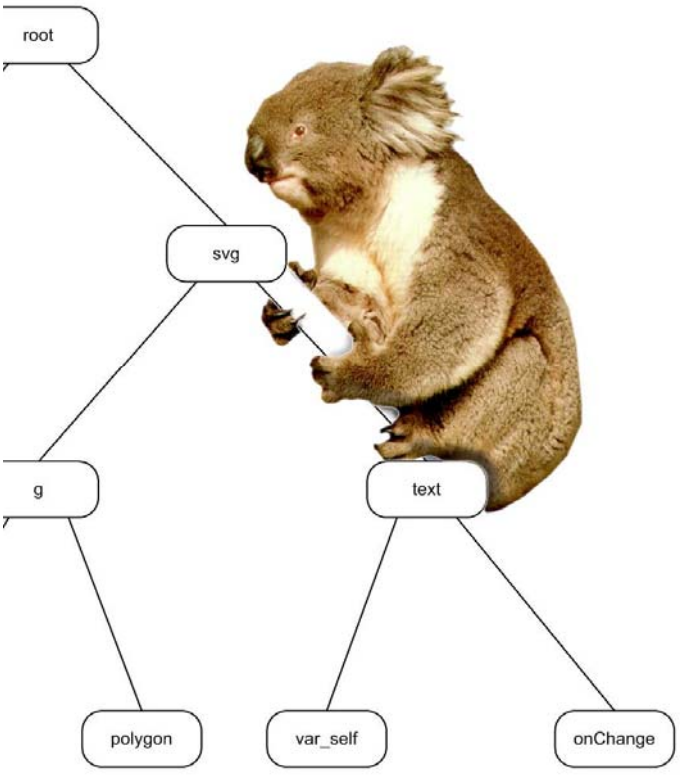
ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Semester project in Computer Science

Synchronization between display objects and representation templates in graphical language construction



Rohrer Fabien
Helg François

February 2006

Advisor : Fondement Frédéric

1. Abstract

To define a language, it is necessary to describe both its abstract syntax and its concrete syntax. A concrete syntax is a "semantically rich" model whose representation is clearly established. Nevertheless, practice shows that synchronizing the abstract syntax and the concrete syntax, or synchronizing the concrete syntax with its representation, requires a lot of work. This project concentrates on defining a graphical concrete syntax for a language, if the abstract syntax is given. We implemented a system capable of keeping synchronized the abstract model of a certain language with its concrete representation. We have based our work on a tool called ProBXS to implement our solution. This tool has been created to display a certain graphical language based on SVG templates and to offer the possibility to the user to interact with it. In this paper, we will expose as a user manual the way to use our tool. Of course we will also deal with theoretical issues, but as the motivation of our advisor was to get a working tool at the end of the project, we have mainly focused our attention on describing the features of our tool implements and the way to use them.

2. Table of contents

3. Table of figures

4. Introduction

4.1. Context

This document is the report of our semester project we've been working on during the first semester of our Master program in Computer Science at the EPFL. We've been assisted by Frédéric Fondement from the LGL laboratory (*Laboratoire de Génie Logiciel – Software Engineering Laboratory*) at the EPFL.

4.2. Generality

This document is both a report of our work and activities during the semester and a user manual written for potential users and future developers. Indeed, we present in the first part of this document a formal and theoretical description of the problematic and an explicit list of the technologies involved in this project. Then, we explain how we have implemented our solution from a technical point of view.

The second part is devoted to the user's manual. We present in that section all the features we have implemented to ProBXS (the tool we used as a basis for our work) and how exactly to use them.

Of course we will conclude this document with a conclusion, explaining what we have learnt, where we had difficulties and a list of potential improvements a future student or developer could make.

4.3. Model-Driven Development

As J2EE and other technologies get more complicated and integration between various technologies (EJBs, web services, databases, etc.) becomes more and more important, it is obvious that things are no longer manageable at the code level. A model-driven approach to software development is needed. OMG recognized this fact, thus the *Model-Driven Architecture* (MDA) – a framework of standards that enable model-driven development – became its mainstream vision. Since MOF is a key component in this framework, implementation of MOF becomes an essential component of MDA tools. As the metadata for all languages an IDE supports are stored in one place (in an MOF repository such as MDR), it is easier to build transformation/synchronization modules between, for example, UML models (capturing an application logic at a higher level of abstraction) and the technologies the application should be deployed to – Java, JSPs, EJBs, database, etc.

5. Problematic

5.1. Description of the problematic

5.1.1. Introduction

In this section, we will present in details the problematic of our project, which is the synchronization between abstract representation (model) and concrete representation of a given language. We will first describe and explain the procedure to define a new graphical language. It will make you aware of some essential terms and concepts involved in this problematic.

We don't pretend to be experts (and we are not!) in language construction, so please read the following chapter as a reminder that gives you the basis to understand the problematic and not as a paper on language construction.

5.1.2. Define a new language

To create a new language, the designer needs to define both its abstract and concrete syntax. The abstract syntax can be seen as a model that the derived language has to follow. The abstract syntax captures the concepts of the language. In the context of XML, a DTD or an XMLSchema plays the role of the abstract syntax.

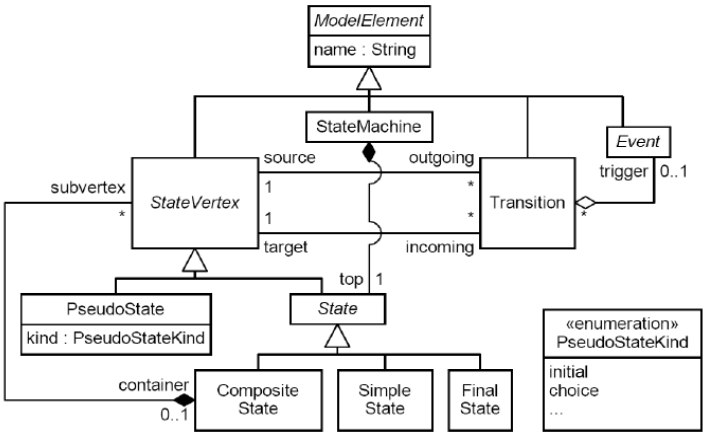


Figure 1: Abstract syntax of the statechart language

Then, it is also required, for a graphical language, to define the "notation" or the representation of the language. This is captured by the concrete syntax.

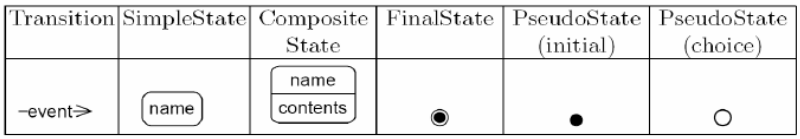


Figure 2 : Concrete syntax of the statechart language

This separation between abstract and concrete syntax is a technique that enables us to define the concepts of the language avoiding taking care of its representation.

5.1.3. Instantiation of a language

Once both the concepts and the graphical representation of the language are set, it is possible to create an instance of the language. It is interesting to see that this time we also get two different views of this specific instantiation. One of them is the "model", which is the abstract representation of the language, containing nevertheless all the semantic (see Figure 3) and the other one is the concrete representation (see Figure 4) based on the graphical templates, defined by the concrete syntax.

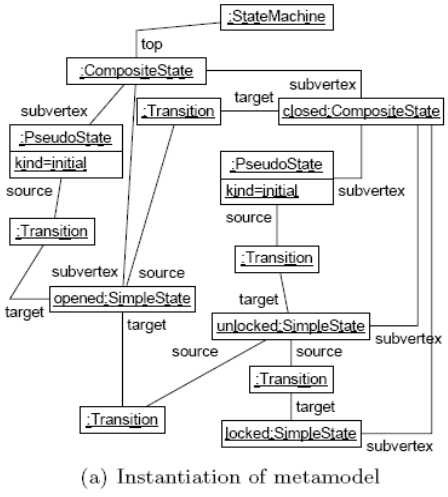


Figure 3 : A specific model of the statechart language

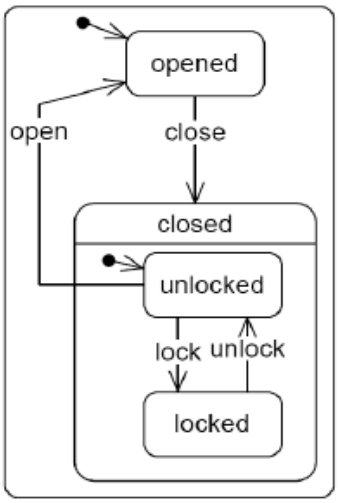


Figure 4 : Concrete representation of the model above.

