LanguageLab 1.1 User Manual

Terje Gjøsæter Andreas Prinz

January 1, 2013

Contents

Preface

This user manual covers version 1.1 of LanguageLab.

The following people have contributed to the design and development of the LanguageLab platform:

- Terje Gjøsæter
- Andreas Prinz
- Samuel Vogel
- Stian Mathias Guttormsen
- Thomas Fauskanger
- Guro Ødesneltveit

Chapter 1

Introduction

In meta-model-based language design, a major challenge is to be able to operate on an adequate level of abstraction when designing a complete computer language. There are several different technologies, meta-languages and tools in use for defining different aspects of a language, that may or may not satisfy the needs of a DSL developer when it comes to abstraction level. Before starting design and development of the LanguageLab workbench, we set out to examine what concepts are needed for defining the different aspects of a computer language, and discuss how to apply them on a suitable level of abstraction. If the abstraction level is too high, the definition of behaviour may be a challenge, and on the other hand if the abstraction level is too low, the language developer will spend too much time on unnecessary details.

In the LanguageLab platform, we set out to facilitate operation on a suitable abstraction level, and also focus on user-friendliness and a low threshold to getting started, in order to make it useful for teaching of meta-modelling. The platform will be open for third party language modules and is intended to facilitate re-use of language modules, modular language development and experiments with multiple concrete syntaxes.

Another goal is to supply some basic guidelines for developing LanguageLab modules that can further add to the features and capabilities of the LanguageLab platform.

1.1 Language Aspects

A description of a modelling language, whether it is a domain specific language (DSL) or a general purpose language, usually involves several different technologies and meta-languages. Traditionally, we are familiar with the distinction between the *syntax* and the *semantics* of a language. The syntax specifies the structure of sentences in the language, while the semantics assign a meaning to the sentences.

In [4], a language definition is said to consist of the following aspects: *Struc-*

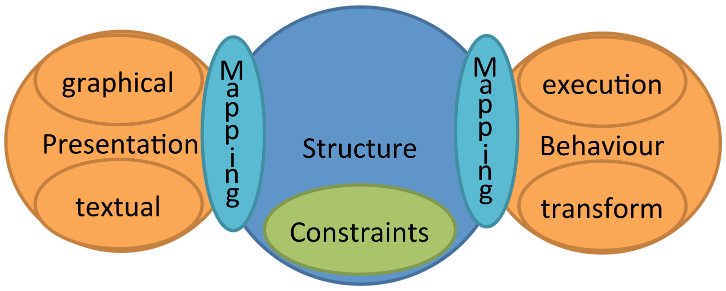


Figure 1.1: The aspects of a modelling language.

ture, *Constraints*, *Presentation* and *Behaviour* (see Figure 1.1).

Structure defines the constructs of a language and how they are related. This is also known as abstract syntax.

Constraints bring additional constraints on the structure of the language, beyond what is feasible to express in the structure itself.

Presentation defines how instances of the language are represented. This can be the definition of a graphical or textual concrete language syntax.

Behaviour explains the dynamic semantics of the language. This can be a transformation into another language (denotational or translational semantics), or it defines the execution of language instances (operational semantics).

These aspects are not always as strictly separated as they seem in the illustration; constraints are shown as overlapping with structure, since constraints interact closely with the structure-related technologies to restrict the set of valid language instances. However, constraints can also be used for defining restrictions for presentations as well as behaviour.

Meta-models define the structure and constraints of a language. For a complete language definition, it is also necessary to define the presentation and behaviour, and relate these definitions to the meta-model, as explained in [2].

The structure is the core of the language; it contains the concepts that should be part of the language, and the relations between them. While traditional grammar-based compiler tools tend to focus on the presentation of the language rather than its structure, a meta-model-based approach to language design facilitates a focus on the structure. Starting from a well-defined language structure, it is convenient to define one or more textual and/or graphical presentations for the language, as well as to define code generation into executable target languages such as Java.

1.2 The Meta-model Architecture

A meta-model architecture or a meta-model hierarchy is a tree of models that are connected by “instance-of" or “conforms-to" relationships.

In the OMG four-layer architecture, every model element on each layer is strictly an instance of a model element of the layer above:

- *M3:* The meta-language for structure, MOF. (There is no M4 because MOF can be used to define itself)
- *M2:* Complete definition of all the aspects of a language.
- *M1:* Language instances (e.g. an UML diagram).
- *M0:* Data or runtime instances or real world objects.

It is common to use meta-models to specify the structure of a language, using existing meta-languages like MOF and Ecore, but they are not expressive enough to handle language aspects like presentation and behaviour.

The approach to the meta-model architecture within the modelling lab at the University of Agder, is based the premise that all aspects of a language should be defined specifically by using suitable meta-languages on the level above, as described in [3]. We see meta-languages as offering *interfaces* that languages on the level below can use, as shown in Figure 1.2.

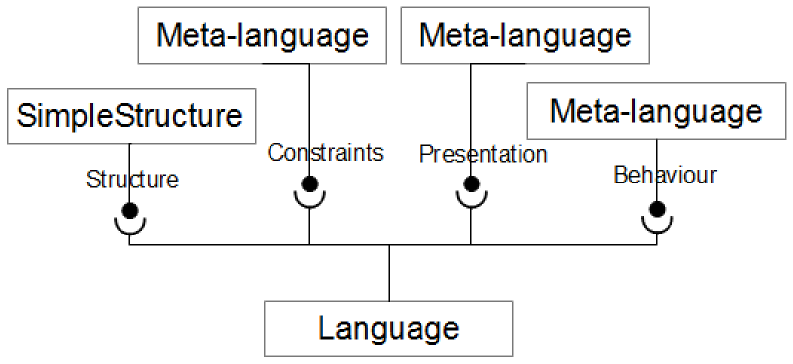


Figure 1.2: The architecture used by the modelling lab at University of Agder.

Thereby, we apply the notion that models can freely be promoted or denoted between levels depending on the intended use, and models in the meta-model architecture are relative to each other based on the relationship between them.

This understanding of the meta-model architecture, forms the basis of the concept of the LanguageLab module, where meta-language modules provide interfaces that language modules can use, and the language modules can again provide interfaces that allows them to be used as meta-language modules.

Chapter 2

The LanguageLab Platform

This manual is aimed at language developers that wish to use the LanguageLab computer language development workbench. LanguageLab is a complete environment for experiments with meta-model-based language specification. Version 1.1 is a prototype that supports some basic functionality.

The manual will describe the functionality of the system as planned. However, the early versions will not implement all of the described functionality. As far as possible, the not-yet-implemented functionality should be described in sufficient detail that it can be implemented at a later stage.

2.1 Design

The LanguageLab language workbench will allow the DSL developer operate on a suitable level of abstraction on all relevant language aspects, and facilitate making and modifying small example languages. In the following, the design for LanguageLab is described.

