superclasses is stored in a collection. The value for superclasses is initialized as an empty collection, as shown in the method initialize. By default, the metaclass of a class-model representing a multiclass (we call them "multi classes") is set as MultiClass, and its superclass as MultiObject.

The method addSuperclass: adds the class-model received as argument to the collection of superclasses. To implement multiple inheritance of methods, we add a reference to each method inherited from the new superclass to the method dictionary of the class. The message at:ifAbsentPut: ensures that the reference is added only if the method is not already present. In this way we do not override local methods with methods defined in superclasses. Conflicting inherited methods, *i.e.* multiple superclasses defining the a method with the same selector, is not explicitly handled. The method inherited is that from the superclass that is before in the collection of superclasses. Notice that we are not creating a copy of the method, instead we are just adding a reference to the method in the superclass into the method dictionary of the subclass.

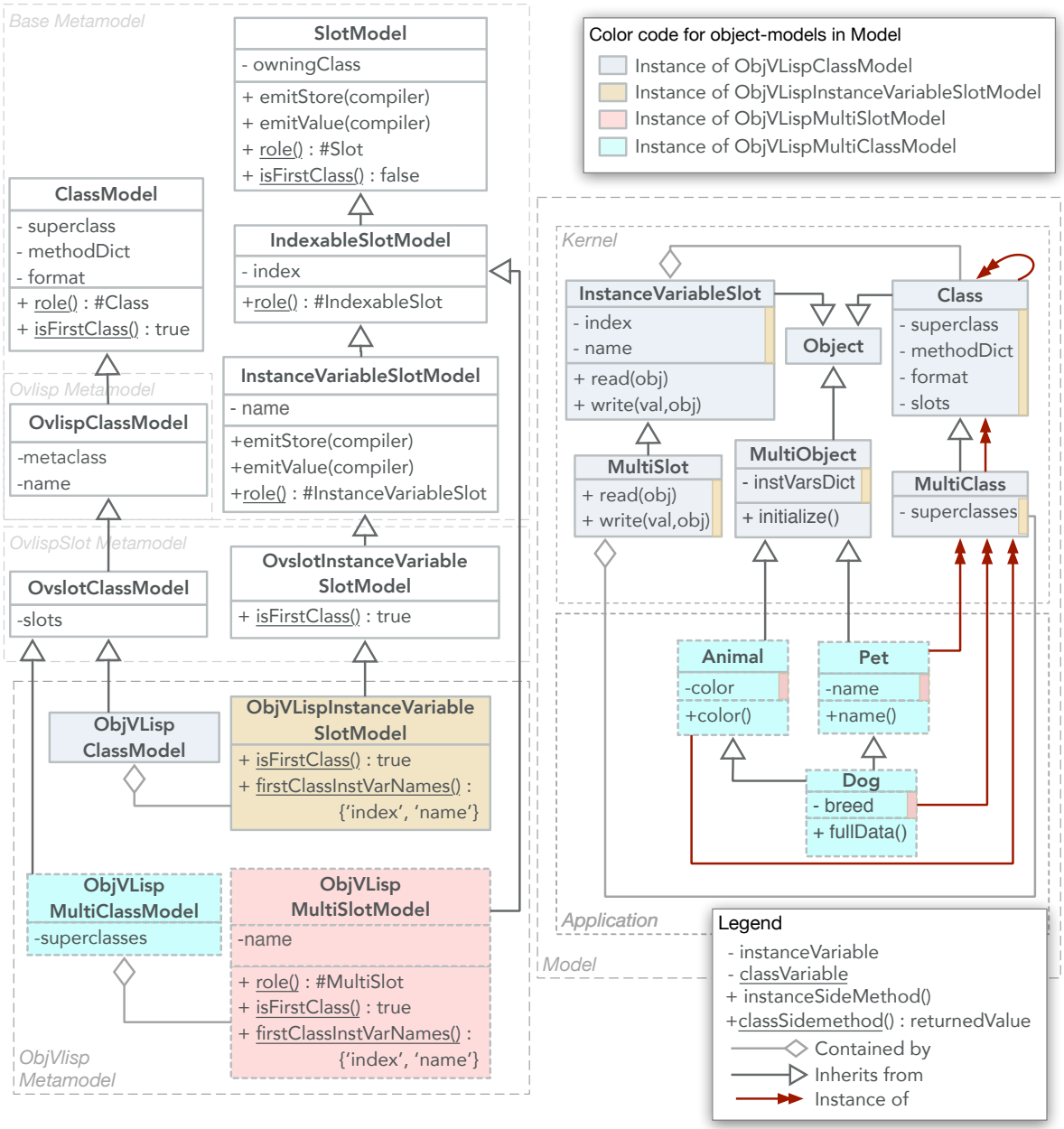


Figure 5.4: *ObjVLisp* metamodel and model.

Multiple inheritance of instance variables relies in the method allSlots, which takes into account extra superclasses for calculating all the slots of a class-model. As it happens with methods, we do not handle conflicts. If two superclasses declare instance variables with the same name, the subclass inherits only one instance variable with that name.

```
1  ObjVLispMultiClassModel >> initialize
2    super initialize.
3    self metaclass: (self language classNamed: #MultiClass).
4    self superclass: (self language classNamed: #MultiObject).
5    superclasses := #()
6
7  ObjVLispMultiClassModel >> addSuperclass: aClassModel
8    superclasses := superclasses asOrderedCollection
9      add: aClassModel;
10     asArray.
11   aClassModel methodDict
12     do: [ :method | self methodDict at: method selector ifAbsentPut: method ]
13
14 ObjVLispMultiClassModel >> allSlots
15   | result extraSlots |
16   " get my slots and slots inherited by normal inheritance "
17   result := super allSlots asSet.
18   " add slots inherited from extra superclasses "
19   extraSlots := self superclasses flatCollect: #allSlots.
20   result := result addAll: extraSlots.
21   ^ result
```

Listing 5.3: Important methods in *ObjVLisp* metamodel

5.2.3 Model

The most important elements in our model are those used to implement reading and writing of instance variables. Their definition is presented in Listing 5.4. The class-model MultiObject declares the instance variable instVarsDict where objects store the value of each one of their instance variables. Every object that is instance of MultiObject, or one of its subclasses, has its own dictionary to store the value of its fields. For example, an instance of Animal that defines its color as "white", has the register #color -> 'white' stored in its dictionary of instances.

The reflective methods read: and write:to: in MultiSlot are executed each time an instance variable declared by a "multi-class" is respectively evaluated or assigned. Users do not need to explicitly call these methods. It is the Pharo compiler who introduces an

instruction that sends the message read: or write:to: when it encounters the name of a slot variable in the code. However, this is not the case for InstanceVariableSlots, for which the reflective message send is not introduced. That is why ObjVLispMultiSlotModel does not inherit from InstanceVariableSlotModel, but from SlotModel. The mechanism to introduce (or not) the reflective message send is defined in the methods emitStore: and emitValue: in SlotModel and its subclasses.

```
1  MultiSlot >> read: anObject
2    ^ anObject instVarsDict at: name
3
4  MultiSlot >> write: aValue to: anObject
5    ^ anObject instVarsDict at: name put: aValue
6
7  MultiObject >> initialize
8    super initialize.
9    instVarsDict := Dictionary new.
10   self class allSlots do: [:slot |
11     instVarsDict at: slot name ifAbsentPut: nil ]
12
13 MultiClass >> allSlots
14   | result extraSlots |
15   result := super allSlots asSet.
16   " add slots inherited from extra superclasses "
17   extraSlots := self superclasses flatCollect: #allSlots.
18   result := result addAll: extraSlots.
19   ^ result
```

Listing 5.4: Important definitions in *ObjVLisp* model

5.2.4 Discussion

The Pharo VM implements single inheritance for methods and instance variables. With *ObjVLisp* we have shown that reflective features in the base metamodel are useful to overcome semantic constrains imposed by the Pharo VM. Our implementation of multiple inheritance is naive, as it does not take into account the resolution of conflicts arising due to collision of names. There is also the efficiency problem, as we use dictionaries for objects to store the values of their variables. However, our goal was to experiment with *MetaL*. We believe that a more robust implementation can be done using *MetaL*.

Multiple inheritance cannot be introduced in classes inside the *kernel*, as the infinite call recursion problem arises (*e.g.* the value of an instance variable is obtained from a dictionary which must read the value of its own instance variables to obtain the required value, the value of a dictionary instance variable is obtained from a dictionary

which needs to read its own instance variables, and so on). However, introducing this problem in a kernel during bootstrap does not make the host system crash, contrary to what would happen in Self-Surgery. Moreover, we can debug guest-language code using *MetaL*'s debugger to find the reason of circularities.

5.3 $Ovlisp_L{}^{ns}$

In Pharo, classes and globals are visible from the whole system and, even though classes are classified in packages, packages do not affect variable scoping. Introducing namespaces in Pharo would increment system modularity and limit dependencies between system parts [Teruel 2012]. In this section, we provide a simplified implementation of namespaces where packages define a scope in between classes and the full environment, resulting in modified class visibility (a class is only visible from classes in the same package). Our implementation of namespaces also provides support for definition of classes with the same name, which is not possible in standard Pharo. Packages are represented as first-class objects and they are visible from the whole system. Referencing a class from an external package is done by sending the message `>` to the package using the name of the class as argument. Importing packages is not covered in this example.

5.3.1 Application

The kernel's entry point, presented in Listing 5.5, tests that definition of multiple classes sharing the same name is possible and that class visibility is limited to packages. We create classes according to Figure 5.5. `Application`, `Multimedia`, and `HairSalon` are packages containing classes. The entry point is installed in the class `Application`. We expect that the result of the message `getClip` is an instance of the class `Clip` in the same package as the receiver's class. Finally, we expect that the last line of code prints `nil`, as `Clip` is not found in package `App`.

5.3.2 Metamodel

MetaL represents packages as object-models by default. Similarly to the slots example presented in Section 5.1, users add the field `packages` to `OvlispNSClassModel`, and create a subclass of `PackageModel` that returns `true` from method `isFirstClass`. Users do not need to initialize the value of the variable `packages`, as classes already know their package. To install a class in the kernel, *MetaL* uses accessor methods (*e.g.* `package`

```
" entry point code "
  | winamp stylist |
  winamp := (Multimedia > #Winamp) new.
  stylist   := (HairSalon > #Hairdresser) new.

  winamp getClip play. "prints 'la la la'"
  stylist getClip putIn.  "prints 'click'"

Hairdresser >> getClip
  ^ Clip new

Winamp >> getClip
  ^ Clip new

" in HairSalon "
Clip >> putIn
  System log: 'click'

" in Multimedia "
Clip >> play
  System log: 'la la la'
```

Listing 5.5: $Ovlisp_L{}^{ns}$ entry point

and package:) to get the instance variables defined in that class and install convert them into kernel-objects. This was explained in Section 4.2.5.

5.3.3 Model

To compile code, the Pharo compiler receives source code together with an environment and a class, both used as context for binding names with variables. The environment is a dictionary containing references to classes and globals that are visible from the compiled code. The environment is used as the last scope for variable lookup (the last place to check). The class also serves as a scope for variable lookup. The compiler sends the message bindingOf: to the class received as context to obtain the variable bound to a given name. In this way, the compiler delegates the responsibility of variable binding to classes and environments.

When a method is compiled during bootstrap, the class of the method together with its environment are given to the compiler to use as context. The class' environment is obtained by sending the message environment to the class.

The default implementation of bindingOf: in ClassModel and LanguageModel is presented in Listing 5.6. In that code we can see that the environment of a class is the language, and the language scans all classes and globals when looking for the binding

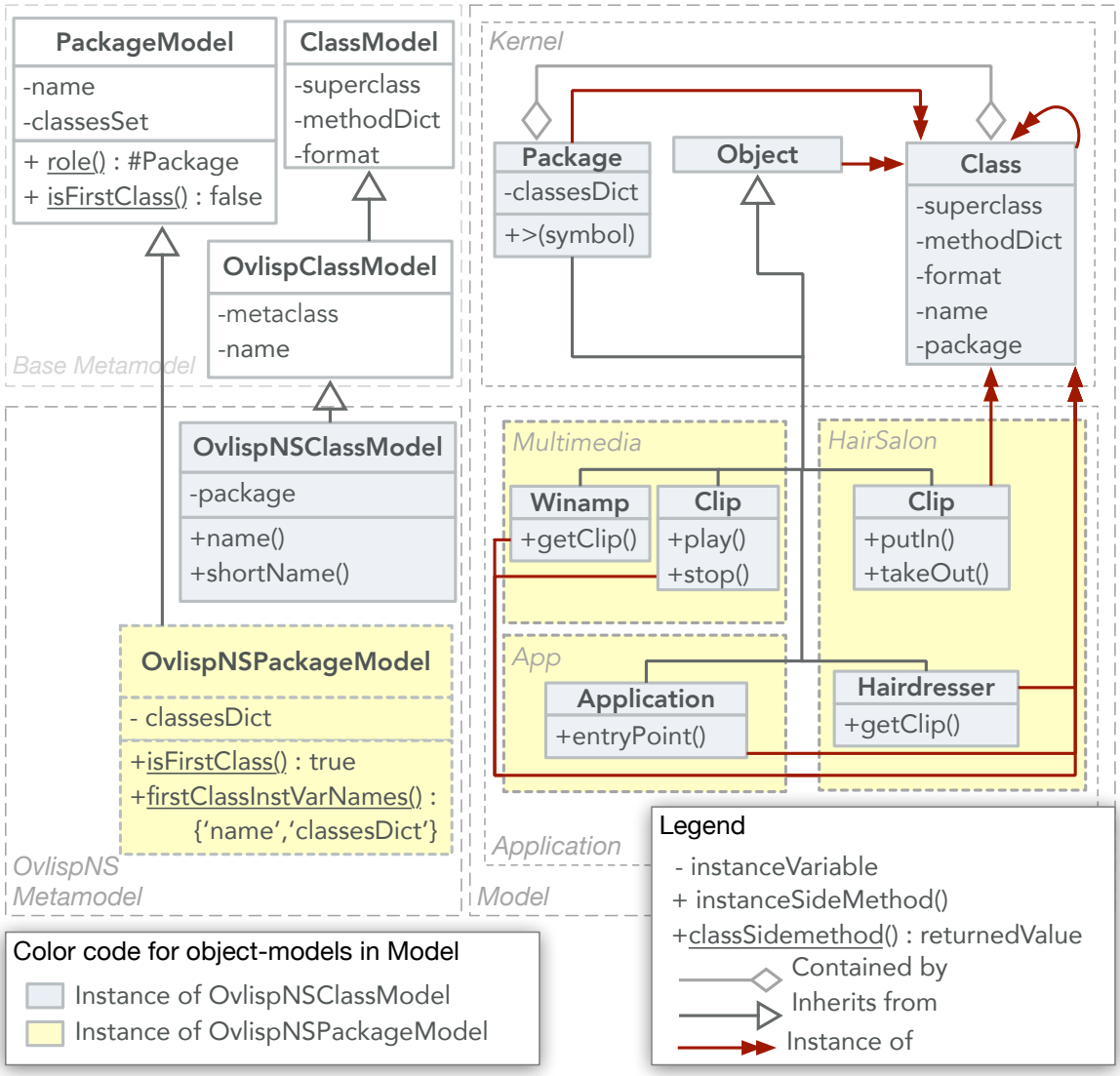


Figure 5.5: $Ovlisp_L{}^{ns}$ metamodel and model.

of a name. The internal lookup of variables in a class is implemented in the method innerBindingOf:, which must be defined by the user. In $Ovlisp_L$ this method returns nil, because instance variables are the only kind of variables and their value is not defined in the class, nor in the environment.

```
1  ClassModel >> environment
2    ^ self language
3
4  LanguageModel >> bindingOf: aSymbol
5    | result |
6    result := self
7      className: aSymbol
8      ifAbsent: [ nil ]
9      ifPresent: [ :class |
10       class ensureRemote. " install remote if it does not exist "
11       class ].
12   result := result
13     ifNil: [ (self globalVariableNamed: aSymbol) ifNil: [ ^ nil ] ].
14   ^ self newGlobalBindingFor: aSymbol
15
16 ClassModel >> bindingOf: varName
17   | aSymbol |
18   aSymbol := varName asSymbol.
19
20   ^ (self innerBindingOf: aSymbol) ifNil: [
21      self environment bindingOf: aSymbol
22   ]
23 ClassModel >> innerBindingOf: varName
24   self subclassResponsibility
```

Listing 5.6: Default variable bindings in *MetaL*.

Restricting class visibility to packages requires the modification of methods used for variable binding. In our implementation, as presented in Listing 5.7, the environment of a class is now its package. Additionally, the method bindingOf: in OvlispNSLanguage-Model does not scan its classes, but instead it scans in packages. Finally, the method bindingOf: in package has been implemented, scanning in classes and, if no class named that way is found, delegating binding to the Language.

```
1  OvlispNSClassModel >> environment
2    ^ self package
3
4  OvlispNSPackageModel >> environment
5    ^ self language
6
7  OvlispNSLanguageModel >> bindingOf: aSymbol
8    (self globalVariableNamed: aSymbol)
9      ifNil: [ (self packageNamed: aSymbol ifAbsent: [nil]) ifNil: [ ^ nil ] ].
10   ^ self newGlobalBindingFor: aSymbol
11
12 OvlispNSPackageModel >> bindingOf: aSymbol
13   | result |
14   result := self
15     classNamed: aSymbol
16     ifAbsent: [ nil ]
17     ifPresent: [ :classModel | classModel ensureRemote ].
18   result := result
19     ifNil: [ (self environment bindingOf: aSymbol) ifNil: [ ^ nil ] ].
20   ^ self environment newGlobalBindingFor: aSymbol
21
22 ClassModel >> bindingOf: varName
23   | aSymbol |
24   aSymbol := varName asSymbol.
25   ^ (self innerBindingOf: aSymbol) ifNil: [
26      self environment bindingOf: aSymbol
27   ]
28
29 OvlispNSClassModel >> innerBindingOf: aSymbol
30   ^ nil
```

Listing 5.7: Variable bindings in $Ovlisp_L{}^{ns}$.

5.3.4 Discussion

With $Ovlisp_L{}^{ns}$ we have introduced static resolution of class names to the language, limiting the visibility of classes to their package. This modification in the architecture cannot be applied in self-surgery, because a part of the system depends on core classes such as Array and Number. Hiding these classes would produce failures in core parts of the system.

With this example we have also shown that modifying the scope of environment and class variables is done in *MetaL* by changing methods in the metamodel. The abstraction level is kept high during the whole process.

5.4 $Candle_L$

$Candle_L$ is a Pharo-like language, *i.e.* same semantics as Pharo but different model of classes, defined by Polito et.al. [Polito 2015] taking *MicroSqueak* as base. Unlike $Ovlisp_L$, it implements implicit metaclasses "à la Pharo", as illustrated in Figure 5.6. This means that for each class a metaclass (whose only instance is the class) is implicitly added to the system [Goldberg 1983][3].

