META PROGRAMMING SYSTEM

STIAN MATHIAS GUTTORMSEN

Bootstrapping LanguageLab to MPS
June 2014 – version 1.00 (final)

UNIVERSITY OF AGDER

Stian Mathias Guttormsen: *Meta Programming System,* Bootstrapping LanguageLab to MPS

IKT411 Advanced Project

SUPERVISORS:
Andreas Prinz
Terje Gjøsæter

ABSTRACT

In this report we took on Meta Programming System (MPS), a language workbench made for the industry, from an academic perspective. We did this to understand whether an academic language workbench, LanguageLab, could possibly be based on MPS in the future. We investigated the bootstrapping of MPS, i. e. how it is implemented in itself. We also showed how models are connected to their metamodels in MPS.

ACKNOWLEDGMENTS

I would like to thank my supervisors Andreas Prinz and Terje Gjøsæter. Their feedback have been *very* valuable. If there are any errors in this paper then it is only my fault.

Stian Mathias Guttormsen
Grimstad, June 2014

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation 1
- 1.2 Problem Statement 2
- 1.3 Contribution 2
- 1.4 State of the Art 2
- 1.5 Outline 3

2 BACKGROUND 5
- 2.1 LanguageLab 5
  - 2.1.1 Language Aspects 5
  - 2.1.2 The LanguageLab Platform 5
  - 2.1.3 The LanguageLab Architecture 6
- 2.2 Meta Programming System 7
  - 2.2.1 Nodes, Concepts and Languages 7
  - 2.2.2 Bootstrap Languages 7
  - 2.2.3 Language Aspects 7
  - 2.2.4 MPS All the Way Down 8

3 MPS DEEP DIVES 11
- 3.1 Models and Their Meta-models 11
  - 3.1.1 The MPS Meta-model 11
  - 3.1.2 An MPS Model and Its Meta-model 11
- 3.2 The MPS Build-process 13
  - 3.2.1 Building MPS Models 13
  - 3.2.2 Bootstrapping MPS 14
- 3.3 Integrating the Core Languages 16
  - 3.3.1 Structure 16

4 DISCUSSION 17
- 4.1 Models and Meta-models in MPS 17
- 4.2 Bootstrapping MPS 18
- 4.3 Integrating the Core Languages 18

5 CONCLUSION 21

BIBLIOGRAPHY 23

LIST OF FIGURES

Figure 1 The modules in LanguageLab form a hierarchy. The core languages (top-level) are bootstrapped, i. e. implemented using themselves. 6

Figure 2 Some of the core concepts of the MPS meta-model. 12

Figure 3 Nodes are instances of concepts, and these concepts are instances (nodes) of a ConceptDeclaration concept. 12

Figure 4 The build process of an MPS language. 13

Figure 5 The bootstrapping of MPS from a meta-modeling point of view. 15

Figure 6 The structure is visible in the logical view in MPS. 16

Figure 7 The Abstract Syntax Tree (AST) shown in the logical view conforms to the MPS meta-model. 19

ACRONYMS

AST	Abstract Syntax Tree
DSL	Domain-Specific Language
EMF	Eclipse Modeling Framework
LOP	Language-Oriented Programming
LWC	Language Workbench Challenge
MPS	Meta Programming System

1 INTRODUCTION

The advent of Language-Oriented Programming (LOP) marked a shift in programming paradigms: Instead of entire systems being written in a single general-purpose language, LOP described an approach where each component is implemented using a Domain-Specific Language (DSL) [2, 16]. In order to support LOP and the development and composition of DSLs the language workbench came to life [4].

1.1 MOTIVATION

LanguageLab is a language workbench that takes a meta-modeling approach to language design. It is a proof-of-concept workbench based on research on abstractions and model-driven language specification at the University of Agder [5]. The aim is for LanguageLab to be used as a teaching tool in courses about computer languages at the university.

With LanguageLab there is a clear distinction between the language aspects; abstract syntax, concrete syntax and semantics are all defined using separate DSLs. Moreover, these languages are *small* and relatively easy to understand, making them ideal for illustrating important concepts such as self-referential bootstrapping, i. e. how they are defined using themselves.

Aside from the importance of language workbenches in general, we believe that LanguageLab can prove a valuable tool for exploring the meta-model side of language specification. It seeks to provide its users with a clear view of the meta-model hierarchy, hopefully helping students to understand what meta-modeling is all about.

The current version of LanguageLab is an Eclipse application powered by the Eclipse Modeling Framework (EMF), i. e. it is not built on top of an existing language workbench. Much time is therefore spent on developing the *workbench*, rather than the *ideas* and *principles* behind the platform. Although these subjects are closely connected, it may be beneficial if LanguageLab could be ported to an existing workbench.

The Meta Programming System (MPS) is one of the many alternatives and one we believe that could be used as a basis for LanguageLab. It is an industrial-strength, model-based and open-source language workbench developed by JetBrains[1]. Not only is MPS a mature platform (in development since 2003), but *"the NYoSh experiment"*[11] shows it can be used as a foundation for other workbenches as well.