5.1.4. Description of the problematic

If you have read the previous chapter, you must remember that there are two different ways to represent an instance of a given language, its abstract representation (model) and its concrete representation. Imagine that the user or the designer changes something in the model or in the graphical representation. For example he changes the name of a state in the statechart language. Our work now is to keep synchronized both these representations! It means that if something changes in one representation (in the previous example the name of a state), the other representation needs to change at the same time to keep synchronized!

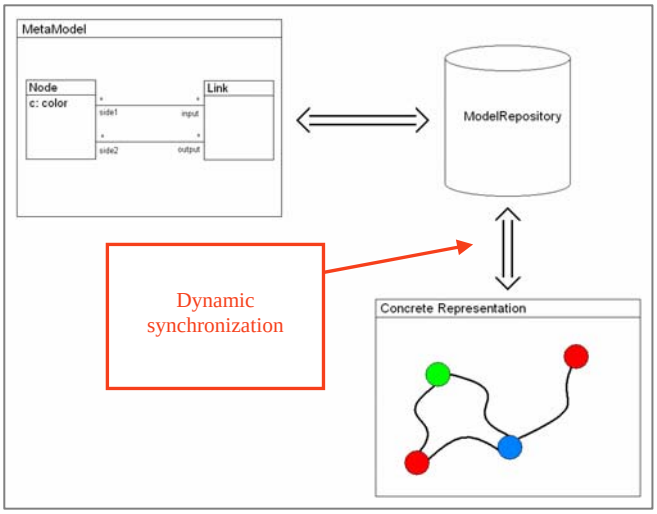


Figure 5 : View of the problematic

As our work is mostly practical work, we don't really have anything to add from a theoretical point of view. You will find what our project is really about in the following section.

5.2. Brief description of the project

This project is focused on adding new features to the existing tool called ProBXS (a brief description of this tool can be founded in the next chapter). We have basically added a way to represent and store the model and keep it synchronized with the graphical representation built with ProBXS. All the technical details can be found in chapter 7: Solution and resolution.

5.3. ProBXS

ProBXS is a tool coded by Fabrice Hong and Frédéric Fondement which provides an environment to create and handle the graphical representation of a given language. The graphical representation is based on SVG templates which are defined by the creator of the language and the dynamic behavior is provided by the implementation of the DoPIdom architecture (More information on [DPI]).

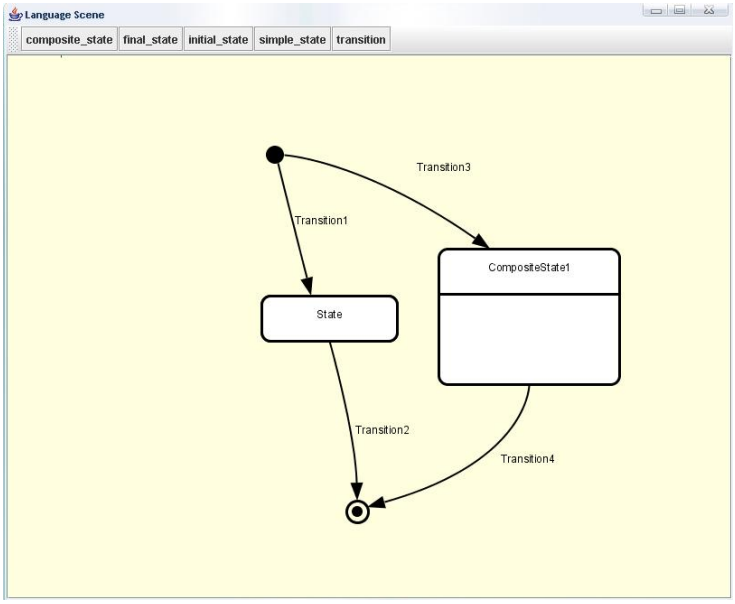


Figure 6 : snapshot of an utilization of ProBXS

6. Technologies used

This chapter contains a brief description of all the technologies which are used in our project. If the description is not sufficient, the chapter, in any case, contains a link to a web site to obtain more information.

6.1. XML

XML (e**X**tensible **M**arkup **L**anguage) is a W3C standard. It is a very simple and flexible text format and is used as a base to create specialized languages. XML language syntax is based on mark-ups that form structured data and can be used to represent any kind of concept. (More information on [XML])

6.1.1. Objective and utility

The initial objective is to facilitate the exchange of structured data on the Web. The main goal is to separate the data and its representations. Thus XML stores document data and increases portability between systems thanks to its format which is in UNICODE. There exist many tools on the Web that can load, import, manipulate and save XML.

XML also provides a support to define new languages in an easy way. Nowadays there exist many of these derivated languages that we call XML dialects. They follow formal syntaxes which are defined by DTD (Document Type Definition) or an XML Schema.

6.1.2. Functioning

The character encoding is defined in the first statement of the document, by default UTF-8 is used, which is a particular transcription of UNICODE.

The document is structured in Elements that are defined by means of start and end mark-ups. Elements can nest other elements. The whole set of elements in the document are contained in a unique element called “root”. Apart from elements, XML documents contain different kind of data:

- Comments (that are not part of the data)
- Processing instructions
- “Character calls” (to represent characters that don’t exist in the used encoding
- “entity calls” (kind of text macros)

6.2. SVG

SVG (Scalable Vector Graphics) is a XML language which describes vectorial graphics or graphical applications in two dimensions. It is used to display vector graphic on the Web and has more or less the same capacities of definitions than Macromedia Flash. (More information on [SVG])

6.2.1. Advantages

- open source
- resolution independent / bandwidth
- text based
- support transparency
- can be read by many tools
- standard

We will use this standard because it is well known, and that means that a lot of free resources can be found giving the possibility to edit, display and interact with it. One of the main characteristics that differentiates ProBXS from other diagram tools is that the representations are fully customized by the user. So with SVG, the user will be able to design his own component with the tool that he wants. Moreover, whereas vector graphics similar to Flash store their documents in binary, SVG stores them in human readable/modifiable texts.

6.2.2. Quick Example

The following lines provide a simple example of SVG utilization:

```
>    <?xml version="1.0" encoding="UTF-8"?>
>    <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
     "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11-flat-20030114.dtd">
>    <svg xmlns="http://www.w3.org/2000/svg" width="500" height="500"
     xmlns:xlink="http://www.w3.org/1999/xlink">
>      <g>
>            <rect width="100" height="100" fill="black"/>
>            <text x="20" y="50" style="font-family:Arial; fill:red">
>                  Hello world
>            </text>
>      </g>
>    </svg>
```

This code produces the following graphic:

Figure 7 : Example of a SVG graphical representation

6.3. DOM

To parse and manipulate the XML structure of SVG, an advanced tool is needed. Basically, two toolkits are well known: SAX and DOM. The latter offers an easily handled tree interface that represents XML structure. For this project, the use of a tree structure to represent the SVG components and their hierarchy is clearly unavoidable. For this reason and because of the numbered advantages that DOM offers over SAX, the project will be partly based on DOM. (More information on [DOM])

6.3.1. Using DOM

At first the application has to load the SVG document using the parser. It will then build a tree of elements called nodes. There exist many different kinds of nodes, for example:

- Text nodes
- Attribute nodes
- Element nodes
- Comments nodes

The nodes are given methods by means of interfaces in order to be accessed, or to navigate through the tree. This allows applications to handle XML quite easily.

6.3.2. Advantages of DOM

- Robust and complete API for manipulating the tree.
- Relatively simple to modify the data structure and extract data.
- Isolated DOM nodes we want to deal with are isolable and can be used as a vector of information between classes.
- Each component of the tree is represented by classes. This allows to subclass and therefore to wrap DOM nodes to run other mechanisms on the tree structure.

6.4. DoPIdom

DoPIdom is a java implementation of the DPI model based on DOM and SVG. The aim of the DPI model (Documents, Presentations, and Instruments) is to provide an alternative to current application-centred environments by introducing a model based on documents and interaction instruments.

DPI makes it possible to edit a document through multiple simultaneous presentations. The same instrument can edit different types of content, facilitating interaction and reducing the user's cognitive load. DPI includes a functional model, aimed at the user interface designer, which describes implementation principles in terms of properties, services and representations. The DPI model offers a first but essential stage in designing and implementing a new generation of document-centred environments based on a new interaction paradigm.