Same as in Pharo, implicit metaclasses are named after their only instance name, adding the suffix 'class' (*e.g.* Point class is Point's metaclass). $Candle_L$ implements class variables in the same way Pharo does. However, $Candle_L$ classes differ from Pharo metaclasses in their definition, declaring a different set of instance variables and they do not support Traits.

5.4.1 Application

```
| jack spike |
System log: Dog numberOfDogs asString. " prints 0 "
jack := Dog new.
System log: jack numberOfDogs asString. " prints 1 "
spike := Dog new.
System log: spike numberOfDogs asString. " prints 2 "

Dog class >> new
  NumberOfDogs := NumberOfDogs + 1.
  ^ super new

Dog class >> numberOfDogs
  ^ NumberOfDogs

Dog >> numberOfDogs
  ^ Dog class numberOfDogs
```

Listing 5.8: $Candle_L$ entry point testing class variables.

5.4.2 Metamodel

We extend *MetaL*'s base metamodel as illustrated in Figure 5.6. The class CandleClassModel represents normal classes in $Candle_L$'s, while CandleMetaclassModel is used to

[3] Smalltalk-80: the Language and its Implementation, chapter 5

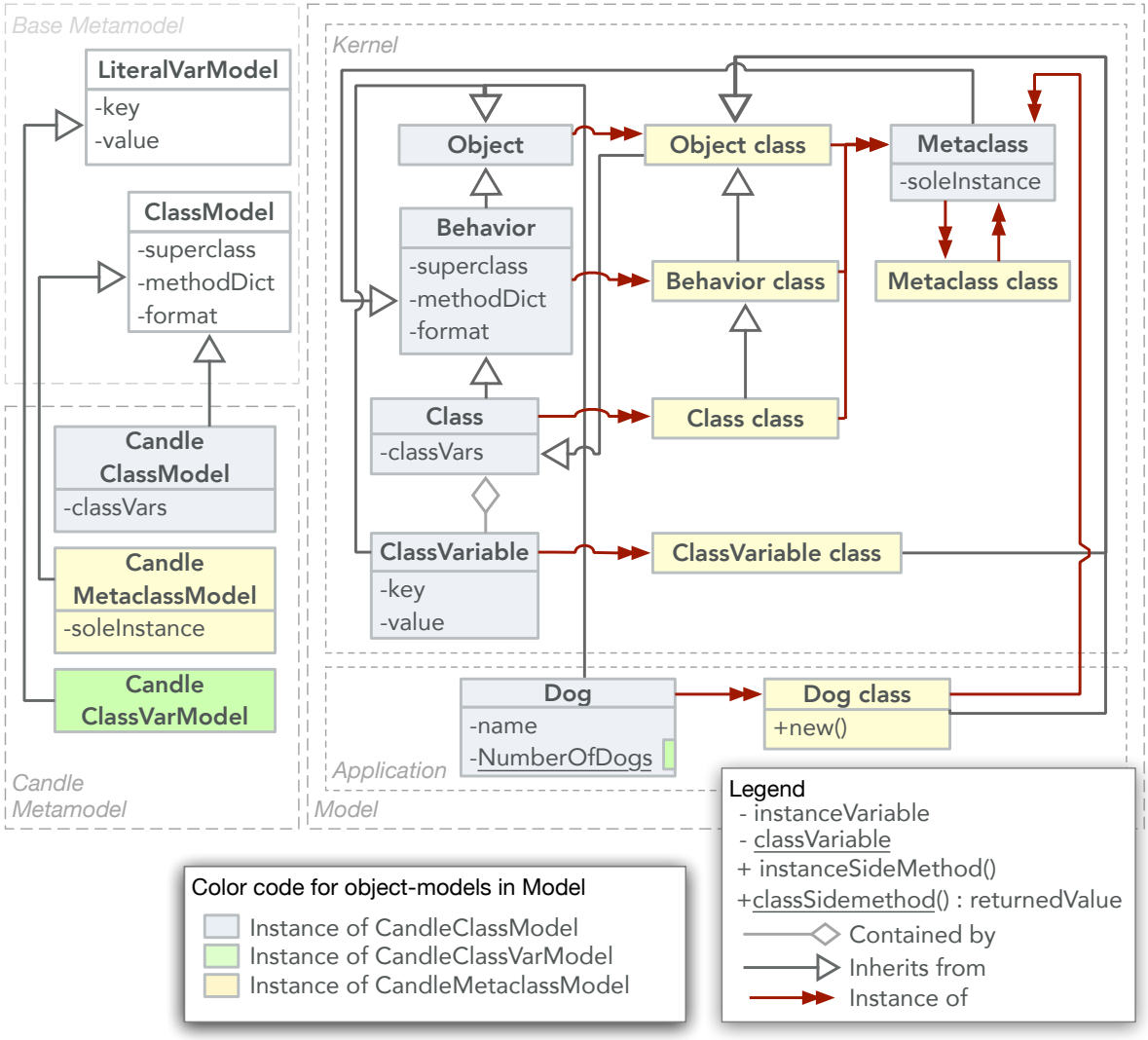


Figure 5.6: $Candle_L$ metamodel and model.

represent their implicit metaclasses. In Figure 5.6 classes having the suffix 'class' in their name are instances of CandleMetaclassModel (we call them "meta-classes"), all others are instance of CandleClassModel (we call them simply "classes"). Finally, the class CandleClassVariableModel represents class variables.

5.4.3 Model

$Candle_L$ and $Ovlisp_L$ models are different at the core level. $Candle_L$'s model core construction must be specified. However, following steps about model automatic completion and kernel generation do not need intervention form the user.

Initialization of class-models. All meta-classes have Metaclass as their metaclass. On their side, classes have as metaclass a meta-class with the same name as them plus the suffix 'class'. This is expressed in methods metaclass as presented in Listing 5.9. Similarly to $Ovlisp_L$, classes have Object as their superclass by default. This is not the case for meta-classes, whose superclass is Object's metaclass, *i.e.* Object class. Therefore, we override the method defaultSuperclass in CandleMetaclassModel. Each time a class is created, its meta-class should be also created. This is implemented in the method initialize in CandleClassModel. Finally, to automatize the process of choosing the class in the metamodel that should be used to create a given class-model, *MetaL* offers the hook classModelFor: where the decision is taken based on the name of the new class-model.

Creation of core class-models . From the previous initialization methods, it is possible to infer which are the core-classes in our model. Core-classes, as explained in Section 4.2.4.1 are those needed to initialize other classes. Therefore, Object, Class, Metaclass, and Object class are core, and therefore they must be created by us in the method createCoreClasses as presented next. Behavior is not core, its introduction can be done in later transformations.

Something to remark is the introduction of the user-defined role #Metaclass. *MetaL* allows the definition of custom-roles, which must be mapped to a class in the metamodel. In this case, the class CandleMetaclassModel is the one representing the role #Metaclass. There fore the class-model with role #Metaclass is automatically transformed by *MetaL*, adding the instance variable soleInstance. The source-code of methods to be added to the class can be defined in the method firstClassBasicMethods (in the class side of the class). In this way, when extending a metamodel, the new language inherits also the definition of special method-models.

```
CandleClassModel >> metaclass
  ^ self language ensureClassNamed: self name, self language metaclassSuffix

CandleMetaclassModel >> metaclass
  ^ self language classWithRole: #Metaclass

CandleMetaclassModel >> defaultSuperclass
  ^ (self language classWithRole: #ProtoObject) metaclass

CandleClassModel >> initialize
  super initialize.
  self metaclass initializeWithSoleInstance: self
  classVars := Dictionary new

CandleMetaclassModel >> initializeWithSoleInstance: aClassModel
  soleInstance := aCandleClass.
  superclass := soleInstance superclass
    ifNil: [ self parent classWithRole: #Class ]
    ifNotNil: [ soleInstance superclass metaclass ]

CandleLanguageModel class >> classModelFor: aString
  ^ (aString endsWith: self metaclassSuffix)
    ifTrue: [ CandleClassModel ]
    ifFalse: [ CandleMetaclassModel ]

CandleLanguageModel class >> metaclassSuffix
  ^ ' class'
```

Listing 5.9: Initialization of class-models in $Candle_L$.

```
CandleLanguageModel >> createCoreClasses
  | object class metaclass objectClass |
  object    := self basicNewClassWithRole: #ProtoObject.
  class     := self basicNewClassWithRole: #Class.
  metaclass := self basicNewClassWithRole: #Metaclass.

  self addClass: object.
  self addClass: class.
  self addClass: metaclass.

  object superclass: nil.
  class superclass: object.
  metaclass superclass: object.

  objectClass := (self basicNewClassNamed: object name, self metaclassSuffix)
    parent: self;
    yourself.
  objectClass superclass: class.
```

```
19  self addClass: objectClass.
20
21  metaclass initialize.
22  object initialize.
23  class initialize.
24  objectClass initialize
```

Initialization of class variables. We create the class-model Dog, which declares the instance variable name and the class variable NumberOfDogs. This last variable must be initialized during bootstrap, otherwise our application fails the first time Dog is instantiated because the message + is send to nil. In *MetaL*, the initial value of a literal variable can be one of the following:

- A host object mapped to a role in *MetaL*, *e.g.* arrays, dictionaries, integers, strings, etc.
- Reflective code that returns the value when evaluated.
- A block using non-reflective instructions to obtain the value from the kernel.

MetaL initializes the values of variables automatically in the kernel. No custom initialization instructions are needed. In this example we use reflective code to define the initial value of NumberOfDogs 0. Using a reflective instruction is an overkill in this case, we do it to illustrate the idea.

```
1  CandleLanguageModel >> transform
2    | dogClass classVar |
3    dogClass := self ensureClassNamed: #Dog.
4    dogClass addSlot: #name.
5    classVar := CandleClassVar named: #NumberOfDogs parent: dogClass
6    classVar initializationCode: '1 — 1'.
7    dogClass classVars at: #NumberOfDogs put: classVar.
```

Definition of class variable semantics. The mechanisms used by the compiler to do name binding were described in Section 5.3.2. In $Candle_L$, we implement the logic for class variable lookup in the method innerBindingOf: in the class CandleClassModel and CandleMetaclassModel. If the lookup fails in the class and its superclasses, the search continues in the environment.

Method innerBindingOf: in class CandleClassModel is presented in Listing 5.10. If the variable is found in the dictionary of class variables, return its binding. Dictionaries in Pharo define the method bindingOf:, which returns an association (key -> value) if

the key exists, otherwise it returns nil. In $Candle_L$, class variables are inherited by the subclasses of a class. Therefore, if the binding is not found, we continue the lookup in the superclass.

If the receiver is a class, the method innerBindingOf: defined in CandleMetaclassModel is used to calculate the variable binding. This method simply returns the binding calculated in its instance side.

```
1  LanguageModel >> bindingOf: aSymbol
2    | result |
3    result := self
4      classNamed: aSymbol
5      ifAbsent: [ nil ]
6      ifPresent: [ :class |
7        class ensureRemote.
8        class ].
9    result := result
10     ifNil: [ (self globalVariableNamed: aSymbol) ifNil: [ ^ nil ] ].
11   ^ self newGlobalBindingFor: aSymbol
12
13 ClassModel >> bindingOf: varName
14   | aSymbol |
15   aSymbol := varName asSymbol.
16
17   ^ (self innerBindingOf: aSymbol) ifNil: [
18      self environment bindingOf: aSymbol
19   ]
20
21 CandleClassModel >> innerBindingOf: aSymbol
22   (self classVars bindingOf: aSymbol)
23     ifNotNil: [ :binding | ^ binding ].
24   self superclass
25     ifNotNil: [ :supercl | ^ supercl innerBindingOf: aSymbol ].
26   ^ nil
27
28 CandleMetaclassModel >> innerBindingOf: varName
29   ^ self soleInstance innerBindingOf: varName
30
31 Dog >> numberOfDogs
32   ^ NumberOfDogs
33
34 Dog class >> new
35   NumberOfDogs := NumberOfDogs + 1.
36   ^ super new
```

Listing 5.10: Metamodel definitions for variable binding in base metamodel and $Candle_L$

5.4.4 Discussion

With $Candle_L$ we show that *MetaL* is not limited to $Ovlisp_L$ like languages and that defining a language with core differences in its model can be done with low efforts. Even though $Candle_L$'s metamodel introduces a new abstraction: metaclasses, *MetaL*'s automatic features continue to work. Using *MetaL* we were able to modify the structure of classes and metaclasses without the need to completely understand the constrains imposed by the VM, showing that *MetaL* removes the need of knowledge about the VM implementation. There was no need to specify custom initialization instructions, even when $Candle_L$ introduces different circular referencing loops, *i.e.* Metaclass and Metaclass class. In this way we have shown that *MetaL* reduces the need from users to handle Bootstrap circular process. These benefits come from *MetaL*'s usage of roles to access classes in a generic way, and also from automatic detection of cycles in the model, adapting kernel initialization behavior according to each case.

The most challenging part in $Candle_L$ definition is model initialization. A correct identification of core-classes is essential to build the model. This can be challenging to unexperienced users, and they should base their first kernels in existing examples. Defects introduced in the process of model construction are debugged with the host debugger. We have shown that defining the semantics for a new kind of variable is done at a high-level of abstraction in a hook of the metamodel. Finally, we have shown that *MetaL* is capable of generating a Pharo-like language.

5.5 $Owner_L$

Ownership has been heavily studied in literature from a type perspective. In addition some works such as the one of [Gordon 2007, Teruel 2015] proposed language designs to take advantages of ownership in the context of dynamically-typed languages. Constraints faced by Teruel et al. in their implementation set an interesting case for language kernel implementation. Showing how *MetaL* supports such language is the point of this section.

$Owner_L$ is a language core that provides an access control policy to reflective operations in a similar way as mentioned in [Teruel 2015]. Reflection is a powerful feature but it might lead to violations in encapsulation and even to security vulnerabilities.

5.5.1 Application

To understand the problem let's start with an example of how reflection breaks object's encapsulation. The example in Listing 5.11 is used to illustrate this, and it is installed as

the kernel's entry point. We have two Person objects *Alice* and *Bob*, each with its own private reference to its own wallet object. In Smalltalk (contrary to Java) encapsulation is strict. It forbids instances of the same class to access the internal state of each other. However, the method Person»steal:, presented below, shows that by using the reflective method instVarNamed:, it is possible for an instance of Person to violate encapsulation and withdraw money from the credit card of another instance of Person. Our goal is to forbid this kind of behavior. Therefore, we expect that an exception is raised when this application is executed in the $Owner_L$ kernel.

```
1  Person>>initialize
2     wallet := Wallet new.
3     wallet add: CreditCard new
4
5  Person>>steal: aPerson
6    notMyWallet := aPerson instVarNamed: #wallet.
7    notMyCard := notMyWallet instVarNamed: #creditCard.
8    notMyPin := notMyCard instVarNamed: #pin.
9    notMyCard withdraw: 1000 euros pin: notMyPin
10
11 alice := Person new.
12 bob := Person new.
13 alice steal: bob
```

Listing 5.11: $Owner_L$ entry point, where reflection is used to break object. In $Owner_L$ this code should raise an exception.

In our example we make Alice steal from Bob. We see in the code that once *Alice* has access to Bob's wallet leaked reference, she is able to introspect the wallet in the same way she did with Bob, allowing her to even execute unauthorized actions, such as withdrawing money from his credit card. This example is not exclusive to Pharo since equivalent implementations of the message instVarNamed: exist in other languages that offer reflective capabilities such as *Python*, *Ruby* and *Java*.

Conceptual Strategy This problem of possible violations of the encapsulation has been addressed in the literature. For example, Teruel et al. [Teruel 2015] proposed to implement a language that provides an access-control policy to reflective calls, based on object dynamic ownership. This access-control policy dynamically determines whether an object is authorized to perform reflective operations on another object. The policy takes into account the relationships between objects. In our example, the relationship between *Bob* and his wallet is not the same as the relationship between *Alice* and *Bob*'s wallet. The difference is that *Bob* owns its wallet while *Alice* does not.

Formally, the transitive relationship of ownership is as follows: An object A *owns* an object B if:

- A and B are the same object.
- A instantiated B in this case we say that A is the *direct owner* of B.
- A owns an object C that is the *direct owner* of B.

For the sake of simplicity we limit our example to control access to reflective operations of introspection, leaving outside reflective modifying capabilities.

The access-control policy establishes that A is allowed to send reflective messages to B if and only if A *owns* B.

Implementation Strategy Strategies to implement this control policy in a programming language are many and varied. One of them is saving the owner of each object in a hash table. However this strategy does not scale well with the number of objects. Another possibility is to encapsulate the reflective operations in separate objects, so called *metaobjects*, each of which would keep a reference to the object it controls. This strategy is extensively discussed in [Teruel 2015], where they describe a mirror-based architecture for controlling access to reflection.

Unlike [Teruel 2015], our strategy does not depend on mirrors to control access to reflection. Instead, we chose to implement a simpler strategy, described below:

- Add an instance variable in each object, which saves a reference to the object's owner.
- In the method that creates instances, add an instruction that sets the value of the instance variable owner.
- Add an owner verification step in reflective methods controlled by our policy.

However, adding an instance variable definition in the class Object (root of the inheritance hierarchy) is forbidden by the VM, because the VM imposes that instances of certain classes with a role have specific structure in memory. For example: instances of the class Array, String and Float.

To overcome the limitations presented above, we build a model where the owner is declared as an extra instance variable in classes where this action is not forbidden by the VM. We extend the structure for classes that do not support the declaration of extra instance variables. We add in their metaclasses a new instance variable named

instancesDict that maps all their instances to their corresponding owner. These dictionaries must be initialized during the kernel generation through imperative initialization instructions.

5.5.2 Metamodel

Figure 5.7 shows $Owner_L$'s metamodel and model. In the model, the instance variable owner has been added only to classes where the VM allows it. This model also includes the instance variable instancesDict defined in Class to store ownership relationships of objects whose layout prevents them from storing this information themselves.

5.5.3 Model

The method OwnershipLModel»transform, shown in the following code extract, applies transformation operations to the model after VM required roles have been loaded.

```
1 OwnershipLModel>>transform
2   (self classWithRole: #ProtoObject) inHierarchyAddInstVarOwner.
3   self modifyReflectiveMethods.
4   self modifyInstantiationMethods
```

Adding owner instance variable and accessors. The method inHierarchyAddInstVarOwner, implemented by us and presented in the following code extract, shows how the instance variable owner is added to the full class hierarchy starting in its root: Object. When trying to add the instance variable owner to a class where the VM constraints forbids it, *MetaL* signals the exception ForbiddenAddInstVar. In this case, the instance variable is not added to the class.