[1] http://www.jetbrains.com/mps/

1.2 PROBLEM STATEMENT

MPS represents the state-of-the-art in language workbenches made for the industry, while LanguageLab is a proof-of-concept workbench that stems from research and academics. We find it interesting to investigate MPS from an academic point of view to see how the underlying theoretical ideas are put into practice. This to find limits and possibilities to use MPS as basis for LanguageLab Specifically, we want to:

1. Find out how MPS implements the idea of a model being an instance of another model (its meta-model).
2. Understand the build-process in MPS, i.e. how the bootstrapping of MPS works.
3. Examine some MPS core languages and show how they are connected to the MPS platform core.

1.3 CONTRIBUTION

The contributions of this paper are the following:

1. We compare an academic language workbench (LanguageLab) with one from the industry (MPS).
2. We reveal parts of the MPS meta-model.
3. We show how the connection between a model and its meta-model is done in MPS.
4. We report the steps involved in the build-process of the MPS platform.

1.4 STATE OF THE ART

Regarding the current state of LanguageLab, it has DSLs for defining the abstract syntax and semantics of languages using a built-in editor. The editor is a projectional tree-editor, i. e. the models are edited via a tree-like projection. A DSL for defining conrete textual syntaxes is currently under development. Semantics are primarily defined using a model-to-model transformation DSL, although more complex behavior can be created using hand-written Java.

LanguageLab is internal to the University of Agder, but language workbenches in general are an active area of research, with a continuous effort being made to improve the way DSLs can be defined and used. This means there are a few different workbenches, each with various design principles and feature-sets.

A comparison between MPS, Actifsource and Spoofax reveals that, among the three, only MPS could be viable for porting LanguageLab; Actifsource is not really a *language* workbench but is used to create domain models, while Spoofax takes a text-based approach to DSLs [12].

MPS is also a participant of recent years Language Workbench Challenge (LWC)[2], an annual workshop that surveys the state of the art in language workbenches. The 2013 LWC concludes with a feature model for workbenches, and shows that MPS is one on the most feature-rich alternatives [3].

The LWC proceedings also notes that half of the participants are tools for the industry, while the other half are products from academia. With MPS being a tool for the real world, it seems that its documentation focus more on how things are done, instead of how it relates to meta-modeling and language theory.

We have not been able to find any research that goes into the bootstrapping process of MPS, let alone identifying its meta-model. But some information on modular languages (as implemented in MPS) can be found (e. g. in *"DSL Engineering"*[15] by Völter et al.).

1.5 OUTLINE

The rest of this report is structured as follows. We begin with some background on language workbenches in Chapter 2, specifically LanguageLab and MPS. We then proceed to explore MPS and present our findings in Chapter 3. In Chapter 4 we discuss our results, before concluding the report in Chapter 5.

2 `http://www.languageworkbenches.net/`

2 BACKGROUND

There are many different approaches to language workbenches. LanguageLab and MPS share some of their features, but whereas the former is relatively simplistic in its design, the latter is rich in features but also quite complex. We will now take a closer look at these two workbenches before diving deeper into MPS in Chapter 3.

2.1 LANGUAGELAB

LanguageLab attempts to demonstrate meta-modeling concepts and is not a general-purpose language workbench. It is a proof-of-concept that takes a modular approach to language definition, where a language is described using several meta-languages. Each of these meta-languages are generally used to describe a particular *aspect* of the new language.

2.1.1 *Language Aspects*

LanguageLab considers a language definition to consist of four aspects: structure, presentation, behavior and constraints. Their definitions are as follows [6, 7].

STRUCTURE defines the constructs of a language and how they are related, i.e. what the language can express.

BEHAVIOR defines the semantics, i.e. how it can be executed or transformed into another language.

PRESENTATION defines how language instances are represented, e.g. textual or graphical editors.

CONSTRAINTS brings additional constraints on the structure.

The way that LanguageLab approaches language definition, and how these aspects are defined in the application, is known as the LanguageLab platform.

2.1.2 *The LanguageLab Platform*

A language definition (with all its aspects) are contained in a language *module*. Module is also the term used of models, meta-languages, meta-meta-languages and so on, i. e. it is a collective term for anything created in LanguageLab.

A module in LanguageLab is (the root element of) an instance of the LanguageLab meta-model [5]. In the module all aspects of a language is defined, and also references to all the languages used to define the module are kept. For instance, the structural elements of a language are instances of *Type*, and the behavior are *Operation* instances.

Everything that can be expressed in LanguageLab stems from the LanguageLab meta-model. The functionality of LanguageLab is thus limited to the meta-model, and this is referred to as the *platform functionality*.

2.1.3 *The LanguageLab Architecture*

A LanguageLab module is specified using one or more meta-modules. Any module can in theory be used either as a module or a meta-module, and it is only determined by the way the module is loaded into LanguageLab. A meta-module in LanguageLab provides an interface that can be used by other modules. The modules then form a hierarchy, as shown in Figure 1.

