

Final Bachelor's Paper

Oryx Stencil Set Specification

Nicolas Peters

Supervisors
Prof. Dr. Mathias Weske
Hagen Overdick
Gero Decker

30 June 2007

Abstract

Business Process Management is a hot topic in the IT industry and in computer science. One important part of this subject is Business Process Modeling, which means the modeling of processes assisted by description or notation languages, for example the Business Process Modeling Notation (BPMN) [1]. The project "B7 - Browser-based Business Process Editor" of the department "Business Process Technology" of the Hasso-Plattner-Institute in Potsdam, Germany is about developing a web-based application for modeling processes with graphical notation languages such as BPMN. The result of the project is an application named "Oryx".

The project described above provides the framework for this and three other final bachelor's papers. This document is a manual for how to describe a graphical notation language (stencil set) so that you can use it in Oryx. It explains the tasks of a stencil set development step by step. After reading this document you should be able to implement a stencil set for Oryx on your own.

Zusammenfassung

Business Process Management ist ein sehr aktiver und aktueller Forschungsbereich in den Computerwissenschaften. Ein wichtiger Teil dieses Bereichs ist das Business Process Modeling, also das Modellieren von Prozessen mit Hilfe von Beschreibungs- oder Notationssprachen, z. B. die Business Process Modeling Notation (BPMN) [1]. In dem Projekt "B7 - Browser-basierter Geschäftsprozesseditor" des Fachbereichs "Business Process Technology" am Hasso-Plattner-Institut in Potsdam geht es um die Entwicklung einer web-basierten Anwendung für die Modellierung von Prozessen mit graphischen Notationssprachen wie BPMN. Herausgekommen ist eine Anwendung mit dem Namen „Oryx“.

Dieses Projekt bildet den Rahmen für diese und drei weitere Bachelorarbeiten. Dieses Dokument ist eine Anleitung, wie eine grafische Notationssprache (Stencil Set) zu beschreiben ist, um sie in Oryx nutzen zu können. Schritt für Schritt werden alle notwendigen Aufgaben erklärt. Nach dem Lesen dieser Arbeit sollten Sie in der Lage sein, selbstständig ein Stencil Set zu implementieren.

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Potsdam, den 30. Juni 2007

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Chapter Overview	2
2	What is an Oryx Stencil Set?	3
2.1	Requirements	4
2.2	Technologies	4
3	Getting Started with Stencil Sets	7
3.1	Creating a Stencil Set	7
3.2	How to load a Stencil Set in Oryx?	8
3.3	Example	8
4	Creating Graphical Representations	11
4.1	Restrictions	12
4.2	Nodes	12
4.3	Edges	18
4.4	Picture Files	21
4.5	Implementation Remarks	21
5	Creating a Stencil Set Description	23
5.1	Stencil Set Description	23
5.2	Stencil Description	24
5.3	Property Description	29
5.4	Property Item Description	34
5.5	Implementation Remarks	37

6	Creating Rules	39
6.1	Connection Rules	40
6.2	Cardinality Rules	41
6.3	Containment Rules	44
6.4	Rules Extensions	44
6.5	Stencil Set Extensions	46
6.6	Implementation Remarks	47
7	Outlook	49
A	API	51
A.1	ORYX.Core.StencilSet	51
A.2	ORYX.Core.StencilSet.Stencilset	52
A.3	ORYX.Core.StencilSet.Stencil	53
A.4	ORYX.Core.StencilSet.Property	54
A.5	ORYX.Core.StencilSet.PropertyItem	55
A.6	ORYX.Core.StencilSet.Rules	55
B	The Workflow Nets Stencil Set	59
	Bibliography	67

1. Introduction

Business Process Management is a hot topic in the IT industry and in science. One important part of this subject is Business Process Modeling. That is the modeling of processes with the help of description or notation languages, for example the Business Process Modeling Notation (BPMN) [1]. The project "B7 - Browser-based Business Process Editor" of the department "Business Process Technology" of the Hasso-Plattner-Institute in Potsdam, Germany is about developing a web-based application for modeling processes with graphical notation languages such as BPMN. The project focuses on the front end of the application. The back end was developed by another project of the department. The result of the project is an application named "Oryx" (see figure 1.1 for a screenshot of Oryx).

The project described above provides the framework for this and three other final bachelor's papers. This document is a manual for how to describe a graphical notation language (stencil set) so that you can use it in Oryx. It explains the tasks of a stencil set development step by step. After reading this document you should be able to implement a stencil set for Oryx on your own.

The final bachelor's paper "Oryx - Dokumentation" by Willi Tscheschner [2] describes the core concepts and the architecture of the application and it gives a review of the whole project. Next, "Oryx - Embedding Business Process Data into the Web" by Martin Czuchra [3] is about the internal data format and the client/server communication of Oryx. It introduces a way for embedding data into web applications, as well as concepts for accessing and storing the data on the web. At last, Daniel Polak describes the implementation of a stencil set for BPMN in his paper "Oryx - BPMN Stencil Set Implementation" [4].

1.1 Motivation

One of the main requirements of Oryx was to use generic approaches to be extendable and flexible. Therefore, we have implemented a plug-in concept to easily extend the editor's functionality. Second, we have introduced a data management layer to ease the replacement of the backend storage system. And at last, we have invented a generic approach for implementing modeling notation languages for Oryx. The

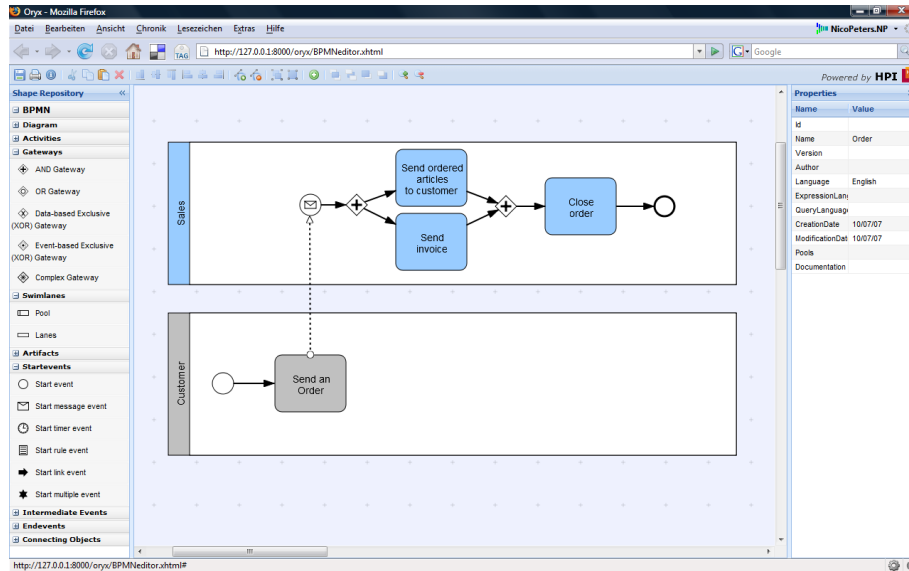


Figure 1.1: A screenshot of Oryx with the BPMN stencil set.

design goal of the stencil set specification was to be able to implement all constructs of the Business Process Modeling Notation (BPMN) [1] [4] without using BPMN dependent approaches in the specification that are only useful for this one stencil set.

1.2 Chapter Overview

Each of chapters three to six deals with one task within a stencil set implementation. To better understand those tasks I will provide an implementation for a workflow nets [5] stencil set throughout these chapters (see section 3.3 for more details on the example).

In chapter 2 "What is a Stencil Set?" I will give a brief introduction of the idea of stencil sets and the technologies used for describing a stencil set. After this I will present the first steps when creating a stencil set in chapter 3 "Getting Started with Stencil Sets". The following three chapters address different tasks within a stencil set development. First, chapter 4 "Creating Graphical Representations" explains how to describe a stencil's design and how to prepare it that way that, for example, Oryx can resize graphical representations. Second, chapter 5 "Creating a Stencil Set Description" presents how to describe a stencil set, its stencils and the stencils' properties. At last, chapter 6 "Creating Rules" introduces the definition of rules that are applied to the stencils.

The following chapter, "Outlook", is about features that could be implemented in future developments of Oryx.

At the end of the document, in appendix A "API" I will describe the classes and methods of the `ORYX.Core.StencilSet` package that is the implementation of the stencil set specification. Finally, appendix B "The Workflow Nets Stencil Set" offers the complete implementation of the workflow nets stencil set.

2. What is an Oryx Stencil Set?

An Oryx stencil set is basically a description of a set of graphical objects and rules that specify how to relate those graphical objects to others. A graphical object is a node or an edge. Edges connect two graphical objects. A stencil set can be loaded and used in the Oryx editor to build process models. The stencil set for the Business Process Modeling Notation (BPMN) by Daniel Polak [4] is one example. An important feature is that a stencil does not only have a graphical representation, but also additional properties that can later be used by other applications or Oryx extensions. Those properties can also manipulate the graphical representation, for example changing the color or setting the visibility.

Oryx' internal data format is based on the Resource Description Framework (RDF) [6] or more precisely eRDF [7] for embedding RDF into XHTML (for detailed information about the data format and the data API see [3]). Oryx only knows resources and literals that are accessed using triples with a subject, a predicate and an object. The subject is the resource you are interested in. The object is a literal or another resource and the predicate specifies the relationship between the subject and the object. With these triples you can easily build up a data hierarchy. The API to access those resources can also be used by other applications for example a process execution engine.

Therefore, each shape (nodes and edges) of a diagram is mapped to a resource. All properties of a shape are saved as literals or resources inside that resource. From this angle a stencil set is a form of a resource type definition. It defines which data has to be added to the resource and what data the editor can expect when loading a resource of a certain type. To be able to relate to all concepts of the stencil set specification you have to keep in mind that everything is a resource. Oryx offers a view on a resource representation that contains references to other resources. A diagram is neither just an Oryx document nor a black box to other applications. Instead, it is a resource and you could build another stencil set that uses these diagrams as nodes and relate them. And because of that you also have to specify a stencil in your stencil set that can include all the other stencils and is used as the top most resource type for diagrams that uses your stencil set. The editor's canvas will represent this top most resource and all the properties and rules that are defined

for that stencil are applied to it. This differs Oryx from other modeling tools and is maybe confusing at the beginning. To fully understand all concepts of Oryx see [2] and [3].

2.1 Requirements

The main requirements of the stencil set specification are as follows:

- It should offer a generic API to be able to include every graphical notation language in Oryx.
- It should ease the inclusion of new graphical notation languages in Oryx.
- It should describe how to specify the graphical representation and additional properties of stencils.
- It should describe how to specify rules that assist the modeler with creating processes.

These requirements are rough instructions we always had to keep in mind during the development. The concrete concepts have been arisen from the incremental iterative development of this specification.

2.2 Technologies

Three technologies are used when describing a stencil set: the JavaScript Object Notation (JSON) [8] for the description of stencils, stencils' properties and rules. Second, the Scalable Vector Graphics (SVG) [9] for the description of the graphical representation. And at last JavaScript for functions that are used as callbacks for stencil set specific actions (see sections 5.2 and 6.4).

2.2.1 JSON

The chosen description language has to satisfy the following requirements:

- It should be powerful enough so that the possibilities of a stencil set are not limited by the language
- It should be human-readable, as well as machine-readable
- It should be a lightweight description language (minimal writing to add information)
- It should be easy and high-performance to process the data with JavaScript

All this is met by JSON, the JavaScript Object Notation. With its simple structure of name/value pairs it is easy to use and flexible enough for our requirements. One of the main advantages is that JSON is a subset of JavaScript. So, it can be used directly in Oryx that is written in JavaScript.

To give you an example of JSON, let us define a JSON object for describing a person with her or his name, the city she or he is living in and a list of her or his favorite meals. We just define three name/value pairs for each property - "name", "city" and "favoriteMeals". The last attribute is an array of string values:

```
{
  "name": "Nico",
  "city": "Potsdam",
  "favoriteMeals":
  [
    "Pizza",
    "Meat"
  ]
}
```

2.2.2 SVG

After evaluating several technologies (see [2]) we have decided to use SVG for the visualisation in Oryx. So, it is obvious to use SVG for the description of a stencil's graphical representation, too. The description can directly be used in Oryx. Furthermore, SVG is extendable, so that we can include additional information.

