A Nossa Universidade

Colégio dos Jesuítas
Rua dos Ferreiros - 9000-082, Funchal

Tel: +351 291 209400
Fax: +351 291 209410
Email: gabinetedareitoria@uma.pt

DM

WAMM
Wiki Aided Meta Modelling
Vitor Manuel Freitas Nóbrega

UNIVERSIDADE da MADEIRA

WAMM
Wiki Aided Meta Modelling

DISSERTAÇÃO DE MESTRADO

Vitor Manuel Freitas Nóbrega
MESTRADO EM ENGENHARIA INFORMÁTICA

UNIVERSIDADE da MADEIRA
A Nossa Universidade
www.uma.pt

janeiro | 2014

DM

Nome do Projecto/Relatório/Dissertação de Mestrado e/ou Tese de Doutoramento | Nome do Autor

UMa

DIMENSÕES: 45 X 29,7 cm
PAPEL: COUCHÊ MATE 350 GRAMAS
IMPRESSÃO: 4 CORES (CMYK)
ACABAMENTO: LAMINAÇÃO MATE

NOTA*
Caso a lombada tenha um tamanho inferior a 2 cm de largura, o logótipo institucional da UMa terá de rodar 90º , para que não perca a sua legibilidade|identidade.

Caso a lombada tenha menos de 1,5 cm até 0,7 cm de largura o laoyut da mesma passa a ser aquele que consta no lado direito da folha.

WAMM
Wiki Aided Meta Modelling

DISSERTAÇÃO DE MESTRADO

Vitor Manuel Freitas Nóbrega
MESTRADO EM ENGENHARIA INFORMÁTICA

ORIENTADOR
David Sardinha Andrade de Aveiro

Abstract

All over the world, organizations are becoming more and more complex, and there's a need to capture its complexity, so this is when the DEMO methodology (Design and Engineering Methodology for Organizations), created and developed by Jan L. G. Dietz, reaches its potential, which is to capture the structure of business processes in a coherent and consistent form of diagrams with their respective grammatical rules.

The creation of WAMM (Wiki Aided Meta Modeling) platform was the main focus of this thesis, and had like principal precursor the idea to create a Meta-Editor that supports semantic data and uses MediaWiki.

This prototype Meta-Editor uses MediaWiki as a receptor of data, and uses the ideas created in the Universal Enterprise Adaptive Object Model and the concept of Semantic Web, to create a platform that suits our needs, through Semantic MediaWiki, which helps the computer interconnect information and people in a more comprehensive, giving meaning to the content of the pages.

The proposed Meta-Modeling platform allows the specification of the abstract syntax i.e., the grammar, and concrete syntax, e.g., symbols and connectors, of any language, as well as their model types and diagram types. We use the DEMO language as a proof-of-concept and example.

All such specifications are done in a coherent and formal way by the creation of semantic wiki pages and semantic properties connecting them.

Keywords: Organizational Engineering, DEMO Methodology, Semantic MediaWiki, SVG-Edit, SVG meta-editor, UEAOM.

Acronyms:

SMW - Semantic MediaWiki

SVG - Scalable Vector Graphics

DEMO - Design & Engineering Methodology for Organizations

UEAOM – Universal Enterprise Adaptive Object Model

Resumo

Em todo o mundo, as organizações estão se tornando cada vez mais complexas, e existe a necessidade de capturar a sua complexidade, e é quando a metodologia DEMO (Design e Metodologia de Engenharia de Organizações), criada e desenvolvida por Jan LG Dietz, atinge o seu potencial, que é o de capturar a estrutura de processos de negócios de uma forma coerente e consistente na forma de diagramas com as suas respetivas regras gramaticais.

A criação da plataforma WAMM (Wiki Meta Aided Modeling) será o foco principal desta tese, e teve como principal precursor a ideia de criar um meta-editor que suporte dados semânticos e utilize o MediaWiki.

Este protótipo Meta-Editor usa o MediaWiki como recetor de dados, e usa as ideias criadas no Universal Enterprise Adaptive Object Model e o conceito de Semantic Web, para a criação de uma plataforma que se adapte às nossas necessidades, através do Semantic MediaWiki, o qual ajuda o computador a interligar a informação e as pessoas de uma forma mais compreensiva, dando significado ao conteúdo das páginas.

A plataforma de Meta-modelagem proposta permite a especificação da sintaxe abstrata, isto é, a gramática e a sintaxe concreta, por exemplo, de símbolos e conectores, de qualquer linguagem, tal como os seus tipos de modelos e de diagramas. A linguagem DEMO é utilizada como prova do conceito e exemplo.

Todas as especificações são feitas de forma coerente e formal, através da criação de páginas Wiki e das propriedades semânticas associadas a elas.

Palavras-chave: Engenharia Organizacional, Metodologia DEMO, MediaWiki Semântico, SVG-Edit, SVG meta-editor, UEAOM.