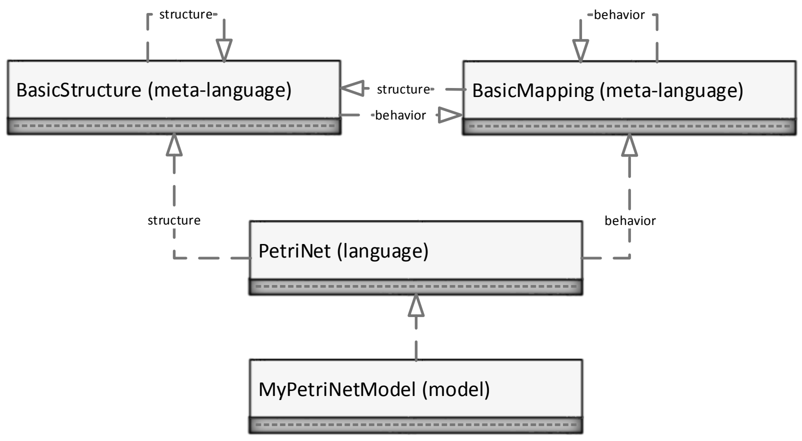


Figure 1: The modules in LanguageLab form a hierarchy. The core languages (top-level) are bootstrapped, i. e. implemented using themselves.

The languages at the top of the model hierarchy are implemented using themselves, e. g.the structure of the *BasicStructure* language is defined using BasicStructure. The primitive languages in LanguageLab are therefore said to be *bootstrapped*.

2.2 META PROGRAMMING SYSTEM

MPS is another model-based language workbench. Compared to LanguageLab it is a huge system, and there is no obvious MPS metamodel that shows the boundaries of the system (i. e. is it not clear how the core MPS platform functionality can be separated from the entire system).

2.2.1 *Nodes, Concepts and Languages*

A model in MPS is always in the form of an Abstract Syntax Tree (AST). The AST is a tree structure of *nodes*, and each of these nodes are instances of some *concept*. A *language* in MPS is just a collection of concepts and other aspect information e. g. editors [13].

2.2.2 *Bootstrap Languages*

MPS includes a *devkit*, i. e. a collection of exported languages (and/or solutions), that includes all the bootstrap languages. It is conveniently named *bootstrap-languages*[1]. The devkit includes twenty different languages, and some of these are used to describe the different language aspects in MPS. We will not investigate the languages that are not directly represented as language aspects.

2.2.3 *Language Aspects*

MPS specifies a total of nine different language aspects, and the "*MPS User Guide for Language Designers*"[13] defines them as follows.

STRUCTURE describes the nodes and structure of the language AST. This is the only mandatory aspect of any language.

EDITOR describes how a language will be presented and edited in the editor.

ACTIONS describes the completion menu customizations specific to a language, i. e. what happens when you type Control + Space.

CONSTRAINTS describes the constraints on AST: where a node is applicable, which property and reference are allowed, etc.

BEHAVIOR describes the behavioral aspect of AST, i. e. AST methods.

TYPESYSTEM describes the rules for calculating types in a language.

INTENTIONS describes intentions (context dependent actions available when light bulb pops up or when the user types Alt + Enter).

1 jetbrains.mps.devkit.bootstrap-languages

PLUGIN allows a language to integrate into the MPS integrated development environment.

DATA FLOW describes the intented flow of data in code. It allows you to find unreachable statements, uninitialized reads etc.

Already, just from their definitions, we see that some of these aspects are very specific to MPS; actions, intentions, plugins and data flow are not what we consider to be core aspects of a language, but rather utilities that help bridge the gap between *language functionality* and *platform functionality*. In other words, MPS uses these aspects to create a consistent and complete editing experience, and with it comes an entirety to the MPS platform.

We will for the most part ignore the aforementioned MPS-specific aspects, and focus on those that are acknowledged in language theory: structure, editor, constraints and behavior. We consider the typesystem aspect to be closely related to constraints, and as such we will not go much into the typesystem either.

2.2.4 *MPS All the Way Down*

The concept of bootstrapping is also applied in MPS, where the core languages are implemented using themselves. However, in MPS it is not only the languages that are said to be bootstrapped, but the platform itself.

MPS is written in *BaseLanguage*, and BaseLanguage is Java 1.6 implemented in MPS, i. e. MPS is written in a language developed in MPS. Thus the bootstrapping of MPS can be compared to that of other reflective systems; the evolution of the language (e. g. BaseLanguage) must take into account the system in which it is developed (MPS), so as not to break the system. Changing a feature in the language could introduce bugs into the system unless the system also was updated to account for the change.

For instance, in Smalltalk systems like Pharo[2], the development of the language has traditionally been a series of changes. Each of the changes need not be compatible with each other, and each change could introduce a subtle bug that may break the system some time down the line. This is explained in "*Bootstrapping Reflective Systems: The Case of Pharo*"[9] by Polito et al., where they also propose a boostrapping model that solves the problems inherent to this process of bootstrapping by evolution.

Considering MPS, it is sort of intuitive how the *initialization process* of the system works, i. e. how the first version of the system is built with Java, then subsequent versions are developed using (an almost 1:1 Java clone) BaseLanguage in MPS. Still it is not really clear how

2 `http://www.pharo.org`

MPS supports its own evolution, i. e. how the *bootstrap process* works, because not all parts of the platform are written in BaseLanguage.

After investigating how models and their meta-models are connected in MPS, we will take a closer look at the build-process of MPS.

3 MPS DEEP DIVES

We started our journey into MPS by implementing a primitive version of the PetriNet language.[1] It is a simple language with *places* and *transitions* between them, and is somewhat similar to state machines. We used version 3.0 of MPS during this project.