DoPIdom propose an architecture in which Components and behaviours are defined separately. (More information on [DPI])

6.4.1. DoPIdom and SVG

SVG is a good base to work on, but the problem is that it's by far not sufficient just to describe and to work with graphical concrete syntax that represents instances. Interactive and dynamics components are needed, but SVG is static!

It is possible to define some simple interactions with the SVG standard, but it will not provide the complex one required by the graphical edition of a language.

DoPIdom wraps XML/SVG elements represented in the DOM tree, in a new kind of component which can consume or produce actions or queries that will modify or obtain information.

6.5. Batik

Batik is a java toolkit for applications or applet that want to use images SVG format for various purposes, such as viewing, generation or manipulation.

In our case we will need the parsing functionalities, and the renderer module which use a swing derivated canvas that can display SVG components.

There are some other functionalities like a SVG generator that translate the usual graphic interface of swing in SVG language, or even a browser, which is like a little application to load SVG documents.

When actions modify the DOM tree, Batik updates the display immediately.

For more information, see [Bat].

6.6. Technologies for constructing metamodels

6.6.1. MOF

MOF (MetaObject Facility), an adopted OMG standard, is a specification which provides a metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems. With MOF, we can import and export models, handle, store and load a repository, etc. A lot of technologies (like XMI and JMI) are based on it. (More information on [MOF])

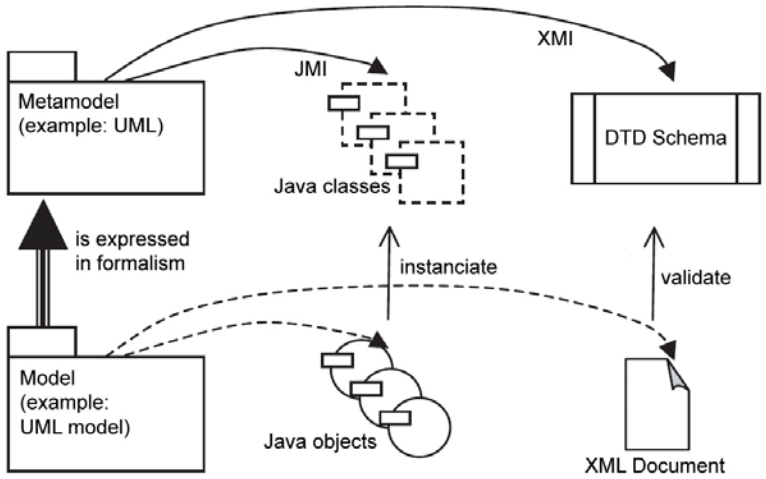


Figure 8 : This diagram shows JMI and XMI mapped the MetaModel and the Model

6.6.1.1. XMI

XMI is the OMG's XML-based standard format for metamodel storage and transmission. XMI has an expression power comparable to UML. We can see on Figure 8 how the XMI technology maps the model and its model. More information on the XMI specification web site [XMI]

6.6.1.2. JMI

Official definition: ”The JMI (Java Metadata Interface) specification enables the implementation of a dynamic, platform-independent infrastructure to manage the creation, storage, access, discovery, and exchange of metadata. JMI is based on the Meta Object Facility (MOF) specification from the Object Management Group (OMG), an industry-endorsed standard for metadata management. The MOF standard consists of a set of basic modelling artefacts described using UML. Models of any kind of metadata (called metamodels) can be built up from these basic building blocks. JMI defines the standard Java interfaces to these modelling components, and thus enables platform-independent discovery and access of metadata. JMI allows for the discovery, query, access, and manipulation of metadata, either at design time or runtime. The semantics of any modelled system can be completely discovered and manipulated. JMI also provides for metamodel and metadata interchange via XML by

using the industry standard XML Metadata Interchange (XMI) specification. More information on the JMI specification web site [JMI].”

6.6.2. MDR

As its name suggests, MDR is a metadata repository. Because it implements MOF, it is able to load any MOF metamodel (description of metadata) and store instances of that metamodel (the metadata conforming to the metamodel). Metamodels and metadata can be imported into/exported from MDR using XML that conforms to the XMI standard. Metadata in the repository can be managed programmatically using the metamodel-specific or reflective JMI API.

6.7. Technologies to communicate with metamodels

6.7.1. Dynamic Java : Koala

Koala is a dynamic Java source interpreter. It is completely free and distributed with its Java sources (see [Koa]). With this tool we can interpret scripts when wanted during the execution of another program.

Koala interprets scripts written in DynamicJava. DynamicJava is a derived Java. So they have some differences (These differences come directly from the web site of Koala):

- Statements and expressions can be written outside classes, in the top-level environment.
- The variable declaration is optional. When the left part of an assignment is an unknown identifier, a variable is defined. The type of this variable is the type of the right part of the assignment.
- The dynamic casts are optional.
- The *package* clause can be used anywhere in the top-level environment to set the current package. The syntax of this clause has been extended : writing *package*; set the current package to the anonymous package.
- C-like functions are supported in the top-level environment. The syntax used to declare a function is the same as the one used to declare a method. The method modifiers (public, static, ...) and the *throws* clause are ignored. Functions can only be used in the top-level environment and in the body of other functions, including itself.
- Anonymous classes defined in the top-level environment can contain references to top-level environment's final variables.
- Inline comments beginning with '#' are allowed.

We can run Koala as a standalone application and this is very useful for the requirements of our project.

6.7.2. Kermeta

Kermeta is a metamodeling language which allows describing both the structure and the behaviour of models. It has been designed to be a common basis to implement Metadata languages, action languages, constraint languages or transformation language as seen in Figure 9.

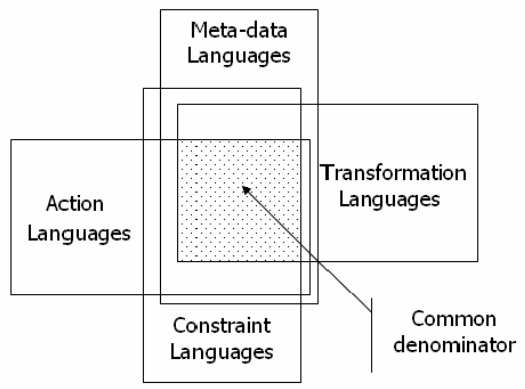


Figure 9 : Kermeta positioning

Kermeta is free and open source, it's adapted to our program, it's powerful and it hides the concept of the repository. Therefore we had reasons to choose it. Nevertheless we haven't chosen it for the final implementation. More details in chapiter 7.2.4.

7. Solution and resolution

7.1. Architecture of ProBXS before our work

7.1.1. General architecture

This chapter is a summary of Chapter 3 (Architecture) of [PBX]

In order to program the behaviour of the component we needed an object oriented programming language. For this we chose to develop the project in a Java 1.5 environment.

ProBXS is not a toolkit itself but completes the DoPIdom proposed architecture; that is to say, it adds features and new components. Three main modules run together to make the interaction with SVG components possible:

- Batik to display SVG graphics
- DOM to handle XML/SVG
- DoPIdom to handle interactions with SVG

7.1.2. DoPIdom architecture

Here will be presented the basic functioning of DoPIdom and additional features and concepts that have been added.

The way to create behaviour for SVG element is quite simple with DoPIdom. First of all, a subclass C1 of *Component* has to be created. In the constructor of this class, interfaces that the component must compose are decelerated.

Finally the SVG element must be assigned the wanted class (possible package must appear) with the attribute: *dpi:component*.

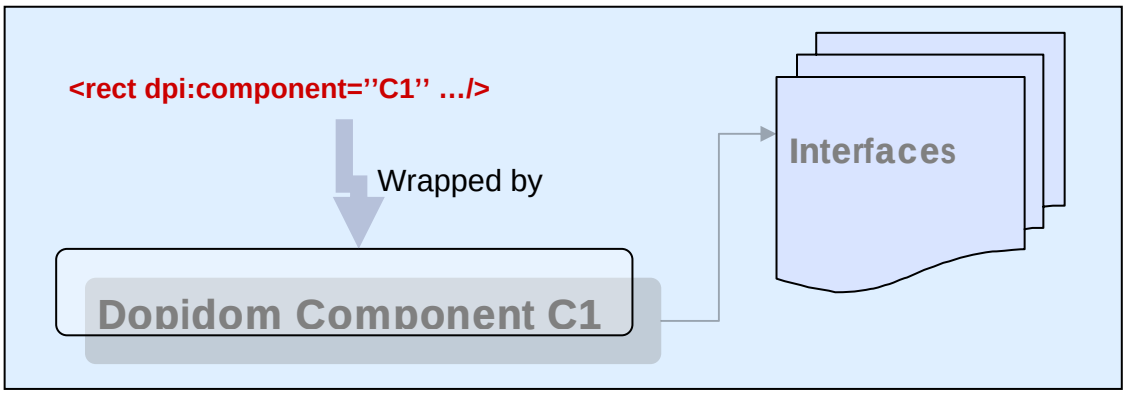


Figure 10 : This figure shows that a component is constituted of some interfaces and that a component attribute called in an SVG template corresponds to a concrete Java Component.