2.1.1 Language Modularity and Instantiation

The most fundamental functionality of the platform is to allow instantiation of modules. When modules are used as languages for defining a new module, they are also called *meta-modules*. A meta-module that supports structure, will allow creation of instances based on its instantiation semantics, through a simple built-in tree editor. A suitable presentation meta-module can be loaded, that will provide an editor that allows for a more user-friendly creation of a module. Then, constraints can be defined on the elements of the module with the help of a constraints meta-module. Finally, a behaviour meta-module can be added to allow defining semantics for the module.

Note that it is also possible to load more than one module for the same language aspect, to provide e.g. different views of the language (usually one presentation module will provide one editor) and/or support for different language features, like e.g. expressions or inheritance. However, for a given structure meta-module, it is necessary to take care that all meta-modules for presentation, constraints and behaviour either are created for this particular structure meta-module, or are adapted to it with suitable mappings.

When a module that has been developed based on the basic meta-modules is itself a meta-module, it can be promoted, or “moved up” a level in the meta-modelling hierarchy, by loading it as a meta-module and use its available interface, in order to create an instance of the meta-module that has been developed. If presentations for the meta-module have been created, those editors can then be loaded. When a new meta-module instance (module) has been created, one may want to execute it. If a meta-module supporting behaviour has been used, one can again “move up” the created module, and execute it in a debugger/runtime-environment.

Figure 2.1 shows how a meta-language for structure can be defined using different meta-modules including a version of itself, following the pattern shown in Figure 1.2.

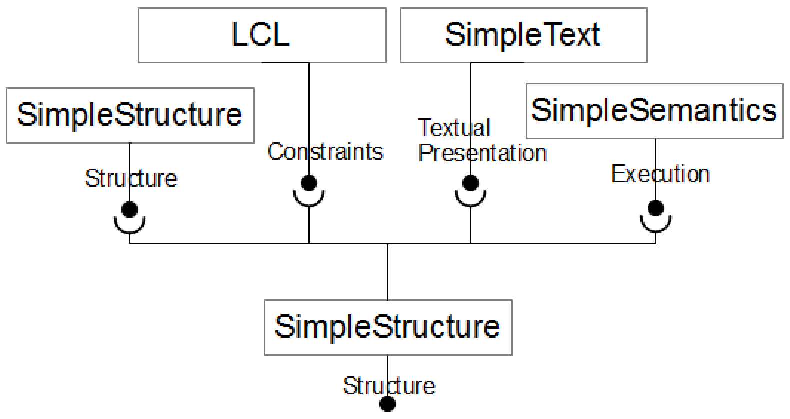


Figure 2.1: The LanguageLab architecture used for defining a meta-module.

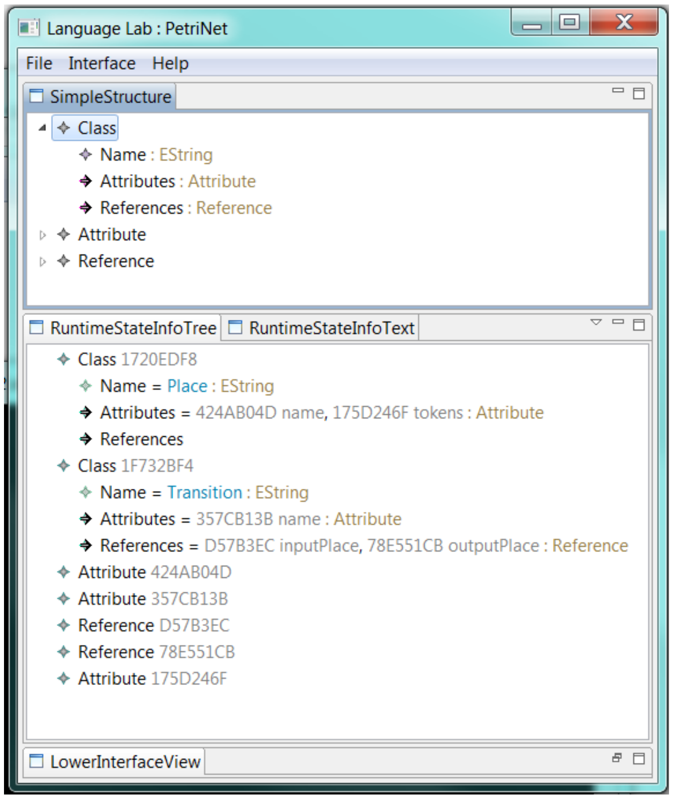
2.2 User Interface

The two main elements of the Language Lab Graphical User Interface are, from top to bottom:

Language Level (upper interfaces) contain the interfaces of loaded meta-modules. Each meta-module exists in a separate tab.

Model level contains the module being developed. The module is an instance of meta-modules loaded on the language level. The model level always includes the platform view (providing a simple tree structure) of the module being developed. Optionally, it may contain other presentations of the module in separate tabs, if the loaded meta-modules support it.

In addition, there is an optional *lower interface view*, for displaying the lower interface of a meta-moduile.



```
Class
    Name : EString
    Attributes : Attribute
    References : Reference
Attribute
Reference
```

```
Class 1720EDF8
    Name = Place : EString
    Attributes = 424AB04D name, 175D246F tokens : Attribute
    References
Class 1F732BF4
    Name = Transition : EString
    Attributes = 357CB13B name : Attribute
    References = D57B3EC inputPlace, 78E551CB outputPlace : Reference
Attribute 424AB04D
Attribute 357CB13B
Reference D57B3EC
Reference 78E551CB
Attribute 175D246F
```

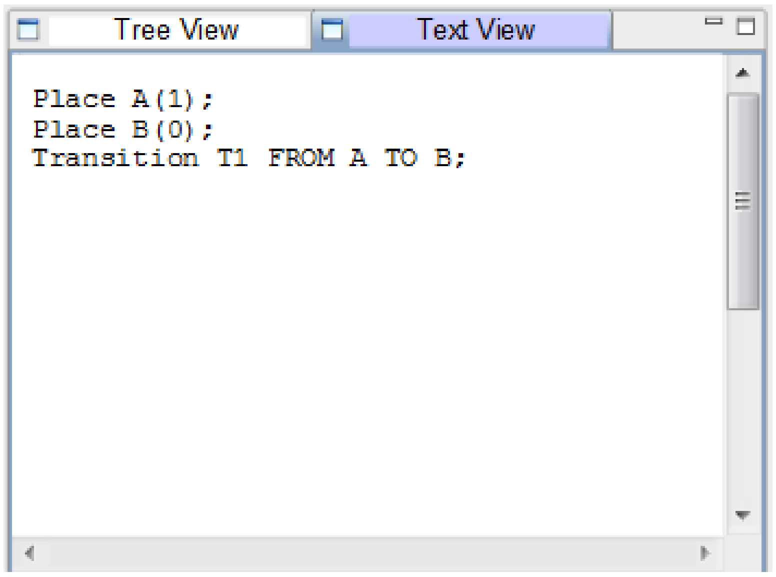
Figure 2.2: LanguageLab user interface.