3.1 MODELS AND THEIR META-MODELS

As part of understanding the connection between a model and its meta-model in MPS, we reveal parts of the MPS meta-model.

3.1.1 *The MPS Meta-model*

We explained earlier how an MPS model, or AST, consists of nodes of different concepts. We found a more complete description of the AST and its elements in the MPS source code.[2]

Figure 2 shows parts of the MPS meta-model describing ASTs. In the figure *SModule* represents a language (or solution) in MPS. The different language aspects are then described in *ModelRoots*, with *SModels* representing the aspects elements, e. g. description of a new concept. The *SNodes* then are the elements consituting the SModels.

We see that SNode keeps a reference to its *SConcept* in Figure 2. This is an implementation detail of the MPS platform, but for language designers working in MPS a node (SNode) is actually considered to be an instance of some concept (SConcept).[3]

3.1.2 *An MPS Model and Its Meta-model*

An MPS model is an instance of another model, i. e. its meta-model. This relationship can be identified in the nodes and concepts of MPS ASTs. The nodes at one level are instances of concepts at the meta-level, as shown in Figure 3. In the figure each class represents a different level in the meta-model hierarchy, i. e. ConceptDeclaration on the level above Concept and Concept on the level above Node.

The concepts are also nodes, i. e. when editing a language in MPS the concepts it offers are described by nodes. But when we use this language to create models the concepts are considered to be concepts

1 PetriNet has been our standard go-to language when testing features in LanguageLab.
2 Overview of ASTs found in org.jetbrains.mps.openapi.module.SRepository
3 Explained in source code org.jetbrains.mps.openapi.language.SAbstractConcept

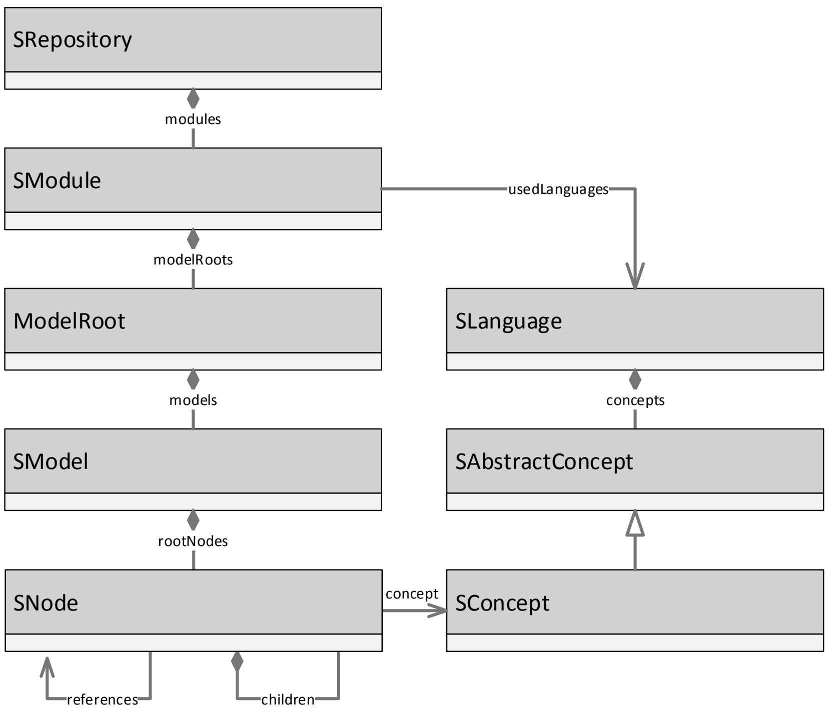


Figure 2: Some of the core concepts of the MPS meta-model.

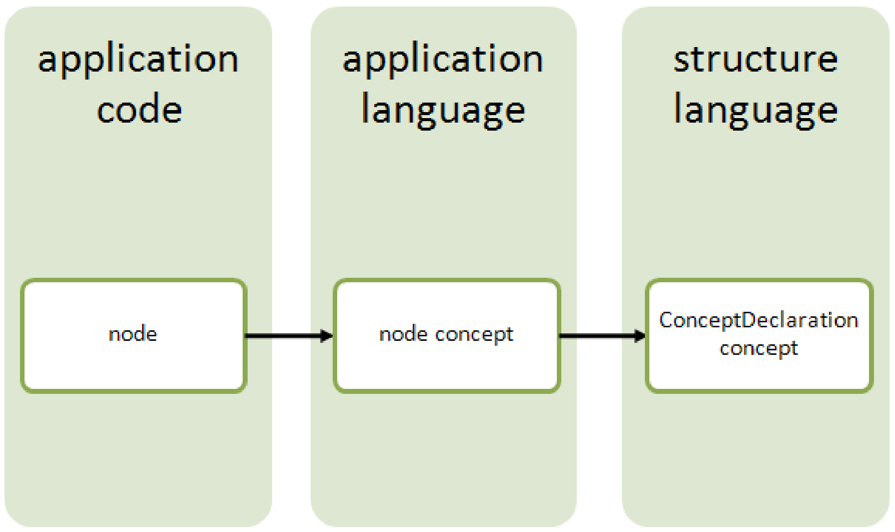


Figure 3: Nodes are instances of concepts, and these concepts are instances (nodes) of a ConceptDeclaration concept.

and not nodes. In other words, the interface provided by a language is its defined concepts, while these same concepts are defined through its defining nodes.