Acrónimos

SMW - MediaWiki Semântico

SVG - Gráficos vetoriais escaláveis

DEMO - Design & Metodologia de Engenharia para Organizações

UEAOM - Modelo Adaptativo de Objetos Universal para Organizações

Contents Index:

Abstract I

Resumo III

Contents Index: V

Table Index: VI

Figures Index: VII

1 Introduction 1

1.1 Motivation 7

1.2 Objectives 8

1.3 Content 9

2 Related work & problem 10

2.1 SVG or Canvas Graphical editor type 10

2.2 Scalable Vector Graphics (SVG) 11

2.2.1 SVG graphical editor 12

2.3 Meta modeling 13

2.4 Meta-Editor 14

3 Context 15

3.1 Enterprise Ontology Theoretical Concepts 15

3.1.1 Enterprise Ontology 15

3.1.2 Factual knowledge 15

3.1.3 Stata and Facta 17

3.1.4 World Ontology 18

3.1.5 The Grammar of WOSL 19

3.2 Basic Ontological Notions 20

3.3 Theoretical Foundations on Models 24

3.4 The universal enterprise adaptive object model 25

3.4.1 Abstract Syntax 26

3.4.2 Concrete Syntax 30

3.5 EE tools supporting DEMO 33

3.6 Realizing OSA with a wiki 35

3.6.1 Fundamental patterns used 36

3.6.2 Creating models and diagrams 37

3.6.3 Page names and semantic properties 39

3.6.4 Semantic Forms 41

4 Implementation 43

4.1 MediaWiki 46

4.2 Configuring and installing MediaWiki 46

4.2.1 Basic Extensions: 46

4.2.2 Parser hooks: ...47
4.2.3 Additional extensions ...47
4.2.4 Parametization ...48
4.3 Semantic Web ...50
4.4 Defining wiki pages and semantic properties ...50
4.5 Templates and properties ...52
4.5.1 Templates examples ...56
4.6 Templates and categories: ...58
4.7 Template for elements ...60
4.8 Forms and templates ...63
4.8.1 Form: OAKRK ...64
4.8.2 Form: OAK-Organizational Artifact Kind ...66
4.8.3 How to use the Forms and edit pages ...67
4.9 Page names, templates and properties connections ...69
4.10 Creating MediaWiki extensions ...74
4.11 SVG-edit ...75
4.11.1 SVG-Edit – Connector point extension ...76
4.11.2 SVG-Edit MediaWiki toolbar ...77
4.11.3 SVG-Edit MediaWiki toolbar use procedure ...77
4.11.4 SVG-Edit attributes ...81
4.11.5 SVG-Edit procedures for creating shape kind ...84
4.11.6 SVG-Edit modifications ...85
4.12 Connector-Edit ...87
4.12.1 Connector-edit implementation ...91
4.12.2 Adapting extension Anywebsite for Connector-Edit ...95
5 General Arquitecture ...96
6 Conclusion ...97
7 Bibliography ...99
8 Appendix ...101

Table Index:

Table 1 - Comparison between SVG and Canvas ... 10
Table 2 - Semantic properties examples ... 52
Table 3 - Template properties specification examples ... 55
Table 4 - Templates and category for each ... 59
Table 5 - Elements and their attributes ... 60
Table 6 - SVG-Edit file names, description and changes done ... 85
Table 7 - Connector kind template properties specification ... 89
Table 8 - Connector-Edit, files and usage description ... 94

Figures Index:

Figure 1 - Simplified version of the UEAOM ... 1
Figure 2 - SVG-Edit interface ... 4
Figure 3 - Connector Edit interface ... 4
Figure 4 – Examples of pages created in the Meta-Editor with their specific interconnections ... 6
Figure 5 - Performance Comparison between Canvas and SVG [3] ... 11
Figure 6 - The meaning triangle ... 16
Figure 7 - The ontological parallelogram ... 17
Figure 8 - Statum type declarations ... 19
Figure 9 - Example of a reference law ... 19
Figure 10 - Example of a dependency law ... 20
Figure 11 - Example of a factum type ... 20
Figure 12 - The meaning triangle 2 ... 21
Figure 13 - The ontological parallelogram 2 ... 21
Figure 14 - The model triangle ... 21
Figure 15 - Model triangle applied to organizations ... 21
Figure 16 - Meaning triangle applied to a transaction OA ... 21
Figure 17 - Model triangle applied to the organization space ... 21
Figure 18 - Type Square ... 25
Figure 19 - Universal Enterprise Adaptive Object Model ... 28
Figure 20 - UEAOM - Abstract Syntax classes ... 29
Figure 21 - DEMO Ontological Meta-Model and UAOM classes used to represent it . 29
Figure 22 - UEAOM Concrete syntax ... 30
Figure 23 - DEMO concrete diagrams example and UEAOM classes used to represent them ... 32
Figure 24 - Overview of the SMW Pages and their Relation with DEMO Models and UEAOM ... 35
Figure 25 - Semantic box for CA02-driver_shape ... 39
Figure 26 - DIAGRAM TEMPLATE page view ... 41
Figure 27 - DIAGRAM TEMPLATE edit view ... 42
Figure 28 - Fluxogram of the creation protocol ... 43
Figure 29 - UEAOM – Page creation protocol to follow and line explanation ... 44