The accessors owner and owner: are different according to whether the instance variable owner was successfully added to the class or not. In classes where the addiction of the variable owner is successful, accessors simply return or modify this variable. In classes where the addition of owner fails, the accessors owner: and owner use the variable instanceDict to store and access the information about the object's owner.

Modifying reflective methods. Methods in the language model are loaded from the basic language definition source code. Methods Object»instVarNamed: and Object»instVarNamed:put: are modified to implement the control restriction policy, as shown in the next code extract. We focus on instVarNamed: as an example. We

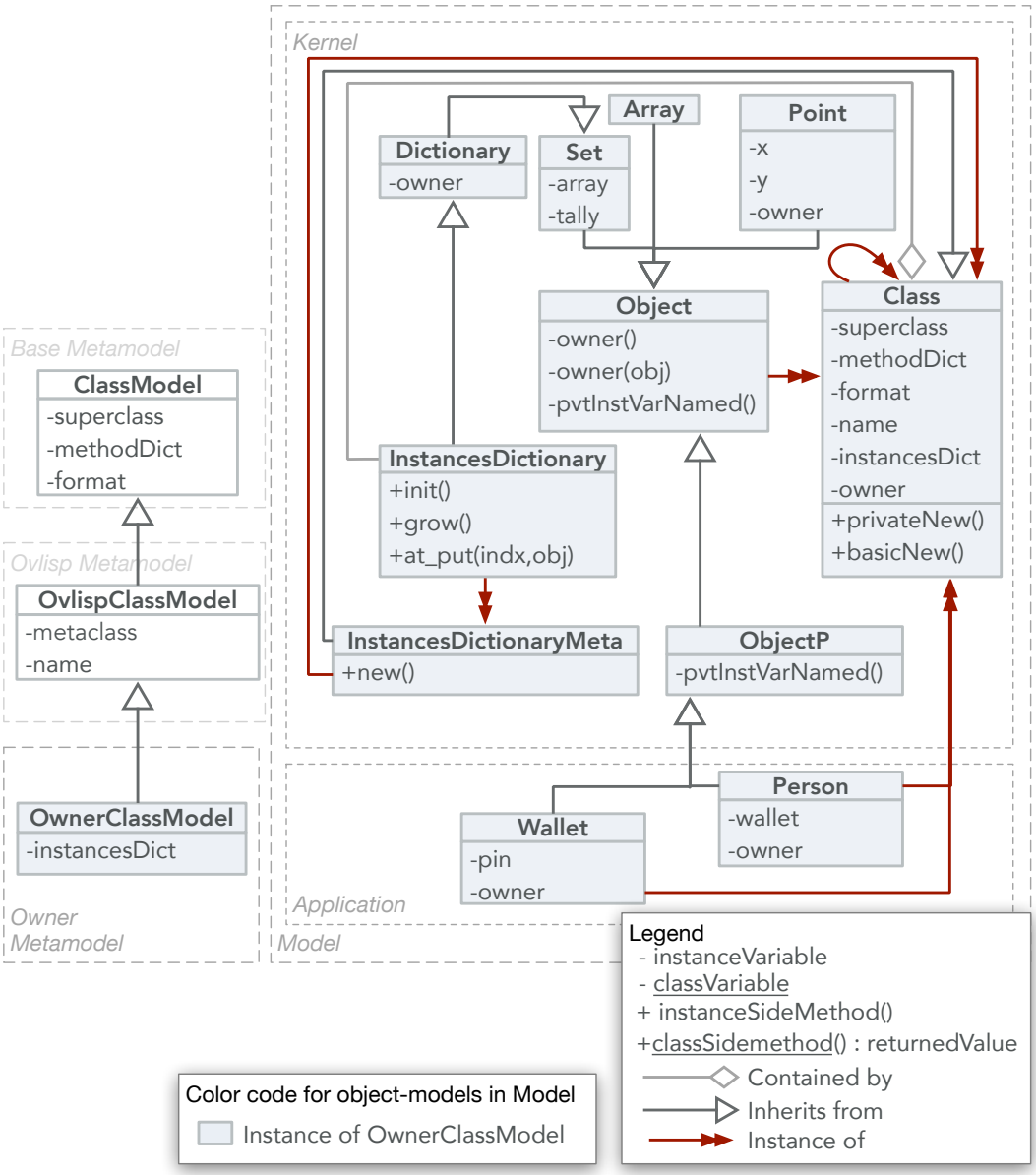


Figure 5.7: $Owner_L$ metamodel and model.

```
OwnershipLClassModel>>inHierarchyAddInstVarOwner
  | setter getter |
  [ self addInstVar: #owner.
    setter := 'owner
      ^ owner'.
    getter := 'owner: anObject
      owner := anObject' ]
  on: ForbiddenAddInstVar
  do:
  [ setter:= 'owner
      ^ self class instancesDict at: self'.
    getter:= 'owner: anObject
      ^ self class instancesDict at: self put: anObject' ].

  self methodModel fromSource: getter).
  self methodModel fromSource: setter).
  self subclasses do: [ :subclass |
    subclass inHierarchyAddInstVarOwner ]
```

Listing 5.12: User defined method to add instance variable owner and accessors.

change the method's source code to signal an exception if the object who sent the constructor message is not the owner of the object who received this message. Otherwise, the private version of the same method is executed. To prevent application classes from using accessing the private versions of reflective methods, we create the class ObjectP that inherits from object and overrides private methods. Application classes inherit from ObjectP.

```
OwnershipLModel>>modifyReflectiveMethods
  | classModel |
  classModel := self classNamed: #Object.
  classModel
    renameMethod: #instVarNamed:
    as: #privateInstVarNamed:.
  classModel
    addLocalMethodFromSource:
  'instVarNamed: aName
    ^ self owner = thisContext sender sender receiver
      ifFalse: [ ForbiddenOperationException signal ]
      ifTrue: [ self privateInstVarAt: aName ]'
```

Modifying instantiation methods. To detect which methods in the language model create instances, we query the model searching for methods using VM primitives associated to instance creation, we implement the code in 5.13. As a result we obtain the next methods: Class » basicNew, and Class » basicNew:. We rename these methods into privateNew and privateNew: respectively. We create new methods basicNew, ba-

sicNew:, which use their private versions to create the new instance and then set their owner.

According to the ownership model, the owner of an object is the object who instantiated it. However, methods basicNew, basicNew: are not directly used. They are used through constructor methods. Therefore, the real owner is the receiver of the method that sends the constructor message. This is reflected in Listing 5.13 line 7. The following code extract shows the implementation of the new method basicNew as an example. This implementation ensures that each time a new instance is created, its owner is automatically set as the object who sent the message that sent the message basicNew.

```
1  owner allMethods
2    select: [ :method | method sendsInstantiationPrimitive ]
3
4  OwnershipLModel>>modifyInstantiationMethods
5    | classObject |
6    classObject := self classNamed: #Object.
7    classObject renameMethod: #basicNew: into: #privateNew.
8    classObject addMethodFromSource:
9    'basicNew
10     ^ self privateNew
11       owner: thisContext sender sender receiver;
12       yourself'
```

Listing 5.13: Finding instantiation method-models.

5.5.4 Kernel Initialization

Infinite recursion during instance creation. Each time a class is installed in the kernel, its instancesDict variable must be initialized containing a new Dictionary. Our first implementation uses a reflective instruction as follows.

```
1  OwnerLClass >> installCustom
2    | dictionary |
3    dictionary := self evaluateCode: 'Dictionary new'.
4    self remote instancesDict: dictionary.
```

However this implementation is incorrect because it produces an infinite call stack problem at the moment instancesDict is created. This occurs because the bootstrapped language is reflective. As dictionary are used to store the relationship between objects and owners, we need to provide an implementation of dictionary that avoids creating

objects using the standard API. Because the standard API creates new dictionaries to hold owners of objects created by the first dictionary producing an infinite recursion.

Creating new dictionaries to store the relationship between objects and owners, would require creating a dictionary to initialize instancesDict of a class Dictionary. If this creation uses the updated constructor with owner support, it would require initializing instancesDict of class Dictionary, hence leading to an infinite recursion.

Solving infinite recursion during instance creation. To solve this problem, we must find methods creating instances, which are used by dictionaries. We must provide an implementation that does not produce this problem, while keeping our restriction policy valid.

We simulate the execution of the following code that creates a new dictionary and adds 1000 values in it. By implementing the hook markIn: we mark methods sending messages basicNew, and basicNew:.

```
1  owner evaluateCode:
2    '| d |
3    d := Dictionary new.
4    1 to: 1000 do: [ :i |
5      d at: i put: i + 1 ]'.
```

The method marking process is implemented in one of *MetaL* hooks, which is executed when the corresponding message-model is accessed during the simulated interpretation of code. Its implementation is presented below.

```
1  " retrieve marked messages "
2  owner allMethods select: [ :method | method properties at: #sendsNew ifAbsent: false ]
3
4  OwnerLMethod >> markIn: currentContext
5      {(self language classNamed: #Class) localMethods at: #basicNew .
6       (self language classNamed: #Class) localMethods at: #basicNew: .
7       (self language classNamed: #Object) localMethods at: #basicCopy}
8         includes: self
9      ifTrue: [ currentContext sender sender methodModel
10           propertyNamed: #sendsNew put: true ].
```

As a result we get: Set»init:, Set » grow, Dictionary»at:put:, and AssociationMeta » key:value: These methods create new instances of Association and Array using messages basicNew and basicNew:. With this information we create a subclass of Dictionary called InstanceDictionary, that overrides the methods at:put, grow, and init: with versions that use privateNew and privateNew: to create new instances. We also create a metaclass for InstanceDictionary, called InstanceDictionaryMeta, which overrides the method new:

to prevent this object from setting an owner.

5.5.5 Discussion

With $Owner_L$ we have implemented an access control policy in a kernel while respecting VM constraints. In this example, we have shown how *MetaL*'s MOP prevents corruption of the model during its construction, using model transformation operations together with validations. Moreover, in Listing 5.12 we show how the language model is manipulated programmatically, while relieving the user from the burden of dealing with VM requirements. The user ignores which classes accept the instance variable owner and relies on the exception signaled by *MetaL* to install the correct accessor method.

One of the challenges generating $Owner_L$ is the infinite recursion problem that arises when using dictionaries that depend, in their turn, on the creation of new dictionaries and so on. We have shown that *MetaL* provides support to solve this problem, limiting the expertise required about the Bootstrap process. Finally, $Owner_L$ shows how *MetaL* prevents failures at run-time, in this case due to infinite instance creation. In consequence, abstraction leaps are limited thanks to tools provided by *MetaL*, such as method marking to perform dynamic execution analysis of guest-language code.

5.6 $Ovlisp_L{}^{dyn}$

$Ovlisp_L{}^{dyn}$ is a extension to $Ovlisp_L$ that implements dynamic temporary variables. This means that the binding for temporaries is dynamic instead of static, allowing reading and writing of temporaries from methods that do not declare them.

5.6.1 Application

To test dynamic temporaries we design the application in Listing 5.14. The method test in TestDyn declares a temporary variable named tmp. This variable is read and written from the methods readTmp and writeTmp: in the class TestExt. None of these methods declares the variable, but they have access to it because they implement dynamic scope for temporary variables.

```
"entry point code"
TestDyn new test

"application definitions"
Object subclass: #TestDyn
  instanceVariableNames: ''

Object subclass: #ExtDyn
  instanceVariableNames: ''

ExtDyn >> readTmp
  ^ tmp

ExtDyn >> writeTmp: value
  tmp := value

TestDyn >> test
  | tmp object |
  object := ExtDyn
  tmp := 'first'
  object readTmp. "<— first"
  object writeTmp: 'second'.
  tmp."<— second"
  object readTmp. "<— second"
```

Listing 5.14: $Ovlisp_L{}^{dyn}$ application

5.6.2 Metamodel

$Ovlisp_L{}^{dyn}$ metamodel and model are presented in Figure 5.8. The most important metamodel classes are OvdynDynClassModel, OvdynMethodModel, and OvdynDynVarModel. They represent classes containing methods with dynamic temporaries, methods with dynamic temporaries, and dynamic temporaries respectively.

Dynamic variables work under the same principles as slots. But, unlike slots, dynamic variables belong to methods instead of classes. A dynamic variable knows its index and name, which are defined at compilation time. If the variable belongs to a method that defines it (*e.g.* tmp in TestDyn » test), then its index is its position in the collection of temporaries declared in the method (in Pharo collections are one based). If the variable belongs to a method that does not define it (*e.g.* tmp in ExtDyn » readTmp), then its index is zero.

5.6.3 Model

When a dynamic method is compiled and a temporary variable node is visited, custom bytecode to read and write the variable is generated as specified in the methods emitValue: and emitStore:. In Listing 5.15 it is shown how these methods generate the custom bytecode. For example, in emitValue: we can see that to read the variable, the message read is sent to the temporary variable itself, which is an instance of the class DynamicVar. Accessing this variable at run-time is possible because during compilation we store the variable in an extra literal. In line 5 the new literal is created. The value of the additional binding is the kernel-object representing the dynamic variable. With

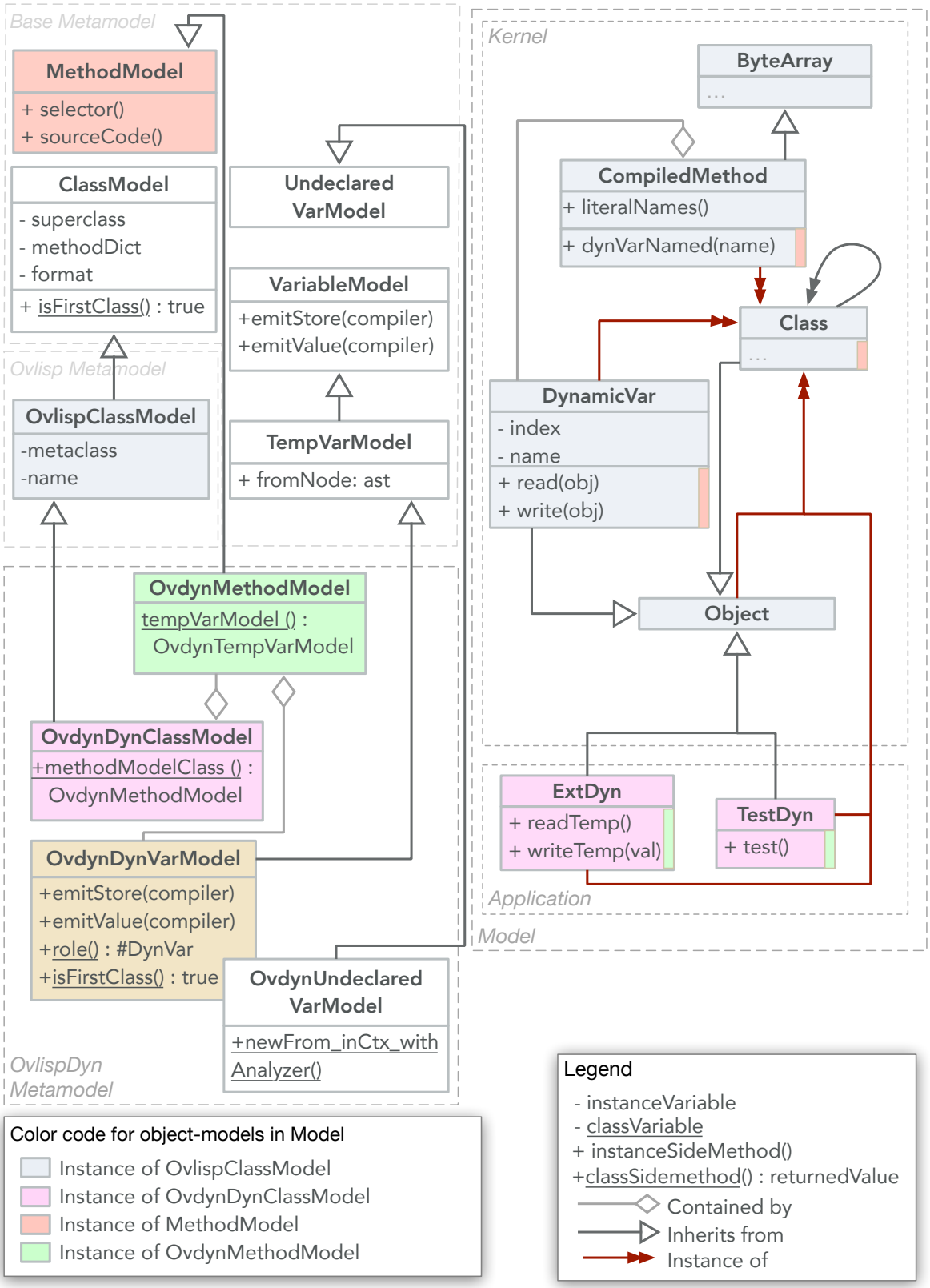


Figure 5.8: $Ovlisp_L{}^{dyn}$ metamodel and model.

the message ensureRemote we ensure that the kernel-object is installed and obtain a mirror pointing to it.

Listing 5.15 shows the methods in DynamicVar used for reading and writing dynamic temporaries. The method read looks for the variable up in the execution contexts before the current one, asking for the variable name to the method associated to each context. Because methods store dynamic variables in an extra literal, they can return the variable when they declare it or return nil when they do not declare it. The implementation for writing is the same as for reading, but one extra argument for the new value is taken into account.

```
1  OvdynDynVarModel >> emitValue: methodBuilder
2    "generate bytecode to call the reflective read
         method of DynamicVar"
3    methodBuilder
4      pushLiteralVariable:
5        (AdditionalBinding
6          key: #dynvar
7          value: self ensureRemote);
8        send: #read
9
10 OvdynDynVarModel >> emitStore: methodBuilder
11   "generate bytecode to call the reflective write
         method of DynamicVar"
12   | tempName |
13   tempName := '0slotTempForStackManipulation'.
14   methodBuilder
15     addTemp: tempName;
16     storeTemp: tempName;
17     popTop;
18     pushLiteralVariable:
19       (AdditionalBinding
20         key: #dynvar
21         value: self ensureRemote);
22     pushTemp: tempName;
23     send: #write:
```

Listing 5.15: $Ovlisp_L{}^{dyn}$ metamodel definition extract

```
1  Object subclass: #DynamicVar
2    instanceVariableNames: 'index name'
3
4  DynamicVar >> read
5    | context var |
6    context := thisContext sender.
7    [ context isNil ] whileFalse: [
8      var := context method dynamicVarNamed: name.
9      (var isNil not) ifTrue: [
10       ^ context tempAt: var index ].
11     context :=  context sender
12   ].
13   ^ nil
14 DynamicVar >> write: anObject
15   | context var |
16   context := thisContext sender.
17   [ context isNil ] whileFalse: [
18     var := context method dynamicVarNamed: name.
19     (var isNil not) ifTrue: [
20       ^ context tempAt: var index put: anObject ].
21     context :=  context sender
22   ].
23   ^ nil
```

Listing 5.16: $Ovlisp_L{}^{dyn}$ model definition extract

Finally, the class OvdynUndeclaredVarModel is used to handle the case when exceptions due to undefined variables arise during the compilation of dynamic methods. The default behavior in *MetaL* is rising an exeption. In this example, when an undefined variable is found in the body of a method, it is replaced by a dynamic variable that is compiled and accesses using the mechanism described before.