A node describing a new concept, i. e. a ConceptDeclaration node, is transformed into a new concept during the build-process of the language.

3.2 THE MPS BUILD-PROCESS

3.2.1 *Building MPS Models*

Starting from version 2.0, MPS includes a *make scripting framework* that is used to describe the build-process of models [10]. The make-system consists of a sequence of steps called *targets*, and a collection of targets (that are related) makes up a *build facet*.

The make framework is extensible [14], but the default configuration used when making a language (that generates Java code) consists of the following build facets (see Figure 4).

1. Generate
2. TextGen
3. JavaCompile
4. ReloadClasses
5. Make

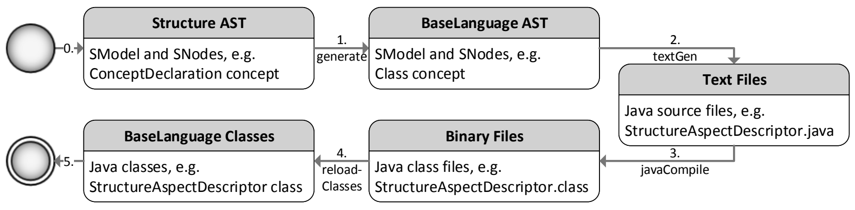

0.
Structure AST
SModel and SNodes, e.g. ConceptDeclaration concept
1. generate
BaseLanguage AST
SModel and SNodes, e.g. Class concept
2. textGen
Text Files
Java source files, e.g. StructureAspectDescriptor.java
3. javaCompile
Binary Files
Java class files, e.g. StructureAspectDescriptor.class
4. reload-Classes
BaseLanguage Classes
Java classes, e.g. StructureAspectDescriptor class
5.

Figure 4: The build process of an MPS language.

The language designer defines the new language in step 0 (in Figure 4), and it is not part of the build process. The way the editing is done is specified by the *editor* aspect of the meta-language used to define the new language. For instance, it is the structure language's editor that is used when editing the structure of the new language.

The generate facet (in step 1) transforms the AST defined previously into another AST. It is similar to the previous step, but here it is the *generator* (of the meta-language used) that determines the transformation.

Next in step 2 is the text-generate facet. It uses the textGen aspect of the language used in the previous step, e. g. if step 1 is a BaseLanguage model, then step 2 is the model-to-text transformation defined by BaseLanguage's textGen aspect.

Step 3 takes the generated text files and compiles them, before step 4 loads them into MPS. The classes loaded into MPS are *concept descriptors* that describe the various aspects of the new language.

This process lies at the heart of anything developed in MPS. Above it is described in the point-of-view of making a new language project, but the process is basically the same for solution projects.

3.2.2 *Bootstrapping MPS*

The bootstrapping of MPS seems to use the same pattern as discussed in the previous section. With that said it is a huge platform, but the build-process of MPS in MPS includes the following.

GENERATE BaseLanguage models from the different solutions (models) that comprise MPS.

TEXTGEN transforms the BaseLanguage models of MPS into Java source files.

JAVACOMPILE compiles the Java source files, resulting in MPS as Java class files (that can be run).

This abovementioned list is a very high level view on the build-process, and it describes the *generate (build)* transformation in Figure 5[4] from the M1 level to M'1 level. The four levels M3-M0 displayed here is inspired by the *four-level meta-model hierarchy* as defined by the Object Management Group [8]:

M3 is the meta-language (or meta-meta-model) level. The MPS core (bootstrap) languages can be said to reside here.

M2 is the language (or meta-model) level, where the BaseLanguage (and other MPS languages) are defined using the meta-languages at the M3 layer.

M1 is the model level, where e. g. BaseLanguage models (programs) are created (using languages at M2).

M0 represents the execution of an M1 model.

These four levels only makes sense when considered part of a platform, i. e. there must be some tool that supports these levels and how they relate to each other. In this case, MPS is the platform that supports

4 Note that the figure does not completely reflect the MPS platform, as some of the core languages also uses BaseLanguage.

this meta-model hierarchy. Thus the four levels can be considered to live inside MPS when running the platform (as depicted by the M'0 level in Figure 5).