Figure 30- MediaWiki installed software 46
Figure 31 - Semantic extensions installed 47
Figure 32 - Parser hooks installed 47
Figure 33 - Other extensions installed 47
Figure 34 –WikiEditor interface 48
Figure 35 - File page with PNG version 49
Figure 36 - Original SVG File 49
Figure 37 – Menu for creating semantic Property with Type options 50
Figure 38 - Overview of page Property: End connector symbol position 51
Figure 39 - List of Templates created in MediaWiki 53
Figure 40 - Template: LANGUAGE page edit 54
Figure 41 - Template Structure for Organizational artifact kind relation kind 56
Figure 42 - Template Structure for Organizational artifact kind 57
Figure 43 - Template Structure for Organizational artifact kind element kind 57
Figure 44 - Template Structure for Diagram kind 57
Figure 45 - Template structure for Connector 57
Figure 46 - UEAOM with the template definition for each class 58
Figure 47 - Template: SHAPE 62
Figure 48 – Component Templates with their associated Categories 62
Figure 49 - List of Forms that exist in MediaWiki 63
Figure 50 - Form: OAKRK page creator 64
Figure 51 - Form: OAK page creator 67
Figure 52 – Special pages - page 68
Figure 53 - Form link page example 68
Figure 54 - Form Edit for OCD page 69
Figure 55 - Pages names, templates and properties connections 70
Figure 56 - Rent a car, ATD example with the selected part used for example 71
Figure 57 – Pages created for Rent a car example and interconnection between pages-1st part 72
Figure 58 - Pages created for Rent a car example and interconnection between pages-2nd part 73
Figure 59 - Form: Shape kind 78
Figure 60 - Shape kind page example 78
Figure 61 - Shape kind edit page example 79

Figure 62 - Shape kind page with SVG-edit application open 79
Figure 63 - Shape kind edit page – example code created 80
Figure 64 - Shape kind page end result example.. 80
Figure 65- Page creation information from Recent changes page................................. 80
Figure 66 - SVG-EDIT special attribute code example 81
Figure 67 - Rescale attribute selection 81
Figure 68 - Move attribute selection 82
Figure 69 - Text position attribute selection.. 82
Figure 70 - Z-Position attribute selection.. 83
Figure 71 - SVG Properties 84
Figure 72 - SVG Dimension definition 84
Figure 73 - Connector kind template.. 87
Figure 74 - Connector Edit Interface.. 91
Figure 75 - Meta-Editor arquitecture.. 96
Figure 76 - Update process for the data in SMWAdmin.. 102

1 Introduction

This project was centered on the creation of WAMM - Wiki Aided Meta Modeling, which essentially is the adaption of Semantic MediaWiki for the creation of a Meta-Editor.

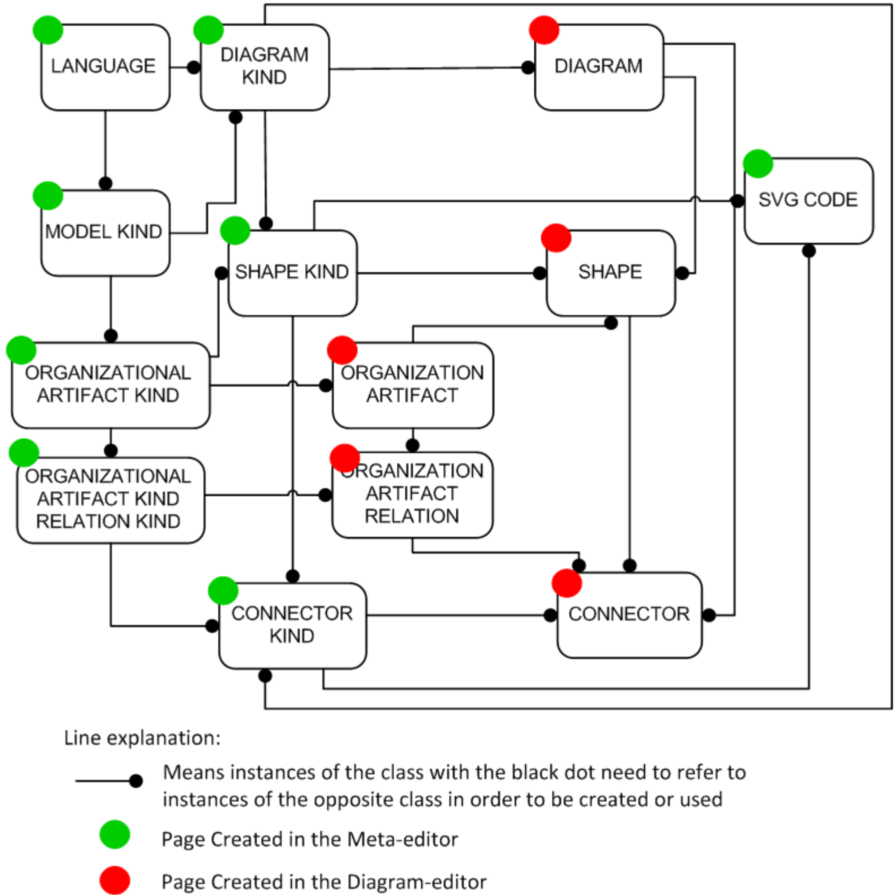


Figure 1 - Simplified version of the UEAOM

This project is based on the adaptation of the UEAOM, which was created during this project and we can see a simplified version in Figure 1 and further on we have the complete version in Figure 19, but just using this simplified version it's possible to explain the basics of the meta-model editor.