5.6.4 Discussion

With $Ovlisp_L{}^{dyn}$ we have generated a kernel where the semantics of temporary variables is modified. Our solution is based in the introduction of reflective calls during compilation time, to make variable evaluation dynamic. This is the same mechanism implemented by slots. We have achieved this in *MetaL* without the need of interacting with compiler code, in great part thanks to the hook for undeclared variables.

The reflective features offered by the VM, such as a reification of the current execution context, were necessary in our implementation. Users who are not familiar with the reflective capabilities offered by the Pharo VM could have trouble finding a good strategy to implement this behavior.

5.7 Experiment by External User

In the Pharo community, the user Erick Stel has used *MetaL* in his project of building Smalltalk environments that use the browser as a view/frontend [Stel 2020]. He uses *MetaL* to generate a minimal Smalltalk image that runs inside the web browser. This image handles all 'view' logic of his application. For running this Smalltalk image in the web browser he uses SqueakJS, a virtual machine implemented in JavaScript that executes in a web browser regular Smalltalk images based on the OpenSmalltalk image format. At run-time, he copies classes and methods from a regular Smalltalk image to the minimal Smalltalk image, to dynamically create the 'view' of the application. So the minimal image is further extended with classes and methods at run-time rather than generation-time. Since both images share the same format, methods are copied based on their bytecode rather than source code. This allows the minimal image to be really minimal: neither parser nor compiler are necessary. This minimal Smalltalk image is 155Kb in size. It is an extension to $Candle_L$, but with some classes/methods removed and others added. With this minimal image it is possible to create WebComponents inside the web browser, handle events and send relevant events back to the Smalltalk image on the 'server'.

5.8 Analysis Of Cognitive Distance In *MetaL*

We rely on the experiments presented in this chapter together with the generation of $Ovlisp_L$ presented in Section 4.2 to analyze cognitive distance in *MetaL*. We use the visual representations introduced in Section 3.4.1 to depict this concept for both Bootstrap and *MetaL*.

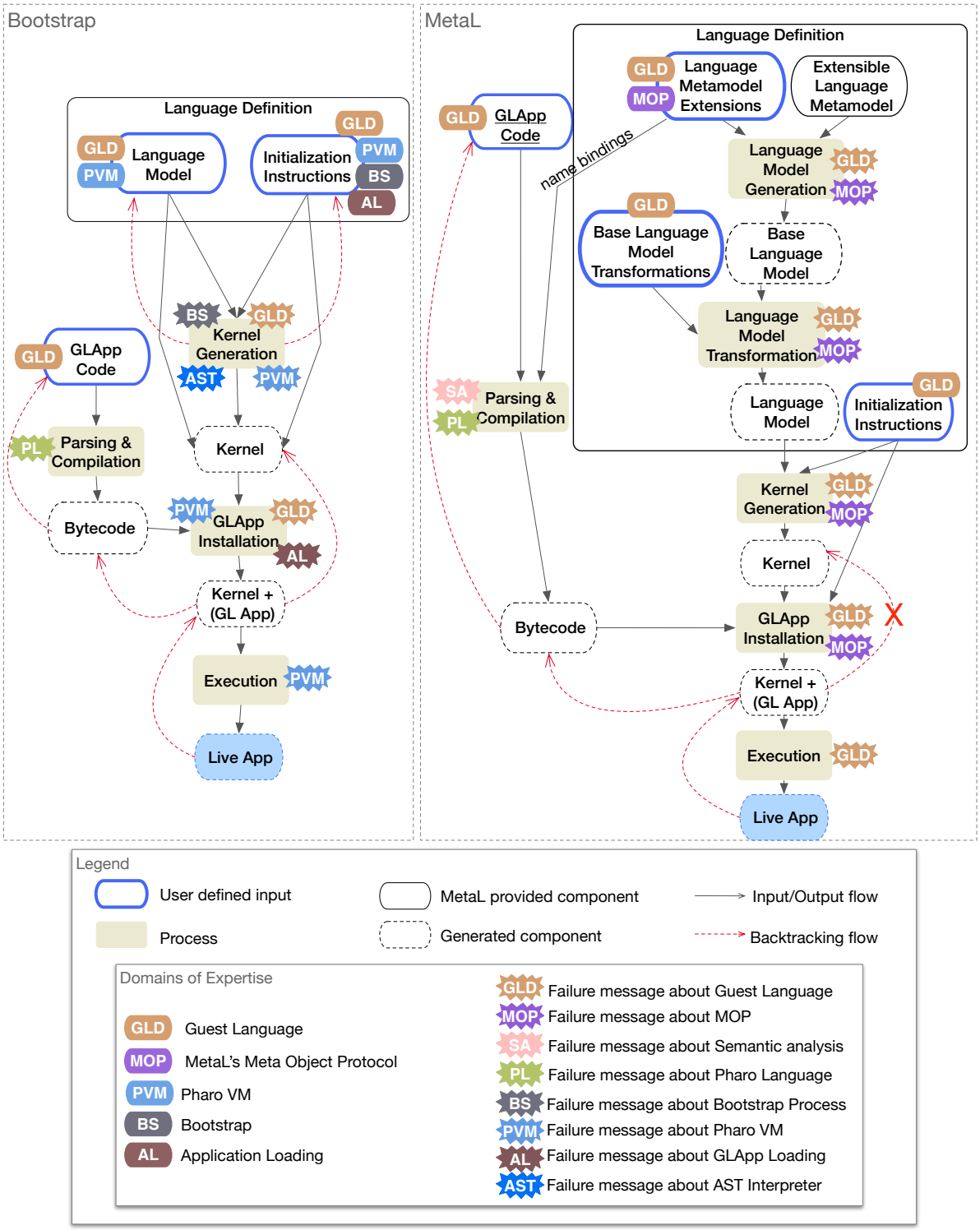


Figure 5.9: Cognitive Distance in Bootstrap.

Figure 5.9 confronts the process diagrams of Bootstrap and *MetaL*. In Bootstrap, multiple domains of expertise are required from developers to achieve both creation of a language definition and debugging failures. This is not the case for *MetaL*, which only requires knowledge about the semantics of the new language and *MetaL*'s MOP. One exception are failures arising during semantic analysis, which require debugging compiler code. These failures occur when users design their own strategies for bytecode compilation (*e.g.* dynamic temporaries in $Ovlisp_L{}^{dyn}$).

The Figure shows that MetaL prevents failures due to VM constraints from arising at run-time, removing the need to debug VM code.

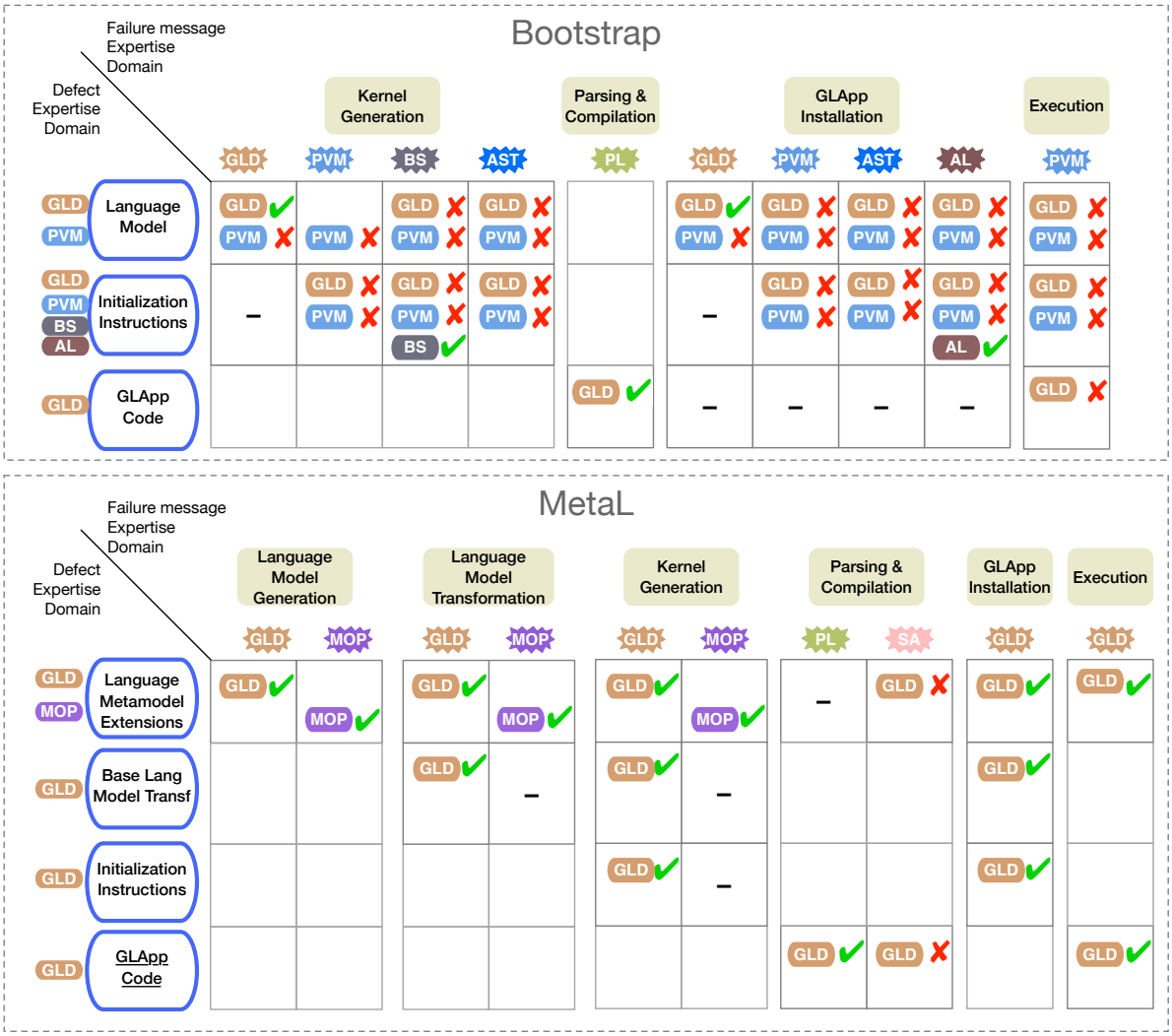


Figure 5.10: Cognitive distance comparison between MetaL and Bootstrap.

Figure 5.10 presents a more detailed visualization for defect backtracking support.

In Bootstrap's diagram the red arrows point at the need for mental defect backtracking. In *MetaL*, mental backtracking of defects due to unfulfilled VM requirements is never necessary. However, defects in the application code can produce failures during execution of the kernel. Most of the times, debugging these failures is possible before generating the kernel, using the guest-language debugger. However, if the application code relies on code that was compiled in a special way (*e.g.* the reflective call to the method `read:` in the class `Slot`), debugging guest-language code is not supported.

5.9 Evaluation of *MetaL*

We have proposed three desirable features for a bootstrap-based LIT in Section 3.5. The following sections analyze the effectiveness of these features in *MetaL* according to the experiments carried on.

5.9.1 Meeting Requirement 1

"Specification abstractions provided by the technique must be succinct, expressive, close to abstractions used for conceptualization of language syntax and semantics."

Languages generated in *MetaL* are reflective and have their semantic expressed in their metaobjects. In *MetaL*, the metamodel is used to specify the structure and behavior of these metaobjects. Defining a metamodel does not require knowledge about any domain other than the new language metaobjects.

We position it in relation to abstractions offered by state of the art LIT in Figure 5.11. *MetaL*'s metamodel has the same level of abstraction as Pharo's metaobjects. We place the metamodel above metaobjects in terms of expressivity because metaobjects modification occurs at run-time, which reduces the range of possible modifications due to metastability issues. *MetaL*, being a bootstrap technique, solves metastability issues, overcoming self-surgery limitations.

MetaL's base reflective model allows simple languages, such as $Ovlisp_L$, to be defined only by the extension of a few classes from the base metamodel and basic model transformations. Thanks to *MetaL*'s MOP, model transformations are expressed in high-level methods that preserve the model compliance with the target VM constraints. This is possible because VM constraints are reified as Roles. Roles are objects in the host encoding low-level information about the VM. In addition to making the MOP aware of VM constraints, roles allow the implementation of a generic bootstrapper, which hides non-customizable kernel initialization instructions and offers only specific extension points for kernel customization. VM constraints remain hidden from the user.

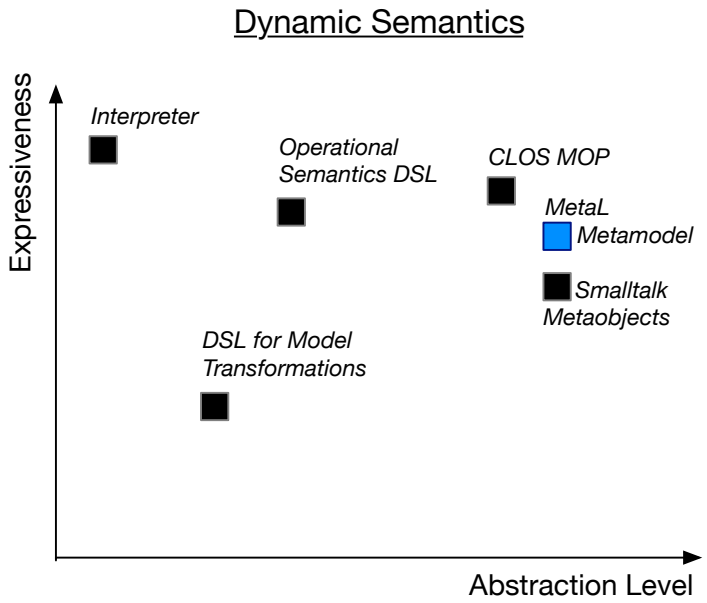


Figure 5.11: Comparison of abstractions for dynamic semantics specification in LIT including *MetaL*

5.9.2 Meeting Requirement 2

"The mapping between abstraction specification and realization must be automatic and efficient."

MetaL is designed to infer the most information from the metamodel as possible. Automatic model completion from metamodel definitions (Section 4.2.4.2) and smart mirrors (Section 4.4.2) apply this concept. *MetaL* is also able to automatically infer which classes must be installed first in the kernel (*e.g.* Class in $Ovlisp_L$, Metaclass and Metaclass class in Pharo) by detecting circular references in the class instantiation chain. *MetaL*'s test of the model and kernel are also automatically extended based in the metamodel definition.

Additionally, smart-mirrors provide high-level means to interact with kernel-objects, inspect them, validate them, and modify them. The effects of user-actions are immediately visible in the kernel, making the development experience more live.

Finally, *MetaL* introduces optimizations to make the development experience more dynamic. Section 3.4 mentions that generating $Ovlisp_L$ in the original Bootstrap implementation takes 32,2 seconds in average. Using the same machine to generate $Ovlisp_L$ in *MetaL* takes 3,6 seconds in average, producing a percentage fall in the kernel generation time of 88,8%. Hardware characteristics are the same as in Section 3.4. Proces-

sor: 2,9GHz quad-core Intel Core i7-7700HQ, cache 6MB. Memory: 16 GB 2133MHz LPDD3. Storage: PCIe SSD 512GB.

5.9.3 Meeting Requirement 3

"In the case the mapping from specification to realization fails, the technique should provide debugging support that answers developer questions formulated at the level of their abstraction specifications."

Generation LIT offer tools for automatic detection of static defects. *MetaL*'s equivalent static checking are the ahead-of-time validations applied to the model. The key difference between Generation LIT and *MetaL* is that the first must deal with the abstraction gap between specifications and realizations. In *MetaL*, most of imperative code written by the user (specification) has the same abstraction level as the generated language (realization). Therefore realization code is debugged at the level of its specification. This feature is characteristic from self-surgery techniques, and it is also present in *MetaL*. In this sense, *MetaL* benefits from the advantages of Self-Surgery Techniques.

Detection of non-static defects is implemented with the help of health tests for the kernel and the basic debugger for guest-language code. Combining both techniques, the debugging process is kept at the same abstraction level as the language definition, *i.e.* model, metamodel, and user-defined kernel initialization instructions.

MetaL's MOP uses Roles to detect invalid model transformation operations at the moment of their execution, Roles are also used to test the model at each step of its construction and to test the kernel during its generation. Failing tests and forbidden model transformations interrupt the bootstrap process displaying an informative error. In this way, *MetaL* reduces the temporal distance between defects and failures. Reducing this temporal gap decreases the difficulty to find defects back in user code [Chis 2015]. Tests in *MetaL* are extensible, users can define their own set of tests according to their needs.

5.9.4 Limitations

During our work with *MetaL* we encountered the limitations presented below. Some of these limitations represent opportunities for improvement and they are used as a motivation to continue with our research.

Semantic Correctness Is Not Ensured Nor Tested. MetaL ensures compatibility with target VM. However, semantic correctness of the generated language is not ensured.

In MetaL, language semantics can only be tested through observation of the result of simulating execution of guest-language. Integrating Unit Tests to be applied to the guest-language before its dump to kernel would help to validate semantic correctness of guest languages.

Advanced Semantic Modifications Require Expertise About The VM Reflective Features. New semantics usually require a combination of the next two activities: Creating/Modifying classes and methods (*e.g.* Prototypes functions and instVars lookup) and taking advantage of reflective capabilities offered by the VM (*e.g.* $Owner_L$ kernel using the pseudo-variable thisContext). In MetaL, even if expertise about the VM implementation is not needed, creating the strategy to implement a specific semantic feature requires familiarity with the reflective capabilities provided by the target VM. This situation is similar to that of self-surgery techniques, where users must know the reflective capabilities of the host language to be able to modify its semantics.

Limited Code Reuse. Specification of semantic features traverses classes, since a single feature is implemented in multiple classes, *e.g.* Slots require the introduction of the class Slot, the addition of an instance variable in Class and respective methods to access slots from Classes. Hence, the code for one concern is scattered among different classes. We believe that overcoming this limitation cannot be done if we continue to stick to the class paradigm. Aspect-oriented programming (AOP) proposes to modularize such cross-cutting concerns into a new abstraction called Aspect. Implementing an aspect-oriented bootstrap is an interesting idea that would rise the abstraction level of the language specification, and would increase code reuse by allowing the definition of semantic features only once, to be re-used in different kernels. We explore this idea in the following chapter.

5.10 Conclusions

This chapter has presented experiments by us and by an external user to validate *MetaL*. For all generated kernels, *MetaL* was able to prevent failures due to VM constraints arising during kernel execution. Therefore, debugging VM code was not necessary. The idea that *MetaL* is easy to use is reinforced by the experiment performed by an external user. The presented experiments foster the idea that Bootstrap-based language implementation technique is possible.