2.2.1 Presentation

Figure 2.2 shows the two main parts of the LanguageLab GUI, the upper part is for meta-modules, and in this case it contains a meta-module interface for structure. The lower part is for showing the module being built from elements of the loaded meta-module(s). In this case, it is a simple petrinet module. In this illustration, the language instance view is based on a built-in tree view, the so-called platform view. It may display other views, editors, debuggers, code generators as tabs depending on the meta-modules used. The following

Figures 2.3 and 2.4 show possible interfaces for working in textual and graphical presentation modes.

LanguageLab allows users to switch between different presentation views of a language instance, that are automatically synchronised with the internal representation. Note that the view must be in a consistent state for successful synchronisation to take place. Future versions of LanguageLab will allow different presentations of a language to support preservation of extra information, elements of the presentation that are also present in some form in other presentations, but not in the structure, as described in [1]. This can be achieved by using a presentation extra-information module.



```
Place A(1);
Place B(0);
Transition T1 FROM A TO B;
```

Figure 2.3: Textual presentation of the language instance.

2.2.2 Constraints

Constraints may be added by using a suitable contraints meta-module (TBD).

2.2.3 Behaviour

There are menu items in the model view's menu to create new module elements, based on types from meta-modules. Basic create-commands are provided by the platform, but may be overloaded by customised versions from meta-modules. Types may also have operations that can be run from this menu, as shown in Figure 2.5.

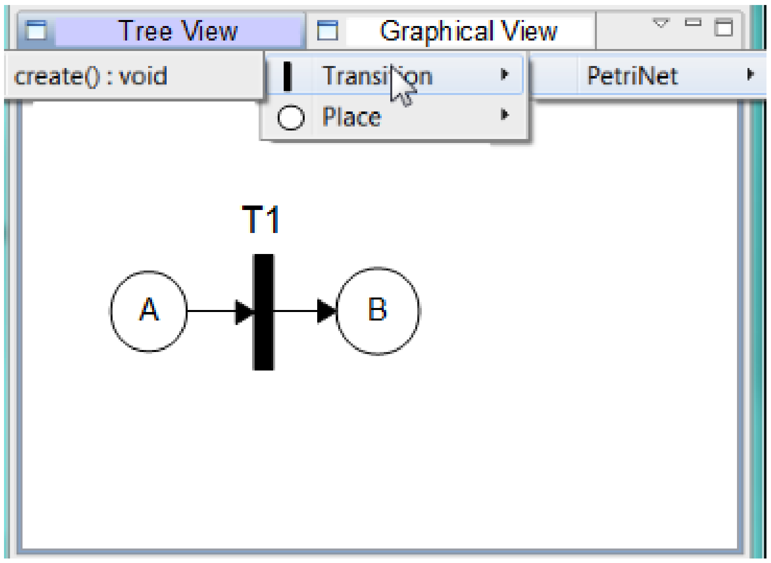


Figure 2.4: Graphical presentation of the language instance.

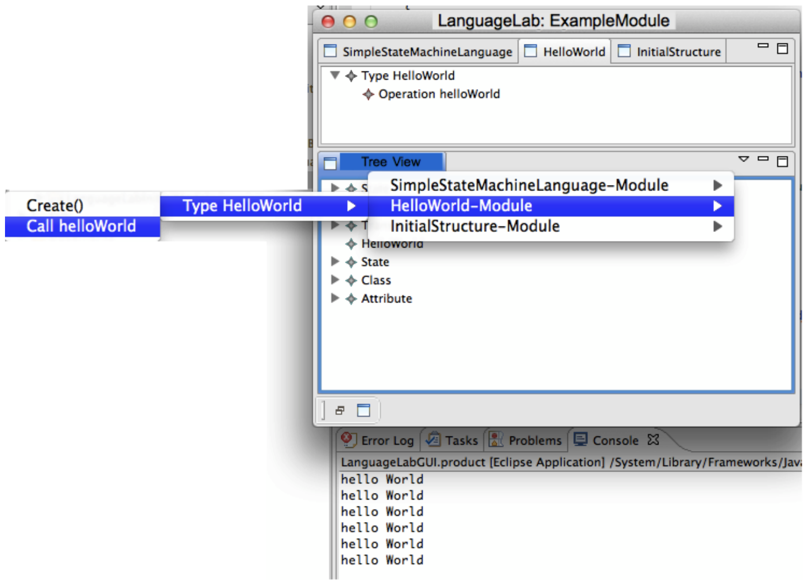
For executing a module, its semantics must be defined using a meta-module for that purpose, and then the module must be loaded as a meta-module for being executed. This is further described in Section 3.3.

Transformations are implemented by using a transformation meta-module for defining a transformation, and the result from executing the transformation on the module is a new transformed module.

Code generators may function either as template-based systems where the code generation is one-way, i.e. the code is read-only, or it may be a full editor that allows for editing the generated code and have the changes applied to the original language instance.

2.2.4 Module System

It is possible to combine elements from different meta-modules, as shown in Figure 2.6 where elements from three different meta-modules are used for one module.



```
LanguageLabGUI.product [Eclipse Application] /System/Library/Frameworks/Java
hello World
hello World
hello World
hello World
hello World
hello World
```

Figure 2.5: Calling an operation.

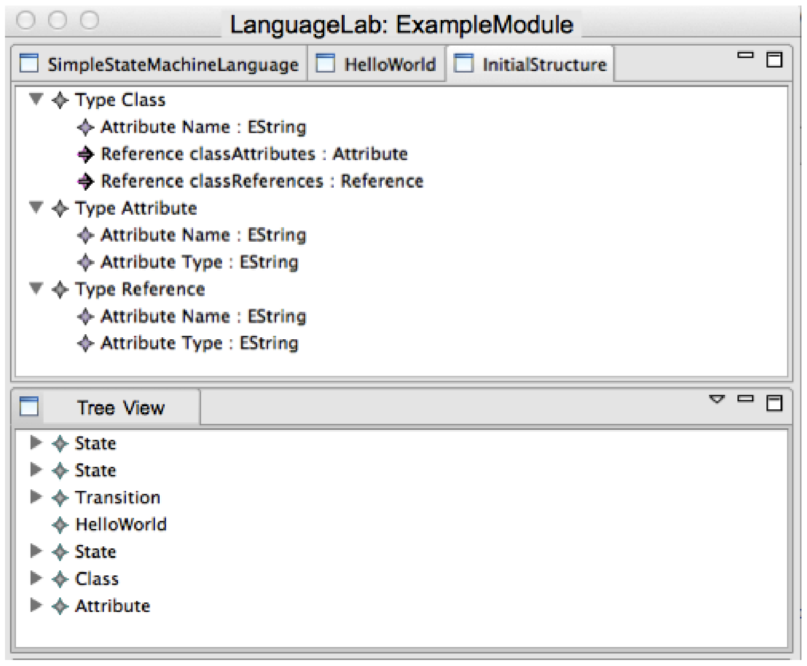


Figure 2.6: Combining elements from different meta-modules.