The classes created in UEAOM were created in order to define the specific aspects of the model and in order to use them, we need to grasp their meaning. For that we need a brief explanation of the concept behind each class used (examples taken from Figure 4):

- Language - specifies which type of language is permitted. Each Language can have multiple Model kinds. Example: «DEMO V3.5»
- Model kind - specifies which kind of model is permitted for a specific Language. Each model can have multiple Diagram kinds. Example: «Construction Model V3.5»

- Diagram kind - specifies the diagrams of a language, for a specific Model kind. For each Diagram Kind there is a set of Shape Kinds and Connector Kinds that are allowed to be used in it. The diagrams created are of the schematic kind, and are used to represent a model, using abstract, graphic symbols. Example: «ATD V3.5».
- Organizational artifact kind - specifies which kind of Organizational Artifact is permitted for a specific Model kind and Diagram kind. Example: «ELEMENTARY ACTOR ROLE V3.5». The OAKs are basic elements that are created for a determined Model specification like "Elementary Actor Role" or a "Transaction" in the "Construction Model". OAKs can be represented by instances of Shape Kind. OAKs may have multiple Organizational Artifact Kinds Element Kinds, such as "actor name" or "actor id".
- Organizational artifact kind relation kind - specifies which kinds of relations are permitted between two Organizational artifact kinds. Example: «TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE V3.5».
- Shape kind - specifies the properties values used to create a shape, for a determined diagram kind and version, in conjunction with its SVG code. Shape kinds are the definition of the OAKs. Example: «ELEMENTARY ACTOR ROLE V3.5 Shape kind».
- Connector kind – specifies the based properties values used to create a connector, for a determined Diagram kind and Version, in conjunction with its SVG code. Instances of Connector Kinds represent Instances of Organizational Artifact Kind Relation Kinds, each Connector kind is associated with an OAKRK, and serve as a connector between OAKs. Example: «EXECUTOR V3.5».
- SVG code- specifies the SVG code used for each of the shapes and connectors. This specification is implemented into a SVG file. Example: «EXECUTOR V3.5.svg»
- Version - specification of the versions associated with each of the previous classes. A Version is needed so each of the classes can be updated. While not displayed in Figure 1, it`s an intricate part of this project. Example: Two Diagram Kinds can have the same name but have different versions, so this concept of version is needed to differentiate them.

For the classes/pages created in the Diagram Editor: Shape, Connector, Diagram, Organization artifact, Organization artifact relation and a few others, are all the

implementation of its associated kind, which is basically an instantiation of their associated kind, for example, the instantiation of the Connector Kind «EXECUTOR V3.5», will receive additional information, like the positioning and it`s associations (diagram, connector kind and Organizational artifact relation), resulting into a Connector page, and an example of that is the page «Connector 2» which is displayed in Figure 58.

For most of the implementation of the UEAOM, in order to facilitate the Meta-Editing, we used templates and forms. Templates are MediaWiki pages created with specific definition using properties names created in the MediaWiki in order to use the resulting structure, inside other pages in order to facilitate for an organized page creation and population of properties for specific pages. These structures contain the name of a property and the allowed values for those properties, while the type of property (numeric, page, string) is defined in the property page itself, they can have different output formats, that is, a different visualization, which normally is in the type of a table or even plain text.

Forms are basically an interface that allows for the editing and creation of pages, and in this case permits for the creation and editing of pages with specific templates created for this project, permitting for the population of property values following a specific set of rules for each property.

Each of the classes in the Figure 1, represent a template created in MediaWiki, with all of them having a corresponding Form, that serves to create pages, except for SVG code class that is the association to the SVG file for the shape kind and connector kind. All of the classes with "kind" in the name and the Language class are created in the meta-editor, by using forms to fill the data for each property, except for the connector kind, which has a different interface (Figure 3), and in the Shape kind template there is a property that is filled after using the SVG-Edit application (Figure 2), which permits for the drawing of SVG based images using an web based interface that allows for the manipulation of the majority of the SVG elements and attributes used in SVG. This application is modified in order to facilitate for additional properties, and in order to use it inside MediaWiki, the SVG Edit extension of MediaWiki was used and also modified.