SVG is a XML data format for describing vector graphics. One of the arguments for SVG was the native support in our main targeted platform, the Mozilla Firefox 2.0+. The manipulation of the shapes at runtime can simply be reduced to DOM manipulations. This is a very common task of web applications, so we didn't have to reinvent the wheel for our goals and we could resort to the experiences of many JavaScript developers.

Although you will have to write XML code directly into the SVG document while creating the stencils' graphical representations, you can also use one of the various SVG drawing tools for the basic design of a stencil.

2.2.3 JavaScript

As mentioned above, JavaScript is used for callback functions that can be specified in a stencil set description. That is those functions are embedded into the JSON object. Actually, JSON does not allow functions as values, because this breaks the language independency. But because we are solely using JSON in a JavaScript environment, we can make use of this approach without getting any problems.

3. Getting Started with Stencil Sets

The most important requirement for creating a stencil set is that you have access to the server where Oryx is installed. Otherwise, you cannot use your stencil set. It is also recommended to have basic knowledge about SVG and JavaScript.

In this chapter, you will learn, how to set up the basic folder structure for a stencil set and how to load a stencil set in Oryx.

3.1 Creating a Stencil Set

A typical stencil set consists of a JSON file and several SVG files, as well as several picture files. The JSON file contains the stencil set description, e.g. a definition of every stencil (see chapter 5). For each stencil in a stencil set one SVG file that contains the graphical representation of the stencil and one picture file that is used as an icon for that stencil is required. To minimize the configuration complexity, a stencil set has to contain two mandatory folders within the same folder as the JSON file resides: `icons` and `view`. Simple to guess, the folder `icons` contains the picture files and the folder `view` contains the SVG files. The name of the JSON file can be selected freely, but it is recommended to use the ending `".json"`. For example, the folder structure for a stencil set `"abc"` looks like this:

```
[abc]
|- abs.json
|- icons
|- view
```

After setting up the folders, the usual way to create a stencil set is to first design the stencils' graphical representations and then to write the stencil set description. However, to ease finding bugs in your stencil set, you can also use an incremental, iterative approach: design one graphical representation of a stencil, include it in the stencil set description and load the stencil set in Oryx to test it.

3.2 How to load a Stencil Set in Oryx?

To use your own stencil set in Oryx, you must put it on the same web server where Oryx is running and add a reference to the stencil set description file in the XHTML document where Oryx is embedded. Create a folder on the web server and copy all the files and folders of the stencil set into the created folder. An example reference looks like this:

```
<a href="./data/stencilsets/abc.json" rel="oryx-stencilset" />
```

Please refer to [2] and [3] for more information about embedding Oryx in an XHTML document.

Another way to load a stencil set is to use the Oryx plugin 'Add Stencil Set'. When Oryx is loaded, click on the button "Add a stencil set" in the toolbar (if the plugin is available) and prompt the relative path to the JSON file into the input box.

As mentioned in chapter 2 Oryx is a view on a resource. For this reason, the XHTML document that is a representation of a resource must have a type. For example, if the example stencil set "abc" has the namespace `http://example.org/abc#` and the top most resource type is defined by the stencil with the id "Diagram", you will have to add the following line to the head of the XHTML document:

```
<meta name="oryx.type" content="http://example.org/abc#Diagram" />
```

If you have problems loading a stencil set, refer to the JavaScript error console of your browser to see, if there is a problem in one of the stencil set's files.

3.3 Example

Because the easiest way to understand a language is to use and see it, I will provide the implementation of a simple stencil set for workflow nets by Wil van der Aalst [5] throughout the following chapters (see figure 3.1 for an example workflow net created with Oryx). The idea of workflow nets is to map the concepts of business process modeling to petri nets. Workflow nets are basically extended petri nets. The example stencil set should provide stencils for activities with triggers, conditions, control flow, input conditions and output conditions, as well as rules to avoid building invalid process models. Additionally, we need a stencil for the resource that can contain

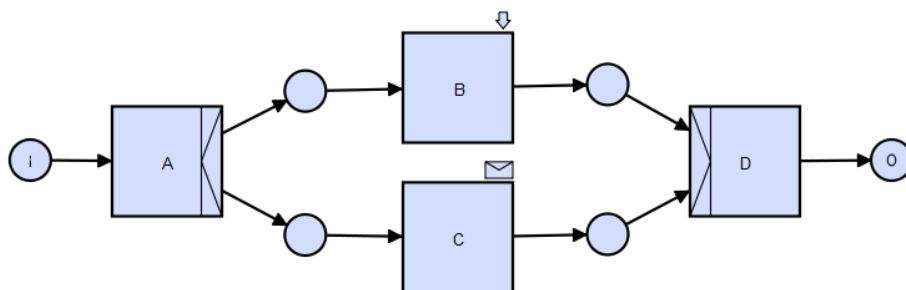


Figure 3.1: An example workflow net created with Oryx.

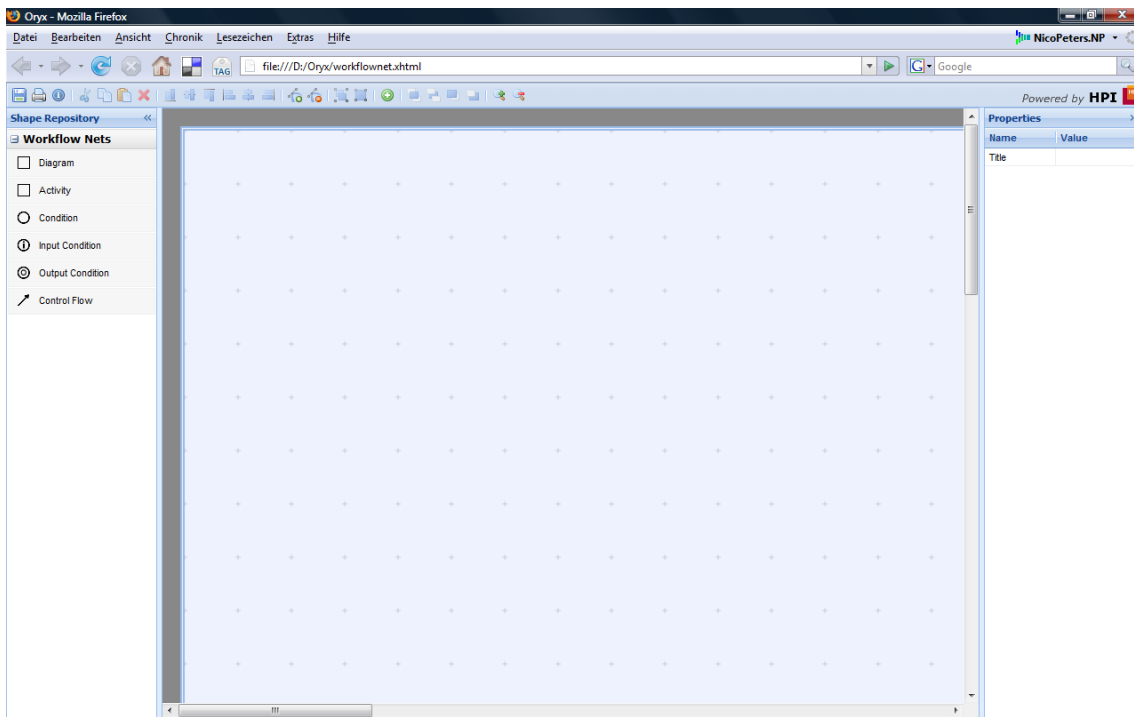


Figure 3.2: Oryx with the workflow net stencil set.

those workflow nets. Therefore, we also have to define a stencil for workflow net diagrams.

To start our example, we create the basic folder structure for the workflow nets stencil set:

```
[workflownets]
|- workflownets.json
|- icons
|- view
```

The file "workflownets.json" is simply an empty text file.

You can find the complete stencil set description for this example in appendix B. If you want to try out the workflow nets stencil set, open the file `Oryx/workflownet.xhtml` on the enclosed CD in a Firefox 2.0+ browser. When Oryx is loaded, the web site will look like figure 3.2.

4. Creating Graphical Representations

As mentioned earlier, the graphical representation of stencils are described by using the Scalable Vector Graphics (SVG) format in version 1.1 [9]. Using SVG has the advantage that you can use a SVG drawing application for designing the graphical representation of stencils and that you can have a look at the graphical representation with any SVG Viewer. On the other hand, the SVG format can directly be used in Oryx, because it uses SVG for visualization.

However, a pure SVG description does not contain enough information for Oryx to manipulate a stencil's graphical representation. You have to add additional information so that the editor can redraw the view the way you want in case of resizing, for example.

To accomplish this task, I have extended the SVG namespace with additional elements and attributes. This does not violate the SVG specification as it explicitly allows extensions.

The first step when introducing an extension, is setting up an own XML namespace in which all our elements and attributes are defined:

```
http://www.b3mn.org/oryx
```

When using our elements and attributes in a SVG document, you have to set a prefix for our namespace:

```
<svg xmlns="http://www.w3.org/svg"
      xmlns:oryx="http://www.b3mn.org/oryx/sss">
  <!--...-->
</svg>
```

I will use the prefix 'oryx' throughout this chapter, but you can choose the prefix freely.

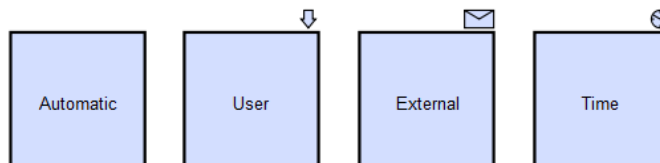


Figure 4.1: Different types of triggers of a workflow net activity.

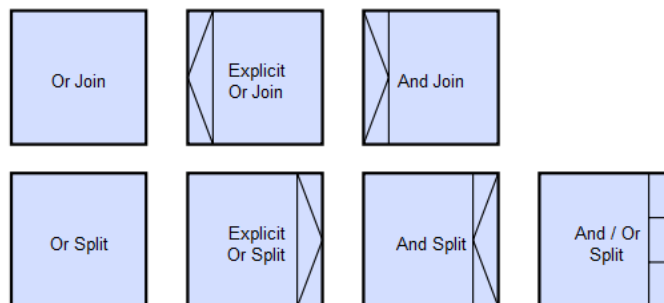


Figure 4.2: Split and join behavior of an activity.

4.1 Restrictions

SVG is a very comprehensive standard for a very complex topic. This may be one reason why many implementations don't fully support all SVG constructs. And the Mozilla Firefox browser 2.0+ (our main targeted platform for Oryx) lacks in supporting the complete SVG specification, too. [10] offers information about what has already been implemented in Firefox. So, when designing stencils, always keep in mind not to use elements and attributes that the Mozilla Firefox Browser does not understand.

Furthermore, the Oryx editor restricts the usage of some SVG constructs to ease the data handling. However, these restrictions should not limit your creativity. The Oryx specific restrictions are mentioned in the following sections.

4.2 Nodes

The graphical representation of a node is not a static picture. In Oryx it can be resized, text can be attached, colors and opacities can be changed and parts of the node can be hidden. Except resizing, these abilities are not only specified in the SVG document of a node, but you have to prepare the SVG representation for that.

I will use the attributes of the Oryx namespace in the examples or mention them throughout the next paragraphs. The last paragraph contains a list of all defined attributes.