7.1.3. Components

In the modelling language context, every concept is represented by SVG elements that can be wrapped by *Component*s to be given behaviours. We can point out two kinds of behaviours for the components:

- The sharable behaviours : the Actions and Queries (consumed by interfaces)
- The behaviours inherent to the function of the component that are the methods of the component class

Components data :

The state of a DoPIdom component must be able to be completely defined by SVG. Once a component is loaded, the interactions with the components change the data; moreover additional data have to be taken into account.

Afterwards, the entire information about the component has to be stored in SVG. Two kinds of data can be distinguished:

- The Component Definition Data

  Mandatory data about the properties of the object or about the initial state. For instance, the color, the width, the other object that will be used as sub parts (ex: Link Arrow)

- The Class Instance Data

  Optional data that represent the state of some class variable. For instance a number, a list of attached component, the component on which an arrow is sliding.

7.1.4. Interfaces, actions and queries:

Interfaces consume actions or queries. Interface classes define the behaviour whereas actions or queries class are just used as container to send data or get data from interfaces.

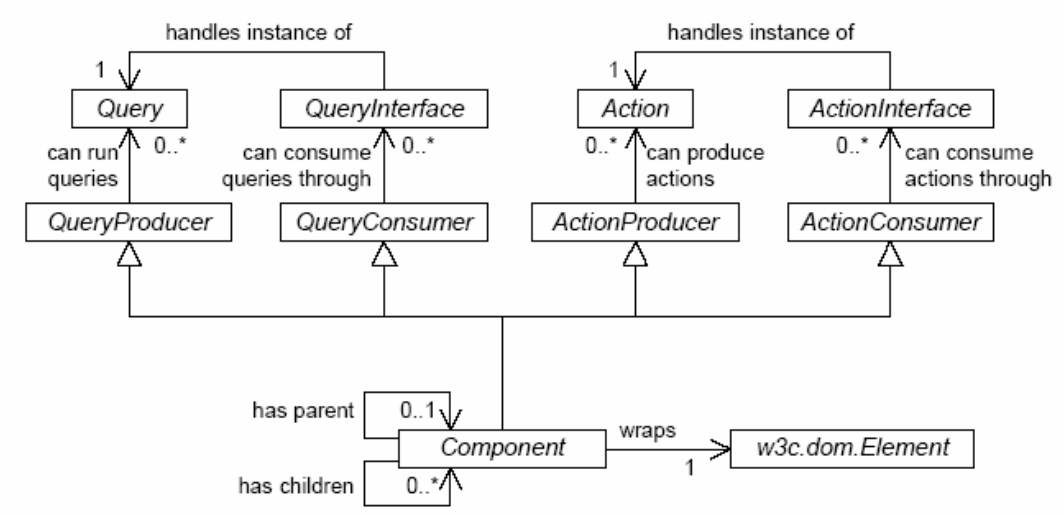


Figure 11 : Architecture of DoPIdom (components, interfaces, actions and queries)

7.1.5. Interactions

In the continuation of ProBXS project, toolbars will be present somewhere in the scene to allow users to choose which kind of cursor they would like to use. To each type of cursor would be associated a certain role, in other words, the ability to trigger specific actions or queries on consumer components.

SVGInteractions :

A SVGInteraction is a class that associate production of queries of actions to mouse triggers such as *mousePressed(), mouseReleased(),* etc. The cursors are related to the SVGInteractions the same way as components are related to interfaces. In the constructor of *SVGCleanToolInteractor*s (cursor), possible SVG interactions are defined.

7.2. Architecture of ProBXS after our work

In order to make the synchronization between the graphical representation and the model, we had to consider different technical aspects.

- Implementation of a model repository constrained by a given meta-model
- Add new instructions into SVG templates which will be interpreted by the “model-synchronization interpreter”.
- Implement the synchronization itself

7.2.1. Implementation of the repository

We had several possibilities to implement the concept of model repository. As a first idea we wanted to use a specific language called Kermeta to realise this task (it offered an interesting level of abstraction, even avoiding the concept of repository), but we had as a need to interpret the code dynamically and unfortunately the environment wasn’t set to offer this possibility (see 7.2.4). We have thus used another architecture based on JMI, MDR & dynamically interpreted Java code. MDR is the repository [MDR], JMI is used as a metamodel and Java is used to realise the synchronization (this last aspect will be covered below).

So when we store the current scene (Ctrl + S), the program generates two files, the model in XMI and of course the scene in SVG.

7.2.2. Add new instructions into SVG templates

Adding new instructions into SVG templates is the way we have found to make the model aware of what it will have to do when a semantically rich change occurs in the graphical representation. In order to do so, we have added into the nodes of SVG templates, from where the corresponding changes occur, attributes containing variables that describe the environment for the considered event and the code that the model will have to interpret when the event is captured.

Here is a more detailed explanation of the global architecture:

- An event is linked with a certain component (stickedEvent is linked with AnchorPoint component, CharacterInsertedEvent is linked with EditableString component, etc.).

- A SVG node (DOM node when the component is added to the scene) can be wrapped by a dpi:component. This feature allows us to exactly know where to put the code into the template. For example if we consider the CharacterInsertedEvent, we know that we will have to add attributes into the node which is wrapped by the component linked with this event, which is EditableString!

```
>    <g dpi:component="test.CompositeContainer" <!-- … -->>
```

- Two kinds of information have to be added into the node

    - Code to interpret, describe in a specified language. Our project allows user to use several languages even in the same SVG template. (see 8.2)

```
>    onEvent="{ Language | codeToInterpret }"
```

    - Variable to set the environment. The content of the variables can change during the execution of the program. For example, the content of the var_self variable will change after the instantiation and will take as a value Object(#uniqueId). Further explanation will be given in 8.3.

```
>    var_self="$s"
```

Here is the final template for an EditableString component.

```
<text onChange="{ Java | self.setName(content); }" var_self="$s" id="123"
    fill="black" dpi:component="test.EditableString" cursor="text" text-
    anchor="middle" y="0" x="0">newString</text>
```

A complete explanation of how to add new instructions conformed to our architecture can be found in the user's manual section.

7.2.3. Implementation of the rising synchronization

This task can be subdivided into a chain of simpler tasks:

- Put listeners on semantically rich events coming from ProBXS and get interesting parameters (new name, new position, new color…).

- Get the script to interpret from the dom-tree.

- Map the variables and set the environments

- Interpretation of the script and update of the model

- Get the environment from the interpreter and update the DOM-tree if needed

7.2.3.1. Listeners on semantically rich events

The rising synchronization starts from a semantically rich change in the graphical representation. In practice, this means that we have to find out where the change comes from and how to capture it and its interesting parameters. A good programming pattern is the listener pattern to achieve this task, and we have indeed implemented a list of listeners listening for semantically rich events! Here is a list of all considered events:

- CharacterDeletedEvent, CharacterInsertedEvent: Appear when a character is changed in a textbox
- ComponentCreatedEvent: Appears when a new component is added into the scene
- DirectionAdjustEvent: Appears when a component changes its direction
- LocateEvent, PositionEvent, TranslateEvent: Appear when a component moves on the scene
- ResizeEvent: Appears when a component is resized
- ContainedEvent: Appears when a component is slided into another component
- StickEvent: Appears when a component is stuck to another one.

The role of the listeners is therefore to capture these events with their parameters. The interaction between ProBXS and the model will be made in the “receiveEvent()” method of the considered listener. For example, let’s imagine that in the statechart language the user decides to change the name of a state. When he effectively changes the text in the textbox, the listener responsible for that event captures the event and treats it in its receiveEvent() method.