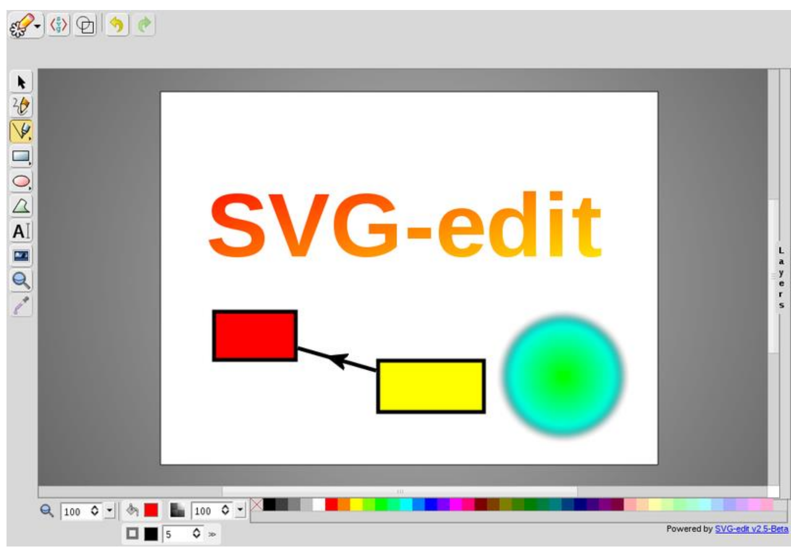


Figure 2 - SVG-Edit interface

This application creates a SVG file, which will be used in “SVG code” property. This part is explained in the chapter, 4.11.3 SVG-Edit MediaWiki toolbar use procedure.

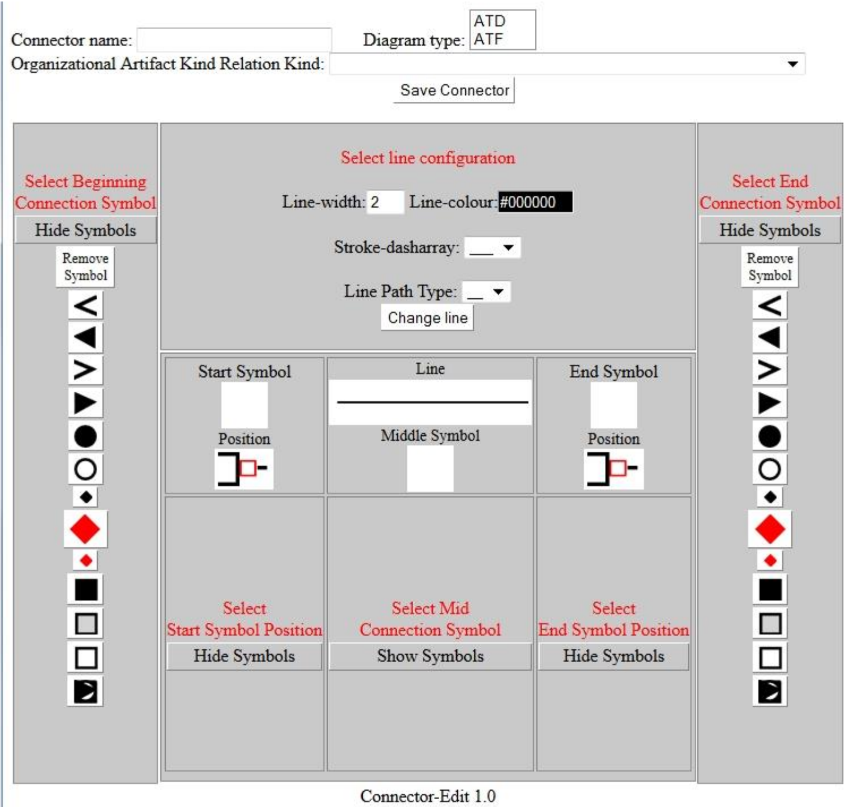


Figure 3 - Connector Edit interface

For the connector kind, a separate interface called “Connector-Edit”, Figure 3, was created in order for it to be easier to interact and to permit for the creation of the SVG file for that connector, which was used in the parallel project WOW-Wiki aided

Organization draWing. This interface permits for the creation of the «Connector kind» template and respective page, with all the values selected in the interface. It also creates a SVG file based on the selection of the SVG symbols and line data used, that will be used in the Diagram Editor, for the icon button and the creating of the connector.

For the rest of the classes, in Figure 1, with a red dot, they are mainly only used by the Diagram editor created in the parallel project WOW-Wiki aided Organization draWing.

On Figure 4 it's possible to view an example of some pages created in the Meta-Editor section of this project, which are displayed in templates, in the form of tables, with a small explanation of the connections between them, which deepens the ideas behind the initial Figure 1.