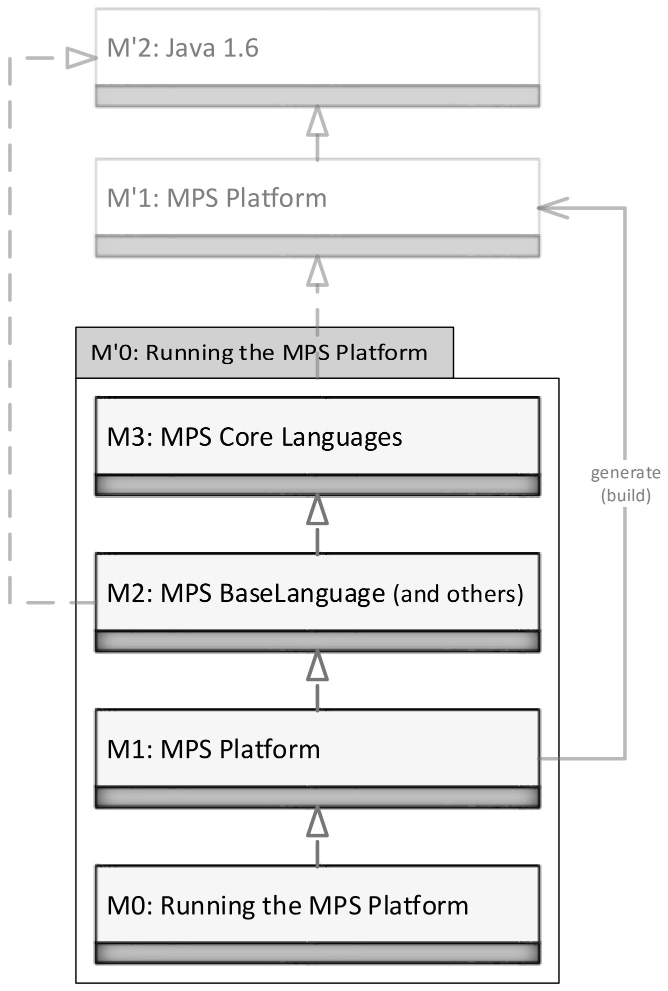

M'2: Java 1.6
M'1: MPS Platform
M'0: Running the MPS Platform
M3: MPS Core Languages
generate (build)
M2: MPS BaseLanguage (and others)
M1: MPS Platform
M0: Running the MPS Platform

Figure 5: The bootstrapping of MPS from a meta-modeling point of view.

The M'0 level in Figure 5 is the running MPS platform, and inside it lives all the models (and languages etc.) that are created in MPS. Bootstrapping MPS then means that the MPS platform can be defined inside MPS itself, i. e. MPS is defined inside M'0, specifically at M1.

When running MPS at the M'0 level, and considering it as the execution of MPS as defined at M'1, we see that we actually have a nested meta-model architecture, where levels (e. g. M'0) can contain other meta-model hierarchies (M3-M0).

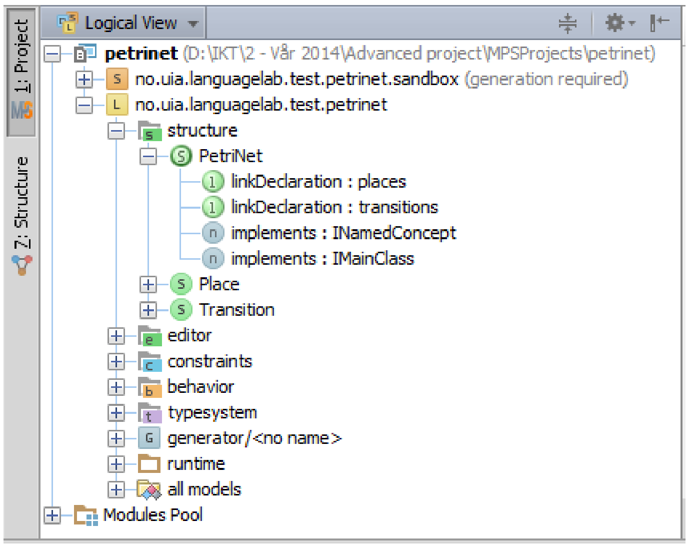
3.3 INTEGRATING THE CORE LANGUAGES

The MPS platform integrates some of the core languages to form an entirety to the platform, thus enhancing the user experience. We mentioned in Section 2.2.3 how some of the language aspects are specifically made for integrating the languages into the platform, but apart from the obvious ones (e. g. plugin and intentions), other aspects are also visible in the platform.

3.3.1 *Structure*

The structure is the heart of MPS models, and it is always visible in the project outline (logical view) when working in MPS (see Figure 6). Along with the structure language, there is also the *SModel* language for working directly with the ASTs in MPS [1].

Unfortunately due to time limitations we were not able to completely investigate the different core languages of MPS to see how they are linked to the platform.



```
petrinet (D:\IKT\2 - Vår 2014\Advanced project\MPSProjects\petrinet)
  no.uia.languagelab.test.petrinet.sandbox (generation required)
  no.uia.languagelab.test.petrinet
    structure
      PetriNet
        linkDeclaration : places
        linkDeclaration : transitions
        implements : INamedConcept
        implements : IMainClass
      Place
      Transition
    editor
    constraints
    behavior
    typesystem
    generator/<no name>
    runtime
    all models
Modules Pool
```

Figure 6: The structure is visible in the logical view in MPS.

4 DISCUSSION

4.1 MODELS AND META-MODELS IN MPS

Despite its practical approach to language definition, MPS languages have a somewhat "pure" representation in MPS; this in the sense that the different language aspects are not discrimated between in the way they are stored and manipulated in MPS. In other words, MPS models—ASTs—consist of *nodes* of different *concepts*, and these nodes and concepts are instances of the same classes in MPS (SNode and SConcept) no matter what language aspect they are describing.