Chapter 3

A LanguageLab Use Case

In this part, concrete examples of LanguageLab usage are presented, to help the developer getting started with LanguageLab. We explain how to install LanguageLab, how to use it for creating a simple module based on an existing meta-module, how to execute the module, and finally how to create a new meta-module. Figure 3.1 shows the relation between the different modules and meta-modules described in this chapter; SimpleStructure (meta-module for structure), PetriNet (a petrinet meta-module), MyPetriNet (an instance of the PetriNet meta-module) and RT_MyPetriNet (runtime-instance of MyPetriNet).

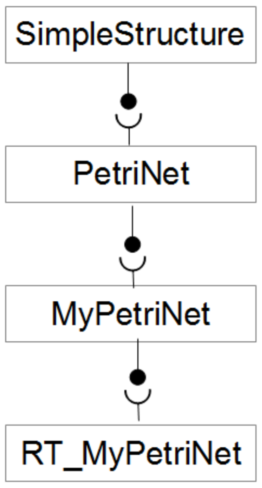


Figure 3.1: The (meta-) modules used in this chapter.

3.1 Installing and running LanguageLab

Download the LanguageLab platform with the initial set of LanguageLab modules. Unzip the downloaded file into a folder of your choice, and run the *LangugeLab* application. LanguageLab is a Java application, requiring Java to be installed on your system.

3.2 Creating a Module

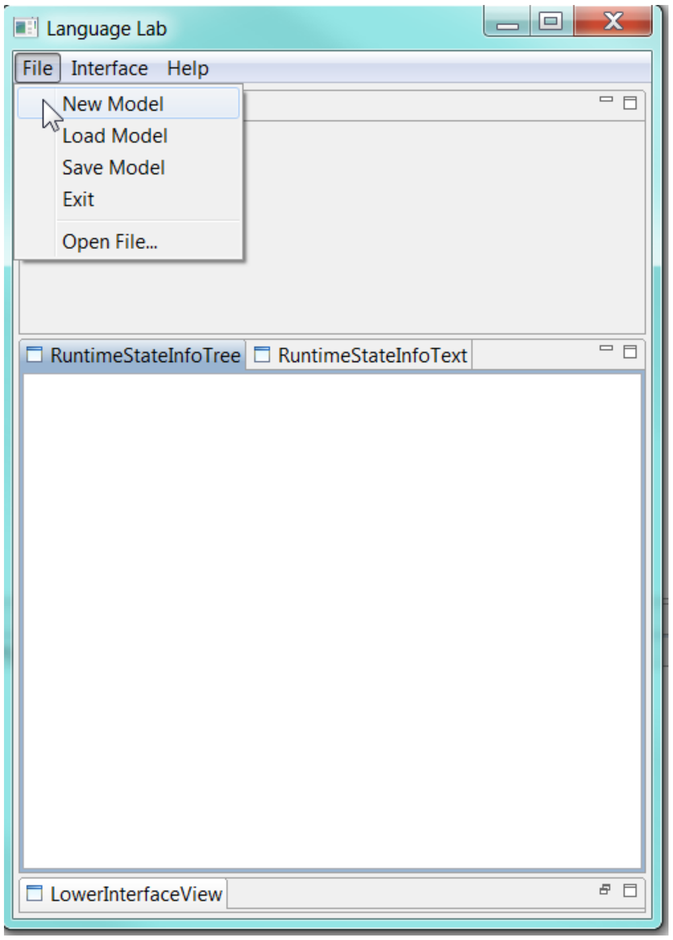


Figure 3.2: Creating a new module.

In this example, we will show how to create a simple module based on an existing meta-module in LanguageLab. The example language is a PetriNet meta-module.

From this meta-module, we will be able to create a PetriNet module with places, transitions and arcs.

We start by creating a new module from the File menu (see Figure 3.2):

File->New Model, Create New Folder.

Select the newly created folder, click OK.

After a new module is created, meta-modules can be loaded from the Interface menu.

Interface->Load Meta-model.

Select InitialModules/LanguageModules/PetriNet, click OK

Figure 3.3 shows how type instances can be created based on the types of the interfaces of the loaded meta-modules. If types contain operations, the operations can be run from the same menu. First create two *Place* elements by twice selecting the Create() operation for the type Place:

View-Menu (the small arrow marked with a rectangle in Figure 3.3) → PetriNet → Place → Create()

Then create a transition in the same way:

View-Menu → → Transition → Create()

When a type instance is created, the attributes and references are not set. By double-clicking on the item, dialog boxes for setting attributes and references will appear, to allow the language developer to set the values of these.

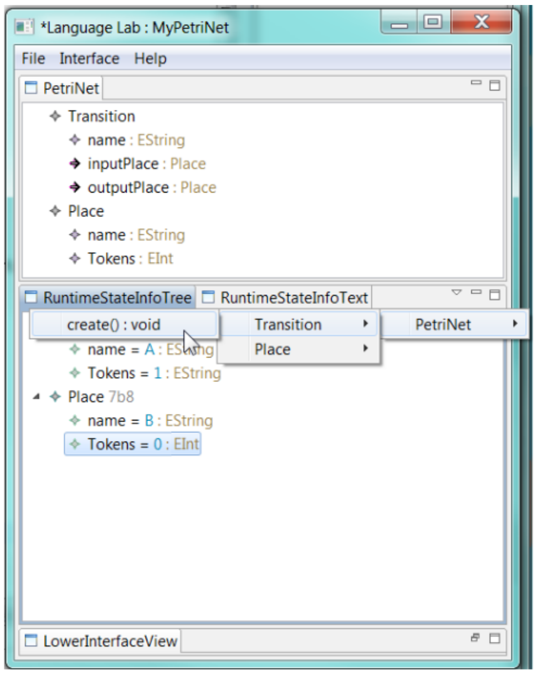


Figure 3.3: Creating module elements from the language interface.

Double click on the name attribute of the first Place, enter the name 'A' in the dialog box, Click OK.