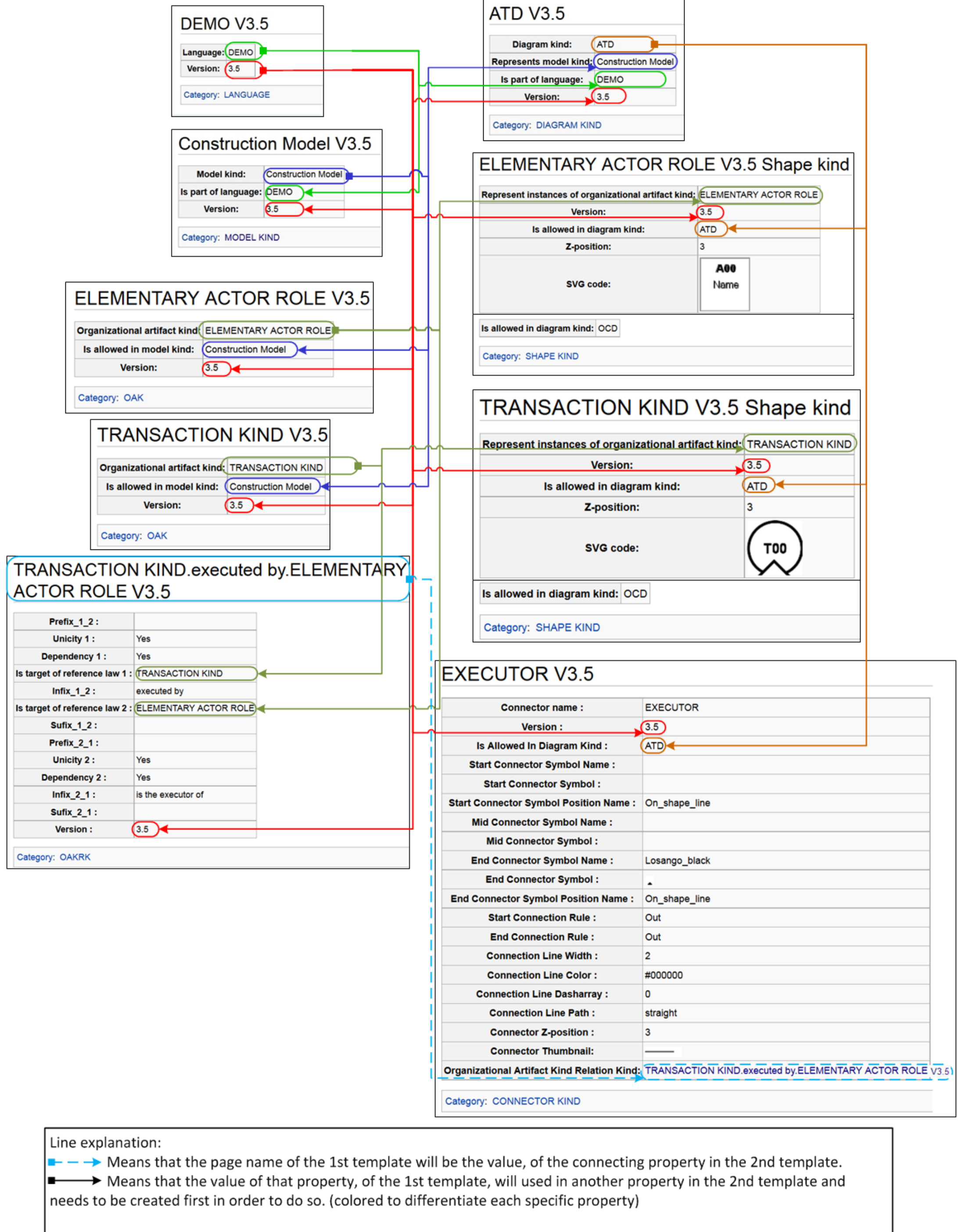


Figure 4 – Examples of pages created in the Meta-Editor with their specific interconnections

For the Meta-editor section, displayed in Figure 4, we basically used the value of the principal property of the template and the value of the version used in the page name, with some divergence between them, better explained in section 4.9. Also it`s possible to see that most of the values are used in similar properties with the main purpose of interconnecting the pages and facilitating for a better organization which hopefully improves the overall search used by the Diagram editor for a determined version.

In the Meta-editor section, displayed in Figure 4, it`s possible to view an example of what to expect in the Diagram Editor, which is the use of the page name in the properties values (the page name «EXECUTOR V3.5», receives, in the property «Organizational Artifact Kind Relation Kind», the value «TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE V3.5» from the OAKRK page name.), and also the pages names used have a different structure since they don`t have the version integrated in the page name, but instead a unique number associated with the type of page it is, for example, the page name «CONNECTOR 1» is a combination between the category name and the unique number. This specification is better explained in section 4.9.

In the next section, the motivation behind this project, the objectives and a synthesis of the content of each chapter of this dissertation are presented.

1.1 Motivation

A large percentage of IT project as high as 75%, fail to reach the expectations of their users. One of the main causes behind that failure is the insufficient or inadequate knowledge of the organizations reality in order for it to be automatized or supported by information system (IS).The Organizational Engineering[7] discipline emerged in the 90s and applies engineering concepts and methods to organizations in order to finally understand and represent the multiple facets of an organization, as well as facilitating the analysis and organizational changes, regardless of the implementation of SIs.

The semantic wikis are easy tools to use, by anyone, regardless of their computer knowledge. It's intended that way in order for everyone to contribute for the creation of a collective consciousness of organizational reality through the use of semantic wikis. This tool allows for a collective distribution of coherent organizational knowledge in the form of Semantic elements and relations in models aligned with organizational reality, that allow for the capture and monitoring of its evolution, as well as developing a more efficient and effective SIs for supporting such a reality .