The MPS meta-model (that describe the models) can be considered simple yet powerful. However, since the language aspects are not that clearly separated in the *meta-model* as such, it takes a different approach than the academic LanguageLab workbench. From our understanding of MPS the language aspects are separated by logic (i. e. behavior) in the platform, and not also through structure as is the case with LanguageLab (e. g. where structure and behavior are *Types* and *Operations* respectively in the LanguageLab meta-model).

To be sure, we have not uncovered the complete MPS meta-model as such, but what we have found suggests that it is somewhat different than LanguageLab in regard to language aspects; it seems LanguageLab is more explicit in defining the language aspects as part of the meta-model.

There is one striking resemblance between MPS and LanguageLab however; the relationship between nodes and concepts in MPS are identical to that of objects and types in LanguageLab. Both workbenches also define new concepts (types) using nodes (objects), but the transformation from a concept declaration node to a new concept is much more involved in MPS.

Everything in MPS revolves around *code generation* with little use of *model interpretation*. This difference is obvious in the way it defines new concepts; in MPS the concept declaration is transformed into BaseLanguage concept descriptors, and these descriptors are then transformed to text (Java source files) and compiled, before loading the compiled descriptors into MPS. In LanguageLab it is a simple matter of transforming objects into types (in the meta-model), without any need for external Java.

4.2 BOOTSTRAPPING MPS

When retrieving the source code of MPS from the repository, it is fully compliant Java code that can be compiled and run. It is known that MPS is written in MPS, and it can be seen that bigger parts of the codebase is generated by MPS.

This brings us to the *initialization process* of MPS, i. e. how the first version of MPS could be built. We mentioned this briefly earlier, and although we have not confirmed it (since we have not been testing MPS during its infancy), it stands reason to believe that everything in MPS was written in plain Java at the beginning (with no generated code). After the platform grew and the BaseLanguage (Java) was implemented in MPS, the development of MPS could be moved from outside (plain Java) to inside MPS itself (with BaseLanguage). Thence the platform could be *bootstrapped* by supporting its own evolution.

The bootstrapping of MPS is a process of evolution through recreation, i. e. for each new version of MPS the entire platform is built (generated and compiled) from scratch. The generation phase of building MPS is a model-to-model transformation that represents the system as BaseLanguage models. With BaseLanguage as the target language it is possible to generate the Java source files.The source files can then be compiled and the system can be run on a Java virtual machine.[1]

The bootstrap process is similar to that described in "*Bootstrapping Reflective Systems: The Case of Pharo*"[9] by Polito et al. There are some major differences though in that MPS generates Java that can be run outside of MPS, while the Smalltalk environment in the aforementioned article keeps closer ties to itself (where the connection to the new system is cut at a later stage in the build process).

4.3 INTEGRATING THE CORE LANGUAGES

Due to time limitations we were not able to investigate the core language of MPS as thoroughly as we wanted. But with some of our knowledge of the MPS meta-model (from Section 3.1.1), we confirmed that the structure shown in the MPS logical view conforms to the MPS meta-model. This is shown in Figure 7.

1 Note that when first retrieving the MPS Java source code it has already been generated by MPS.

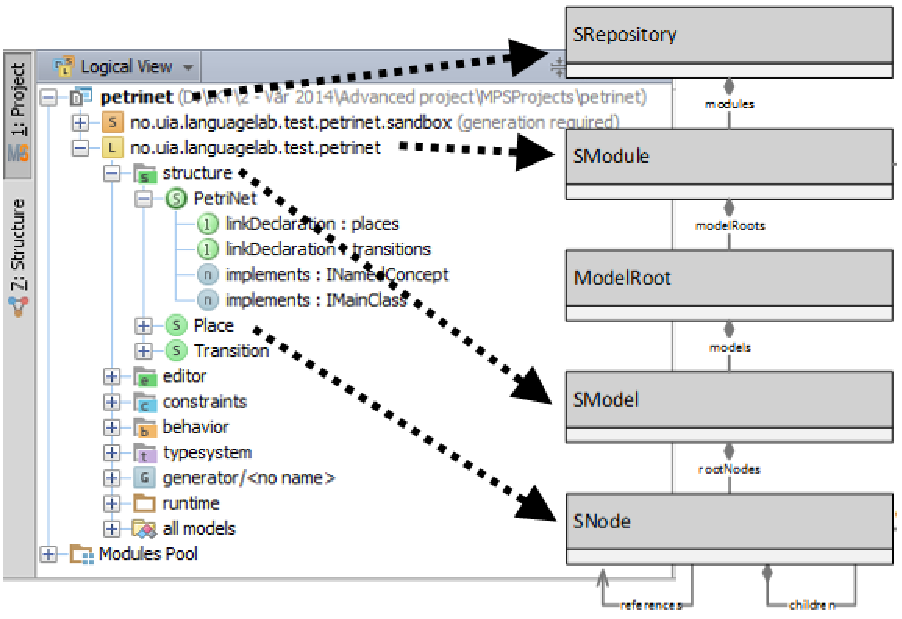


Figure 7: The AST shown in the logical view conforms to the MPS metamodel.

5 CONCLUSION

Parts of the MPS meta-model can be described as models containing nodes and concepts. Different language aspects does not seem to be described by different classes in the meta-model, but rather different instances of the same classes (models, nodes and concepts). Thus the distinction between language aspects does not seem to be structural (from a meta-model point of view), but part of the platform's logic and behavior.