Let us use the example of a workflow nets activity for explaining how to specify the graphical representation of nodes. The basic look of an activity is a rectangle with a centered title text. Additionally, workflow nets activities can also have three different types of triggers (see figure 4.1). Moreover, you can define the split and join behavior of an activity. Those behaviors are represented as different symbols at the left and right side of the activity (see figure 4.2). We will implement all of these symbols step by step. First, there is a basic document structure for a node:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg
  xmlns="http://www.w3.org/2000/svg"
  xmlns:oryx="http://www.b3mn.org/oryx"
  version="1.1">
  <g oryx:minimumSize="50 50" oryx:maximumSize="200 200">
    </g>
</svg>
```

Within the 'svg' element we have to define the Oryx namespace. We have also added one single 'g' element to the document. To ease the data handling for Oryx the complete graphical representation of the node must be added to this one 'g' element. Anything else will be ignored by Oryx. That is do not use any SVG markers, symbols, gradients or anything else that you have to specify outside of this one 'g' element, because this will not be rendered on the Oryx canvas.

The 'g' element is extended by a 'minimumSize' and 'maximumSize' attribute that set the minimum and maximum size the node can be resized. These attributes are not mandatory, so you do not have to set them.

Let us now first add the rectangle to the 'g' element:

```
<rect id="color" oryx:anchors="top bottom left right" x="1" y="16"
  width="80" height="80" stroke="black" fill="#D3DEFF"
  stroke-width="2"/>
```

We simply use a 'rect' element for the rectangle. For Oryx we add the attribute 'anchors' to the element. With these anchors we fix the rectangle to all four sides so that it will always have the same distance to the border of the stencil. The border of a stencil is defined by the smallest rectangle aligned along the axes that includes all SVG shapes of the stencil.

To go on, we now add the SVG shapes that represent the different join behaviors of the activity:

```
<path id="explicitorjoin" oryx:resize="horizontal vertical"
  stroke-linejoin="bevel" oryx:anchors="top bottom left" fill="none"
  stroke="black" d="M16 16 l-15 40 115 40 z" />
<path id="andjoin" oryx:resize="horizontal vertical" stroke="black"
  stroke-linejoin="bevel" oryx:anchors="top bottom left" fill="none"
  d="M1 16 115 40 1-15 40 M16 16 v80" />
```

There are two path elements, one for the explicit or join and another for the and join. If you open the document in a SVG viewer, you will see that both paths are rendered and that is not what we want. You will see in section 5.3 how to use stencil properties to manipulate the graphical representation. However, it is important to add a unique id to every SVG element you want to manipulate with a stencil property.

We have set the 'resize' attribute to "horizontal" so that the width is recalculated when the stencil is resized. The height is controlled by the anchors "top" and "bottom". Because of these two anchors, the path will be stretched vertically, if the stencil's height changes.

In the same way we append elements that represent the split behaviors:

```
<path id="explicitorsplit" oryx:resize="horizontal vertical"
  stroke-linejoin="bevel" oryx:anchors="top bottom right" fill="none"
  stroke="black" d="M66 16 v80 l15 -40 z" />
<path id="andsplit" oryx:resize="horizontal vertical"
  stroke-linejoin="bevel" oryx:anchors="top bottom right" fill="none"
  stroke="black" d="M81 16 l-15 40 l15 40 m-15 0 v-80" />
<path id="andorsplit" oryx:resize="horizontal vertical"
  stroke-linejoin="bevel" oryx:anchors="top bottom right" fill="none"
  stroke="black" d="M66 16 v80 m0 -26.666 h15 m-15 -26.666 h15" />
```

Finally, the elements for the triggers are still left:

```
<path id="external" oryx:anchors="top right" fill="#D3DEFF" stroke="black"
  d="M61 1 h20 v12 h-20 v-12 m20 0 l-10 6 l-10 -6" />
<path id="user" oryx:anchors="top right" fill="#D3DEFF" stroke="black"
  d="M71 1 v6 h-2 l5 6 l5 -6 h-2 v-6 h-6" />
<g id="time">
  <circle oryx:anchors="top right" fill="#D3DEFF" stroke="black"
    cx="75" cy="7" r="6" />
  <path oryx:anchors="top right" fill="none" stroke="black"
    d="M75 1 v2 M81 7 h-2 M75 13 v-2 M69 7 h2 M75 7 l3 -2 M75 7 l-5 -3" />
</g>
```

All attributes of the Oryx namespace are optional. If you do not set any Oryx attributes, you will get a non-resizable stencil.

If you want to use SVG 'image' elements, you have to reference the image files with an absolute URI or relatively to the SVG file in the 'view' folder (see chapter 3). If you open the SVG file in a SVG viewer and the image is shown, then it should be shown in Oryx, too.

4.2.1 Text

Now, there is only one brick left, the text element:

```
<text id="caption" x="41" y="56" oryx:align="middle center"></text>
```

Text is rendered with SVG text elements, but Oryx extends these element with attributes for the alignment and the rotation of the text. The alignment uses the specified coordinates (attributes 'x' and 'y') as the reference point. A value of 'middle center' means that the horizontal center and vertical middle point of the text will be positioned on the reference point. Anchors can also be defined for text elements, as well as a rotation angle that specifies the rotation around the reference point. If you are familiar with SVG you may wonder about why I have added an own attribute for the rotation, although the SVG specification offers several attributes for laying out text like rotation, top to bottom alignment and so on. The reason is that the Firefox browser does not support any of those attributes, but the goal was to offer at least a way for rotating text.

If you want to set the text element's value using a property, you have to set the id of the text element (see section 5.3).

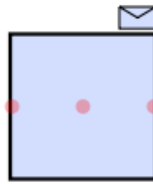


Figure 4.3: An activity with its magnets.

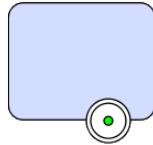


Figure 4.4: A circle docked to a rectangle and the docker is shown (the small, green circle).

4.2.2 Magnets

With magnets you can define special points on a node where you can dock other nodes or edges to connect them. This is part of the dockers-magnets-concept implemented in Oryx (see [2]). You can connect a docker to any point on a node, but magnets help the user creating nicer looking models. If you do not define a magnet for a node, it will have a default magnet in the center. Magnets are specified with 'magnet' elements, for example:

```
<oryx:magnets>
  <oryx:magnet oryx:cx="41" oryx:cy="56" oryx:default="yes"/>
  <oryx:magnet oryx:cx="2" oryx:cy="56"/>
  <oryx:magnet oryx:cx="80" oryx:cy="56"/>
</oryx:magnets>
```

The 'magnets' element must be a direct child of the 'svg' element. These are the magnets for our workflow net activity: one on the left, one in the center and one on the right (see figure 4.3). The centered magnet is marked as the default magnet. In cases where the editor automatically docks an edge to a node the default magnet is used.

4.2.3 Docker

As mentioned in the previous sub section, dockers are the other part of the dockers-magnets-concept. A docker is a control object to connect an edge to a node or in this case to connect two nodes. A node can have at most one docker that can be defined by using a 'docker' element:

```
<oryx:docker oryx:cx="16" oryx:cy="16" />
```

A 'docker' element does not need more information than its position. Docking nodes on nodes can be used as a shortcut for connecting two nodes with an edge. The node with the docker plays both the role of the edge and of the target. In figure 4.4 the rectangular shape is the source and the circular shape is both the edge and the target. These connections also need a connection rule to work (see chapter 6).

4.2.4 Attributes

align

- Value: left | center | right, top | middle | bottom
- Initial: left bottom
- Optional: yes
- Applies to: SVG text elements

Description: 'align' specifies the alignment of the text around the reference point that is defined by the 'x' and 'y' attribute of the text element. The values "left", "center" and "right" set the horizontal alignment, whereas "top", "middle" and "bottom" set the vertical alignment. You can set at most one value for the horizontal alignment and at most one value for the vertical alignment.

anchors

- Value: bottom, top, left, right
- Initial: -
- Optional: yes
- Applies to: SVG shapes, SVG text elements

Description: 'anchors' specifies zero or more anchors. If an anchor is set and the stencil is resized, the distance between the SVG shape and the stencil's border in the direction of the anchor will stay the same. If at least one SVG element has set the anchors "top" and "bottom" or "left" and "right" the stencil can be resized.

cx

- Value: float
- Initial: -
- Optional: no
- Applies to: Oryx docker elements, Oryx magnet elements

Description: 'cx' defines the horizontal position of a docker or magnet.

cy

- Value: float
- Initial: -
- Optional: no
- Applies to: Oryx docker elements, Oryx magnet elements

Description: 'cy' defines the vertical position of a docker or magnet.

default

- Value: yes | no
- Initial: no
- Optional: yes
- Applies to: Oryx magnet elements

Description: 'default' marks a magnet as the default magnet. The default magnet is used when the editor does not know the position a docker should be docked at. If more than one or no magnet is marked as the default magnet, one of them is randomly chosen.

maximumSize

- Value: float float
- Initial: -
- Optional: yes
- Applies to: SVG g elements

Description: 'maximumSize' defines the maximum size the node can be resized, if the node is resizable. The first value defines the maximum width, the second the maximum height. If this attribute is not defined, the maximum size will not be limited. A value smaller than its corresponding value in minimumSize is an error. The 'g' element this attribute is added must be the one 'g' element that contains the node's complete visualization.

minimumSize

- Value: float float
- Initial: 1 1
- Optional: yes
- Applies to: SVG g elements

Description: 'minimumSize' defines the minimum size the node can be resized, if the node is resizable. The first value defines the minimum width, the second the minimum height. A value below one or a value greater than its corresponding value in maximumSize is an error. The 'g' element this attribute is added must be the one 'g' element that contains the node's complete visualization.

resize

- Value: vertical, horizontal
- Initial: -
- Optional: yes
- Applies to: SVG shapes

Description: 'resize' specifies, if a SVG shape is horizontally and/or vertically resizable. If this property is not set, the SVG shape will not be resizable.

rotate

- Value: float
- Initial: 0
- Optional: yes
- Applies to: SVG text elements

Description: 'rotate' specifies the degrees the text is rotated around the reference point that is defined by the 'x' and 'y' attribute of the text element.



Figure 4.5: A simple edge.

Note that if no SVG shape has set the 'resize' attribute or the anchors "top" and "bottom" or "left" and "right", the stencil can not be resized. If you set the anchors "top" and "bottom", the resize "vertical" attribute has no effect. This is also true for the anchors "left" and "right" and the resize "horizontal" attribute.

4.3 Edges

When defining edges for a stencil set, you have to adhere to more conventions. Usually, edges are lines of different forms with decorations attached to it (for example arrowheads). In Oryx, edges should also have the possibility to be divided into sections and to control the sections separately. This makes it possible to draw not only straight edges, but also edges with corners.

We start creating the workflow net control flow edge with setting up the basic structure of an SVG document for edges:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg
  xmlns="http://www.w3.org/2000/svg"
  xmlns:oryx="http://www.b3mn.org/oryx"
  version="1.0">
  <g>
  </g>
</svg>
```

You can see that the basic structure is the same than for nodes. We have a SVG document with exactly one 'g' element. But this 'g' element is only allowed to contain paths. You have to specify at least one SVG 'path' element, but an edge can consist of more than one path. For the control flow we just need to add one path:

```
<path d="M50 50 L100 50" stroke="black" fill="none"
  stroke-width="2" marker-end="url(#arrowEnd)"/>
```

The 'path' element defines a horizontal black line. If you load this edge in Oryx, you will get a black line with a docker at each end (see figure 4.5). Dockers are created on every point you specify in the 'd' attribute of the path. For example, if you set the 'd' attribute to "M50 50 L75 50 L100 50", three dockers will be created. You can only dock the first and the last docker of an edge to another shape. The other dockers are just used for positioning the edge. If you want to prevent the adding of dockers, you can set the 'allowDockers' attribute of the Oryx namespace with the value "no" on a path.

Now, we have to add an arrowhead to the control flow. To achieve this we use the SVG concept of markers. Markers are graphical elements that can be attached to



Figure 4.6: An edge with an arrowhead.

any 'path', 'polyline', 'polygone' and 'line' element. The most important feature of markers is that they can be aligned to the direction of a path. This, for example, automatically renders arrowheads aligned to the path element. You can use any SVG shapes, as well as SVG text elements within the marker element. For more information on markers see [9].

Markers are defined within the 'defs' element of a SVG document. The definition of the arrowhead for the control flow is as follows:

```
<defs>
  <marker id="arrowEnd" refX="10" refY="5" markerUnits="userSpaceOnUse"
    markerWidth="10" markerHeight="10" orient="auto">
    <path d="M 0 0 L 10 5 L 0 10 z" fill="black" stroke="black"/>
  </marker>
</defs>
```

The arrowhead is simply drawn with a 'path' element inside the 'marker' element. If you have a look again at the 'path' element that defines our edge, you can see the attribute 'marker-end' that points to the marker we just defined. The result is shown in figure 4.6.

So, with the attributes 'marker-start' and 'marker-end' you can add markers at the start and the end of the edge. To attach markers along an edge, you have two possibilities. First, you can use the 'marker-mid' attribute that adds the referenced marker to every point of a path that is not the first or the last point. Second, you can specify more than one path and use the attributes 'marker-start' and 'marker-end'.