By using the Organizational Engineering and Semantic MediaWiki, it should be able to interconnect the ideas behind Organizational Engineering and a well-established platform like Semantic MediaWiki to create a coherent Meta-modeling Editor platform which will serve as the foundation for a diagram editor.

1.2 Objectives

The main objective of this project is the analysis and development of a Meta Editor prototype, based on Semantic MediaWiki that allows for creation and editing of new types of diagrams, respective symbols and connectors with their respective grammar and syntactic laws. All of this focused on the DEMO methodology (Design and Engineering Methodology for Organizations) created and developed by Jan L. G. Dietz [7] with emphasis in SVG vector graphics.

The main challenges are defining wiki pages and semantic properties using a standard nomenclature to represent organizational fact types in order to store and connect all information needed to generate diagrams, the creation and editing of SVG based shapes using a SVG graphical editor, the use of Semantic Forms extension to manage forms, which helps to create and edit the semantic properties' values, and the creation of an interface for the creation of connectors.

There was another project, which was being created in parallel with this one, that is essentially the creation of a diagrams editor which uses this current platform (Wiki Aided Meta Modeling), created in this project, as a database for reading the data for the shape kinds, connector kinds and a few other specific information needed in order to create diagrams and will use this platform for saving all the data, using for that, the existing templates and forms created for that purpose, so it will basically be running in the background of that diagram editor platform.

The objectives generated from the main objective and challenges were:

- Use SVG based images, by that using SVG-edit and it's extension for MediaWiki.
- Adapt the SVG-edit extension with various modifications to its core in order to adapt it to our needs. This adapted extension permitted the creation of shapes with new SVG attributes created for better organization in the creation of diagrams for DEMO, which will be later used for background for a SVG diagram editor.
- Create a Connector-Edit interface that will facilitate the creation of connector kinds.
- Create the templates and forms that should be used for the creation of the models.

- Inter-connect the UEAOM with the templates and properties, by that creating a standard nomenclature.
- Create the templates and forms needed for the parallel project WOW-Wiki aided Organization draWing.

If the objectives are achieved, the prototype for a Meta Editor based on the Universal Enterprise Adaptive Object Model should be working.

1.3 Content

The First section will cover all the related work and the existing problems, the explanation for some initial choices made, like the type of vectorial graphics used and why the UEAOM was used.

Next there is the chapter Context, were all the basis notions behind this Meta editor, like the basic ontological notions, the UNIVERSAL ENTERPRISE ADAPTIVE OBJECT MODEL (UEAOM) and how it is intended to work, and all notions about enterprise ontology.

The last section and the most important, the implementation section, explains how the implementation of UEAOM into MediaWiki was accomplished, by that explaining how the nomenclature, the properties, templates and forms used, were created, and specially some of the alterations done to SVG-Edit and the creation of Connector-Edit. Also, in this section, there’s an explanation on how to use the editor for the creation of all the pages, and how to create pages with the appropriate template: Language, Model kind, Diagram kind, organization artifact kind, Shape kind, Connector kind between a few other that must be created in order for the Meta-editor to work, which is explained in detail, by giving examples, and information on how to create them and how they are created inside MediaWiki. This last section will be divided, essentially, in the Semantic MediaWiki part of the project, the SVG-section and the Connector-Edit section, with more focus in the MediaWiki part of the project.

2 Related work & problem

2.1 SVG or Canvas Graphical editor type

When deciding on which the graphical editor to choose from, there was two obvious choices Canvas or SVG, both of them widely used on the Internet.

For the editing of the shapes kinds in the Meta-editor and eventually the Diagram editor we had to choose between:

- SVG, which is the standard for vectorial based graphics.
- Canvas, which is an element of HTML5 and is a Pixel based graphics.

Both are the most commonly used on the Internet in terms of image manipulation, in all sorts of sites, which make them the ideal choices. While initially Canvas happear to be Vectorial, because it can draw lines and shapes, it isn`t because canvas elements are bitmaps while the SVG elements maintains vectorial elements, canvas doesn`t. For example if we draw a shape in a canvas element it will turn into a collection of pixels while if we draw something into a SVG element it will maintain it`s vectorial identity.

During the research, it was found out that there is a lot of criterion used to compare SVG to Canvas [3], and a small selection of those criterions are displayed in Table 1 and Figure 5.

Table 1 - Comparison between SVG and Canvas

Criterion	SVG	Canvas
Line codes	Lower number of lines codes in general	Higher number of lines codes in general
Events	Can process each event for each sub element separate	Can only process events for the entire screen
Basic forms	Rectangle, circle, ellipse, …	Only rectangle
Screen size	As the size of the screen get bigger, the rendering time increases slowly	As the size of the screen get bigger, the rendering time increases exponentially
Object number	As the amount of objects on screen increases, the rendering time increases exponentially	As the amount of objects on screen increases, the rendering time increases slowly
Best visualization format	Better scalability	Bad scalability
Static images	Better because of scalability	Degradation because of bad scalability
Pixels manipulation (visual effects)	Worse performance	Better performance
Mathematical and animation manipulation	Worse performance	Better performance

Basically canvas is better for complex scenes, real time mathematical animations, video manipulation and high performance (filters, ray tracers) while SVG is better used for static Images and high fidelity documents for viewing and printing which is essential to this project [3].