One of the key positive notes on MPS is that the platform is bootstrapped. The bootstrapping process helps prove the platform's potential, and the process is interesting in itself. But it is not clear how much it would actually benefit LanguageLab to be based on MPS and its ways; for LanguageLab to demonstrate the ideas and concepts behind it, it is more important for the languages inside the platform to be bootstrapped than the platform itself.

In closing it does not seem to be worth the risk to try and use MPS as a basis for LanguageLab. One of the determining factors is that it is hard to separate meta-levels in MPS, while LanguageLab focuses on very clear boundaries between models, languages and meta-languages. Another thing is how pretty much everything in MPS revolves around Java (BaseLanguage) and code generation. We feel this would be a step in the wrong direction for LanguageLab, where the goal is to always work on a suitable level of abstraction; the inability to completely escape Java seems to go against these principles.

BIBLIOGRAPHY

[1] Fabien Campagne. *The MPS Language Workbench: Volume I*. Vol. 1. The MPS Language Workbench. Fabien Campagne, 2014. URL: `http://books.google.no/books?id=nvcEAwAAQBAJ` (cit. on p. 16).

[2] Sergey Dmitriev. "Language Oriented Programming: The Next Programming Paradigm". In: *JetBrains onBoard* 1.2 (2004) (cit. on p. 1).

[3] Sebastian Erdweg et al. "The State of the Art in Language Workbenches". In: *Software Language Engineering*. Ed. by Martin Erwig, RichardF. Paige, and Eric Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer International Publishing, 2013, pp. 197–217. ISBN: 978-3-319-02653-4. DOI: `10.1007/978-3-319-02654-1_11`. URL: `http://dx.doi.org/10.1007/978-3-319-02654-1_11` (cit. on p. 3).

[4] Martin Fowler. "Language Workbenches: The Killer-App for Domain Specific Languages". In: (2005). URL: `http://www.issi.uned.es/doctorado/generative/Bibliografia/Fowler.pdf` (visited on 04/04/2014) (cit. on p. 1).

[5] Terje Gjøsæter and Andreas Prinz. *LanguageLab 1.1 User Manual*. University of Agder, 2013. URL: `http://brage.bibsys.no/xmlui/handle/11250/134943` (visited on 03/28/2014) (cit. on pp. 1, 6).

[6] Stian Mathias Guttormsen. "Describing Objects in LanguageLab". In: (2013). Bachelor's Thesis (cit. on p. 5).

[7] Jan P. Nytun, Andreas Prinz, and Merete S. Tveit. "Automatic Generation of Modelling Tools". In: *Model Driven Architecture - Foundations and Applications*. Ed. by Arend Rensink and Jos Warmer. Vol. 4066. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 268–283. ISBN: 978-3-540-35909-8. DOI: `10.1007/11787044_21`. URL: `http://dx.doi.org/10.1007/11787044_21` (cit. on p. 5).

[8] Object Management Group. *UML 2.0 Infrastructure Specification*. 2003. URL: `www2.informatik.hu-berlin.de/sam/lehre/MDA-UML/UML-Infra-03-09-15.pdf` (visited on 06/06/2014) (cit. on p. 14).

[9] Guillermo Polito et al. "Bootstrapping Reflective Systems: The Case of Pharo". Anglais. In: *Science of Computer Programming* (Jan. 2014). URL: `http://hal.inria.fr/hal-00903724` (cit. on pp. 8, 18).

[10] Alexander Shatalin. *What's new in MPS 2.0*. Confluence - JetBrains, 2011. URL: `http://confluence.jetbrains.com/display/MPS/What's+new+in+MPS+2.0` (visited on 06/03/2014) (cit. on p. 13).

[11] Manuele Simi and Fabien Campagne. "Composable Languages for Bioinformatics: The NYoSh Experiment". In: *PeerJ* 2 (Jan. 2014), e241. ISSN: 2167-8359. DOI: `10.7717/peerj.241`. URL: `http://dx.doi.org/10.7717/peerj.241` (cit. on p. 1).

[12] Roman Stoffel. "Comparing Language Workbenches". In: *MSE-seminar: Program Analysis and Transformation*. 2010, pp. 18–24 (cit. on p. 3).

[13] Eugene Toporov, Vaclav Pech, and Alexander Shatalin. *MPS User Guide for Language Designers*. Confluence - JetBrains, 2013. URL: `http://confluence.jetbrains.com/display/MPSD30/MPS+User's+Guide+(one+page)` (visited on 05/29/2014) (cit. on p. 7).

[14] Markus Völter. *HowTo – Integrating into the MPS Make Framework*. Confluence - JetBrains, 2011. URL: `http://confluence.jetbrains.com/display/MPSD30/HowTo+--+Integrating+into+the+MPS+Make+Framework` (visited on 06/04/2014) (cit. on p. 13).

[15] Markus Völter et al. "DSL Engineering: Designing, Implementing and Using Domain-Specific Languages". In: *Implementing and Using Domain-Specific Languages. dslbook. org* (2013) (cit. on p. 3).

[16] Martin P. Ward. "Language Oriented Programming". In: *Software-Concepts and Tools* 15.4 (1994), pp. 147–161 (cit. on p. 1).