We have obtained a partial answer for RQ1: metamodels are effective abstractions for language specification in that they have the same abstraction level as metaobjects and surpass the latter in their expressiveness. However, we have identified an inherent

limitation about code reuse. The class paradigm hinders the modularity of specifications, producing repetition of code in kernels that share part of their semantics. This limitation motivates the next Chapter, where we explore the application of the aspect-oriented paradigm into Bootstrap.

Chapter 6

Aspect-Oriented Bootstrap: *AspectMetaL*

Contents

In the previous chapter, we have presented our experience using *MetaL*, an implementation of our approach for bootstrapping kernels with a low cognitive load on developers. We have confirmed that kernel generation tasks are kept at a high abstraction level and that the definition of languages other than Pharo-*like* languages is possible. But we have identified one important limitation: code reuse is hindered because the specification for a single semantic feature is scattered among different classes in the metamodel.

In this chapter, we present *AspectMetaL*, an extension to *MetaL*, which allows specification of kernels at the level of semantic features. To achieve this *AspectMetaL* uses

aspect-oriented programming. In particular, *AspectMetaL* is implemented in *PHANtom* [Fabry 2012], an aspect language for Smalltalk. *AspectMetaL* is a proof of concept of AOP's applicability to Bootstrap.

We start this chapter by giving a brief introduction to aspect-oriented programming and *PHANtom* in Section 6.1. This is followed by an overview of *AspectMetaL*'s approach for kernel specification in Section 6.2. To expose the most important features of *AspectMetaL* we demonstrate its use in the generation of an example kernel in Section 6.3. A discussion about *AspectMetaL* is offered in Section 6.4. Finally, our main conclusions about applying AOP to Bootstrap are presented in Section 6.5.

6.1 AOP in a Nutshell

AspectMetaL uses aspect-oriented programming to solve *MetaL*'s limitations to code reuse. This section provides a brief introduction to the most relevant concepts of AOP, particularly applied to *PHANtom*. *PHANtom* [Fabry 2012] is a dynamic aspect language for Smalltalk, chosen by us to implement *AspectMetaL*. A more complete description of *PHANtom* and AOP is offered in Appendix D.

Aspect-Oriented Programming (AOP) is a paradigm that seeks to solve the issue of code for one concern scattered among different classes by proposing a new kind of abstraction: the *aspect* [Kiczales 1997]. Aspects represent application features which are typically scattered across methods and classes. An aspect not only implements the behavior of a concern, but it also handles the moment at which the behavior should be executed. In *PHANtom*, the execution steps of an application correspond to method executions. Aspects are able to identify in the stream of method executions those which are interesting to them. They achieve this with the help of *pointcuts*, which work as queries to select join points. The behavior of the aspect is implemented in *advices*. An aspect can define multiple advices and multiple pointcuts. The aspect links advices with pointcuts such as when the application execution reaches a join point that matches a specific pointcut, the advices linked to the pointcut are executed.

In a nutshell, pointcuts define the when, advices define the what, and aspects use advices together with their pointcuts to define cross-cutting functionalities of an application.

6.2 *AspectMetaL* Overview

AspectMetaL is built on top of *MetaL*. The main difference between *AspectMetaL* and *MetaL* is that the first has a first-class representation of the metamodel. The metamodel is manipulated by aspects. Having a first-class metamodel allows implementing an architecture to organize the aspects in charge of its modification. In *AspectMetaL*, one aspect defines one semantic feature and metamodels are defined by the set of aspects used to transform an existing metamodel. Unlike *MetaL*, users do not need to create new classes to conform the metamodel of the new language. Instead, they select aspects among those provided by *AspectMetaL* or create their own. Aspects take care of cross-cutting concerns for implementation of semantic features because they are able to modify multiple classes in an existing metamodel and even to create new ones if required. These modifications result in the metamodel that is used to build the model and eventually to generate the kernel.

To make our explanations more concrete, let us consider the language $Ovlisp_L$, a class-based language with explicit metaclasses presented in Section 4.2. Having explicit metaclasses means that each class is instance of one metaclass, and that one metaclass can have multiple instances. Therefore, unlike Pharo, there is no parallel hierarchy between classes and metaclasses. In $Ovlisp_L$, Class is the first metaclass and it is instance of itself.

6.2.1 General Process

Listing 6.1 shows the code to define and generate the $Ovlisp_L$ kernel in *AspectMetaL*, and Figure 6.1 illustrates the process corresponding to the execution of this code.

```
1 aspect := SimpleLanguageAspect on: LanguageModel with: ClassModel.
2 metamodel := LanguageMetamodel
3   newFrom: { aspect }
4   withName: 'Ovlisp'
5   withEntryPoint: '...some code...'.
6 metamodel install. "1. aspects modify the base metamodel"
7 metamodel model build. "2. construct the model"
8 metamodel model generateWriteAndExecute. "3. generate & execute the kernel"
9 metamodel uninstall "4. revert the metamodel to its original state"
```

Listing 6.1: $Ovlisp_L$ definition and kernel generation in *AspectMetaL*.

The first line of Listing 6.1 shows the creation of the only aspect used to specify $Ovlisp_L$. This aspect is an instance of the class SimpleLanguageAspect, which is prede-

fined in *AspectMetaL*. The constructor of this class, on:with:, receives as its arguments the two classes that are modified by the aspect.

The second line shows that the metamodel is represented by an instance of LanguageMetamodel. This class is provided by *AspectMetaL*. The constructor method newFrom:withName:withEntryPoint: receives as its first argument the collection of aspects defining the language and as its second and third, the name and the entry point code for the kernel.

When the message install is sent to the metamodel (1), the aspect is deployed, modifying the classes LanguageModel and ClassModel. Besides from changing their structure (*e.g.* adding/removing instance variables), the aspect installs specific behaviors according to its requirements. The method model provides access to the language model, which is an instance of LanguageModel. Hence, all tools to build and modify the model, to debug guest-language code, to generate and execute the kernel (2 & 3), etc are available just as in *MetaL*. Finally, sending the message uninstall reverts the base metamodel to its original state.

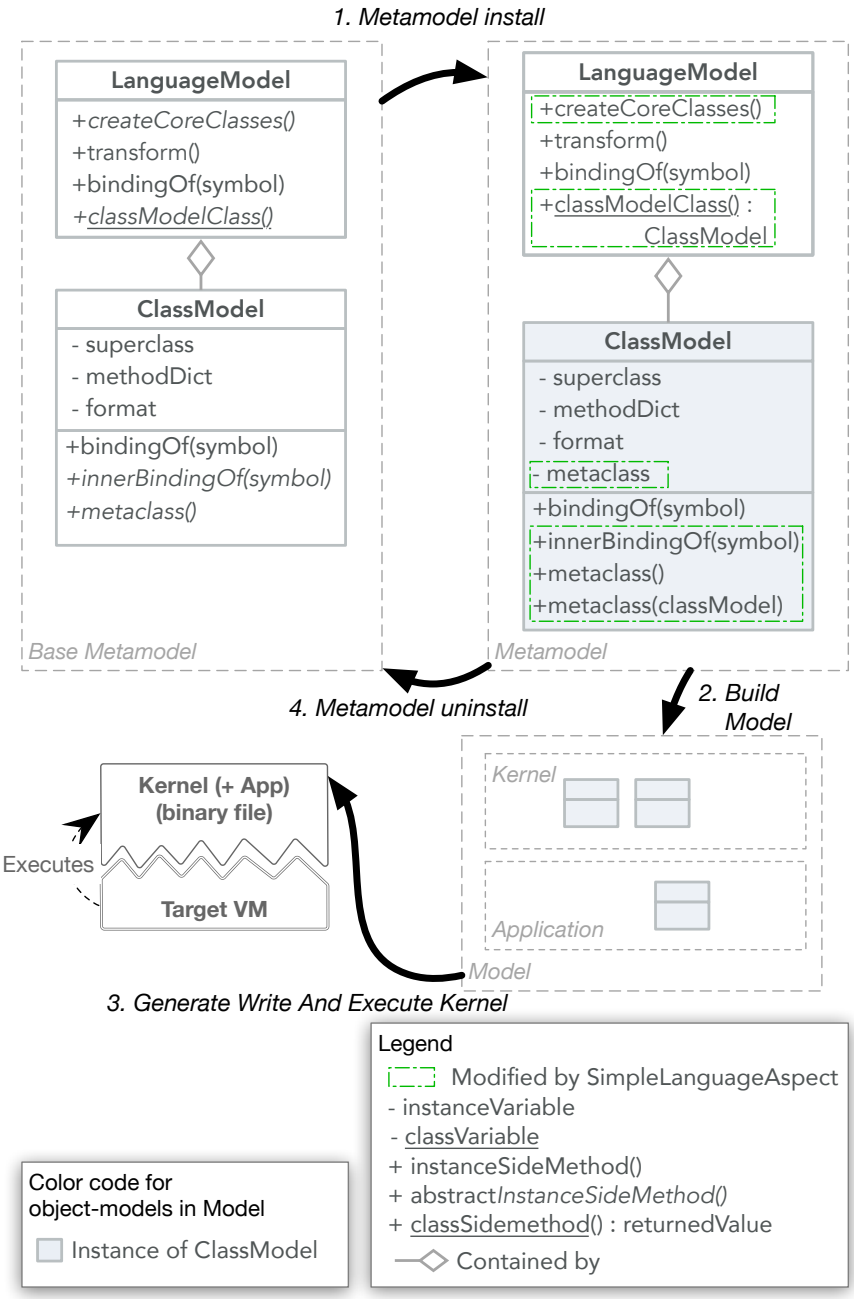


Figure 6.1: *AspectMetaL* process overview. The base metamodel is automatically modified by aspects and used for model construction and kernel generation. Then it is restored to its original state.

6.2.2 *AspectMetaL* Aspects

As illustrated in Figure 6.1, the only aspect used to define $Ovlisp_L$ is an instance of SimpleLanguageAspect. This class adds the instance variable metaclass to the class ClassModel and modifies two of its methods: innerBindingOf: and metaclass. It also modifies methods from the class LanguageModel: createCoreClasses and classModelClass. These modifications alter the model construction and the kernel generation process. Listings 6.2 and 6.3 offer a more complete description of these modifications.

Aspects classification. Up to this time, the aspects we have worked with in *AspectMetaL* there classified in two groups.

- *Field Aspects*: they implement semantic features as fields in classes and/or metaclasses (*e.g.* class vars, slots, class pools, etc).
- *Language Aspects*: associated to semantic features affecting relationships between classes (*e.g.* implicit and explicit metaclasses)

Multiple aspects defining a kernel. A language more complex than $Ovlisp_L$ requires additional aspects to define its metamodel. While SimpleLanguageAspect belongs to the first category, InstVarSlotsAspect belongs to the second one. The latter defines instance variables as first-class objects represented by slots. This aspect is also predefined in *AspectMetaL*. Both aspects are used in the definition of $Ovlisp_L{}^{slot}$, a kernel presented in Section 5.1 where instance variables are first-class objects represented by slots. In *AspectMetaL*, the $Ovlisp_L{}^{slot}$ metamodel definition and kernel generation is as follows.

```
1 metamodel := LanguageMetamodel
2   newFrom: { SimpleLanguageAspect on: LanguageModel with: ClassModel .
3              InstVarSlotsAspect on: ClassModel }
4   withName: 'OvlispSlots'
5   withEntryPoint: '...some code...'.
6 metamodel model build.
7 metamodel model generateWriteAndExecute.
8 metamodel uninstall "uninstall before generating another kernel"
```

InstVarSlotsAspect's constructor on:, takes only one class as an argument. This is that class which declares the new kind of field. *E.g.* in $Ovlisp_L{}^{slot}$ the class ClassModel declares a new instance variable named slots for class-models to store the collection of slots representing their instance variables.

AspectMetaL provides only a small set of predefined aspects. But even if they were numerous, they would not cover the full range of possible semantic features for kernels. Users are able to define their own aspects by subclassing existing ones. In the following section we use an example to continue our presentation of *AspectMetaL*, describing the creation of an aspect defined by the user.

```
ObjectModel subclass: #ClassModel
  instanceVariableNames: 'superclass
  methodDict
  format'
  classVariableNames: ''

ClassModel >> metaclass
  self subclassResponsibility

ClassModel >> innerBindingOf: aSymbol
  self subclassResponsibility

LanguageModel >> classModelClass
  self subclassResponsibility

LanguageModel >> createCoreClasses
  self subclassResponsibility
```

Listing 6.2: $Ovlisp_L$ metamodel before modification

```
ObjectModel subclass: #ClassModel
  instanceVariableNames: 'superclass
  methodDict
  format metaclass'
  classVariableNames: ''

ClassModel >> metaclass
  ^ metaclass

ClassModel >> innerBindingOf: aSymbol
  ^ nil

LanguageModel >> classModelClass
  ^ ClassModel

LanguageModel >> createCoreClasses
  | objectModel classModel |
  objectModel := self basicNewClassWithRole:
      #ProtoObject.
  classModel := self basicNewClassWithRole:
      #Class.

  self addClass: objectModel.
  self addClass: classModel.

  objectModel initialize.
  classModel initialize.
```

Listing 6.3: $Ovlisp_L$ metamodel after modification

6.3 *AspectMetaL* By Example: Generating $Ovlisp_L{}^{scv}$

To proceed with *AspectMetaL* description, we use the definition of a new language, called $Ovlisp_L{}^{scv}$, to present a more in-depth view of aspects, demonstrating how user defined aspects are created in *AspectMetaL*.

6.3.1 Semantic Features

$Ovlisp_L{}^{scv}$ (*i.e.* $Ovlisp_L$ with slots & class variables) is a class-based language with explicit metaclasses (like $Ovlisp_L$ described in Section 6.2.1), first-class instance variables as slots (like $Ovlisp_L{}^{slot}$ described in Section 6.2.2), and class variables. Class variables, like in $Candle_L$ (presented in Section 5.4), are variables declared and defined in a class, and shared by the all instances of that class. The original design of ObjVLisp, as described in [Cointe 1987], states that the class variables of a class are the instance variables of its metaclass. We take a different approach: each class stores in one of its instance variables a dictionary that contains its class variables.

6.3.2 Metamodel Definition

The $Ovlisp_L{}^{scv}$ metamodel takes as input one aspect per each semantic feature: SimpleLanguageAspect for explicit metaclasses, InstVarSlotsAspect for first-class instance variables, and ClassVarsAspect for class variables.

Listing 6.4 presents the metamodel definition together with a simple entry point and application created to show that class variables are shared among different instances of the same class. The class Screen has three class variables named Red, Green, and Blue which are initialized to their corresponding values in the method initializeClassVars. The method displayMacaw in the class Screen shows a tricolored macaw on the screen. The second time the macaw is displayed, the color blue is replaced by yellow.

This code reveals what is expected from the class ClassVarsAspect, showing that the class should not only handle metamodel structural modifications to let classes store their class variables, but it should also support the declaration and initialization of class variables and their addition to specific classes. In the example code, the three class variables for the class Screen are declared in the collection cvCustom and they are given to the aspect cvAspect through the message customFields:. Class variables are initialized in a reflective instruction, which is also given to cvAspect through the message initializationInstructions:.

```
" Class vars declaration "
cvCustom :=
  { #Screen -> { #Red . #Green . #Blue } } .

" Aspects "
cvAspect  :=
  (ClassVarsAspect on: ClassModel)
    customFields: cvCustom;
    initializationInstructions:
        'Screen new initializeClassVars'
    yourself.

slotAspect := InstVarSlotsAspect
    on: ClassModel.

langAspect := SimpleLanguageAspect
    on: LanguageModel
    with: ClassModel.

" Entry point "
code := '| screen |
          screen := Screen new.
          screen displayMacaw.
          Screen new blue: (Color r: 1 g: 1 b: 0).
          screen displayMacaw'
```

```

" Metamodel "
metamodel :=
  LanguageMetamodel
    newFrom: { langAspect .
               slotAspect .
               cvAspect }
    withName: 'OvlispSCV'
    withEntryPoint: code


" Application "
Screen >> displayMacaw
  self displayFeathersIn: Blue .
  self displayWingsIn: Green .
  self displayBodyIn: Red .

Screen >> blue: aColor
  Blue := aColor

" Class vars initialization "
Screen >> initializeClassVars
  Red   := Color r: 1 g: 0 b: 0.
  Green := Color r: 0 g: 1 b: 0
  Blue  := Color r: 0 g: 0 b: 1
```

Listing 6.4: $Ovlisp_L{}^{scv}$ metamodel and application

Figure 6.2 presents the metamodel that we want to obtain for $Ovlisp_L{}^{scv}$. This metamodel is the result of all modifications performed by the three aspects defining the kernel. Modifications are marked placing them inside dotted line rectangles and they are classified according to the aspect that originates them. Each modification category has a different style of rectangle.

6.3.3 Definition Of A New Aspect

AspectMetaL offers an abstract class called FieldAspect for users to define new aspects by subclassing it. We create the class ClassVarsAspect and give an implementation for the inherited abstract methods as shown in Listing 6.5. The metamodel modifications produced by these methods correspond to those in blue dashed line rectangles illustrated in Figure 6.2.