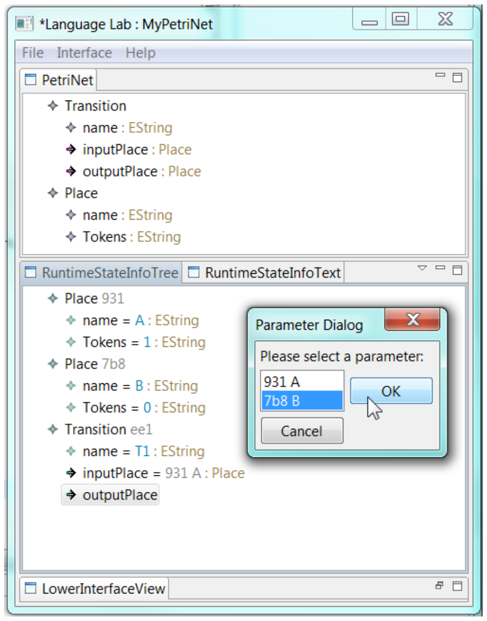
Double click on the name attribute of the second Place, enter the name 'B' in the dialog box, Click OK.

We now have two named places and a transition. Add one token to place A by double clicking on the tokens attribute and enter the number '1'.

Double click on the name attribute of the transition, enter the name 'T1' in the dialog box, Click OK.

Set Place A as inputPlace for the Transition: **Double click on the inputPlace reference of Transition T1, select the ID of the Place A in the dialog box, Click OK.**

Set Place B as outputPlace for the Transition: **Double click on the outputPlace reference of Transition T1, select the ID of the Place B in the dialog box, Click OK, as shown in Figure 3.4.**



```
Transition
    name : EString
    inputPlace : Place
    outputPlace : Place
Place
    name : EString
    Tokens : EString
```

```
Place 931
    name = A : EString
    Tokens = 1 : EString
Place 7b8
    name = B : EString
    Tokens = 0 : EString
Transition ee1
    name = T1 : EString
    inputPlace = 931 A : Place
    outputPlace
```

Please select a parameter:

```
931 A
7b8 B
```

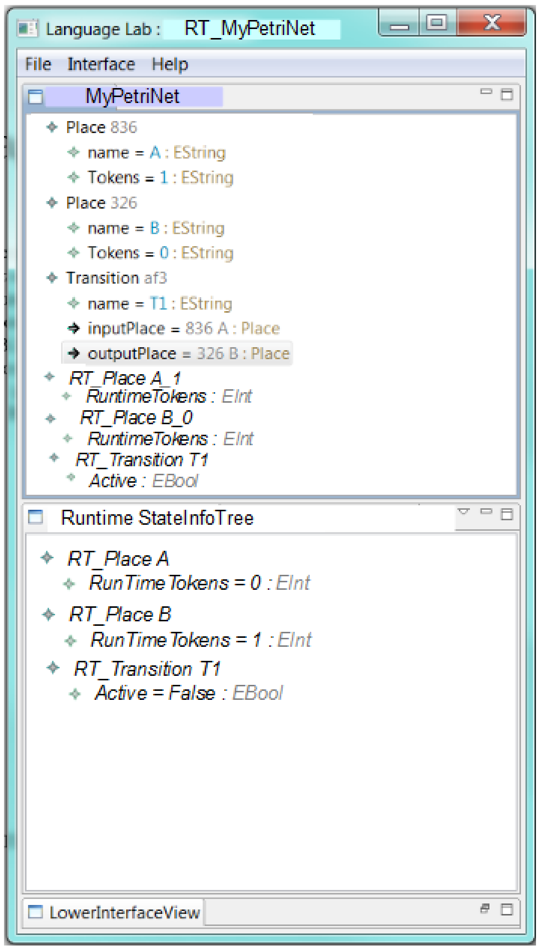
Figure 3.4: Dialog for selecting the target Transition of an Arc from a Place.

We may now want to call the operation **PrepareToRun()** to prepare the module for execution, as described in the next section. We now have a complete model, and may save the module for later use:

File->Save Model

3.3 Executing a Model

The execution requires the module to be loaded on the language level. Then its runtime operations can be executed, e.g. run(), step(), reset(). Execution/debugging (based on operational semantics) and code generation will usually be displayed in a separate tabbed view. Debugging is shown in the form of an extended version of the platform view showing the stepwise state of a language instance during runtime. Start, stop and reset operations as defined by the PetriNet meta-module are called through the Runtime View menu.



```
Place 836
    name = A : EString
    Tokens = 1 : EString
Place 326
    name = B : EString
    Tokens = 0 : EString
Transition af3
    name = T1 : EString
    inputPlace = 836 A : Place
    outputPlace = 326 B : Place
RT_Place A_1
    RuntimeTokens : EInt
RT_Place B_0
    RuntimeTokens : EInt
RT_Transition T1
    Active : EBool
```

```
RT_Place A
    RunTimeTokens = 0 : EInt
RT_Place B
    RunTimeTokens = 1 : EInt
RT_Transition T1
    Active = False : EBool
```

Figure 3.5: platform-view debugger.

The debugger will show the runtime-structure in the platform view, with access to values of runtime variables, as shown in Figure 3.5. Runtime elements are marked with italics font.

3.4 Creating a New Language with LanguageLab

Using the SimpleStructure meta-module, we define the structure of PetriNet in the same way as we defined the module above, adding Place and Transition, with the above described attributes and transitions.

If we want to use a module as a meta-module, we may open it as a meta-module from the interface menu. For this to be meaningful, the lower interface of the module has to be populated.

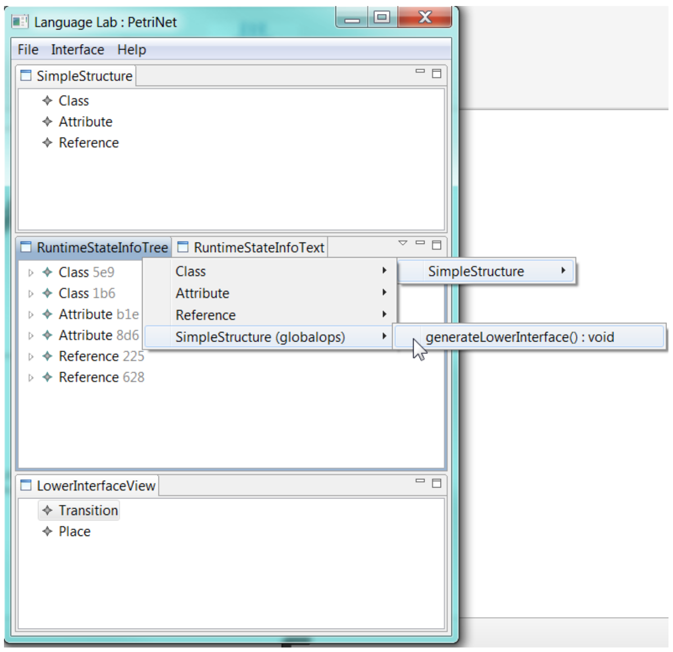
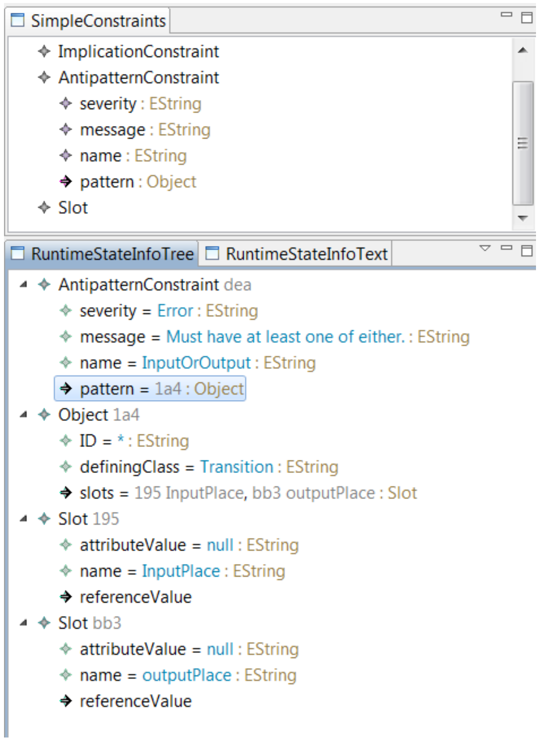