If you are using the 'marker-mid' attribute you can influence the rendering of the marker by adding the attributes 'size' and 'minimumLength'. With these attributes you are able to define edges with a fixed design. For example, the following marker is stretched to 100% of the path's length, it is attached to, but will be hidden, if the path's length is below 10:

```
<marker id="mid" oryx:size="100%" oryx:minimumLength="10"
  refX="5" refY="5" markerUnits="userSpaceOnUse"
  markerWidth="10" markerHeight="10" orient="auto">
  <rect x="0" y="0" width="10" height="10" fill="black" stroke="black"/>
</marker>
```

Note that if you allow to add dockers at runtime to the path on which you have set the 'marker-mid' attribute, the marker will be rendered at the position of each docker.

The other way to attach markers along the edge is to define the edge with more than one path. For example, the following edge definition results in figure 4.7:

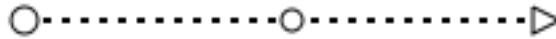


Figure 4.7: An example for an edge defined with two paths.

```

<svg
  xmlns="http://www.w3.org/2000/svg"
  xmlns:oryx="http://www.b3mn.org/oryx"
  version="1.0"
  oryx:edge="edge" >
  <defs>
    <!--Markers' definition is not shown here-->
  </defs>
  <g id="edge">
    <path d="M10 50 L110 50" stroke="black" fill="none" stroke-width="2"
      stroke-dasharray="3, 4" marker-start="url(#start)"
      marker-end="url(#mid)" />
    <path d="M110 60 L210 50" stroke="black" fill="none" stroke-width="2"
      stroke-dasharray="3, 4" marker-end="url(#end)" />
  </g>
</svg>

```

The circle in the middle of the edge is attached as the end marker of the first 'path' element. In this approach, dockers are created at each point of each path. So, there is a docker at the point where the circle is attached. This makes it possible to move the circle along the edge. In this approach the dockers at the beginning and the end of each path cannot be dynamically deleted at run time, because otherwise the markers would disappear and cannot be restored.

4.3.1 Attributes

allowDockers

- Value: yes | no
- Initial: yes
- Optional: yes
- Applies to: SVG path elements

Description: 'allowDockers' set if dockers can dynamically be added to that path.

minimumLength

- Value: float
- Initial: 0
- Optional: yes
- Applies to: SVG marker elements

Description: 'minimumLength' specifies the path's minimum length so that the marker will be rendered. If the path is less long than this attribute's value, the marker will not be visible. This attribute will only be considered, if the marker is referenced from a 'marker-mid' attribute.



Figure 4.8: The icon set for the workflow nets stencil set.

size

- Value: percent
- Initial: -
- Optional: yes
- Applies to: SVG marker elements

Description: 'size' specifies the size of the marker element relative to the path's length. The value must be a number between 0 and 100 followed by a percent sign. If the attribute is not set, the marker's size will not be changed. This attribute will only be considered, if the marker is referenced from a 'marker-mid' attribute.

4.4 Picture Files

Besides the SVG files you have to create a picture file for each stencil. The files should have a size of 32 to 32 pixel. The files format must be a common format for pictures on the web such as JPEG, PNG or BMP. Those pictures are used as icons for the stencils in the user interface. Figure 4.8 shows the icons for the workflow nets stencil set.

4.5 Implementation Remarks

This definition of a stencil's graphical representation shall give you the possibility to design any stencil you like. However, we had some problems during the implementation, particularly because the Mozilla Firefox 2.0+ Browser does not fully support the SVG specification [10].

If you add text elements to your stencils, please define the 'font-size' attribute in each element. Otherwise, the font size is set to a default font size specified in the configuration file of Oryx. The text flickering when rendering text elements is based upon the bug 293581 of the Mozilla Firefox browser. Furthermore, another bug in the Firefox browser causes that only the text stroke is rendered on Apple Computers with Mac OS X.

Do not use transformations in the SVG documents, because they will not be considered in case of resizing the shapes.

Although you can define a position for the docker of a node, the docker will always be positioned in the center of the node. The consideration of the docker's attributes is simply not implemented.

Avoid using SVG circle elements within your SVG files, if you want to resize the circle elements. There is a bug in the update algorithm for circle elements. Instead, use ellipse of path elements.

The attributes 'size' and 'minimumLength' of a marker element are not implemented. So, it is not possible to define edges with a fixed design.

Although you can use paths to draw not only lines, but also curves, arcs and so on, Oryx only uses the 'LineTo' command (letters 'L' and 'l'). You should not use any other commands, because this can cause an unwanted result.

5. Creating a Stencil Set Description

In chapter 4 I described how to create the graphical representation of stencils. But with all these SVG and picture files the work is not done, yet. Some more information about the stencils and their relationships have to be added to the stencil set in order to use it in Oryx. The stencil set description contains all the information. It is a single file in JSON format (see section 2.2 for more information on JSON).

5.1 Stencil Set Description

A stencil set consists of a set of stencils, rules and additional attributes. The basic structure of our Workflow Nets example looks like this:

```
{
  "title": "Workflow Nets",
  "namespace": "http://www.example.org/workflownets",
  "description": "Simple stencil set for Workflow Nets.",
  "stencils": [/*...*/],
  "rules": {/*...*/}
}
```

The title simply names the stencil set. The provided namespace is the unique identifier for this stencil set. You can also provide a short description of the stencil set. Furthermore, the stencil set contains an array of stencils and a rules object. Both stencils and rules are described later in this chapter.

5.1.1 Attributes

description

- Type: string
- Initial: ""
- Optional: yes

Description: 'description' is an optional short description of the stencil set.

namespace

- Type: string
- Initial: -
- Optional: no

Description: 'namespace' is a mandatory attribute that is the unique identifier for this stencil set. This specification requires to use URIs for this attribute. The identifier is very important for avoiding conflicts, if you use more than one stencil set.

rules

- Type: object
- Initial: an empty object
- Optional: yes

Description: 'rules' is an object that specifies the rules for a stencil set, e. g. which nodes can be connected. The rules object and how to create rules is explained in chapter 6.

stencils

- Type: array
- Initial: an empty array
- Optional: yes

Description: 'stencils' is an array of stencil objects. The stencil objects are explained in section 5.2.

title

- Type: string
- Initial: ""
- Optional: yes

Description: 'title' names the stencil set.

5.2 Stencil Description

Each object in the stencils array of the stencil set description is one stencil. Let us have a look at the object for a workflow net activity:

```
/*...*/
"stencils":
[
  {
    "type":"node",
    "id":"Activity",
    "title":"Activity",
    "description":"An atomic activity.",
```

```

    "view": "activity.svg",
    "icon": "activity.png",
    "roles": ["activitySource", "activityTarget"],
    "properties": [/*...*/]
  }/*,
  ...*/
]/*,
...*/

```

Every stencil has to have an id that is unique within its stencil set. The type attribute specifies that the activity is a node. Next, there is a title and a short description that can be used by the Oryx user interface. You also have to reference the SVG and the picture file for that stencil. We have also added two roles to the stencil, "activitySource" and "activityTarget". Roles are used to specify rules so that you can define rules that affect a group of stencils (see chapter 6 for further information). Properties can be defined to add additional information to the stencil. The description of properties is explained in section 5.3.

This simple example does not use all attributes a stencil can have, but the most important ones. A detailed explanation for all available attributes can be found in the next subsection.

5.2.1 Attributes

description

- Type: string
- Initial: ""
- Optional: yes

Description: 'description' is a short description of the stencil.

groups

- Type: array
- Initial: an empty array
- Optional: yes

Description: 'groups' specifies zero or more groups (defined as string) that can be used to group the stencils of a stencil set. For example, this attribute defines the groups in the Oryx Shape Repository.

icon

- Type: string
- Initial: -
- Optional: no

Description: 'icon' sets the name of the picture file of a stencil. The file has to be located in the "icons" folder of the stencil set.

id

- Type: string
- Initial: -
- Optional: no

Description: 'id' is the unique identifier for a stencil.

properties

- Type: array
- Initial: an empty array
- Optional: yes

Description: 'properties' is an array of zero or more property objects that define additional information of a stencil. The property description is explained in section 5.3.

roles

- Type: array
- Initial: an empty array
- Optional: yes

Description: 'roles' specifies zero or more roles (defined as string) of a stencil. Roles are used to specify rules. There are no predefined roles with special abilities. The id of a stencil is an implicit role.

title

- Type: string
- Initial: ""
- Optional: yes

Description: 'title' names the stencil.

type

- Type: "node" | "edge"
- Initial: -
- Optional: no

Description: 'type' sets, if a stencil is a node or an edge.

view

- Type: string
- Initial: -
- Optional: no

Description: 'view' sets the name of the SVG file that contains the graphical representation of a stencil. The file has to be located in the "view" folder of the stencil set.

serialize

There are three more optional attributes that are different from the others: 'serialize', 'deserialize' and 'layout'. These attributes are different, because their values are JavaScript functions. Note that using these attributes requires a good knowledge of JavaScript programming.

The 'serialize' function of a stencil is called, whenever a shape is persisted into the DOM. The values of all properties are serialized automatically, so you don't have to do this yourself. Furthermore, all necessary information to restore the model are saved, too. This callback gives you the possibility to persist additional information and manipulate the serialized data that for example other applications call for. To define the function add the following to a stencil object:

```
{
  /*...*/
  "serialize":function(shape, data) {/*...*/},
  /*...*/
}
```

The parameter 'shape' is of type `ORYX.Core.Shape` and is the object that represents the stencil on the canvas at run time. See [2] for information about the API of the parameter. 'data' is an array of objects with the properties 'name', 'prefix', 'value', and 'type' that contains all information that is automatically serialized by Oryx. For example

```
[
  {
    name:outgoing,
    prefix:oryx,
    value:5,
    type:'literal'
  }
]
```

is an array with one object. The 'prefix' is the eRDF schema and 'name' is the identifier for that data in the schema. The property 'value' contains the value you want to add. The 'type' property can either be 'literal' or 'resource'. It is needed for the internal eRDF data format. In short, 'literal' sets to store the value as a literal, whereas 'resource' defines the value to be a reference. For more information on eRDF and the internal data format, see [3].

The return value must be of the same type as the 'data' parameter. The usual way is to return the 'data' parameter after adding objects to it and manipulating the existing objects in it. Keep in mind that Oryx needs the already defined information for restoring the model (see table 5.1 for an overview of the triples that are serialized by Oryx). So, if you manipulate any data, you will probably have to reset these changes during the deserialization.

As an example, let us define a callback for the workflow net activity stencil that counts all outgoing control flows and adds it to the serialized data:

Prefix	Name	Type	Description
oryx	bounds	literal	The value is a comma separated list of four numbers that define the shape's bounds.
	docker	literal	If the shape is a node and if it has a docker, the position of the docker will be serialized.
	dockers	literal	If the shape is an edge, the position of all dockers will be serialized.
raziel	outgoing	resource	For each outgoing shape a reference to the outgoing shape is stored.
	parent	resource	A reference to the parent shape.

Table 5.1: The triples that are serialized for each shape.

```

/*...*/
"serialize":function(shape, data) {
  var numOfOutgoings = shape.getOutgoingShapes().length;
  data.push({
    name:"numOfOutgoings",
    prefix:"oryx",
    value:numOfOutgoings,
    type:"literal"
  });
  return data;
},
/*...*/

```

deserialize

As mentioned above, the 'deserialize' attribute is also a JavaScript function that is called while loading an object of a stored model. In detail, it is called before Oryx processes the serialized data. So, it can be used to manipulate the data before Oryx treats it. To add a deserialize callback, add the following to a stencil's definition:

```

{
  /*...*/
  "deserialize":function(shape, data) { /*...*/ },
  /*...*/
}

```

The signature is the same than the one of the 'serialize' attribute. 'shape' is of type `ORYX.Core.Shape` and 'data' is an array of objects with the same structure as described for the 'serialize' function. The return value must also be of the same type than the 'data' parameter.

layout

The third function is called whenever the position and size of a shape is recalculated. This callback can be used to manipulate the bounds of the current shape and its child shapes. The definition is as follows:

```

{
  /*...*/
  layout: function(shape) { /*...*/ },
  /*...*/
}

```

The parameter is of type `ORYX.Core.Shape`. It defines the context the function is called in. If a shape changes the layout algorithm of the parent shape, grandparent shape and so on will also be called, beginning with the top most callback. It is recommended not to use this function for anything else than manipulating the position and size of shapes.

Examples for using this callback can be found in the BPMN stencil set [4].