The first definition presented is that of FieldAspect as offered by *AspectMetaL*. This

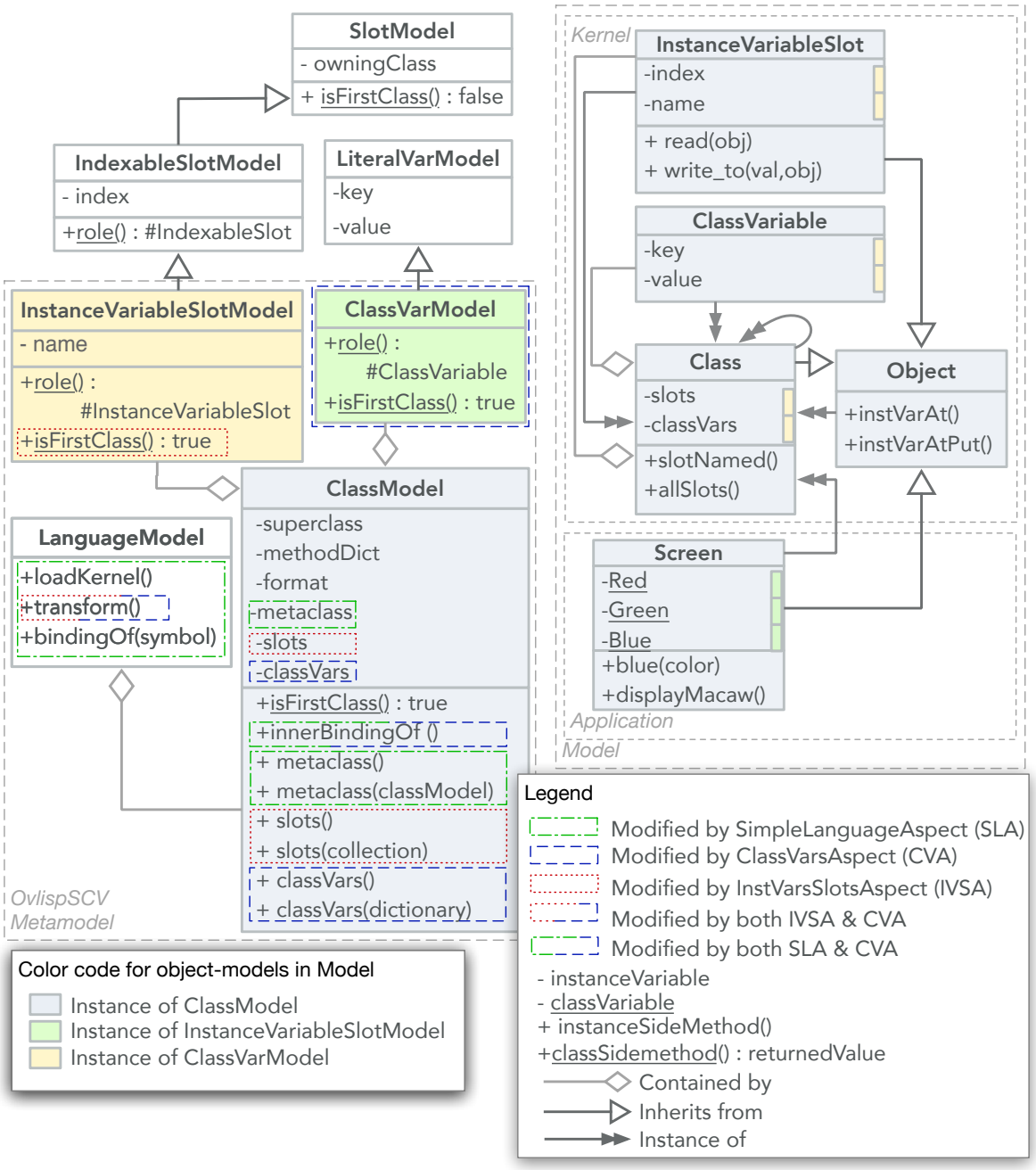


Figure 6.2: $Ovlisp_L{}^{scv}$ metamodel after aspects modifications and model

class declares three instance variables:

- classModelClass to store the class of class-models which are capable of having class variables. This is ClassModel in $Ovlisp_L{}^{scv}$.

- fieldModelClass to store the class of object-models representing class variables. This is ClassVarsModel in $Ovlisp_L{}^{scv}$.
- customValues to store the collection of associations declaring class variables for class-models. Set up in Listing 6.4, line 8.

MetaL's base metamodel does not include a class representing class variables. Therefore, ClassVarsAspect is responsible for its creation and subsequent elimination. This is done in the methods setUpNewClasses and tearDownNewClasses, which are executed when the aspect is deployed and undeployed.

```
1  " Abstract aspect provided by AspectMetal "
2  PhAspect subclass: #FieldAspect
3    instanceVariableNames: 'classModelClass
         fieldModelClass customValues'
4
5  FieldAspect subclass: #ClassVarsAspect
6    instanceVariableNames: ''
7
8  ClassVarsAspect >> setUpNewClasses
9    fieldModelClass := self class
10     installClass: #ClassVarModel
11     superclass: LiteralVarModel  "<-from base
         metamodel "
12
13 ClassVarsAspect >> tearDownNewClasses
14   self class uninstallClass: fieldModelClass
15
16 ClassVarsAspect >> fieldName
17   ^ #classVars
18
19 ClassVarsAspect >> fieldModelClassRole
20   ^ #ClassVariable
21
22 ClassVarsAspect >> fieldModelClassIsFirstClass
23   ^ true
24
25 ClassVarsAspect >> fieldValueFor: aClassModel
26   ^ Dictionary new  " Default value for field "
27
28 ClassVarsAspect >>
29 fieldValueFor: aClassModel
30 from: aValue " Custom value for field "
31   self assert: aValue isCollection.
32   " from aValue create class-var-model dict "
33   ^ (aValue
34       collect: [ :each | fieldModelClass
35                   named: each
36                   parent: aClassModel ]) asDictionary
37
38 ClassVarsAspect >>
39 transformLanguage: aLanguageModel
40   | cvModel |
41   cvModel := aLanguageModel classWithRole:
         #ClassVariable.
42   cvModel removeLocalMethod:  (cvModel
         localMethodNamed: #classVars:)
43
44 ClassVarsAspect >>
45 innerBindingOf: aSymbol in: aClassModel
46   ^ aClassModel classVars bindingOf: aSymbol
```

Listing 6.5: User defined ClassVarAspect in *AspectMetaL*.

The method fieldName returns the name of the instance variable to be added in classModelClass (*i.e.* in ClassModel). We name it classVars. The method fieldModelClassRole determines the value to be returned by the method role in ClassVarModel class. In our example the role is named 'ClassVariable'. We also implement the method fieldModelClassIsFirstClass to return true. This causes *AspectMetaL* to modify the method isFirstClass in ClassVarModel class making it return true. As explained

in Section 5.1.2), when this method returns true, a class-model mapping the class fieldModelClass (*i.e.* ClassVarModel) is automatically created and added to the language model. This new class-model is named after the role of the metamodel class it maps, *i.e.* 'ClassVariable'.

The method fieldValueFor: returns the default value of the instance variable classVars for class-models with no custom values associated. This value is an empty dictionary. The method fieldValueFor:from transforms the collection received as its second argument into a dictionary of object-models, each representing a class variable. This value is set in Listing 6.4 line 8. These object-models are instances of ClassVarModel and in the model section of Figure 6.2 are those variables with a solid green vertical rectangle to their right in the class-model Screen.

Class-models mapping metamodel classes (*i.e.* Class, InstanceVariableSlot, and ClassVariable) have accessor methods for their instance variables automatically generated by *MetaL*. Late modifications to the model are implemented in the hook transform in LanguageModel as explained in Section 4.2.4.3. Adding custom behaviors to this method from an aspect is done by implementing them in the method transformLanguage:. *AspectMetaL* sets up the aspects to execute custom behaviors at the right time.

Finally, the method innerBindingOf:in: implements behaviors to be added to the method innerBindingOf: in ClassModel. This method is fundamental for the lookup of variables during bytecode compilation and AST interpretation (which is necessary to evaluate reflective instructions). A complete explanation of this method was offered in Section 5.3.3. Our implementation looks for the variable name in the dictionary of instance variables. This is done by sending the message bindingOf: to the dictionary containing class variables. If the variable is not found, this method returns nil.

6.3.4 Aspect Reuse

In the previous sections we have described the implementation of an aspect that implements class variables and we used it to generate $Ovlisp_L{}^{scv}$. However, this aspect is not specific for this kernel. We can reuse ClassVarsAspect each time we want to generate a new language with, among other features, class variables. Unless the metamodel of the language defines extra representations of class-models, ClassVarsAspect can be directly used to generate different kernels. Otherwise, the aspect can be extended.

Reusing an existing aspect. The simplest example is $Ovlisp_L{}^{cv}$, a language with the same semantics as $Ovlisp_L{}^{scv}$ but without first-class instance variables as slots. $Ovlisp_L{}^{cv}$ metamodel definition is as follows[1].

[1]Declaration and initialization of custom class variables were removed for simplicity

```
cvAspect   := ClassVarsAspect on: ClassModel.
langAspect := SimpleLanguageAspect on: LanguageModel with: ClassModel.

metamodel := LanguageMetamodel
    newFrom: { langAspect . cvAspect }
    withName: 'OvlispCV'
    withEntryPoint: '...some code...'
```

This code works because the implementation of ClassVarsAspect is completely decoupled from InstVarSlotsAspect, as well as from any other field aspect. This is not exclusive of ClassVarsAspect. Field aspects are independent from each other, unless they extend the behavior of an existing aspect.

Extending aspects. In some languages the implementation of class variables provided by ClassVarsAspect is not enough and needs to be extended. For example, implementing class variables in $Candle_L$ (described in Section 5.4) requires modifying two metamodel classes: CandleClassModel and CandleMetaclassModel. While ClassVarsAspect is able to modify CandleClassModel accordingly to class variable semantics in $Candle_L$, modifications required by Metaclass are out of ClassVarAspect's scope. We recognize two ways to extend[2] aspects.

Create complementary aspect(s) for the original aspect. One alternative is to create a new aspect, that we call ClassVars1Aspect, to modify CandleMetaclassModel. Implementing class variables in $Candle_L$ would need two aspects, as presented below.

```
cvAspect  := ClassVarsAspect on: CandleClassModel.
cv1Aspect := ClassVars1Aspect on: CandleMetaclassModel.
```

Subclass the original aspect. Another alternative is to create ClassVars1Aspect as a subclass of ClassVarsAspect, in which case the previous problem does not exist. Implementing class variables in $Candle_L$ would need only one aspect that takes two classes as its arguments, as presented below.

```
cv1Aspect := ClassVars1Aspect on: CandleClassModel on: CandleMetaclassModel.
```

[2]We use the term "extend" in the colloquial sense, we are not necessarily referring to subclassing

6.3.5 Deployment Ordering

An important issue when working with multiple aspects is that their order of execution can modify the resulting behavior. This problem is also present in our example kernel $Ovlisp_L{}^{scv}$, where both SimpleLanguageAspect and ClassVarAspect specify different behaviors for the method innerBindingOf: in ClassModel.

While the behavior from SimpleLanguageAspect is returning nil (see Listing 6.3), the one from ClassVarAspect searches for the name of the variable in the dictionary of class variables. If SimpleLanguageAspect is executed before ClassVarAspect, the method innerBindingOf: always returns nil.

The default aspect ordering for execution in *AspectMetaL* is defined by the order of aspects in the collection given as an argument to the metamodel constructor method. But this mechanism is not enough when aspects have more than one conflicting method. For example, in the method innerBindingOf: we want to execute ClassVarAspect before SimpleLanguageAspect, but in the method transform of the class LanguageModel we may want the opposite.

AspectMetaL takes advantage of the mechanisms for behavior execution ordering offered by *PHANtom* to allow users define the ordering of aspects when modifying the same method. The code extract below shows how this is done. The messages pcInnerBindingOf and pcTransformLanguage are sent to cvAspect but they could have been sent to langAspect or slotAspect with the same results. In each case the returned value is the *PHANtom* pointcut associated to the message innerBindingOf: in ClassModel or transform in LanguageModel accordingly. More about *PHANtom* is presented in Appendix D.

```
1  " Aspects "
2  cvAspect  :=
3    (ClassVarsAspect on: ClassModel)
4      customFields: cvCustom;
5      initializationInstructions:
6          'Screen new initializeClassVars'
7      yourself.
8
9  slotAspect := InstVarSlotsAspect
10     on: ClassModel.
11
12 langAspect := SimpleLanguageAspect
13     on: LanguageModel
14     with: ClassModel.
15
16 cvAspect pcInnerBindingOf precedence: #('ClassVarAspect' 'SimpleLanguageAspect').
17 cvAspect pcTransformLanguage precedence: #('SimpleLanguageAspect' 'ClassVarAspect' 'InstVarSlotsAspect').
```

6.4 Discussion

Our analysis of applying AOP to Bootstrap provides some answers and also raises several questions that require further experiments to be answered.

6.4.1 Abstractions In *AspectMetaL*

MetaL's aspects are as expressive as the metamodel, as they are able to represent the same range of semantic features. But their abstraction level is much higher than that of the metamodel as depicted in Figure 6.3.

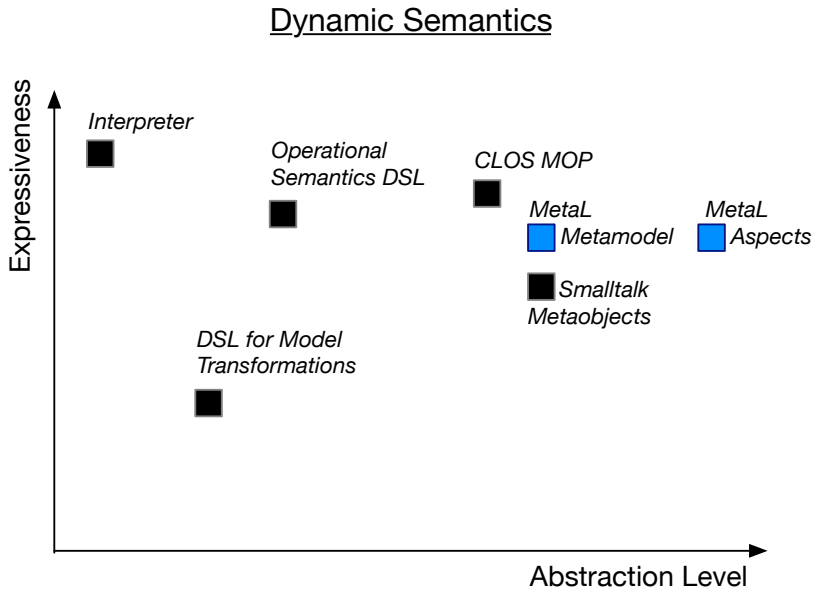


Figure 6.3: Comparison of abstractions for dynamic semantics specification in LIT including *AspectMetaL*

6.4.2 Code Reuse In *AspectMetaL*

We discuss aspect reuse compared to metamodel reuse, considering also aspect extensions.

***AspectMetaL* improves code reuse over *MetaL*.** According to our experience with *AspectMetaL* we believe that *AspectMetaL* increases code reuse compared to *MetaL*.

To make this idea concrete, consider the languages $Ovlisp_L{}^{scv}$ and $Ovlisp_L{}^{cv}$ discussed in Section 6.3.4. Passing from $Ovlisp_L{}^{scv}$ to $Ovlisp_L{}^{cv}$ in *MetaL* is done by subclassing and implementing again ClassModel and LiteralVariable. This is necessary because the subclass of ClassModel in $Ovlisp_L{}^{scv}$ contains a partial implementation of slots. For this reason, we cannot reuse $Ovlisp_L{}^{scv}$ implementation of class variables. By contrast, $Ovlisp_L{}^{cv}$ definition in *AspectMetaL* reuses the full implementation of class variables created for $Ovlisp_L{}^{scv}$.

As a result, the combination of existing aspects provides a very high abstraction level for language specification based on semantic features. However, some aspect combinations are not possible. In these cases, extending an aspect is necessary. Below we discuss the two methods for aspect extension proposed in Section 6.3.4.

Problems of complementary aspect for aspect extension. The problem with this approach is that the complementary aspect depends on the original one. In particular, the class ClassVarModel is created by ClassVarsAspect but ClassVars1Aspect depends on it to create class-var-models. This situation brings us the following questions: What are the different dependency relationships between aspects in the context of Bootstrap? Metamodels describe relationships between class-models: containment, instance of, superclass, etc. Are there equivalent relationships arising among aspects?

Problems of aspect subclassing for aspect extension. The problem with this approach is that it makes aspects grow in complexity. The new class would declare a new instance variable to save a reference to the extra metamodel class. But what would happen if we wanted to implement $Candle_L$ with multiple inheritance and class variables? The implementation of multiple inheritance, as described in Section 5.2, requires the creation of a metamodel class to represent class-models which provide multiple inheritance. In the example from Section 5.2 this extra class is named ObjVLispMultiClassModel. $Candle_L$ with multiple inheritance would have at least three metamodel classes representing class-models. Should we subclass ClassVars1Aspect to define a new aspect whose constructor takes three arguments? At which point should we stop making aspects grow? Should we give each modified class as an argument, or should we use dependency relationships among metamodel classes to deduce other arguments from only one? (*e.g.* the method classModelMetaclassClass in CandleLanguageModel returns CandleMetaclassModel).

6.4.3 Limitations

We have identified two main limitations of applying AOP to Bootstrap, as presented below.

***AspectMetaL* and *MetaL* cannot share the same metamodel.** In *MetaL*, users create subclasses of metamodel classes. The new classes declare custom instance variables, *e.g.* a subclass of ClassModel declares an instance variable named slots. As a result, *AspectMetaL* cannot transform ClassModel to declare the instance variable slots, because one of its subclasses declares it already. Given this situation, *AspectMetaL* and *MetaL* cannot share the same metamodel. We suggest two possible solutions for this problem. The first alternative is to duplicate the base metamodel. The second is to modify *AspectMetaL*, such as it does not take an existing metamodel as its starting point. Instead of using aspects to modify an existing metamodel, a set of aspects would define the metamodel from scratch. Exploring proper abstractions to represent the metamodel modularly and a protocol to make the metamodel parts interact and integrate to each other is necessary.

Efficiency and debugging tools. *PHANtom* implementation uses method-wrappers to execute behaviors defined by aspects during the execution of the bootstrap process. As a result, the call stack provided by the host debugger is polluted with several messages unrelated to bootstrap, but sent by *PHANtom*. This causes debugging tasks to become difficult, because stepping through the bootstrap execution cannot be done efficiently. On the other hand, *PHANtom* efficiency is not optimal yet, because the generation of join points and invocation of advices is based on meta-level operations. Both of these problems are recognized in [Fabry 2012], where they mention that these problems arise due to implementation concerns. They propose solutions, such as creating a special syntax and developing tooling support for *PHANtom*. However, these problems are not specific to *PHANtom*. Computational overhead [Maes 1988] and the unclear separation of base and meta-level code [Bracha 2004] are well known problems that arise when controlling the runtime behavior of reflective facilities. Papulias et al. [Papoulias 2017] propose a set of meta-level control dimensions that they call the *reflectogram*. They argue that reifying the reflectogram is useful to control the meta-level execution, providing proper control of the underlying language semantics, *i.e. PHANtom* in our case.

6.5 Conclusions

This chapter explored the application of AOP to Bootstrap as a means to solve the code reuse limitation found in *MetaL*. To do this we have developed an aspect-oriented layer on top of *MetaL* that allows language specification at the level of semantic features. These features are represented by aspects, which apply modifications to an existing metamodel according to the semantic features they implement.

We have performed brief experiments to test this approach. Based on our results, we believe that code reuse is improved, because aspects are weakly coupled between each other. On the contrary, user defined metamodel classes, as in *MetaL*, are tightly coupled to other classes, hindering code reuse. This experience has given us the insight to pose questions about abstractions to represent semantics, aspects, and their effects to the metamodel. However, to find their their answer we need to do further experiments.

Conclusions

Chapter 7

Contents

In this chapter we summarize the contributions of this dissertation and present our future work. We start presenting our two main contributions: *MetaL* and *AspectMetaL*, to then answer our research questions and evaluate our hypothesis. We close this dissertation with the presentation of our future work.

7.1 Contributions

This dissertation is a contribution to the fields of programming language bootstrap and programming language engineering. In particular, we studied the design of a language implementation technique based on bootstrap of language kernels. To the best of our knowledge we are the first to explore this possibility.