Figure 3.6: Generating a lower interface based on language module semantics.

Figure 3.6 shows how the SimpleStructure module may be used to define the structure of the PetriNet language and generate a lower interface offering place and transition elements that can be used by other modules, thereby allowing it to be used as a meta-module. The lower interface is expanded in this figure, for illustrative purposes.

Note that if we want to be able to execute a module, we also have to add execution semantics and a runtime environment to its meta-module, or we may alternatively transform it into another form that we are able to execute. Textual or graphical presentations and constraints may also be added to the module.

Constraints for PetriNet like “A transition must have at least one inputPlace or one outputPlace” may for example be expressed using the SimpleConstraints module antipattern constraint as shown in Figure 3.7.



```
ImplicationConstraint
AntipatternConstraint
    severity : EString
    message : EString
    name : EString
    pattern : Object
Slot
```

```
AntipatternConstraint dea
    severity = Error : EString
    message = Must have at least one of either. : EString
    name = InputOrOutput : EString
    pattern = 1a4 : Object
Object 1a4
    ID = * : EString
    definingClass = Transition : EString
    slots = 195 InputPlace, bb3 outputPlace : Slot
Slot 195
    attributeValue = null : EString
    name = InputPlace : EString
    referenceValue
Slot bb3
    attributeValue = null : EString
    name = outputPlace : EString
    referenceValue
```

Figure 3.7: A constraint for PetriNet.

Chapter 4

Implementation Details

The starting point for the LanguageLab application is an Eclipse/EMF-based tree-view model editor, generated from an Ecore model of the proposed LanguageLab module format. The EMF-based editor plugins were extended with a front-end carrying a full graphical user interface, implemented as a standalone Eclipse RCP application, enabling the language developer to use it independently of Eclipse. Although RCP applications do not carry the weight of the full Eclipse workbench, they can still build on all available features of Eclipse, such as the JFace widget toolkit, the plug-in architecture of Eclipse, and EMF, allowing a relatively rapid development of the LanguageLab platform.

4.1 Structure and Design of LanguageLab Modules

As shown in the model of the LanguageLab in Figure 4.1, the `Module` may provide an interface where the types that are supported by the module are defined. It may also have an interface bound to the provided interfaces of other modules and thereby use them as meta-modules. The `Interface` consists of `Types` that can have `Attributes`; supporting basic built-in types: `EBoolean`, `EString`, `EInt`. `Reference` refers to typed objects. `Operation` elements may have operation implementation and code (Java class files).

This implements the architecture described in Section 1.2 by allowing a module to both use interfaces from meta-modules as well as offer interfaces for other modules to use. This facilitates not only a module per language aspect, but also other variants such as modules supporting particular language features that can be used as building blocks, as a starting point for creating a partly customised DSL with some stock features.