Here is a typical code (receiveEvent() method from the considered listener) to treat an event: In this example (CharacterInsertedEvent)

```
>    public void receiveEvent(Event e) {
>
>            // Get the event. (Allow us to access to its parameters)
>            final CharacterInsertedEvent cie = (CharacterInsertedEvent)e;
>            HashSet<String[]> listSpec = new HashSet<String[]>();
>
>            // Set a variable “content” which contains the new text
>            String[] v1  =  { "content",
>                ManipString.createSerialization ("String", cie.getNewText ())};
>
>            listSpec.add(v1);
>
>            // Call a function which set & call the right interpreter
>            ListenerFunctions.foo ("onChange",
>                (Node) cie.getTextInsertedComponent().getWrappedElement(),
>                listSpec);
>
>    }
```

7.2.3.2. Get the script to interpret and the environment from the dom-tree

We have described in section 7.2.2 and in the user’s manual how to encode instructions into SVG templates. The problem now is that we want to get this information in order to interpret it.

Considering that during the execution of the program, we only deal with a DOM tree (and no longer with SVG files), we only need to find a way to reach the right node within the tree. It can be easily done because we know which component raises the event and therefore we can have direct access to its wrapped element where all the information is encoded.

7.2.3.3. Map the variables and set the environments

Once we have all the information from the DOM tree and from the parameters of the event (special variables), we need to deserialize the information in order to make it understandable by the chosen interpreter!

```
>    onChange="{ Java | self.setName(content); }"
```

The interpreter is chosen by reading the name before the ‘|’ character. In the example, the Java interpreter will be used. Then, everything will be re-encoded according to that language (variables and expression). For example, var_isSource = “Boolean(false)”, will be replaced by Boolean isSource = new Boolean(false), before the interpretation.

(An explanation of how we can retrieve which element in the model corresponds to its graphical representation can be found in chapter (Influence on our architecture 7.2.4.3).

7.2.3.4. Interpret the script and update of the model

The interpreter does its job; it interprets the script linked with the considering event. In our implementation, it modifies the XMI file contained in the MDR repository. At the end of the interpretation, the model is synchronized with the graphical representation and the environment is sent back to ProBXS in case that some attributes have to be updated in the DOM tree.

7.2.3.5. Get the environment from the interpreter and update if needed the dom-tree

For some events, it can occur that some variables contained in the DOM tree have to be updated. So we recover the output variables of the interpretation and replace all the corresponding local variables (attributes of both the current node and its children recursively). We have to serialize the variables before storing them in the DOM tree. It will help us to recognize both objects in the future. To illustrate this procedure, see 8.3 How to use variables.

7.2.4. Storing and loading a scene

One of the main feature of ProBXS before our work was to store and load a specific scene. We have kept this feature and add the possibility to save the corresponding model.

7.2.4.1. Save a scene and the corresponding model

When the user uses the store command (Ctrl + S); ProBXS stores the current scene in a SGV file. We have added the following feature: store the corresponding model in a XMI file. This model is updated during the execution of the program by using the rising synchronization described above.

7.2.4.2. Load a scene and its corresponding model

When the user uses the load command (Ctrl + L); ProBXS loads the SVG file, which becomes a DOM-tree and the corresponding model. In theory, the loaded scene can keep the synchronization with its model even after a store and load manipulation.

7.2.4.3. Influence on our architecture

During the execution of the program, the synchronization between a graphical component and its abstract representation in the model is established by using a hashmap containing as a key the refMofId (unique ID during the execution of the program) of the object in the model and as an element the object itself. The trick is that we also store the key in the DOM tree during the creation of the component. In order to do so, we change the value of the variable representing the self value by: Object(#refMofId). For example var_self = “Object(.000000318)”. Thus, this architecture allows us, when a change occurs in the graphical representation, to retrieve which object will change in the model.

The problem is that if we store the scene and the model, we loose all the dynamic context (the hashmap in particular) and we need to find a way to keep the synchronization even in a “static” context (when both the model and the scene are

saved). We thought that the refMofId would gave us a real unique id (not just for a single instantiation) allowing to ensure the synchronization in any case, but this attribute was less powerful than what we expected (the id is unique just in the context of its instantiation) and at this state of the project, we haven't find another way to implement this feature and therefore it is only possible to store a scene with its model a single time. (No loading possible!)

7.3. Problems with Kermeta

In theory Kermeta would have been in charge of all the difficult operations of our project.

- Write the metamodel in Kermeta and generate his corresponding ecore metamodel.
- Handle the repository model which is linked with the metamodel.
- Insert scripts into the attributes.
- Run it as a standalone application

Chapters 7.3.1 and 7.3.2 show how we thought of using Kermeta at the beginning of our project and chapter 7.3.3 describes why we have abandoned this technology.

7.3.1. Creating model and metamodel

The following code shows how we can create a metamodel (displayed in Figure 12) in Kermeta :

```
>    require kermeta
>    using kermeta::standard
>
>    class Node
>    {
>      atribute c :  color
>      reference input : set Link[0..*]#side1
>      reference output : set Link[0..*]#side2
>    }
>
>    class Link
>    {
>      reference side1 : set Node[0..*]#input
>      reference side2 : set Node[0..*]#output
>    }
```

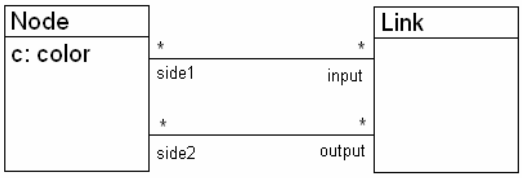


Figure 12 : simple example of a metamodel

The next step is to transform this metamodel into an Ecore file because Kermeta needs to have it for handling the model. This transition is simple because it belongs to the eclipse environment; we must only select the corresponding tool and the transformation Kermeta2Ecore is done automatically.

To finish, we can instantiate the Ecore metamodel in an empty XMI model using “creating dynamic instance” tool. Like Kermeta2Ecore transformation tool, it belongs to the eclipse environment.

Once the model is instantiated, using the following commands, we can load the model, handle it and store it:

```
> var statemachine : StateMachine
> var repository : EMFRepository init EMFRepository.new
> var resource : EMFResource init repository.createResource("./model.xmi",
  "./metaModel.ecore")
> resource.load
> statemachine ?= resource.instances.one
> // handle the statemachine model
> // …
> resource.saveWithNewURI("./newModel.xmi")
```

7.3.2. Handling the model

We wanted to use Kermeta queries like we use Koala queries, that is:

```
> onCreation="{ Kermeta | var t : Transition init Transition.new }"
```

The advantage with Kermeta is that we don’t need to care about the model repository. It’s the internal Kermeta interpreter problem.

To do these queries, we have to handle the Kermeta scripts during the running of our program. So, like with Koala, we have to have a standalone application to integrate Kermeta in our project

7.3.3. Problems

We have spent a lot of time to render the environment of Kermeta utilizable, but in each step of this work we have found other annoying problems. These problems are principally due to the youth of this language. You will find in the list the main problems that we have met:

- Bugs in the transformation Kermeta2Ecore

  When we want to use the Kermeta2Ecore transformation, some bugs appear in the resulting Ecore file. First of all, in the Kermeta file, there is no way to specify the name space of the resulting Ecore file. This name space is indispensable to load the metamodel in a Kermeta load instruction. So we have to introduce it by hand. A second problem is that the *reference* key word is understood like the *attribute* key word. So the expression power is reduced because we can’t modelize certain link of a UML model. For these reasons, the transformation Kermeta2Ecore is not safe and therefore not usable. But

these bugs are signalled to the Kermeta developers and will be most likely corrected on the next versions.

- No standalone application

  As said before, the possibility of a standalone version of the program is indispensable for running our project. Kermeta actually does not offer this possibility. Kermeta Developers are actually working on a standalone version that is not finished yet. When we noticed this lack, we tried to manipulate Kermeta sources to render it able to interpret scripts, but there are to many complicated dependencies in Kermeta; we didn't find a conclusive way to do so.

- Problems in saving the model

  A big problem was that the saving of a model was capricious. Sometimes, without apparent reasons, we couldn't easily store a model, and that was annoying because it was the true aim of our project.

- Detection of fault not enough precise

  We spent a lot of time resolving small bugs in Kermeta scripting because the error detection was not precise. Sometimes, the error detection contented itself with detecting a fault without saying where the error was.

- Lack of documentation

  At the beginning of the project, we spent a lot of time learning the specifications of Kermeta because the documentation was too short and obsolete (the documentation was written for an older version).

Considering all these problems and the lack of time we had to do this project, we decided to abandon the implementation of Kermeta and focalize ourselves on Koala. It was nevertheless a pity to abandon Kermeta because this language was (and is still) promising, but it was simply too young yet for our utilization.