5.3 Property Description

Let us have a look at a property of the workflow nets activity that sets the caption:

```

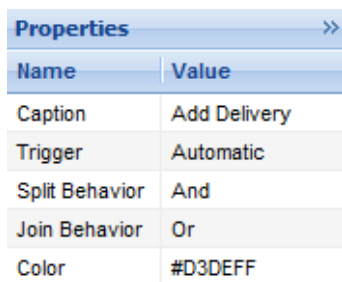
"properties": [
  {
    "id": "caption",
    "type": "String",
    "title": "Caption",
    "value": "",
    "description": "",
    "readonly": false,
    "optional": true,
    "refToView": "caption",
    "wrapLines": true
  } /*,
  ... */
]

```

Properties also have an id that must be unique within its stencil. The 'type' attribute specifies that it is a "String" property. The 'value' attribute sets the default value of this property. If the 'optional' attribute is set to false, you will have to set a default value. 'wrapLines' is an attribute, that is only useful, if the property is of type "String". Most property types have special attributes that are listed in the next subsection.

The most interesting attribute is 'refToView'. It references a SVG element in the graphical representation of the stencil. In this example, the value "caption" is an id of a SVG text element. If a text element with this id is found in the graphical representation, the value of this property will be rendered on the canvas. Dependant on the property's type you can reference different types of SVG elements (see table 5.2).

The properties are accessible in the property window of the Oryx user interface (see figure 5.1. The user interface avoids that the user enters invalid values. For example, a property of type "Color" can be set with a color palette (see figure 5.2).



Name	Value
Caption	Add Delivery
Trigger	Automatic
Split Behavior	And
Join Behavior	Or
Color	#D3DEFF

Figure 5.1: The property window and the properties of an activity.

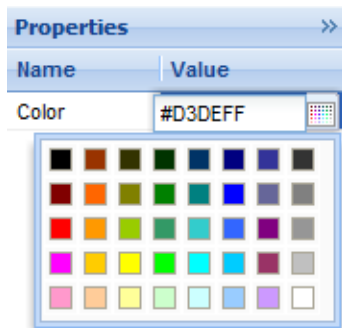


Figure 5.2: A color property.

Property type	SVG elements	Description
Boolean	SVGElement	Sets the visibility of any SVG element. In case of edges you can reference SVG marker elements, but not the child elements of a marker.
Choice	SVGTextElement SVGElements	The property's value is rendered in the text element. Each property item can reference a SVG element. Only the chosen one is shown, the others are hidden.
Color	SVGElement	Sets the 'fill' and/or 'stroke' attribute of a SVG element.
DateTime	-	-
Float	SVGElement SVGTextElement	Sets the 'fill-opacity' and/or 'stroke-opacity' attribute of a SVG element. The property's value is rendered in the text element.
Integer	SVGTextElement	The property's value is rendered in the text element.
String	SVGTextElement	The property's value is rendered in the text element.
Url	SVGAElement SVGImageElement	Sets the 'href' attribute. Sets the 'href' attribute.

Table 5.2: Property types and SVG element types

5.3.1 Attributes

dateFormat

- Type: string
- Initial: "m/d/y"
- Optional: yes

Description: 'dateFormat' sets the format of a date. See table 5.4 for a full list of the format syntax. For example, the initial format looks like "01/18/07".

description

- Type: string
- Initial: ""
- Optional: yes

Description: 'description' is a short description of the property.

fill

- Type: boolean
- Initial: false
- Optional: yes

Description: 'fill' specifies, if the property sets the 'fill' attribute of the referenced SVG element. Only useful, if the property is of type "Color".

fillOpacity

- Type: boolean
- Initial: false
- Optional: yes

Description: 'fillOpacity' specifies, if the property sets the 'fill-opacity' attribute of the referenced SVG element. Only useful, if the property is of type "Float". You should set the 'min' attribute to 0.0 and the 'max' attribute to 1.0, because other values are not allowed in the 'fill-opacity' attribute.

id

- Type: string
- Initial: -
- Optional: no

Description: 'id' is the identifier for a property. Make sure that all properties of a stencil have different ids.

items

- Type: array
- Initial: an empty array
- Optional: yes

Description: 'items' is an array of property item objects. If the property is of type "Choice", this array contains the items of the choice list. See section 5.4 for more information on property items.

length

- Type: integer
- Initial: -
- Optional: yes

Description: 'length' specifies the maximum length of the property's value, if the property is of type "String". If the attribute is not set, the length is not limited.

max

- Type: integer
- Initial: -
- Optional: yes

Description: 'max' specifies the maximum value of the property, if the property is of type "Integer" or "Float". If the attribute is not set, the value is not limited.

min

- Type: integer
- Initial: -
- Optional: yes

Description: 'min' specifies the minimum value of the property, if the property is of type "Integer" or "Float". If the attribute is not set, the value is not limited.

optional

- Type: boolean
- Initial: true
- Optional: yes

Description: 'optional' specifies, if the property must have a value. If 'optional' is set to false, you must set a valid value in the attribute 'value'.

prefix

- Type: string
- Initial: "oryx"
- Optional: yes

Description: 'prefix' sets in which eRDF schema a property is stored. This is very important, if you want to build process models with Oryx that can later be used by other applications. Note that using another prefix requires that an eRDF schema is defined for that prefix. Please refer to [3] for more information about eRDF and the storage format used by Oryx.

readonly

- Type: boolean
- Initial: false
- Optional: yes

Description: 'readonly' specifies, if the property can be changed by the user.

refToView

- Type: string | array
- Initial: ""
- Optional: yes

Description: 'refToView' specifies an id as string or an array of ids of SVG elements in the graphical representation of a stencil. If this attribute is set, the property will manipulate the graphical representation at run-time, e. g. changing the color or rendering text. Dependant on the property's type you can reference different types of SVG elements (see table 5.2).

stroke

- Type: boolean
- Initial: false
- Optional: yes

Description: 'stroke' specifies, if the property sets the 'stroke' attribute of the referenced SVG element. Only useful, if the property is of type "Color".

strokeOpacity

- Type: boolean
- Initial: false
- Optional: yes

Description: 'strokeOpacity' specifies, if the property sets the 'stroke-opacity' attribute of the referenced SVG element. Only useful, if the property is of type "Float". You should set the 'min' attribute to 0.0 and the 'max' attribute to 1.0, because other values are not allowed in the 'stroke-opacity' attribute.

title

- Type: string
- Initial: ""
- Optional: yes

Description: 'title' names the property.

type

- Type: "Boolean" | "Choice" | "Color" | "DateTime" | "Float" | "Integer" | "String" | "Url"
- Initial: -
- Optional: no

Type	Attribute
Boolean	-
Choice	items
Color	fill stroke
DateTime	dateFormat
Float	fill-opacity max min stroke-opacity
Integer	max min
String	length wrapLines
Url	-

Table 5.3: Property types and their attributes.

Description: 'type' is a mandatory attribute that specifies the type of the property. Depending of the type, other attributes are useful or not (see table 5.3).

value

- Type: string | boolean | integer | float
- Initial: ""
- Optional: yes

Description: 'value' specifies the default value of the property. If 'optional' is set to false, this attribute is mandatory.

wrapLines

- Type: boolean
- Initial: false
- Optional: yes

Description: 'wrapLines' specifies, if line feeds are allowed in properties of type "String".

5.4 Property Item Description

Property items are only necessary, if a property is of type "Choice". A property of this type must have an attribute 'items' that is an array of property item objects.

In our workflow nets example, let us have a look at the property of the activity stencil for setting the trigger (see figure 5.3):

```
"properties": [
  /*...*/
  {
    "id":"trigger",
```

Format	Output	Description
d	10	Day of the month, 2 digits with leading zeros
D	Wed	A textual representation of a day, three letters
j	10	Day of the month without leading zeros
l	Wednesday	A full textual representation of the day of the week
S	th	English ordinal day of month suffix, 2 chars (use with j)
w	3	Numeric representation of the day of the week
z	9	The julian date, or day of the year (0-365)
W	01	ISO-8601 2-digit week number of year, weeks starting on Monday (00-52)
F	January	A full textual representation of the month
m	01	Numeric representation of a month, with leading zeros
M	Jan	Month name abbreviation, three letters
n	1	Numeric representation of a month, without leading zeros
t	31	Number of days in the given month
L	0	Whether its a leap year (1 if it is a leap year, else 0)
Y	2007	A full numeric representation of a year, 4 digits
y	07	A two digit representation of a year
a	pm	Lowercase Ante meridiem and Post meridiem
A	PM	Uppercase Ante meridiem and Post meridiem
g	3	12-hour format of an hour without leading zeros
G	15	24-hour format of an hour without leading zeros
h	03	12-hour format of an hour with leading zeros
H	15	24-hour format of an hour with leading zeros
i	05	Minutes with leading zeros
s	01	Seconds, with leading zeros
O	-0600	Difference to Greenwich time (GMT) in hours
T	CST	Timezone setting of the machine running the code
Z	-21600	Timezone offset in seconds (negative if west of UTC, positive if east)

Table 5.4: Date parsing and format syntax. The format specification is the format specification of the Ext JavaScript library (<http://extjs.com>) that is used for the user interface of Oryx.

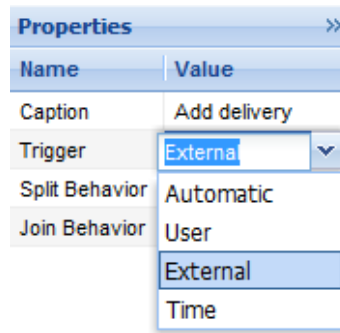


Figure 5.3: Setting the trigger in the property window.

```

"type": "Choice",
"title": "Trigger",
"value": "Automatic",
"optional": false,
"items": [
  {
    "title": "Automatic",
    "value": "Automatic",
    "refToView": "automatic"
  },
  {
    "title": "User",
    "value": "User",
    "refToView": "user"
  },
  {
    "title": "External",
    "value": "External",
    "refToView": "external"
  },
  {
    "title": "Time",
    "value": "Time",
    "refToView": "time"
  }
]
}
]

```

You can choose between "Automatic", "User", "External" and "Time". The default value is set in the 'value' attribute of the property. The attribute simply contains the value of the selected property item. Like stencil and property objects a property item has a title and a value. The value of a property item is mandatory and each value must be unique within a property. Each property item has a 'refToView' property. The referenced SVG element of the selected property item will be rendered, whereas the referenced elements of the other property items will be hidden.

5.4.1 Attributes

refToView

- Type: string
- Initial: ""
- Optional: yes

Description: 'refToView' specifies an id of a SVG element in the graphical representation of a stencil. If this attribute is set, the property item will manipulate the graphical representation at run-time. If this property item is selected, the SVG element will be shown, otherwise hidden.

title

- Type: string
- Initial: ""
- Optional: yes

Description: 'title' names the property item.

value

- Type: string | boolean | integer | float
- Initial: ""
- Optional: no

Description: 'value' specifies the value of the property. It is a mandatory attribute and all property item values of a property must be unique.

5.5 Implementation Remarks

Almost everything of this part of the stencil set specification is implemented in Oryx. The application expects exactly the document structure described above. There are only some drawbacks when using the 'layout' callback. Using this function can be very tricky, because for example you have to consider not to break the minimum or maximum size of a shape. It can happen that a shape will not be redrawn, although your layout algorithm has changed it. However, do not call the 'update' method on a shape within your 'layout' function, because that can result in an infinite loop.

Hopefully, further development on Oryx will include the implementation of automatic layouting algorithms and within this scope a refactoring of the current layouting implementation.

6. Creating Rules

In chapter 5 we explained how to write a stencil set description except one part, the rules. Rules are applied to stencils and should help the user of Oryx to build valid process models. For example, you can specify that an edge with id "A" can only connect nodes with id "B".

There are different kinds of rules. Connection rules specify which nodes can be connected by which edge. Cardinality rules set minimum and maximum cardinalities of stencils, as well as incoming and outgoing edges. At last the containment rules specify possible containment relationships between stencils.

If there is a rule that you cannot express with these three types of rules, you can use the rules extensions. These extensions consist of JavaScript functions that are automatically called by Oryx when checking rules. However, using the extensions requires a good knowledge about JavaScript programming.