Our technique focuses on cognitive distance reduction for the final user. Combining different tools and techniques we improve the effectiveness of provided abstractions and reduce the efforts to generate kernels from their specifications.

To achieve effective abstractions we reify the requirements of the target virtual machine into high-level representations that are used to ensure the structural soundness of kernels, guaranteeing compatibility with the target VM. To reduce the efforts to produce the kernel, we provide debugging tools for specifications that take advantage from the benefits of reflective systems. When specifications are written in the same language that is being generated, the abstraction gap between specifications and realizations (*i.e.* implementation) is nonexistent.

The two main contributions of this work are *MetaL* and *AspectMetaL*. With *MetaL* we studied the application of concepts from self-surgery to bootstrap to achieve cognitive distance reduction. We also explored the boundaries for the range of different semantics supported by the technique and target VM. With *AspectMetaL* we explored the integration of the aspect-oriented programming paradigm into bootstrap improving code reuse and easing abstraction selection from users.

We proceed to summarize the main characteristics of *MetaL* and *AspectMetaL*, and then we answer the research questions and consider the hypotheses proposed in Section 1.4 and Section 1.5.

7.1.1 *MetaL*

MetaL is a framework for bootstrapping kernels that focuses on cognitive distance reduction [Hernández Phillips 2021]. With *MetaL* we show that it is possible to benefit from the reflective capabilities of the guest-language to make its development style close to self-surgery. In particular, abstraction gaps were effectively reduced and external code debugging was avoided in great part thanks to reification of VM constraints, which allows having a MOP that ensures model correctness by construction and also forcing early manifestation of failures through automatic validations [Hernández Phillips 2019, Polito 2021].

The experience performed by the external user is a good sign of *MetaL*'s effectiveness in reducing the cognitive distance. Even if his sole experience is not enough evidence to prove the success of our approach, we consider it relevant, considering that Pharo's bootstrap has been available for several years but its complexity often discourages people from start experimenting with it.

With *MetaL*, besides confirming the importance of effective abstractions and debugging support to decrease cognitive distance, we remark the big impact that meaningful and immediate feedback has on the technique. Performing kernel bootstraps while able to see the kernel and interact with it in a "live" way, marks a turning point in the process.

7.1.2 *AspectMetaL*

AspectMetaL is an aspect-oriented layer on top of *MetaL* that raises the abstraction level of specifications, offering kernel specification at the level of semantic features. *AspectMetaL* allows users to combine these features while never modifying the metamodel, improving code reuse.

AspectMetaL provides evidence that AOP is useful for implementing bootstraps of language kernels because its capacity to automatically extend the metamodel allows to reify the metamodels and their modifications.

Thanks to AOP, metamodel modifications are expressed modularly, which not only improves code reuse but also makes the implementation of each semantic feature transparent by isolating it in a single aspect. This is the opposite from what we have in

MetaL, where it is hard to reunite and understand the implementation of a single semantic feature because its code is spread in several classes.

7.1.3 Research Questions & Hypotheses

We implemented *MetaL* and *AspectMetaL* to answer research questions presented in Section 1.4 and validate hypotheses presented in Section 1.5.

Answering RQ1: What abstractions for language specification reduce the cognitive distance in a bootstrap-based LIT?

The main abstraction provided by *MetaL* is the metamodel: a meta-representation of metaobjects. Kernels generated by us and presented in Chapter 5 were all specified using a metamodel, showing its flexibility. In our experience generating these kernels, we never needed to become aware of VM implementation details. Moreover, we were able to generate $Owner_L$ bypassing one VM constraint by dynamically capturing an exception signaled by *MetaL*, relieving us from the burden of knowing which classes signaled the exception.

During our experience generating kernels in *MetaL*, abstractions to hide VM constraints and the bootstrap process have shown flexible and capable of adapting accordingly to the user-defined metamodel. For example, most of the generated kernels introduced modifications to Class (*e.g.* new instance variables), even though this class-model has an associated role. Its customization is possible because the role #Class allows extensions, but this is not always the case. Roles such as Array, which do not accept changes and signal an exception when an operation tries to change it.

We acknowledge that generating seven kernels plus one external experiment to validate *MetaL* is not enough to ensure the framework's generality. It is possible that we are missing to test semantic features whose specification in the metamodel is not easy to implement. Nevertheless, we claim that Bootstrap cognitive distance is reduced when the same kernel is generated in Bootstrap and *MetaL*.

In *AspectMetaL*, aspects are the main abstraction for kernel specification. Each aspect represents a single semantic feature. The specification of a kernel in *AspectMetaL* is declarative code that states which aspects compose the kernel. The abstraction level of kernel specifications in comparison to metamodels drastically rises in *AspectMetaL*. Experiments presented in Chapter 6 show how aspects are fully reused in combination with other aspects to define new kernels. This cannot be done when using the metamodel.

Our answer to RQ1 recognizes that both *MetaL* and *AspectMetaL* abstractions re-

duce cognitive distance, and both have their own limitations. We believe that there is not a single answer and that the chosen solution depends on the application, on the target users, and the context. For example, *AspectMetaL* would be effective to introduce students to Bootstrap, while *MetaL* would be more appropriate for researchers prototyping new languages.

Answering RQ2: What mechanisms support the automatic mapping from specification to realization in a bootstrap-based LIT?

In *MetaL*, mapping from specifications to realizations is implemented in the generic bootstrap process. In the case this process fails, *MetaL* provides debugging tools that intend to emulate the those offered by self-surgery techniques. These debugging tools focus on: first, early manifestation of corruption to shorten the distance between defects and failures; second, on closing the abstraction gap between specifications and realizations; and third, on immediate feedback for the user through high-level mirrors to inspect and manipulate kernel-objects.

To evaluate the previous strategy we rely on our experience generating kernels presented in Chapter 5 together with the example kernel $Ovlisp_L$ presented in Chapter 4 Model corruption was always detected before kernel generation starts. Kernel corruption was always detected during kernel generation, stopping the process. Debugging VM code was never needed.

The debugger for reflective code together with smart-mirrors have shown effective in closing the abstraction gap between specifications (reflective instruction) and realizations (kernel-object targeted by smart-mirror). From our experience we recognize that the previous elements are among the most useful to generate kernels.

We acknowledge that our own experience with the previous debugging tools is not enough to validate their effectivity in a general context. However, we state that they are effective to support the generation of kernels presented in this dissertation.

For the moment we provide a limited answer to RQ2: combining early detection of corruption through automatic tests, a debugger for reflective code, and smart-mirrors supports automatic mapping from specification to realization for the example kernels presented in this work.

7.2 Future Work

Improve Kernel Sources Versioning. Currently we use the metamodel. Model transformations are either stored in methods in the metamodel or the resulting model is saved to disk in Tonel format. While the first method does not escalate well with the number of model transformations, the second requires from the user to manage code versioning manually because there is no way to do it from the host. *MetaL* is missing tools for versioning the source code of kernels.

Kernel as a First-Class Object. Such as classes create objects, "meta-kernels" would create kernels. Gemerating that interact with each other could be used in interesting scenarios, such as decentralized computation and IoT. If kernels are first-class then they would be able to define their own protocols for communication with other kernels. Finding suitable abstractions to model kernels and their source code is necessary.

Pharo bootstrap. Bootstrapping Pharo in *MetaL* would improve its efficiency. It would also help developers to make Pharo's kernel smaller, because model and kernel health are monitored, and debugging tools are available. A smaller kernel means better maintainability and thus evolution support for the language.

Tests to check semantic correctness of guest-language. *MetaL* ensures compatibility with target VM, however, semantic correctness of the generated language is not ensured. In *MetaL*, language semantics can only be tested through observation of the result of simulating execution of guest-language. Integrating Unit Tests to be applied to the guest-language before its dump to kernel would help to assure semantic correctness of produced kernels.

***PHANtom* Debugging Support.** *AspectMetaL* current debugging support is not enough, and all its benefits risk being ignored if its debugging support does not improve. Implementing proper debugging for *PHANtom*, or for any other solution that relies on reflective functionalities (such as reflexivity and the reflectogram) depends on solving the meta-execution control problem.

Associated Publications

Chapter A

A.1 Journals

Carolina Hernández Phillips, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, Stéphane Ducasse, Pablo Tesone
Language Runtime Generation: Reducing the Cognitive Distance between Defects and Failures. Science of Computer Programming, 2021 **(to be submitted)**.

A.2 Conferences

Guillermo Polito, Pablo Tesone, Stéphane Ducasse, Luc Fabresse, Théo Rogliano, Pierre Misse-Chanabier, and Carolina Hernández Phillips
Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8 Managed Programming Languages (MPLR) 2021.

A.3 Workshops

Carolina Hernández Phillips, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, Stéphane Ducasse, Pablo Tesone
Challenges Bootstrapping Reflective Kernels. IWST - International Workshop on Smalltalk Technology, Colocated within the 27th International Smalltalk Conference - 2019, 2019.

A.4 Vulgarization

Carolina Hernández Phillips
MetaL : un framework pour bootstrapper de petites images Pharo. Programmez! Le magazine des développeurs. 11/12 2020 Numero 243.

MetaL MOP

ObjectModel class	
named: aString parent: aLanguageModel	constructor
role	returns the role of the metamodel class (*e.g.* #Class for ClassModel)
isFirstClass	returns true if the metamodel class should map a class-model in the model
ObjectModel	
test	tests this model-object according to the VM constraints
install	generates the remote kernel-object in the kernel
ensureRemote	installs the remote kernel-object if not installed it, and returns it
remote	returns a mirror to the corresponding kernel-object
role	returns the role of the receiver
role: aRole	sets the role of the receiver
markIn: aContext	hook containing action to be executed when the object-model is accessed during interpretation of guest-code
environment	returns the receiver's environment scope for compilation
parent	returns the object-model that is the parent of the receiver
language	returns the language-model to which the receiver ultimately belongs
debugCode: aString	opens the guest-language debugger to debug the code received as argument
evaluateCode: aString	evaluates the guest-language code received as argument and returns a mirror pointing to the resulting kernel-object

Table B.1: Core *MetaL* MOP Methods in ObjectModel.

LanguageModel class	
classModelClass	returns the class in the host system used to instantiate class-models
newWithName: name withEntryPoint: someCode	constructor, returns an empty language model
LanguageModel	
build	builds the basic model and applies user-defined transformations
transform	user-defined transformations object-models in the language
createCoreClasses	user-defined initialization for the first object-models in the language
classNamed: aString	returns the class in the language with that name
classWithRole: aString	returns the class in the language with that role
newClassNamed: aString	creates the class-model
newClassWithRole: aString	creates a class-model according to the role received as a parameter
basicNewClassNamed: aString	creates a class-model omitting initialization
ensureClassNamed: aString	creates the class-model if not defined, and adds it to the language
addClass: aClassModel	adds a class to the language
allClasses	returns the collection of all classes in the language
allGlobals	returns the collection of globals in the language
bindingOf: aSymbol	returns binding of a variable if found in the receiver's scope
browse	open a code browser showing all packages, classes and methods in the model

Table B.2: Core *MetaL* MOP Methods in LanguageModel.

ClassModel class	
methodModelClass	returns the class in the metamodel representing method-models in this class
instanceVariableSlotModel Class	returns the class in the metamodel representing instance variable models in this class
packageModelClass	returns the class in the metamodel representing package-models associated to this class
defaultFormat	returns the default layout associated to the role associated to this class
ClassModel	
superclass: aClassModel	sets the superclass
superclass	returns the superclass
addLocalMethod FromSource: source selector: symbol	creates a new method-models and adds it to the methodDict
addMethod: aMethodModel	adds a method-model to the methodDict
localMethods	returns method-models defined in methodDict
methodDict	returns the dictionary where the class stores its methods
install	installs this class in the runtime as instance of it metaclass
format	returns the layout of instances of this class
format: anInt	sets the layout of instances of this class
metaclass	returns this receiver metaclass
bindingOf: aSymbol	returns the binding of a variable if found in the receiver's scope
innerBindingOf: aSymbol	returns the local binding of a variable if found in the receiver's scope

Table B.3: Core *MetaL* MOP Methods in ClassModel.

MethodModel class	
tempVarModelClass	returns the metamodel class to model temporaries
MethodModel	
fromSource: code	constructor

Table B.4: Core *MetaL* MOP Methods in MethodModel.

MetaL Roles

Role	Class format	Instance variables	Class table index	S.O.A. index	Required to load kernel
Array	indexable, no inst vars	*forbidden*	16 & 51	8	*Yes*
Association	fixed size	key, value			
Bitmap	32-bit indexable	*forbidden*		5	
ByteString	8-bit indexable	*forbidden*			
ByteSymbol	8-bit indexable	*forbidden*			
BlockClosure	indexable, with inst vars	outerContext startpc numArgs	37	37	*Yes*
ByteArray	8-bit indexable	*forbidden*	50	27	
ByteString	8-bit indexable	*forbidden*	52	7	
ByteSymbol	8-bit indexable	*forbidden*			
Character	immediate	*forbidden*	2	20	
CompiledMethod	8-bit indexable	*forbidden*		17	
Context	indexable, with inst vars	sender pc stackp method closureOrNil receiver	36	11	*Yes*

Table C.1: Roles in *MetaL* for a 32 bits Pharo VM. "S.O.A" stands for *special objects array*. Indexes are 1 based (part 1).

Role	Class format	Instance variables	Class table index	S.O.A. index	Required to load kernel
Dictionary	fixed size	tally array			
False	zero sized	*forbidden*			
Float	32-bit indexable	*forbidden*	34	10	*Yes*
LargeNegativeInteger	8-bit indexable	*forbidden*	32	43	*Yes*
LargePositiveInteger	8-bit indexable	*forbidden*	33	14	*Yes*
Message	fixed size	selector args lookupClass	35	16	
Metaclass	fixed size	superclass methodDict format			
MethodDictionary	indexable, with inst vars	tally array			
Point	fixed size	x y	53	13	
Process	fixed size	nextLink suspendedContext priority myList		28	
ProcessList	fixed size	firstLink lastLink			
ProcessScheduler	fixed size	suspendedProcessLists activeProcess			
SmallInteger	immediate	*forbidden*	1 & 3	6	*Yes*
True	zero sized	*forbidden*			
Semaphore	fixed size	firstLink lastLink excessSignals		19	
UndefinedObject	zero sized	*forbidden*			

Table C.2: Roles in *MetaL* for a 32 bits Pharo VM. "S.O.A" stands for *special objects array*. Indexes are 1 based (part 2).

Chapter D

Aspect-Oriented Programming (AOP)

AspectMetaL uses aspect-oriented programming (AOP) to solve *MetaL*'s limitations to code reuse. This section provides a brief introduction to AOP followed by the introduction of *PHANtom*, the aspect language *AspectMetaL* is implemented in.

D.1 AOP Concepts

Aspect-Oriented Programming (AOP) is a paradigm that seeks to solve the issue of code for one concern scattered among different classes by proposing a new kind of abstraction: the *aspect*. An aspect not only implements the behavior of a concern, but it also handles the time at which the behavior should be executed. To achieve this, the execution steps of an application are conceptually reified in *join points*. Aspects are able to identify among the stream of join points those which are interesting to them with the help of *pointcuts*, which work as queries to select join points. The behavior of the aspect is implemented in *advices*. An aspect can define multiple advices and multiple pointcuts. The aspect links advices with pointcuts such as when the application execution reaches a join point that matches a specific pointcut, the advices linked to the pointcut are executed.

D.2 *PHANtom*

PHANtom [Fabry 2012] is a dynamic aspect language for Smalltalk. Its constructs are first-class objects. Aspects are as classes in that they can be instantiated and their behavior is present in methods or blocks of code. *PHANtom* has no special syntax. Instead, aspects and their components are built by instantiating Smalltalk objects. *PHANtom* is dynamic because it allows for clases and aspects to be added, removed and changed at run-time.

We present a brief introduction to *PHANtom* constructs, since they are used in our description of *AspectMetaL* implementation.

Join points. They correspond to method executions. Meaning that each time an object receives a message, that is a join point. A stream of join points is produced during the execution of an application. They are characterized by the type of the receiver and by the selector of the message.

Pointcuts. They are responsible of determining when the behavior of an aspect is executed. They are predicates over the stream of join points. The pointcut in the example below matches all executions of the method with selector build where the receiver of the method is an instance of LanguageModel.

```
pc := PhPointcut receivers: 'LanguageModel' selectors: 'build' context: #(receiver)
```

The last argument of the constructor method makes the receiver of the message available for advices linked to this pointcut.

Advices. They define the behavior to be executed when a pointcut matches. This behavior is defined either in a regular method or in a block. The following example shows an advice whose behavior is specified in a block.

```
adv := PhAdvice
    before: pc
    advice: [ :ctx | Transcript show: ('Building model: ', ctx receiver name); cr. ]
```

The advise adv is executed (*i.e.* its block is evaluated) right before the execution of matches of the pointcut pc. The receiver of the message is obtained from the context variable.

Class modifiers. They are used to add and remove variables and methods in classes, enabling modular class extensions. The class modifier in the example below adds a new instance variable named 'metaclass' to the instance side of the class ClassModel.

```
cm := PhClassModifier new
    on: (PhPointcut receivers: 'ClassModel' selectors: #any asParser); "static pointcut"
    addNewInstanceVar: 'metaclass';
```

The pointcut in the first argument matches any method in the class ClassModel, because the expression #any asParser acts like a wildcard. The modification is performed only once.

Aspects. Aspects are classes implementing cross-cutting behavior using pointcuts, advices, and class modifiers. An aspect contains a collection of class modifiers and

advices, and each advice references a pointcut. Aspects need to be deployed to become active. When they are undeployed, the affected classes return to their original state.

In AOP, when multiple aspects define behaviors on the same pointcut, dynamic control of aspect execution becomes relevant. *PHANtom* allows for advice execution ordering, a key point for specification of semantic features in kernels.

Bibliography

[Aksit 1992] Mehmet Aksit, Lodewijk Bergmans and Sinan Vural. *An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach*. In O. Lehrmann Madsen, editeur, Proceedings ECOOP '92, volume 615 of *LNCS*, pages 372–395, Utrecht, the Netherlands, June 1992. Springer-Verlag.

[Beck 1993a] Kent Beck. *Instance specific behavior: Digitalk implementation and the deep meaning of it all*. Smalltalk Report, vol. 2(7), May 1993.

[Beck 1993b] Kent Beck. *Instance specific behavior: How and Why*. Smalltalk Report, vol. 2(7), May 1993.

[Beizer 1990] Boris Beizer. Software testing techniques (2nd ed.). Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[Bennett 1987] John K. Bennett. *The Design and Implementation of Distributed Smalltalk*. In Conference proceedings on Object-oriented programming systems, languages and applications, OOPSLA '87, pages 318–330, New York, NY, USA, 1987. ACM.