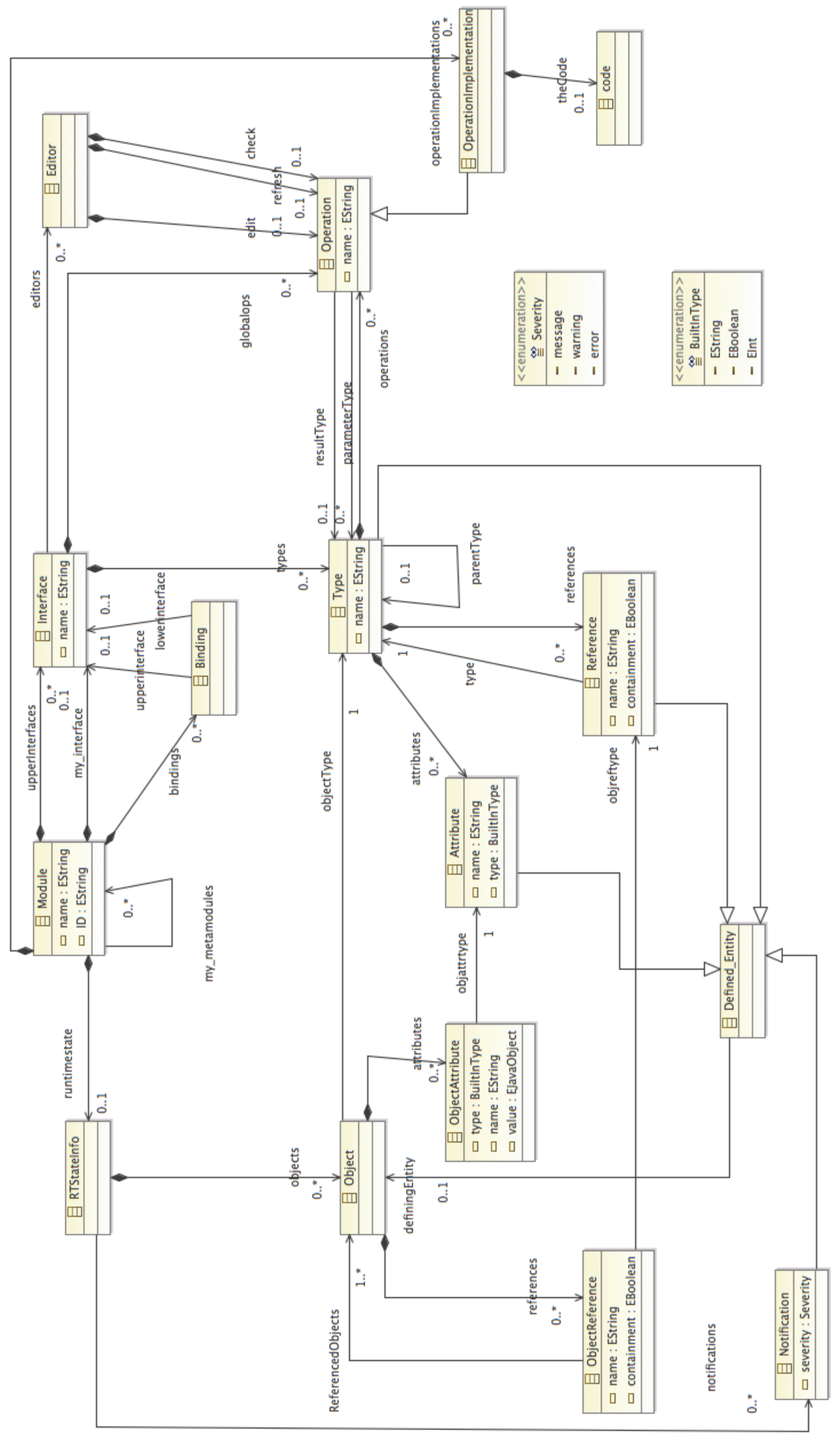


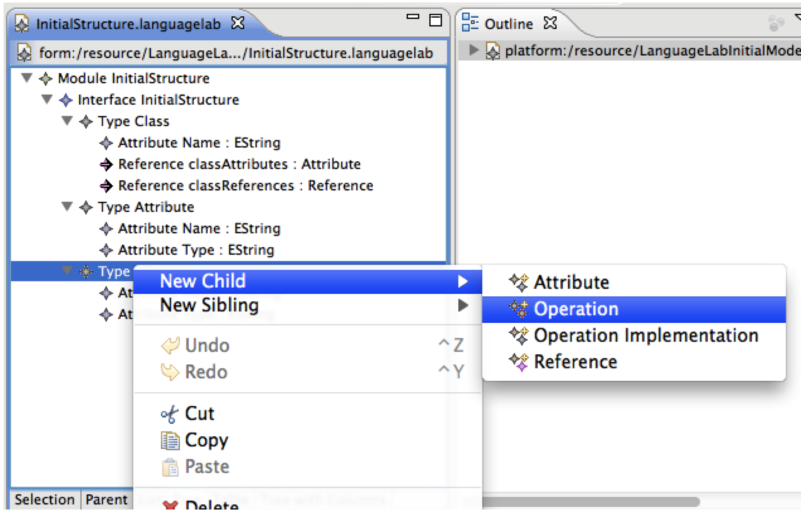
Figure 4.1: The Language Lab Ecore model

4.2 Internal Representation Language

The LanguageLab platform provides a basic representation language that is derived from the LanguageLab module model described above, with `Type`, `Attribute`, `Reference`, and `Operation`. This representation language is supported by the platform view, allowing any type of LanguageLab module to be displayed.

4.3 Manual Module Creation

The Eclipse/EMF-based tree-view model editor, generated from an Ecore model of the proposed LanguageLab module format, was used for creating some basic meta-modules for initial bootstrapping and testing of the system, as can be seen in Figure 4.2.



InitialStructure.languagelab
form:/resource/LanguageLa.../InitialStructure.languagelab
Module InitialStructure
Interface InitialStructure
Type Class
Attribute Name : EString
Reference classAttributes : Attribute
Reference classReferences : Reference
Type Attribute
Attribute Name : EString
Attribute Type : EString
Type

Outline
platform:/resource/LanguageLabInitialMode

Figure 4.2: EMF-based LanguageLab module editor

For the purpose of bootstrapping, operations can be added in the form of java classes to the module. When they are available, the operations can be executed from the module menu. This method may be used for defining the behaviour of a language. However, for more proper and developer-friendly behaviour handling, a behaviour DSL module allowing for defining behaviour on a higher level of abstraction should be used.

Chapter 5

Initial Set of LanguageLab Meta-Modules

5.1 SimpleStructure

SimpleStructure is a simple basic module that allows for creating simple module structures. It contains the object-oriented basics with classes, attributes and references, but not inheritance in this initial version. Figure 5.1 shows the module loaded as a meta-model in LanguageLab, offering `Class` with `Name`, and it also has references to its member `Attributes` and `References`; in addition there are the `Attribute` and `Reference` themselves, each with `Name` and `Type`.

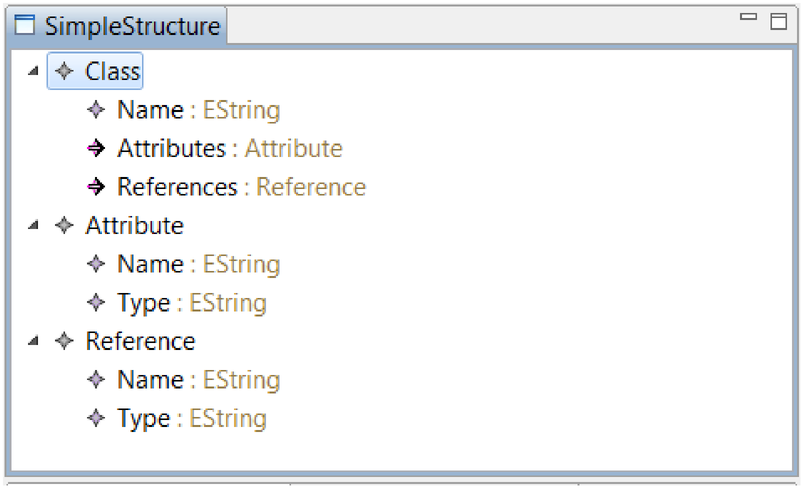
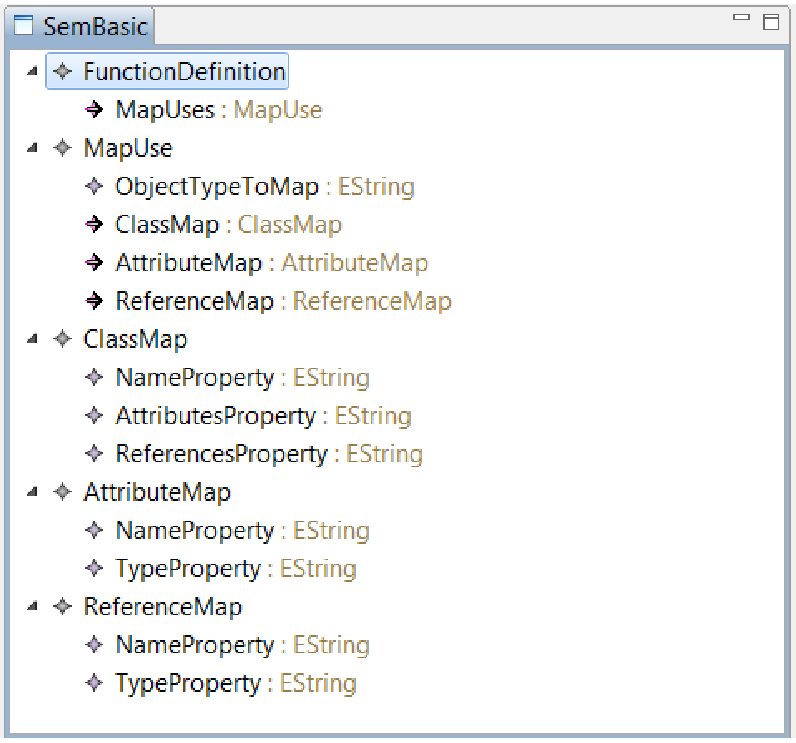


Figure 5.1: The SimpleStructure meta-language interface.