Rules are specified in a JSON object that is the value of a stencil set's 'rules' attribute (see section 5.1). The basic structure of a stencil set with its rules looks like this:

```
{
  "title": "Workflow Nets",
  "namespace": "http://www.example.org/workflownets",
  "description": "Simple stencil set for Workflow Nets.",
  "stencils": [/*...*/],
  "rules":
  {
    "connectionRules": [/*...*/],
    "cardinalityRules": [/*...*/],
    "containmentRules": [/*...*/],
    "canConnect": function(args) { return args.result; },
    "canContain": function(args) { return args.result; }
  }
}
```

Each type of rule is an own attribute. The structure of the attributes' values are described in the following sections.

The last two attributes' values are JavaScript functions. These are the callbacks for the rules extensions and will be explained in section 6.4.

All these attributes are optional, but connection and containment rules uses a white list approach. If you do not specify any rules, you will not be able to do anything with the stencils.

The rules are based on roles. Each stencil can have any number of roles, specified in their 'roles' attribute. The id of a stencil is also a role that is unique for each stencil. With roles you can specify rules that apply to a set of stencils. That eases defining the right rules and minimizes the typing work for the stencil set designer.

6.1 Connection Rules

As mentioned earlier, connection rules specify which nodes can be connected. For our example, the workflow nets, we have to specify rules that guarantee that we can only build bipartite graphs. That means, activities and conditions have to alternate. The rules look like this:

```
"connectionRules": [
  {
    "role": "controlflow",
    "connects": [
      {
        "from": "activitySource",
        "to": "conditionTarget"
      },
      {
        "from": "conditionSource",
        "to": "activityTarget"
      }
    ]
  }
]
```

Remember that we added the roles "activitySource" and "activityTarget" to the activity stencil in section 5.2. These roles can now be used to specify the rules. First, the role the rules are applied to have to be set. The 'connects' attribute is an array of objects with a 'from' and a 'to' attribute. These rules specify that a 'controlflow' can connect an 'activitySource' to a 'conditionTarget' and a 'conditionSource' to an 'activityTarget'. But how to specify that an input condition is not allowed to have an incoming control flow and an output condition is not allowed to have an outgoing control flow? This is very simple. Whereas the condition has both the role 'conditionSource' and 'conditionTarget', the input condition only has the role 'conditionSource' and the output condition only has the role 'conditionTarget'. These are all connection rules we need for our example.

6.1.1 Attributes

role

- Type: string
- Initial: -
- Optional: no

Description: 'role' specifies the role the connection rules are applied to.

connects

- Type: array
- Initial: an empty array
- Optional: yes

Description: 'connects' is an array of objects with the attributes 'from' and 'to'. Each of those objects is one connection rule.

from

- Type: string
- Initial: -
- Optional: no

Description: 'from' specifies the role that is the source of the connection. Any stencil with that role is a possible source.

to

- Type: string | array
- Initial: -
- Optional: no

Description: 'to' specifies the role or an array of roles that are the target of the connection. Any stencil with one of the roles is a possible target.

6.2 Cardinality Rules

In a workflow net there must be exactly one input condition and exactly one output condition. This can be expressed with cardinality rules:

```
"cardinalityRules": [  
  {  
    "role":"inputcondition",  
    "minimumOccurrence":1,  
    "maximumOccurrence":1  
  },  
  {  
    "role":"outputcondition",  
    "minimumOccurrence":1,  
    "maximumOccurrence":1  
  },  
]
```

The 'role' attribute specifies the role we want to express rules for. You can set the occurrence bounds with the attributes 'minimumOccurrence' and 'maximumOccurrence'. We could have also added a role 'iocondition' to both the input and output condition stencil and specify a cardinality rule for this role. But this example should remind you that you can also use a stencil's id as a role.

The context of the occurrence rules is always the parent shape. That means for example that when dragging a shape A over a shape B and B is allowed to contain A (see section 6.3) than the minimum and maximum occurrence for A in B is checked, but not the occurrence on the canvas.

But there is more you can express with cardinality rules. It is possible to specify, how much incoming or outgoing edges of one type a stencil can have. Let us assume that it would not be allowed that an input condition has more than one outgoing control flow. The rule would look like this:

```
{
  "role":"inputcondition",
  "minimumOccurrence":1,
  "maximumOccurrence":1,
  "outgoingEdges": [
    {
      "role":"controlflow",
      "maximum":1
    }
  ]
}
```

You can also do the same for incoming edges with the attribute 'incomingEdges'.

6.2.1 Attributes

role

- Type: string
- Initial: -
- Optional: no

Description: 'role' specifies the role the rules are applied to.

minimumOccurrence

- Type: integer
- Initial: 0
- Optional: yes

Description: 'minimumOccurrence' is the minimum number a stencil must occur in a diagram. A negative number or a number greater than the value of 'maximumOccurrence' is an error.

maximumOccurrence

- Type: integer
- Initial: -
- Optional: yes

Description: 'maximumOccurrence' is the maximum number a stencil can occur in a diagram. A negative number or a number smaller than the value of 'minimumOccurrence' is an error. If this attribute is not set, the maximum number is not limited.

outgoingEdges

- Type: array
- Initial: an empty array
- Optional: yes

Description: 'outgoingEdges' is an array of objects with the attributes 'role', 'minimum' and 'maximum'. It specifies how much outgoing edges a stencil is allowed to have.

incomingEdges

- Type: array
- Initial: an empty array
- Optional: yes

Description: 'incomingEdges' is an array of objects with the attributes 'role', 'minimum' and 'maximum'. It specifies how much incoming edges a stencil is allowed to have.

role (within 'outgoingEdges' or 'incomingEdges')

- Type: string
- Initial: -
- Optional: no

Description: 'role' specifies the role the 'minimum' and 'maximum' attributes are applied to.

minimum

- Type: integer
- Initial: 0
- Optional: yes

Description: 'minimum' is the minimum number of outgoing or incoming edges with the specified role a stencil must have. A negative number or a number greater than the value of 'maximum' is an error.

maximum

- Type: integer
- Initial: -
- Optional: yes

Description: 'maximum' is the maximum number of outgoing or incoming edges with the specified role a stencil is allowed to have. A negative number or a number smaller than the value of 'minimum' is an error. If this attribute is not set, the maximum number is not limited.

6.3 Containment Rules

Parent-child-relationships are specified with containment rules. With these rules you can define which nodes can be added to another node. Edges do not need containment rules, because they have implicit parents dependant on the nodes' parents an edge is connected to. In our workflow net example we have to specify a containment rule for the workflow net diagram stencil. This stencil defines the resource's type that can contain workflow nets. The id of the stencil is "diagram".

```
"containmentRules": [
  {
    "role": "diagram",
    "contains": [
      "activity",
      "condition",
      "inputcondition",
      "outputcondition"
    ]
  }
]
```

The role we want to specify the child roles for is "diagram". The 'contains' attribute specifies an array of roles that the "diagram" role can contain.

Keep in mind that if the resource you are currently looking at with Oryx is not of type "diagram" of this stencil set, you cannot model a workflow net on the canvas.

6.3.1 Attributes

role

- Type: string
- Initial: -
- Optional: no

Description: 'role' specifies the role the rule is applied to.

contains

- Type: array
- Initial: -
- Optional: no

Description: 'contains' is an array of roles (specified as strings). All stencils that have at least one of these roles can be added to a stencil with the role of the 'role' attribute.

6.4 Rules Extensions

In the last three sections, I explained how to express connection, cardinality and containment rules. However, not all rules can be expressed with those constructs. That is why I introduced rules extensions. As mentioned above, these extensions are JavaScript functions that are automatically called by Oryx while checking the rules. There are two callbacks, 'canConnect' and 'canContain'. Each callback gets one object as a parameter, but the content of the object is callback specific.

It is only possible to express more restrictive rules than the existing ones. From this it follows that if a connection rule prohibits a specific connection, the callback will not be called, because it is not allowed to cancel the restriction.

6.4.1 Connection Extension

If you want to extend the connection rules, you have to implement the 'canConnect' callback. Every time a connection will be tested, the callback will be applied to the current stencils. The signature is as follows:

```
"rules":
{
  /*...*/
  "canConnect": function(args) { /*...*/ }
  /*...*/
}
```

'args' is an object that contains all necessary information to check connections. The parameter contains references to objects of type `ORYX.Core.StencilSet.Stencil` and `ORYX.Core.Shape`. It is extremely important to get to know the APIs of the `ORYX.Core.StencilSet` package (see appendix A), as well as the `ORYX.Core.Shape` class (see [2]) before working with the rules extensions. In short, the `ORYX.Core.StencilSet` package are classes for accessing stencil set descriptions. Whereas the `ORYX.Core.Shape` class represents an object on the canvas of the running Oryx editor.

The following properties are specified in the parameter 'args':

edgeShape: The shape object of the edge. 'edgeShape' is not mandatory. So, you cannot rely on the existence.

edgeStencil: The stencil object of the edge. This property is always available.

sourceShape: This is the shape object the start of the edge is or shall be connected to. This property is not mandatory.

sourceStencil: The stencil object of the source of the connection. If 'sourceShape' is set, this property will be defined, too. But it is also possible that only 'sourceStencil' is defined and 'sourceShape' is not.

targetShape: The same for 'sourceShape' applies to 'targetShape'.

targetStencil: Again, the same for 'sourceStencil' applies to 'targetStencil'.

Note that at least the source or the target is defined, but it is not mandatory that both are specified. For example, if the editor wants to find out all possible outgoing edges of a node, a target will not be set.

The return value is a boolean that set if the connection is allowed.

As an example, let us assume a rule for conditions that says that the ratio of the number of incoming control flows and outgoing control flows must either be 0:N, 1:N, N:0, N:1 (N is any positive integer). That is if there is more than one incoming control flow, there can only be at most one outgoing control flow and vice versa.

```
"canConnect": function(args) {
  if(args.sourceShape &&
    args.sourceStencil.id() === args.sourceStencil.namespace() +
    "condition" &&
    args.sourceShape.getIncomingShapes().length >= 2 &&
```

```

    args.sourceShape.getOutgoingShapes().length >= 1) {

        return false;
    }
    if(args.targetShape &&
        args.targetStencil.id() === args.targetStencil.namespace() +
            "condition" &&
        args.targetShape.getOutgoingShapes().length >= 2 &&
        args.targetShape.getIncomingShapes().length >= 1) {

        return false;
    }
    return true;
}

```

6.4.2 Containment Extension

The 'canContain' function is called whenever the containment rules are checked. The signature is very similar to the signature of the 'canConnect' function:

```

"rules":
{
    /*...*/
    "canContain": function(args) { /*...*/ }
    /*...*/
}

```

But, the parameter's properties are different:

containingShape: The shape object of the parent shape candidate. This property is not mandatory.

containingStencil: This property references the stencil object of the parent candidate. 'containingStencil' is always defined.

containedShape: The reference to the shape object, that should be added to the parent candidate. This property is not mandatory.

containedStencil: 'containedStencil' references the stencil object of the child candidate and is always available.

Again, the return value is a boolean that set if the parent-child relationship is allowed.

6.5 Stencil Set Extensions

If you want to create a stencil set that extends an existing stencil set, you have to define rules to add the new stencils to stencils of the existing stencil set and to connect stencils of both stencil sets. This can be done by using globally unique roles. To make a role globally unique simply prepend the namespace of the stencil set the role is defined in followed by "#" to the role. For example, we want to introduce data objects to the workflow nets stencil set. We define a new stencil set and create a new stencil with the id "dataobject". We then define two rules in the new stencil set. The first rule specifies that we can add our data object to the diagram stencil of the workflow nets stencil set. The next rule set, that we can connect activities and data objects with control flows.

```
{
  /*...*/
  "containmentRules": [
    {
      "role": "http://www.example.org/workflownets#diagram",
      "contains": ["dataobject"]
    }
  ],
  "connectionRules": [
    {
      "role": "http://www.example.org/workflownets#controlflow",
      "connects": [
        {
          "from": "http://www.example.org/workflownets#activity",
          "to": "dataobject"
        },
        {
          "from": "dataobject",
          "to": "http://www.example.org/workflownets#activity"
        }
      ]
    }
  ]
}
```

6.6 Implementation Remarks

Oryx uses and understands most of the rules specified in the previous sections. However, the 'minimumOccurrence' and 'minimum' attribute is not considered in Oryx. The reason is very simple: we were running out of time and minimum cardinality rules are not the most important ones. Furthermore, it is not easy to handle these rules. What should the editor do, when the user creates a new empty diagram? In this situation all minimum cardinality rules are broken. We could have implemented to automatically create all shapes till no minimum cardinality rule is broken. But because we have no automatic layouting of graph elements, the result would be ugly.