8. User Manual

8.1. How to create a new language

To define a new complete language, user needs to describe some specific files. All these files have to be declared in the same directory. The following snapshot shows the directory of the statechart language:

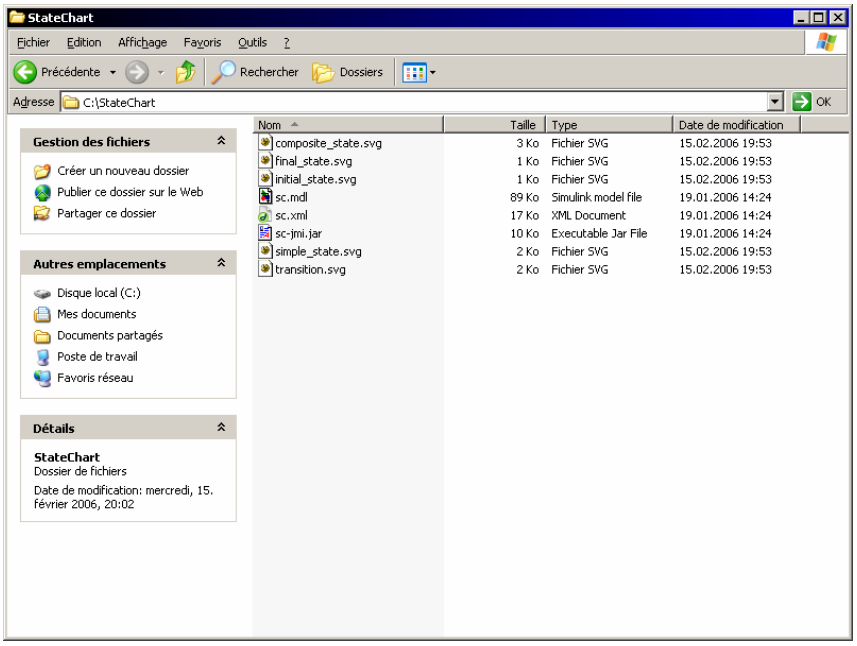


Figure 13 : Snapshot of the repertory contains our statechart language.

A new language is defined by:

- The SVG templates

  These files represent each object of the concrete representation in SVG. They capture the “notation” of the language (see 5.1.2). Thanks to the DoPIdom architecture, these templates can have a dynamic behavior. Specific attributes can define some new properties to the parts of the objects such as render it translatable or allow a string to be editable. User can also define other properties to communicate with the metamodel and the model. The following chapters explain how to do so.

- The metamodel

  The metamodel is written in XMI for practical reasons. There are a lot of tools which allow easy creation of a metamodel in XMI. When the metamodel is created, user has to generate the corresponding metamodel in JMI (Java interfaces allowing handling a derived model). This metamodel allows an easy communication with the program which is written in Java.

8.2. How to use the Java Interpreter

Our project allows user to write some code into the SVG templates. This code is interpreted by a specific interpreter and allows handling the model. The form of this attribute is:

> onEvent="{ Language | codeToInterpret }"

The event is called when the specific interface is used. For example, the component EditableString has the interface CharacterInsertable which can be handled by the attribute onChange. The list of all the possible components and interfaces is described bellow.

> **onChange**="{ Language | codeToInterpret }"

To call the dynamical Java interpreter, we can replace "Language" by "Java", "java", "Koala" or "koala". Example:

> onEvent="{ **Java** | codeToInterpret }"

With this interpreter, we can write almost any code which is by the standard Java. This interpreter allows even certain liberties. For example it's not necessary to declare the type of the objects. All useful things are implemented: casting, management of packages, all the standard types … But the aim of this report is not to enter into the details of Koala. Here is an example of possible code but to have more information see the official web site of Koala [Koa].

> onEvent="{ Java | **import java.math.*; c = new Double(Math.sqrt(2)); System.out.println(c.toString());** }"

Look out; we remind that there is some interdict characters in the XML value of an attribute. These characters have to be defined differently otherwise they enter in conflict with the XML specifications. User has to change the following characters:

Character	Corresponding character in SVG
&	&amp;
<	&lt;
>	&gt;
"	&quot;
'	&apos;

So, an intuitive code likes:

> onEvent="{ Java | System.out.println ("Hello world!"); }"

has to be change like following :

```
>    onEvent="{ Java | System.out.println ( &quot;Hello world! &quot; ); }"
```

Each event (interface) has some specific internal key words. These special variables allow user to handle specific inaccessible values. For example, the interface CharacterInsertable has the key word "content" which contains the content of the string. So the following code will display the new content of a string:

```
>    onChange="{ Java | System.out.println(content); }"
```

A very important key word is "model". This key word is present in all interfaces. It allows the user to have the model repository (the "instanciation" of the metamodel).

```
>    onChange="{ Java |  model.getState.setName(content); }"
```

8.3. How to use variables

Our project allows the user to define some variables as SVG attributes. These variables are useful in a lot of cases. They don't depend on the interpreter. The aim of a variable is to be used into a code to be interpreted. That furnishes a bigger expression power. This chapter explains how to use variables.

A variable has either the form:

```
>    var_varName = "type(value)"
```

or:

```
>    var_varName = "$name"
```

First we explain the first form. This is used for direct creation of a variable. The "type" can be:

- Boolean
- Integer
- Double
- String
- Set
- Bag
- Sequence
- OrderedSet
- Object (which represent an object of the model)

And the "value" is the initialization value of the type. Examples:

```
>    var_isCentered="Boolean(true)"
>    var_percentage="Double(3.32)"
```

```
>   var_listPoints="Set( OrderedSet(Integer(13), Integer(26)),
    OrderedSet(Integer(22), Integer(13)) )"
```

An Object can't be initialized manually because the "value" of an Object is a unique ID which isn't known after the initialization of the program; user has to choose the second form.

The following code shows an event which uses a variable:

```
>   onChange="{ Java | System.out.println(hw); }" var_hw="String(Hello
    World !)"
```

When a code attribute is executed, it generates some output variables. These variables can be recovered using the second form of the declaration of a variable. To recognize the variables we use the symbol $. The output variables have an influence only on their own attributes and children; they act locally. The following example shows a variable named t created in a code attribute and recovered below in a variable named self:

```
>   <svg onCreation="{Java| t = model.getSimpleState().createSimpleState();}"
    <!-- … --> />
>   <!-- … -->
>       <text onChange="{Java| self.changeName(content);}" var_self="$t"
>       <!-- … --> />
>   <!-- … -->
>   </svg>
```

To finish with the definition of variables, we can also change the value of a variable. In the following example, if we do the first event before the second, the displayed value will be "changed value":

```
>   onEvent1="{Java| test = "changed value";}" var_test="initial value"
>   onEvent2="{Java| System.out.println(test);}"
```

8.4. How to use the dynamic components (specifications)

In this part, all used components and interfaces are described. The examples are inspired by our study case; the statechart language which is described in the beginning of the report. The used language is Java (Koala). To have more information on the different components and interfaces, see the final report of ProBXS [PBX].

8.4.1. Components

The components define the behavior of a specific tag in SVG. Each component uses different interfaces which provide its “laws”. The different components can be called in the SVG templates as follows:

```
>    dpi:component="test.EditableString"
```

The following table shows a non-exhaustive list of components which are commonly used. These are all the components used in our study case, the statechart language. The aim of this list is to allow user to create new features easily. To have more information on the different components, see [PBX].