5.2 SemBasic

SemBasic is a simple (work-in-progress) module for defining instantiation semantics of a language by providing a mapping between a given structure (e.g. SimpleStructure) and the platform module structure given in Figure 4.1 above, thereby facilitating the creation of a new language interface from elements of a structure model. Figure 5.2 shows the interface of the SemBasic module that contains FunctionDefinition, MapUse, ClassMap, AttributeMap and ReferenceMap. The function makeSem() is used for generating a java version of the modeled mapping function that may be used with a Structure meta-language like SimpleStrucrure for creating a lower interface based on a structure.



```
SemBasic
◢ FunctionDefinition
    MapUses : MapUse
◢ MapUse
    ObjectTypeToMap : EString
    ClassMap : ClassMap
    AttributeMap : AttributeMap
    ReferenceMap : ReferenceMap
◢ ClassMap
    NameProperty : EString
    AttributesProperty : EString
    ReferencesProperty : EString
◢ AttributeMap
    NameProperty : EString
    TypeProperty : EString
◢ ReferenceMap
    NameProperty : EString
    TypeProperty : EString
```

Figure 5.2: The SemBasic meta-language interface.

5.3 SimpleText

Not yet implemented.

5.4 SimpleGraphics

Not yet implemented.

5.5 SimpleInstance

SimpleInstance allows for defining an instance structure with `Objects` that have `definingClass`, and a set of `Slots`, as shown in Figure 5.3. The reason for `Slot` having both attributeValue and referenceValue, is that it was created using a meta-module that did not support inheritance, and this option was chosen for accomodating slots with either attribute nature and reference nature.

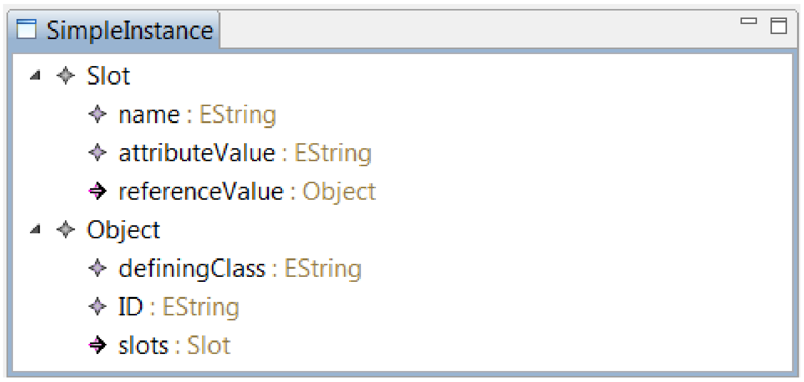


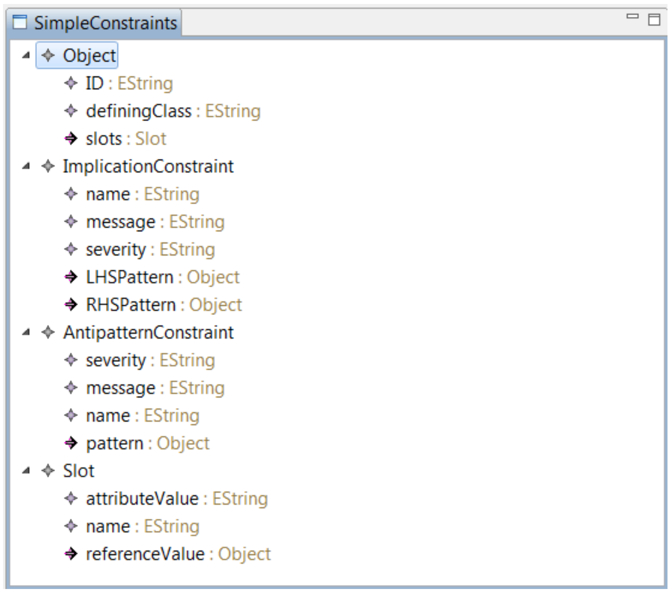
Figure 5.3: The SimpleInstance meta-language interface.

5.6 SimpleConstraints

The meta-module for constraints is a (work-in-progress) module that allows one to add constraints that are connected to elements of a module. The current version of SimpleConstraints extends SimpleInstance and allows to define two types of constraints; AntiPatternConstraints that defines a model-instance pattern that is not allowed.The other option is the ImplicationConstraint, where the precence of one pattern, implies the existence of another pattern. Both constraint types use a SimpleInstance-like way of defining the pattern, and allow the setting of severity level and message to give in case the constraint is broken. The SimpleConstraints module generates a checkConstraints function to use for checking the defined constraints.

5.7 SimpleTransform

Not yet implemented.



SimpleConstraints

- Object
  - ID : EString
  - definingClass : EString
  - slots : Slot
- ImplicationConstraint
  - name : EString
  - message : EString
  - severity : EString
  - LHSPattern : Object
  - RHSPattern : Object
- AntipatternConstraint
  - severity : EString
  - message : EString
  - name : EString
  - pattern : Object
- Slot
  - attributeValue : EString
  - name : EString
  - referenceValue : Object

Figure 5.4: The SimpleConstraints meta-language interface.

Bibliography

[1] Terje Gjøsæter and Andreas Prinz. Preserving non-essential information related to the presentation of a language instance. In *Proceedings of NIK 2009*, 2009.

[2] Anneke Kleppe. A language is more than a metamodel. In *ATEM 2007 workshop*, 2007. Available at http://megaplanet.org/atem2007/ATEM2007-18.pdf.

[3] Liping Mu, Terje Gjøsæter, Andreas Prinz, and Merete Skjelten Tveit. Specification of modelling languages in a flexible meta-model architecture. In Ian Gorton, Carlos E. Cuesta, and Muhammad Ali Babar, editors, *ECSA Companion Volume*, ACM International Conference Proceeding Series, pages 302–308. ACM, 2010.

[4] Jan Pettersen Nytun, Andreas Prinz, and Merete Skjelten Tveit. Automatic generation of modelling tools. In Arend Rensink and Jos Warmer, editors, *ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2006.