7. Outlook

The goal of this final bachelor's paper is to specify a way for defining stencil sets for Oryx and to give the stencil set designer a huge flexibility. We achieved to implement most of the specification during the project. As mentioned in the sections "Implementation Remarks" there are some features that we haven't implemented. Beside those feature we have had some ideas that I did not specify in this document. In this chapter I will present some of the ideas for extending this specification in the future.

Extending SVG support

As mentioned in chapter 4 not all SVG constructs are supported for nodes or edges. This could be extended in the future. Additionally, Oryx could not only support SVG representations, but also XHTML representations or representations drawn with the HTML canvas tag. Of course, this entails changes in several core classes of Oryx, but it would greatly extend the flexibility.

Inheritance

If you have several very similar stencils, you will have to describe the properties for each stencil separately. Introducing stencil inheritance would ease the stencil set description. You simply describe the properties in one stencil and the other stencils can inherit those properties. Additionally, the possibility to define abstract stencils (like abstract classes in Java) is a useful feature. Those abstract stencils do not need a graphical representation, but they can define properties other stencils can inherit.

Grammar

The current specification of the rules aims at supporting the designer by avoiding invalid constructs while modeling processes. Future specifications could define the description of a complete grammar for on the fly validation of the current process. Specifying the description of a grammar was out of scope of this paper and because the browser's JavaScript environment is single threaded, we did not implement a validator that periodically checks the model. But e.g. Google Gears (<http://code.google.com/apis/gears/>) offers multi threading. Using Google Gears could be a way for improving the editor's performance and implementing a validator.

N:M Containment Relationships

The current specification offers 1:N containment relationships between stencils. That is each object has exactly one parent. But there are use cases for N:M containment relationships. This could be achieved by extending the containment rules with cardinalities and implementing a plugin (see [2]) that offers a user interface for relating a shape to several other shapes. The plugin could also offer an automatic layouting of the related shapes.

Dependencies between Stencil Sets and Plugins

A stencil set's JSON file can easily be extended with additional information. If you want to have a feature for your stencil set that is currently not offered by Oryx, you could extend your stencil set with additional information and implement a plugin that uses the information. To make this possible the stencil set API has to be extended with a method to access those additional attributes (like `stencil.attribute("nameOfAttribute")`). An additional attribute can also be a function to add stencil set specific code. To make sure that a plugin is available when using a specific stencil set, it is necessary to list the required plugins in the stencil set description. Those listed plugins can then be loaded by Oryx beside the stencil set.

A. API

In this appendix I will describe the API of the `ORYX.Core.StencilSet` package. It contains the classes `StencilSets`, `Stencil`, `Property`, `PropertyItem` and `Rules`. Instances of those classes are used in some of the callbacks you can define in a stencil set description (see chapter 5). Most of the classes simply define getter-methods for the attributes specified in the stencil set description. Those getter-methods are only named here. A description of the attributes can be found in chapter 5. Only the class `Rules` differs from the others, because it implements the API Oryx uses for checking the rules. This class is not used within the callbacks of the stencil set description.

The methods are described as follows:

```
methodName([parameter1:parameter type]+)[:Return type]
```

A.1 ORYX.Core.StencilSet

Methods for loading and accessing stencil sets and rules are defined directly in the package.

loadStencilSet(url:String, editorId:String)

Loads a stencil set description from the specified url and assigns the loaded stencil set to the editor with the specified editor id.

rules(editorId:String):Rules

Returns the rules object of an editor specified by 'editorId'. Each editor has exactly one rules object.

stencil(id:String):Stencil

Returns a stencil specified by its id or `undefined`, if no matching object is found. The id of a stencil is the stencil set's namespace followed by a pound and the id of the stencil from the stencil set description.

stencilSet(namespace:String):StencilSet

Returns the stencil set with the specified namespace, if the stencil set is loaded. Otherwise, it returns `undefined`.

stencilSets(editorId:String):Array

Returns a hash map of all stencil sets that are loaded by the editor with the specified `editorId`. The keys are the stencil sets' namespaces.

A.2 ORYX.Core.StencilSet.Stencilset

description():String**edges():Array**

This method returns an array of objects of type `Stencil` of all stencils defined in the stencil set that have the type "edge".

equals(stencilSet:StencilSet):Boolean

This method compares two stencil set objects and returns true, if they are identical. Otherwise, it returns false.

jsonRules():Object

Returns the part for the rules of the JSON object of the stencil set description. For internal use only.

namespace():String**nodes():Array**

This method returns an array of objects of type `Stencil` of all stencils defined in the stencil set that have the type "node".

source():String

Returns the URL the stencil set is loaded from.

stencil(id:String):Stencil

Returns an instance of type `Stencil` specified by the stencil id. The id of a stencil is the stencil set's namespace followed by a pound and the id of the stencil from the stencil set description. If no stencil is found, it returns `undefined`.

stencils():Array

This method returns an array of objects of type `Stencil` of all stencils defined in the stencil set.

title():String

A.3 **ORYX.Core.StencilSet.Stencil**

description():String

deserialize(shape:ORYX.Core.Shape, data:Object):Object

This method is used for accessing the stencil set's 'deserialize' callback. For more information about the structure of the parameter 'data' and the return value, see section 5.2.

equals(stencil:Stencil):Boolean

This method compares two stencil objects and returns true, if they are identical. Otherwise, it returns false.

groups():Array

Returns an array of strings that define the groups the stencil belongs to.

icon():String

Returns the URL of the stencil's icon as a string.

id():String

Returns the unique id of the stencil object. The id of a stencil is the stencil set's namespace followed by a pound and the id of the stencil from the stencil set description.

layout(shape:ORYX.Core.Shape)

This method is used for accessing the stencil set's 'layout' callback.

namespace():String

Returns the namespace of the stencil set the stencil belongs to.

properties():Array

Returns an array of objects of type `Property` of all the stencil's properties.

property(id):Property

Returns an object of type `Property` specified by its id.

roles():Array

Returns an array of strings that specify the stencil's roles.

serialize(shape:ORYX.Core.Shape, data:Object):Object

This method is used for accessing the stencil set's 'serialize' callback. For more information about the structure of the parameter 'data' and the return value, see section 5.2.

stencilSet():StencilSet

Returns the stencil set object the stencil belongs to.

title():String**type():String****view():SVGDocument**

Returns the SVG document that specifies the stencil's graphical representation. The SVG files are loaded when initializing the stencil objects.

A.4 ORYX.Core.StencilSet.Property**dateFormat():String****description():String****equals(stencil:Stencil):Boolean**

This method compares two property objects and returns true, if they are identical. Otherwise, it returns false.

fill():Boolean**fillOpacity():Boolean****id():String****items():Array**

Returns an array of objects of type `PropertyItem` of the property's items that are defined, if the property is of type "Choice".

item(value):PropertyItem

Returns an object of type `PropertyItem` specified by its value.

length():Number**max():Number****min():Number****namespace():String**

Returns the namespace of the stencil set the property belongs to.

optional():Boolean

prefix():String

readonly():Boolean

refToView():String

stencil():Stencil

Returns the stencil object the property belongs to.

stroke():Boolean

strokeOpacity():Boolean

title():String

type():String

Returns one of the following values: "Boolean", "Choice", "Color", "DateTime", "Float", "Integer", "String", "Url"

value():String|Number|Boolean

wrapLines():Boolean

A.5 **ORYX.Core.StencilSet.PropertyItem**

equals(stencil:Stencil):Boolean

This method compares two property objects and returns true, if they are identical. Otherwise, it returns false.

namespace():String

Returns the namespace of the stencil set the property belongs to.

property():Property

Returns the property the item belongs to.

refToView():String

value():String|Number|Boolean

A.6 **ORYX.Core.StencilSet.Rules**

Each Oryx instance has exactly one object of this type that contains the rules of all loaded stencil sets.

canConnect(args:Object):Boolean

This is the core method for checking connection rules. The object 'args' references the source, edge and target objects. The following properties are specified in the parameter 'args':

- edgeShape:** The shape object of the edge. 'edgeShape' is not mandatory.
- edgeStencil:** The stencil object of the edge. This property is not mandatory.
- sourceShape:** This is the shape object the start of the edge is or shall be connected to. This property is not mandatory.
- sourceStencil:** The stencil object of the source of the connection. This property is not mandatory.
- targetShape:** The same for 'sourceShape' applies to 'targetShape'.
- targetStencil:** Again, the same for 'sourceStencil' applies to 'targetStencil'.

Note that at least the source or the target must be defined, but it is not mandatory that both are specified. Furthermore, at least 'edgeShape' or 'edgeStencil' must be defined.

canContain(args:Object):Boolean

This method checks containment rules for the objects specified in 'args':

- containingShape:** The shape object of the parent shape candidate. This property is not mandatory.
- containingStencil:** This property references the stencil object of the parent candidate. This property is not mandatory.
- containedShape:** The reference to the shape object, that should be added to the parent candidate. This property is not mandatory.
- containedStencil:** 'containedStencil' references the stencil object of child candidate. This property is not mandatory.

Note that at least 'containingShape' or 'containingStencil' and 'containedShape' or 'containedStencil' must be defined.

initializeRules(stencilSet:StencilSet)

Initializes the rules for a stencil set. This method is called after loading a stencil set and is for internal use only.

incomingEdgeStencils(args:Object):Array

Returns an array of objects of type `Stencil`. These stencils specify edges that can be connected to the target specified in the 'args' parameter. The following properties are specified in the parameter 'args':

- targetShape:** This is the shape object the end of the edge shall be connected to. This property is not mandatory.
- targetStencil:** The stencil object of the target of the connection. This property is not mandatory.

Note that at least 'targetShape' or 'targetStencil' must be defined.

outgoingEdgeStencils(args:Object):Array

Returns an array of objects of type `Stencil`. These stencils specify edges that can be connected to the source specified in the 'args' parameter. The following properties are specified in the parameter 'args':

sourceShape: This is the shape object the start of the edge shall be connected to. This property is not mandatory.

sourceStencil: The stencil object of the source of the connection. This property is not mandatory.

Note that at least 'sourceShape' or 'sourceStencil' must be defined.

sourceStencils(args:Object):Array

Returns an array of objects of type **Stencil**. These stencils specify all possible target types for the source and edge object specified in 'args'. The following properties are specified in the parameter 'args':

edgeShape: The shape object of the edge. This property is not mandatory.

edgeStencil: The stencil object of the edge. This property is not mandatory.

targetShape: This is the shape object the end of the edge is or shall be connected to. This property is not mandatory.

targetStencil: The stencil object of the target of the connection. This property is not mandatory.

Note that at least 'edgeShape' or 'edgeStencil' must be defined.

targetStencils(args:Object):Array

Returns an array of objects of type **Stencil**. These stencils specify all possible source types for the target and edge object specified in 'args'. The following properties are specified in the parameter 'args':

edgeShape: The shape object of the edge. This property is not mandatory.

edgeStencil: The stencil object of the edge. This property is not mandatory.

sourceShape: This is the shape object the start of the edge is or shall be connected to. This property is not mandatory.

sourceStencil: The stencil object of the source of the connection. This property is not mandatory.

Note that at least 'edgeShape' or 'edgeStencil' must be defined.