[Bergel 2004] Alexandre Bergel, Christophe Dony and Stéphane Ducasse. *Prototalk: an Environment for Teaching, Understanding, Designing and Prototyping Object-Oriented Languages*. In Proceedings of 12th International Smalltalk Conference (ISC'04), pages 107–130, September 2004.

[Biggerstaff 1987] T.J. Biggerstaff and C. Richter. *Reusability Framework, Assessment, and Directions*. IEEE Software, vol. 4, no. 2, pages 41–49, March 1987.

[Biggerstaff 1992] Ted J Biggerstaff. *An assessment and analysis of software reuse*. In Advances in Computers, volume 34, pages 1–57. Elsevier, 1992.

[Bobrow 1988] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonia E. Keene, Gregor Kiczales and D.A. Moon. *Common Lisp Object System Specification, X3J13*. Rapport technique 88-003, (ANSI COMMON LISP), 1988.

[Bolz 2009] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski and Armin Rigo. *Tracing the meta-level: PyPy's tracing JIT compiler*. In ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, pages 18–25, New York, NY, USA, 2009. ACM.

[Borning 1982] Alan H. Borning and Daniel H.H. Ingalls. *Multiple Inheritance in Smalltalk-80.* In Proceedings at the National Conference on AI, pages 234–237, Pittsburgh, PA, 1982.

[Bouraqadi 2000] Noury Bouraqadi. *Concern Oriented Programming using Reflection.* In Workshop on Advanced Separation of Concerns — OOPSLA 2000, 2000.

[Bracha 2004] 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.

[Bravenboer 2008] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas and Eelco Visser. *Stratego/XT 0.17. A language and toolset for program transformation.* Science of computer programming, vol. 72, no. 1-2, pages 52–70, 2008.

[Briot 1989] Jean-Pierre Briot. *Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment.* In S. Cook, editeur, Proceedings ECOOP '89, pages 109–129, Nottingham, July 1989. Cambridge University Press.

[Caballero 2007] Juan Caballero, Heng Yin, Zhenkai Liang and Dawn Song. *Polyglot: Automatic Extraction of Protocol Message Format Using Dynamic Binary Analysis.* In Proceedings of the 14th ACM Conference on Computer and Communications Security, CCS '07, pages 317–329, New York, NY, USA, 2007. ACM.

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

[Chis 2015] Andrei Chis, Marcus Denker, Tudor Girba and Oscar Nierstrasz. *Practical domain-specific debuggers using the Moldable Debugger framework.* Journal of Computer Languages, Systems and Structures, vol. 44, pages 89–113, 2015.

[Cointe 1987] Pierre Cointe. *Metaclasses are First Class: the ObjVlisp Model.* In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 156–167, December 1987.

[Costanza 2008] Pascal Costanza, Charlotte Herzeel, Jorge Vallejos and Theo D'Hondt. *Filtered dispatch.* In Proceedings of the 2008 Symposium on Dynamic languages, pages 1–10, 2008.

[DeMichiel 1987] Linda G. DeMichiel and Richard P. Gabriel. *The Common Lisp Object System: An Overview*. In J. Bézivin, J-M. Hullot, P. Cointe and H. Lieberman, editeurs, Proceedings ECOOP '87, volume 276 of *LNCS*, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[Denker 2008] 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.

[Ducasse 1995] Stéphane Ducasse, Mireille Blay-Fornarino and Anne-Marie Pinna. *A Reflective Model for First Class Dependencies*. In Proceedings of 10th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95), pages 265–280. ACM, October 1995.

[Ducasse 2006] Stéphane Ducasse and Tudor Gîrba. *Using Smalltalk as a Reflective Executable Meta-Language*. In International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006), volume 4199 of *LNCS*, pages 604–618, Berlin, Germany, 2006. Springer-Verlag.

[Ducasse 2017] Stéphane Ducasse, Dmitri Zagidulin, Nicolai Hess, Dimitris Chloupis Originally written by A. Black, S. Ducasse, O. Nierstrasz, D. Pollet with D. Cassou and M. Denker. Pharo by example 5. Square Bracket Associates, 2017.

[Durand 2019] Irène A Durand and Robert Strandh. *Bootstrapping Common Lisp using Common Lisp*. In EUROPEAN LISP SYMPOSIUM, 2019.

[Ernst 998] "Michael Ernst, Craig Kaplan and Craig Chambers". *"Predicate Dispatching: A Unified Theory of Dispatch"*. In "Eric Jul", editeur, "ECOOP '98—Object-Oriented Programming", volume "1445" of *"Lecture Notes in Computer Science"*, pages "186–211". "Springer", "1998".

[Fabry 2012] Johan Fabry and Daniel Galdames. *PHANtom: a modern aspect language for Pharo Smalltalk*. Software: Practice and Experience, pages n/a–n/a, 2012.

[Foote 1989] 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.

[Fowler 2005] Martin Fowler. *Language Workbenches: The Killer-App for Domain-Specific Languages*, June 2005. http://www.martinfowler.com/articles/languageWorkbench.html.

[Fowler 2010] Martin Fowler. Domain-specific languages. Pearson Education, 2010.

[Frakes 2005] William B Frakes and Kyo Kang. *Software reuse research: Status and future.* IEEE transactions on Software Engineering, vol. 31, no. 7, pages 529–536, 2005.

[Gabriel 1991] Richard P Gabriel, Jon L White and Daniel G Bobrow. *CLOS: Integrating object-oriented and functional programming.* Communications of the ACM, vol. 34, no. 9, pages 29–38, 1991.

[Garbinato 1995] Benoit Garbinato, Rachid Guerraoui and Karim R Mazouni. *Implementation of the GARF replicated objects platform.* Distributed Systems Engineering, vol. 2, no. 1, page 14, 1995.

[Goldberg 1983] Adele Goldberg and David Robson. Smalltalk 80: the language and its implementation. Addison Wesley, Reading, Mass., May 1983.

[Gordon 2007] Donald Gordon and James Noble. *Dynamic ownership in a dynamic language.* In Pascal Costanza and Robert Hirschfeld, editeurs, DLS '07: Proceedings of the 2007 symposium on Dynamic languages, pages 41–52, New York, NY, USA, 2007. ACM.

[GraalVM] GraalVM. *GraalVM.* http://www.graalvm.org, visited on 2021-08-31.

[Hernández Phillips 2019] Carolina Hernández Phillips, Guillermo Polito, Luc Fabresse, Stéphane Ducasse, Noury Bouraqadi and Pablo Tesone. *Challenges in Debugging Bootstraps of Reflective Kernels.* In IWST19 - International workshop on Smalltalk Technologies, 2019.

[Hernández Phillips 2021] Carolina Hernández Phillips, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, Stéphane Ducasse and Pablo Tesone. *Language Runtime Generation: Reducing the Cognitive Distance between Defects and Failures. (to be submitted).* 2021.

[Hirschfeld 2008] Robert Hirschfeld, Pascal Costanza and Oscar Nierstrasz. *Context-Oriented Programming.* Journal of Object Technology, vol. 7, no. 3, March 2008.

[JetBrains] JetBrains. *Meta Programming System.* http://www.jetbrains.com/mps.

[Keene 1989] Sonia E. Keene. Object-oriented programming in common-lisp. Addison Wesley, 1989.

[Kiczales 1991] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow. The art of the metaobject protocol. MIT Press, 1991.

[Kiczales 1993] Gregor Kiczales, J.Michael Ashley, Luis Rodriguez, Amin Vahdat and Daniel G. Bobrow. *Metaobject protocols: Why we want them and what else they can do.* In Object-Oriented Programming: the CLOS Perspective, pages 101–118. MIT Press, 1993.

[Kiczales 1997] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier and John Irwin. *Aspect-Oriented Programming.* In Mehmet Aksit and Satoshi Matsuoka, editeurs, Proceedings ECOOP '97, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[Koichi] Sasada Koichi. *Yet Another Ruby VM.* http://www.atdot.net/yarv, visited on 2021-08-31.

[Kokubun] Takashi Kokubun. *Yarv-Mjit compiler (Ruby).* https://github.com/k0kubun/yarv-mjit, visited on 2021-08-31.

[Konat 2012] Gabriël Konat, Lennart Kats, Guido Wachsmuth and Eelco Visser. *Declarative name binding and scope rules.* In International Conference on Software Language Engineering, pages 311–331. Springer, 2012.

[Krueger 1992] Charles W. Krueger. *Software Reuse.* ACM Computing Surveys, vol. 24, no. 2, pages 131–183, 1992.

[LaLonde 1988] Wilf R. LaLonde and Mark Van Gulik. *Building a Backtracking Facility in Smalltalk Without Kernel Support.* In Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, pages 105–122, November 1988.

[LaLonde 1991] Wilf LaLonde and John Pugh. Inside Smalltalk: Volume 2. Prentice Hall, 1991.

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

[Maes 1988] Pattie Maes. *Issues in Computational Reflection.* In D. Nardi P. Maes, editeur, Meta-Level Architectures and Reflection, pages 21–35. Elsevier Science Publishers B.V. (North-Holland), 1988.

[Malenfant 1996] J. Malenfant, M. Jacques and F.-N. Demers. *A tutorial on behavioral reflection and its implementation.* In Proceedings of Reflection, pages 1–20, 1996.

[McAffer 1995] Jeff McAffer. *Meta-level Programming with CodA.* In W. Olthoff, editeur, Proceedings ECOOP '95, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag.

[McCullough 1987] Paul L. McCullough. *Transparent Forwarding: First Steps.* In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 331–341, December 1987.

[Messick 1985] Steven L Messick and Kent L Beck. *Active variables in smalltalk-80.* Rapport technique, Technical Report CR-85-09, Computer Research Lab, Tektronix, 1985.

[Metaborg] Metaborg. *Spoofax.* http://www.metaborg.org, visited on 2021-08-31.

[Metacase] Metacase. *MetaEdit+.* http://www.metacase.com, visited on 2021-08-31.

[Oracle] Oracle. *Java Virtual Machine Startup.* https://docs.oracle.com/javase/specs/jvms/se16/html/jvms-5.html, visited on 2021-08-31.

[Papoulias 2011] Nikolaos Papoulias, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse and Luc Fabresse. *Towards Structural Decomposition of Reflection with Mirrors.* In Proceedings of International Workshop on Smalltalk Technologies (IWST'11), Edingburgh, United Kingdom, 2011.

[Papoulias 2017] Nick Papoulias, Marcus Denker, Stéphane Ducasse and Luc Fabresse. *End-User Abstractions for Meta-Control: Reifying the Reflectogram.* Science of Computer Programming, vol. 140, pages 2–16, 2017.

[Pascoe 1986] Geoffrey A. Pascoe. *Encapsulators: A New Software Paradigm in Smalltalk-80.* In Proceedings OOPSLA '86, ACM SIGPLAN Notices, volume 21, pages 341–346, November 1986.

[PetitParser] PetitParser. *PetitParser.* https://github.com/moosetechnology/PetitParser, visited on 2021-08-31.

[Polito 2015] Guillermo Polito, Stéphane Ducasse, Luc Fabresse and Noury Bouraqadi. *A Bootstrapping Infrastructure to Build and Extend Pharo-Like Languages.* In Onward! 2015, 2015.

[Polito 2021] Guillermo Polito, Pablo Tesone, Stéphane Ducasse, Luc Fabresse, Théo Rogliano, Pierre Misse-Chanabier and Carolina Phillips. *Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8.* In MPLR21, Germany, 2021.

[Rascal] Rascal. *Rascal Metaprogramming Language.* http://www.rascal-mpl.org, visited on 2021-08-31.

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

[Ring2] Ring2. *Ring2 Github Repository.* https://github.com/pavel-krivanek/Ring2, visited on 2021-08-31.

[SDF3] SDF3. *SDF3 Overview.* http://www.metaborg.org/en/latest/source/langdev/meta/lang/sdf3/introduction.html, visited on 2021-08-31.

[Sillito 2008] J. Sillito, G.C. Murphy and K. De Volder. *Asking and Answering Questions during a Programming Change Task.* IEEE Transactions on Software Engineering, vol. 34, no. 4, pages 434–451, jul 2008.

[Smith 1982] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language.* Ph.D. thesis, MIT, Cambridge, MA, 1982.

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

[Spinellis 2018] Diomidis Spinellis. *Modern Debugging: The Art of Finding a Needle in a Haystack.* Commun. ACM, vol. 61, no. 11, pages 124–134, October 2018.

[Stel 2020] Erick Stel. *New type of web application using HTML, CSS and Smalltalk.* https://www.youtube.com/watch?v=qvY7R6te7go, 2020. Accessed: 2021-03-10.

[Teruel 2012] Camille Teruel, Stéphane Ducasse and Marcus Denker. *Toward a modularization of Pharo: Analysis of the design space for a new module system.* In 9ème édition de la conférence MAnifestation des JEunes Chercheurs en Sciences et Technologies de l'Information et de la Communication-MajecSTIC 2012 (2012), 2012.

[Teruel 2015] Camille Teruel, Stéphane Ducasse, Damien Cassou and Marcus Denker. *Access Control to Reflection with Object Ownership.* In Dynamic Languages Symposium (DLS'2015), 2015.

[Ucko 2001] Aaron Mark Ucko. Predicate dispatching in the common lisp object. Master's thesis, Massachusetts Institute of Technology, 2001.

[Ungar 1987] David Ungar and Randall B. Smith. *Self: The Power of Simplicity.* In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 227–242, December 1987.

[Vergu 2015] Vlad Vergu, Pierre Neron and Eelco Visser. *DynSem: A DSL for dynamic semantics specification.* In 26th International Conference on Rewriting Techniques and Applications (RTA 2015). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[Visser 2004] Eelco Visser. *Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9.* In C. Lengauer*et al.*, editeurs, Domain-Specific Program Generation, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[Wegner 1983] Peter Wegner. *Varieties of reusability.* In Workshop on Reusability in Programming, pages 30–44, 1983.

[Willink 2011] Edward Daniel Willink. *Re-engineering eclipse MDT/OCL for xtext.* Electronic Communications of the EASST, vol. 36, 2011.

[Wimmer 2012] Christian Wimmer and Thomas Würthinger. *Truffle: a self-optimizing runtime system.* In Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity, pages 13–14, 2012.

[Wimmer 2019] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B Kessler, Oleg Pliss and Thomas Würthinger. *Initialize once, start fast: application initialization at build time.* Proceedings of the ACM on Programming Languages, vol. 3, no. OOPSLA, pages 1–29, 2019.

[Würthinger 2017] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöss, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon and Matthias Grimmer. *Practical Partial Evaluation for High-Performance Dynamic Language Runtimes.* In PLDI'17, 2017.

[XSemantics] Eclipse XSemantics. *XSemantics.* http://projects.eclipse.org/projects/modeling.xsemantics, visited on 2021-08-31.

[Xtext] Eclipse Xtext. *Xtext.* http://www.eclipse.org/Xtext, visited on 2021-08-31.

[Yokote 1987] Yasuhiko Yokote and Mario Tokoro. *Experience and Evolution of ConcurrentSmalltalk.* In Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, pages 406–415, December 1987.

[Zeller 2005] Andreas Zeller. Why programs fail: A guide to systematic debugging. Morgan Kaufmann, October 2005.

[Zenger 2001] Matthias Zenger and Martin Odersky. *Implementing extensible compilers.* In ECOOP Workshop on Multiparadigm Programming with Object-Oriented Languages. Citeseer, 2001.

Bootstrap-Based Language Development

Carolina Hernández Phillips

Abstract: Programming languages need to evolve as software requirements change, but their prototyping and extension comes at the cost of great development efforts. Bootstrap is a technique to support the evolution of a single language or a family of similar languages, specially used in the implementation of reflective languages. However, bootstrapping new languages is a challenging task due to the lack of proper abstractions for language specification, late manifestation of errors, and abstraction leaps during debugging tasks.

In this dissertation we study the design of a bootstrap based language development technique that supports the generation of multiple languages with low efforts. For this we introduce *MetaL*, a bootstrapping framework where language specification is based on metamodels and reflective instructions. Thanks to its Meta Object Protocol (MOP), *MetaL* ensures model correctness and kernel health, detecting corruption early during the generation process.

To validate our approach, we report on the successful generation of seven object-oriented language kernels, plus an experiment by an external user. These experiments show that *MetaL* reduced developer efforts in each study case and that it is applicable in real world scenarios. Nevertheless, this experience also shows that relying on metamodels for language specification hinders code reuse. To solve the previous limitation we propose *AspectMetaL*, an aspect-oriented layer on top of *MetaL* that allows language specification at the level of semantic features. To validate *AspectMetaL*, we generated 3 kernels by combining both semantic features and learning that AOP is effective for improving code reuse and rising the abstraction level for language specification.

Keywords: bootstrapping, programming language design, reflective languages, language runtime initialization.

Développement de langages basé l'amorçage

Carolina Hernández Phillips

Résumé: Les langages de programmation doivent évoluer au fur et à mesure que les exigences des logiciels changent, mais leur prototypage et leur extension se font au prix de grands efforts de développement. Le bootstrap est une technique permettant de soutenir l'évolution d'un langage unique ou d'une famille de langages similaires. Elle est spécialement utilisée dans l'implémentation des langages réflectifs. Cependant, l'amorcage de nouveaux langages est une tâche difficile en raison du manque d'abstractions appropriées pour la spécification du langage, de la manifestation tardive des erreurs et des sauts d'abstraction pendant les tâches de débogage.

Dans cette thèse, nous étudions la conception d'une technique de développement de langages basée sur l'amorcage qui permet de générer plusieurs langages avec peu d'efforts. Pour cela, nous introduisons *MetaL*, un cadre d'amorcage ou la spécification du langage est basée sur des métamodèles et des instructions réfléchies. Grâce à son Meta Object Protocol (MOP), *MetaL* assure la correction du modèle et la cohérence du noyau, en détectant une corruption au début du processus de génération.

Pour valider notre approche, nous rapportons la génération réussie de sept noyaux de langages orientés objet, ainsi qu'une expérience réalisée par un utilisateur externe. Ces expériences montrent que *MetaL* a réduit les efforts des développeurs dans chaque cas d'étude et qu'il est applicable dans des scénarios du monde réel. Néanmoins, cette expérience montre également que le fait de s'appuyer sur des métamodèles pour la spécification du langage entrave la réutilisation du code. Pour résoudre la limitation précédente, nous proposons *AspectMetaL*, une couche orientée aspect (POA) au-dessus de *MetaL* qui permet la spécification du langage au niveau des caractéristiques sémantiques. Pour valider *AspectMetaL*, nous avons généré 3 noyaux en combinant les deux caractéristiques sémantiques et nous avons constaté que la POA est efficace pour améliorer la réutilisation du code et augmenter le niveau d'abstraction pour la spécification du langage.

Mots clés: bootstrapping, conception de langages de programmation, langages réflectifs, initialisation de l'exécution d'un langage.