	Description	**Interfaces**
AnchorPoint	AnchorPoint represents a part which can be anchored by another component. There are two general uses of *AnchorPoint*. You can use it as a visible shape that can anchor the links, or you can make it unvisible (SVG attribute : visibility=”hidden”) in order to make only a part of another shape link attachable	Hiligthtable, Selectable, BorderFindable, LinePullable, Stickable
Arrow	Component used to handle and represent behaviour of Link’s *arrowStart* and *arrowEnd* according to the body link’s anchor points movement.	Locatable, ArrowBorderSlidable, Selectable, Translatable, DirectionAdjustable, OriginGettable, Stickable
Background	Background represents the background of the SVG view.	Paintable, ColorPickable, Locatable, Resizable, Hiligthtable
BasicContainer	BasicContainer represents a simple container.	Locatable
CompositeContainer	BasicContainer represents a container which has much more behaviors than BasicContainer	Containable, Paintable, ColorPickable, Locatable, ContainedTranslatable,

		Orderable, Frontable, Backable, Forwardable, Backwardable, Resizable, Hiligthtable, Selectable
CurvedLine	Body of the *Link*. Manages the points and handles structures and synchronizes them with the SVG path element.	Selectable
EditableString	EditableString represents a text field which can be modified.	Locatable, CharacterHitable, CharacterInsertable, CharacterDeletable
LineHandle	Used to deform *CurvedLines*. They are synchronized with the handles of a SVG curved line path.	Locatable, Stickable, Hiligthtable, Containable, ContainedHandlable
Link	Represents the groups of information. This group of information allows to instanciate new links from a model and defines the visual characteristics.	ColorPickable, Locatable
Translatable-EditableString	TranslatableEditableString is an EditableString which can be translated.	Locatable, CharacterHitable, CharacterInsertable, CharacterDeletable, Translatable, LinePullable

8.4.2. Interfaces

Each interface described below can be called in the template SVG using the tag “onEvent” if the corresponding component is in use. In the attribute’s value, we can call different special variables which are described in the following table:

	Name of the attribute to interpret (onEvent)	**Special variables**
CharacterDeletable	onChange	model : represent the model. content : content of the string tag.
CharacterInsertable	onChange	model : represents the model. content : content of the string tag.
Containable	onContained	model: represents the model. containedComponents : represents an array of objects which contains all the variables of all the contained objects. containerComponent_* : represents all the variables of the container where * is the name of the variable. (see 8.4.4.4)
DirectionAdjustable	onRotate	model : represents the model. alpha : angle of the object.
Locatable	onPosition	model : represents the model. posX : position of the object on X-axis. posY : position of the object on Y-axis.
Orderable (Not done for theoretical and time reasons)	-	-

Positionable	onPosition	model : represents the model. posX : position of the object on X-axis. posY : position of the object on Y-axis.
Resizable (Look out: this event is allowed but as the idea of resizing is forgotten, this event doesn't work.)	onResize	model : represents the model. sizeX : X-size of the object. sizeY : : Y-size of the object.
Stickable	onStick	model : represents the model. stickedComponent_* : represents all the variables of the sticked component where * is the name of the variable (see 8.4.4.3)
Stickable	onUnStick	model : represents the model. unStickedComponent_* : represents all the variables of the sticked component where * is the name of the variable.
Translatable	onPosition	model : represents the model. posX : position of the object on X-axis. posY : position of the object on Y-axis.

8.4.3. Special Cases

8.4.3.1. ComponentCreated

Definition:

ComponentCreated occurs when a component is created. This is a special case; it's neither a component nor an interface, but its utilization is the same as for another interface.

Name of the attribute to interpret:

onCreation

Special variables:

model : represents the model.

8.4.4. Examples

8.4.4.1. Creation of a new component

The following part of SVG template shows how to create a new component

```
>    <svg onCreation="{Java| s = model.getState().createState(); }"
>            xmlns = "http://www.w3.org/2000/svg"
>            xmlns:xlink="http://www.w3.org/1999/xlink"
>            xmlns:dpi = "http://bod.fr/DPI"
>            xmlns:c="http://mcc.id.au/2004/csvg">
>    <!--…-->
>    </svg>
```

8.4.4.2. String which is editable

The following part of SVG template describes how to create an editable string. The assumption is that the current object is s.

```
>    <text onChange="{ Java | self.setName(content); }" var_self="$s" id="123"
     fill="black" dpi:component="test.EditableString" cursor="text" text-
     anchor="middle" y="0" x="0">newString</text>
```

8.4.4.3. Stick event

The following part of SVG template describes how to create two objects which can be stuck.

Point of view of the stuck object t:

```
>    <!-- … -->
>    <rect var_self="$t" var_isSource="Boolean(true)"
     dpi:component="PBXSComponents.Arrow" <!-- … -->/>
>    <polygon var_self="$t" var_isSource="Boolean(false)"
     dpi:component="PBXSComponents.Arrow" <!-- … -->/>
>    <!-- … -->
```

Point of view of the stickable object s:

```
>    <!-- … -->
>    <rect onStick="{Java|
>        if (((Boolean) stickedComponent_isSource).booleanValue()) {
>            self.getOutgoing().add(stickedComponent_self);
>        } else {
>            self.getIncoming().add(stickedComponent_self);
>        };}"
>     var_self="$s"  dpi:component="PBXSComponents.AnchorPoint"
>     <!-- … -->/>
>    <!-- … -->
```

8.4.4.4. Containable

The following part of SVG templates describes how to create a container and its contained components.

First of all, the point of view of a contained object, we just need to define the variable self in the highest level (the level which contains the component to contain):

```
>     <!-- … -->
>     <svg <!-- … -->>
>       <g var_self="$s" dpi:component="test.CompositeContainer" <!-- … -->>
>           <!-- … -->
>       </g>
>     <!-- … -->
>     </svg>
```

In the container point of view, we recover all contained objects environment by using the special variable *containedComponents*. This variable is a 3 dimensional array of objects. The first dimension correspond to the contained components, the second to the variables of the selected component and the last to contain both the name and the value of the variable. So we can imagine it like that:

```
>     <!-- … -->
>     <rect var_self="$cs" onContained="{Java|
>       self.getSubvertex().clear();
>       Object[][][] o = (Object[][][])containedComponents;
>       for (int i = 0; i&lt;o.length; i++){
>          for (int j = 0; j&lt;o[i].length; j++) {
>             if (o[i][j][0].equals(&quot;self&quot;))
>                {self.getSubvertex().add(o[i][j][1]);
>       } } } }"
>     width="300" height="300" dpi:component="test.BasicContainer"
      visibility="hidden" <!-- … --> />
>     <!-- … -->
```

9. Results

To test our program, we decided to create the state chart language (See the abstract syntax on Figure 1 and the concrete syntax on Figure 2) and to reproduce the same instantiation (See Figure 4). The resulting model has to look like the specific model of Figure 3.

We can see on Figure 14 the SVG scene of our test; the same metamodel instantiation than in Figure 4.

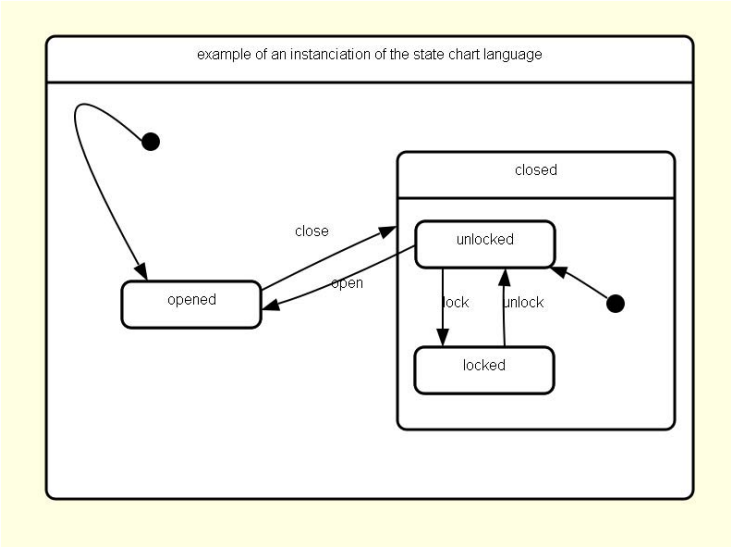


Figure 14 : Resulting SVG scene of our test.