B. The Workflow Nets Stencil Set

This appendix offers the workflow nets stencil set as described throughout this document. Here is a list of all files that belong to the stencil set:

```
[workflownets]
|- workflownets.json
|- view
  |- activity.svg
  |- condition.svg
  |- controlflow.svg
  |- diagram.svg
  |- inputcondition.svg
  |- outputcondition.svg
|- icons
  |- activity.png
  |- condition.png
  |- controlflow.png
  |- diagram.png
  |- inputcondition.png
  |- outputcondition.png
```

The files can be found on the CD in the folder `Oryx/data/stencilsets/`. The file `Oryx/workflownet.xhtml` is an example page that loads Oryx and the workflow nets stencil set.

workflownets.json

```
1  {
2    "title": "Workflow Nets",
3    "namespace": "http://www.example.org/workflownets",
4    "description": "Simple stencil set for Workflow Nets.",
5    "stencils" : [
6      {
7        "type": "node",
8        "id": "diagram",
9        "title": "Diagram",
10       "description": "",
```

```

11     "view": "diagram.svg",
12     "icon": "diagram.png",
13     "roles": [
14     ],
15     "properties": [
16     {
17         "id": "title",
18         "type": "String",
19         "title": "Title",
20         "value": ""
21     }
22     ]
23 },
24 {
25     "type": "node",
26     "id": "activity",
27     "title": "Activity",
28     "groups": [],
29     "description": "An atomic activity.",
30     "view": "activity.svg",
31     "icon": "activity.png",
32     "serialize": function(shape, data) {
33         var numOfOutgoings = shape.getOutgoingShapes().length;
34         data.push({
35             name: "numOfOutgoings",
36             prefix: "oryx",
37             value: numOfOutgoings,
38             type: "literal"
39         });
40         return data;
41     },
42     "roles": ["activitySource", "activityTarget"],
43     "properties": [
44     {
45         "id": "caption",
46         "type": "String",
47         "title": "Caption",
48         "value": "",
49         "description": "",
50         "tooltip": "",
51         "readonly": false,
52         "optional": true,
53         "refToView": "caption",
54         "wrapLines": true
55     },
56     {
57         "id": "trigger",
58         "type": "Choice",
59         "title": "Trigger",
60         "value": "Automatic",
61         "optional": false,
62         "items": [
63         {
64             "title": "Automatic",
65             "value": "Automatic",
66             "refToView": "automatic"
67         },
68         {
69             "title": "User",

```

```
70     "value":"User",
71     "refToView":"user"
72   },
73   {
74     "title":"External",
75     "value":"External",
76     "refToView":"external"
77   },
78   {
79     "title":"Time",
80     "value":"Time",
81     "refToView":"time"
82   }
83 ]
84 },
85 {
86   "id":"split",
87   "type":"Choice",
88   "title":"Split Behavior",
89   "value":"Or",
90   "optional":"false",
91   "items": [
92     {
93       "title":"Or",
94       "value":"Or",
95       "refToView":"orsplit"
96     },
97     {
98       "title":"Explicit Or",
99       "value":"Explicit Or",
100      "refToView":"explicitorsplit"
101     },
102     {
103       "title":"And",
104       "value":"And",
105       "refToView":"andsplit"
106     },
107     {
108       "title":"And / Or",
109       "value":"And / Or",
110       "refToView":"andorsplit"
111     }
112   ]
113 },
114 {
115   "id":"join",
116   "type":"Choice",
117   "title":"Join Behavior",
118   "value":"Or",
119   "optional":"false",
120   "items": [
121     {
122       "id":"c1",
123       "title":"Or",
124       "value":"Or",
125       "refToView":"orjoin"
126     },
127     {
128       "id":"c2",
```

```

129     "title":"Explicit Or",
130     "value":"Explicit Or",
131     "refToView":"explicitorjoin"
132   },
133   {
134     "id":"c3",
135     "title":"And",
136     "value":"And",
137     "refToView":"andjoin"
138   }
139 ]
140 },
141 {
142   "id":"color",
143   "type":"Color",
144   "title":"Color",
145   "value":"#D3DEFF",
146   "refToView":"color",
147   "optional":false,
148   "fill":true
149 }
150 ]
151 },
152 {
153   "type": "node",
154   "id":"condition",
155   "title":"Condition",
156   "groups": [],
157   "description":"A Workflow net condition.",
158   "view":"condition.svg",
159   "icon":"condition.png",
160   "roles": ["conditionSource", "conditionTarget"],
161   "properties": [
162     {
163       "id":"color",
164       "type":"Color",
165       "title":"Color",
166       "value":"#D3DEFF",
167       "refToView":"color",
168       "optional":false,
169       "fill":true
170     }
171   ]
172 },
173 {
174   "type": "node",
175   "id":"inputcondition",
176   "title":"Input Condition",
177   "groups": [],
178   "description":"A Workflow net input condition.",
179   "view":"inputcondition.svg",
180   "icon":"inputcondition.png",
181   "roles": ["conditionSource"],
182   "properties": [
183     {
184       "id":"color",
185       "type":"Color",
186       "title":"Color",
187       "value":"#D3DEFF",

```

```
188     "refToView":"color",
189     "optional":false,
190     "fill":true
191   }
192 ]
193 },
194 {
195   "type": "node",
196   "id":"outputcondition",
197   "title":"Output Condition",
198   "groups":[],
199   "description":"A Workflow net output condition.",
200   "view":"outputcondition.svg",
201   "icon":"outputcondition.png",
202   "roles": ["conditionTarget"],
203   "properties": [
204     {
205       "id":"color",
206       "type":"Color",
207       "title":"Color",
208       "value":"#D3DEFF",
209       "refToView":"color",
210       "optional":false,
211       "fill":true
212     }
213   ]
214 },
215 {
216   "type": "edge",
217   "id":"controlflow",
218   "title":"Control Flow",
219   "description":"",
220   "groups":[],
221   "view":"controlflow.svg",
222   "icon":"controlflow.png",
223   "roles": [],
224   "properties": []
225 }
226 ],
227 "rules": {
228   "connectionRules": [
229     {
230       "role":"controlflow",
231       "connects": [
232         {
233           "from":"activitySource",
234           "to":"conditionTarget"
235         },
236         {
237           "from":"conditionSource",
238           "to":"activityTarget"
239         }
240       ]
241     }
242 ],
243 "cardinalityRules": [
244   {
245     "role":"inputcondition",
246     maximumOccurrence:1
```

```

247     },
248     {
249         "role": "outputcondition",
250         maximumOccurrence: 1
251     },
252 ],
253 "containmentRules": [
254     {
255         "role": "diagram",
256         "contains": [
257             "activity",
258             "condition",
259             "inputcondition",
260             "outputcondition"
261         ]
262     }
263 ]
264 }
265 }

```

activity.svg

```

1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <svg
3      xmlns="http://www.w3.org/2000/svg"
4      xmlns:oryx="http://www.b3mn.org/oryx"
5      version="1.1">
6      <oryx:magnets>
7          <oryx:magnet oryx:cx="41" oryx:cy="56" />
8          <oryx:magnet oryx:cx="2" oryx:cy="56"/>
9          <oryx:magnet oryx:cx="80" oryx:cy="56"/>
10     </oryx:magnets>
11     <g oryx:minimumSize="50 50" oryx:maximumSize="200 200">
12         <rect id="color" oryx:anchors="top bottom left right" x="1" y="16"
13             width="80" height="80" stroke="black" fill="#D3DEFF" stroke-width="2"/>
14         <path id="external" oryx:anchors="top right" fill="#D3DEFF" stroke="black"
15             d="M61 1 h20 v12 h-20 v-12 m20 0 l-10 6 l-10 -6" />
16         <path id="user" oryx:anchors="top right" fill="#D3DEFF" stroke="black"
17             d="M71 1 v6 h-2 l5 6 l5 -6 h-2 v-6 h-6" />
18         <g id="time">
19             <circle oryx:anchors="top right" fill="#D3DEFF" stroke="black"
20                 cx="75" cy="7" r="6" />
21             <path oryx:anchors="top right" fill="none" stroke="black"
22                 d="M75 1 v2 M81 7 h-2 M75 13 v-2 M69 7 h2 M75 7 l3 -2 M75 7 l-5 -3" />
23         </g>
24         <path id="explicitorsplit" oryx:resize="horizontal vertical"
25             stroke-linejoin="bevel" oryx:anchors="top bottom right" fill="none"
26             stroke="black" d="M66 16 v80 l15 -40 z" />
27         <path id="andsplit" oryx:resize="horizontal vertical"
28             stroke-linejoin="bevel" oryx:anchors="top bottom right" fill="none"
29             stroke="black" d="M81 16 l-15 40 l15 40 m-15 0 v-80" />
30         <path id="andorsplit" oryx:resize="horizontal vertical"
31             stroke-linejoin="bevel" oryx:anchors="top bottom right" fill="none"
32             stroke="black" d="M66 16 v80 m0 -26.666 h15 m-15 -26.666 h15" />
33         <path id="explicitorjoin" oryx:resize="horizontal vertical"
34             stroke-linejoin="bevel" oryx:anchors="top bottom left" fill="none"
35             stroke="black" d="M16 16 l-15 40 l15 40 z" />
36         <path id="andjoin" oryx:resize="horizontal vertical" stroke-linejoin="bevel"

```

```

37   oryx:anchors="top bottom left" fill="none" stroke="black"
38   d="M1 16 115 40 l-15 40 M16 16 v80" />
39   <text id="caption" x="41" y="56" oryx:align="middle center"></text>
40   </g>
41 </svg>

```

condition.svg

```

1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <svg
3    xmlns="http://www.w3.org/2000/svg"
4    xmlns:oryx="http://www.b3mn.org/oryx"
5    width="40"
6    height="40"
7    version="1.1">
8    <g>
9      <circle id="color" cx="16" cy="16" r="15" stroke="black" fill="#D3DEFF"
10     stroke-width="2"/>
11   </g>
12 </svg>

```

controlflow.svg

```

1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <svg
3    xmlns="http://www.w3.org/2000/svg"
4    xmlns:oryx="http://www.b3mn.org/oryx"
5    width="500"
6    height="400"
7    version="1.0">
8    <defs>
9      <marker id="arrowEnd" refX="10" refY="5" markerUnits="userSpaceOnUse"
10     markerWidth="10" markerHeight="10" orient="auto">
11        <path d="M 0 0 L 10 5 L 0 10 z" fill="black" stroke="black"/>
12      </marker>
13    </defs>
14    <g>
15      <path d="M50 50 L100 50" stroke="black" fill="none" stroke-width="2"
16     marker-end="url(#arrowEnd)"/>
17    </g>
18 </svg>

```

diagram.svg

```

1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <svg
3    xmlns="http://www.w3.org/2000/svg"
4    xmlns:oryx="http://www.b3mn.org/oryx"
5    version="1.1">
6    <g>
7      <rect oryx:anchors="top bottom left right" x="1" y="16" width="80"
8     height="80" stroke="black" fill="#D3DEFF" stroke-width="2"/>
9    </g>
10 </svg>

```

inputcondition.svg

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <svg
3   xmlns="http://www.w3.org/2000/svg"
4   xmlns:oryx="http://www.b3mn.org/oryx"
5   width="40"
6   height="40"
7   version="1.1">
8   <g>
9     <circle id="color" cx="16" cy="16" r="15" stroke="black" fill="#D3DEFF"
10      stroke-width="2"/>
11     <path d="M16 13 10 2 m0 1 10 5" stroke="black"/>
12   </g>
13 </svg>
```

outputcondition.svg

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <svg
3   xmlns="http://www.w3.org/2000/svg"
4   xmlns:oryx="http://www.b3mn.org/oryx"
5   width="40"
6   height="40"
7   version="1.1">
8   <g>
9     <circle id="color" cx="16" cy="16" r="15" stroke="black" fill="#D3DEFF"
10      stroke-width="2"/>
11     <ellipse cx="16" cy="16" rx="3" ry="4" fill="none" stroke="black" />
12   </g>
13 </svg>
```

Bibliography

- [1] Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, Object Management Group (OMG), February 2006. <http://www.bpmn.org/>.
- [2] Willi Tscheschner. Oryx - Dokumentation, June 2007.
- [3] Martin Czuchra. Oryx - Embedding Business Process Data into the Web, June 2007.
- [4] Daniel Polak. Oryx - BPMN Stencil Set Implementation, June 2007.
- [5] Wil v. d. van der Aalst and Kees v. van Hee. *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. The MIT Press, January 2002.
- [6] Frank Manola Eric Miller. RDF Primer. 2004. <http://www.w3.org/TR/2004/REC-rdf-primer-20040210/>.
- [7] Ian Davis. Rdf In Html, 2006. <http://research.talis.com/2005/erdf/wiki/Main/RdfInHtml>.
- [8] D. Crockford. The application/json Media Type for JavaScript Object Notation (JSON). July 2006. <http://www.ietf.org/rfc/rfc4627.txt?number=4627>.
- [9] Jon Ferraiolo FUJISAWA Jun Dean Jackson. Scalable Vector Graphics (SVG) 1.1 Specification. January 2003. <http://www.w3.org/TR/2003/REC-SVG11-20030114/>.
- [10] Mozilla Developer Center. SVG in Firefox. http://developer.mozilla.org/en/docs/SVG_in_Firefox.