When we store the model of this scene in an XMI file, we expect to obtain the same model representation than these of Figure 3. The XMI generated model is the following:

```
<?xml version = '1.0' encoding = 'windows-1252' ?>
<XMI version = '2.0' xmlns = 'http://www.omg.org/XMI'>
  <sc.StateMachine id = 'a1'>
    <sc.StateMachine.top>
      <sc.CompositeState id = 'a2' name = 'example of an
instanciation of the statechart language'>
        <sc.CompositeState.subvertex>
          <sc.CompositeState id = 'a3' name = 'closed'>
            <sc.StateVertex.incoming>
              <sc.Transition idref = 'a4'/>
            </sc.StateVertex.incoming>
            <sc.CompositeState.subvertex>
              <sc.SimpleState id = 'a5' name = 'unlocked'>
                <sc.StateVertex.outgoing>
                  <sc.Transition idref = 'a6'/>
                  <sc.Transition idref = 'a7'/>
                </sc.StateVertex.outgoing>
                <sc.StateVertex.incoming>
                  <sc.Transition idref = 'a8'/>
                  <sc.Transition idref = 'a9'/>
                </sc.StateVertex.incoming>
              </sc.SimpleState>
              <sc.SimpleState id = 'a10' name = 'locked'>
                <sc.StateVertex.outgoing>
                  <sc.Transition idref = 'a9'/>
                </sc.StateVertex.outgoing>
                <sc.StateVertex.incoming>
                  <sc.Transition idref = 'a6'/>
                </sc.StateVertex.incoming>
              </sc.SimpleState>
              <sc.PseudoState id = 'a11' kind = 'initial'>
                <sc.StateVertex.outgoing>
                  <sc.Transition idref = 'a8'/>
                </sc.StateVertex.outgoing>
              </sc.PseudoState>
            </sc.CompositeState.subvertex>
          </sc.CompositeState>
          <sc.SimpleState id = 'a12' name = 'opened'>
            <sc.StateVertex.outgoing>
              <sc.Transition idref = 'a4'/>
            </sc.StateVertex.outgoing>
            <sc.StateVertex.incoming>
              <sc.Transition idref = 'a13'/>
              <sc.Transition idref = 'a7'/>
            </sc.StateVertex.incoming>
          </sc.SimpleState>
          <sc.PseudoState id = 'a14' kind = 'initial'>
            <sc.StateVertex.outgoing>
              <sc.Transition idref = 'a13'/>
            </sc.StateVertex.outgoing>
          </sc.PseudoState>
        </sc.CompositeState.subvertex>
      </sc.CompositeState>
    </sc.StateMachine.top>
  </sc.StateMachine>
  <sc.Transition id = 'a13'>
    <sc.Transition.source>
      <sc.PseudoState idref = 'a14'/>
    </sc.Transition.source>
    <sc.Transition.target>
      <sc.SimpleState idref = 'a12'/>
    </sc.Transition.target>
  </sc.Transition>
  <sc.Transition id = 'a8'>
    <sc.Transition.source>
      <sc.PseudoState idref = 'a11'/>
    </sc.Transition.source>
    <sc.Transition.target>
      <sc.SimpleState idref = 'a5'/>
    </sc.Transition.target>
  </sc.Transition>
  <sc.Transition id = 'a6' name = 'lock'>
    <sc.Transition.source>
      <sc.SimpleState idref = 'a5'/>
    </sc.Transition.source>
    <sc.Transition.target>
      <sc.SimpleState idref = 'a10'/>
    </sc.Transition.target>
  </sc.Transition>
  <sc.Transition id = 'a9' name = 'unlock'>
    <sc.Transition.source>
      <sc.SimpleState idref = 'a10'/>
    </sc.Transition.source>
    <sc.Transition.target>
      <sc.SimpleState idref = 'a5'/>
    </sc.Transition.target>
  </sc.Transition>
  <sc.Transition id = 'a4' name = 'close'>
    <sc.Transition.source>
      <sc.SimpleState idref = 'a12'/>
    </sc.Transition.source>
    <sc.Transition.target>
      <sc.CompositeState idref = 'a3'/>
    </sc.Transition.target>
  </sc.Transition>
  <sc.Transition id = 'a7' name = 'open'>
    <sc.Transition.source>
      <sc.SimpleState idref = 'a5'/>
    </sc.Transition.source>
    <sc.Transition.target>
      <sc.SimpleState idref = 'a12'/>
    </sc.Transition.target>
  </sc.Transition>
</XMI>
```

And it's exactly the expecting result; that is, the same model than these of Figure 3. So the rising synchronization works well in our language.

10. Conclusion

10.1. The point on the program

As describe in this report, only the rising synchronization has been achieved and implemented. Nevertheless, it is working very well except for the problem of “static” synchronization (see 7.2.4.3).

10.2. Next things to do

Here are some possible things to add after our work.

- The descendant synchronization.
- Loading a model which has been stored by the program is not implemented because there is a problem with the recovering of the object identification.
- Implementation of orderable.

  The interface orderable is not implemented yet. This can pose some problems because there is no implemented scheduling of the whole SVG scene.
- Increase the possible variable types for increasing the expression power.
- Add some new languages like OCL or a new version of Kermeta.
- There are some bugs in the project of Fabrice Hong yet. A good way to make the program better would be to correct them.

10.3. Pros and Cons

- Pros
  - Expression power

    Our project allows user to write his proper requests in Dynamic Java and to define all the variables that he wants. User has at his disposal a new powerful language.
  - Multilanguage

    In a same template, we can imagine having more than one language. We can better make good use of each language; use the appropriate language for each query.
  - User friendliness

    Implementing the rising synchronization for a new language is quite easy; it doesn’t require a lot of knowledge (just the user manual of this paper).

- Cons

    - Too much information in SVG templates

    Even for simple queries, the expressions to interpret come very quick long. It's difficult to write them directly in SVG templates because a simple text editor doesn't recognize the simple coding faults. We can imagine use some external scripts to render that easier.

    - Difficulties updating an unknown language

    If we imagine that somebody has to handle an existing language, he could spend a lot of time to understand what is already done. It's difficult to add some comments into the SVG templates (we can't put them where we want) and in the code attributes.

10.4. Course of our work

We have worked almost four months on this project and of course, as with every project, we had to deal with some difficulties and disappointments during the semester. Here is a quick description of our activities during the semester. You will find how we finally managed to achieve our goals and requirements.

First of all, we had to become familiar with the "world" of language construction and model-driven language construction. This topic was roughly new for us! We had some basics, thanks to the Software Engineering course, but we almost spent one month and a half reading papers, checking technical websites and having a look on how ProBXS was coded. Once we were aware of all the technical background, we presented in our first oral presentation the problematic and a potential solution we had thought about. The difficulties really appeared at this step of the project, because we wanted to use a specific language (Kermeta) to implement our solution. This language had very interesting features, but we had much trouble to integrate it with ProBXS. We even contacted the developers asking for help, but we finally noticed that we spent much more time trying to resolve Kermeta's bugs than really working on our project. Therefore, we changed our mind and with the help of Frédéric we decided to use another architecture (MDR, JMI & Dynamic Java), which was working very well. Since then we have finally had the opportunity to really work on the requirements of the project and we succeeded in achieving half of our starting requirements. Indeed, because of this waste of time due to Kermeta and considering the three oral presentations and this paper to write, it was then only possible for us to achieve the rising synchronization. Anyway, we had a wonderful time working on this project. Frédéric, in spite of a small lack of availability at the beginning of the project, was always open to answer our questions and we would like to thank him very much!

10.5. What we have learnt

- New skills in language construction theory and involved technologies
- Better knowledge of the Eclipse environment
- How to implement new features to an existing project
- Work on a big project which means
    - Work as a team
    - Classify ideas
    - Reflex of writing down problems
    - Concrete representation of the difference between finding theoretical solutions and having to deal with their implementations.
- Improve oral and written English skills.
- Improve oral presentation skill.

11. Index

12. Bibliography

12.1. Site Web

[XML] The definition of XML and all other useful information on:

Http://www.w3.org/XML/

[XMI] XMI specification web site:

http://www.omg.org/technology/documents/formal/xmi.htm

[JMI] JMI specification web site:

http://java.sun.com/products/jmi/

[SVG] The definition of SVG and all other useful information on:

http://www.w3.org/Graphics/SVG/

[DOM] The definition of DOM and all other useful information on:

http://www.w3.org/DOM/

[Bat] Official web site of Batik:

http://xml.apache.org/batik/

[MOF] Official web site of MOF:

http://www.omg.org/mof/

and the current specification:

http://www.omg.org/docs/formal/02-04-03.pdf

[Koa] Official web site of DynamicJava : Koala:

http://koala.ilog.fr/djava/

[Ker] Kermeta official web site:

http://www.kermeta.org/

12.2. Papers

[DPI] *Olivier Beaudoux*, DoPIdom : Une boîte à outils pour la conceptiond'interfaces centrées sur les documents XML, French, 2004, 4 pages.

http://www.eseo.fr/~obeaudoux/publications/beaudoux-ihm04.pdf

[PBX] *Fabrice Hong*, Provide Behaviour to XML-SVG, Semester Project, summer 2005, 53 pages.

http://lglpc35.epfl.ch/viewcvs/*checkout*/dopidom/documents/Projet%20de%20semestre%20-%20Fabrice%20Hong.doc?rev=1.10

[MDR] *Martin Matula*: NetBeans Metadata Repository, 3/3/2003

http://mdr.netbeans.org/MDR-whitepaper.pdf