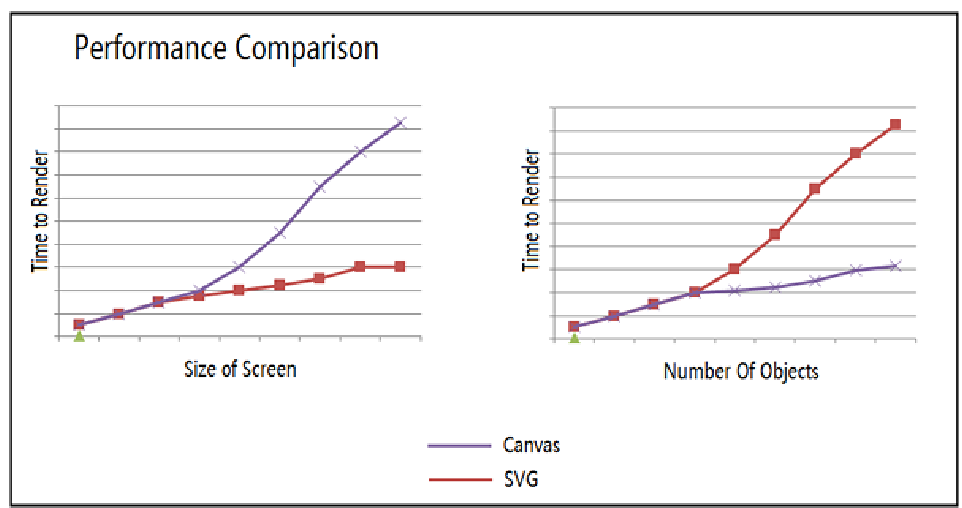


Figure 5 - Performance Comparison between Canvas and SVG [3]

2.2 Scalable Vector Graphics (SVG)

The SVG, In general, this type of File format support works in the majority of the top vector graphics editors as both import or export and compared to other vector image formats, it’s the most well-known and used since it was considered an open standard by World Wide Web Consortium (W3C) [4], these type of images are saved in XML text files with the extension .svg and has all the functionalities of the xml and has its own particular features associated to the drawing of two-dimensional graphics that can support interactivity and animation with some limitations and in combination with other technology it can be used to create a graphical editor.

One important fact about SVG is that there are some attributes that can be used inside other attribute like the “style” attribute that can take some attribute like “stroke”, “stroke-width” and few other, while this makes implementing SVG dependent on the way someone wants to code it, the browser can read both ways, which makes SVG somewhat dynamic in the structuring of the SVG element, so it’s something people want to pay attention to and specially this project since it’s very SVG dependent.

2.2.1 SVG graphical editor

After exploring the options and advantages between a SVG graphical editor and a Canvas graphical editor, the decision was made to go for a SVG graphical editor.
In order to create a graphical editor from scratch or reuse some graphical editor, some decisions were made; the initial idea of creating graphical editor from scratch would probably be too time consuming and takes us from the main objective that was to create a prototype Meta Modeling Editor. So it was necessary to find a compatible SVG graphical editor and in that search some possibilities where found and by some luck we found an editor that was already compatible with MediaWiki, but before reaching that solution there were a few other options viewed.
In the search for web graphical editors, there was an editor that popped out and it was SVG-edit which is a web-based vector graphics editor that uses only JavaScript, HTML5, CSS and SVG. This editor is the basis for many projects and by chance there was already an extension for MediaWiki that used the same editor which is called Extension: SVG-Edit and provided in-browser creation and editing of uploaded SVG files in MediaWiki, by using the open-source SVG-edit widget. One of projects mentioned before was Method Draw which basically uses the SVG-Edit and improves its overall looks and usability while removing some of the options that were required for this project.
Some projects that are included in SVG-Edit are JQuery which is a very popular library, jpicker for fill/stroke picker, jGraduate for picking gradient for the SVG, jQuery UI to make all the dialog boxes between a few others, all of whom are free and usable by anyone because of its MIT License (MIT), which means everyone can practically alter the code as they wish and even sell the result of those alterations if they want to, but a person needs to mentioned that SVG-Edit was used in that creation.
While it was a strong solution to our problem there was a need to search for additional possibilities, in that search, it was noticed that there were alot of vector graphics editors, but most of them, were not web-based.
One of the most used in the web is the Google drawings but it’s not permitted to use any of its code, so while it’s a very good tool it’s not the best example, to base anything besides the ideas.

Another SVG editor which can be used online is RichDraw – Simple VML/SVG Editor, which has some functionalities but it has many issues that a person doesn't find in the SVG-Edit.

There were a few projects used to create SVG diagrams, one of those was Accidentsketch and like the name says, it's used to draw accidents and helpful to create SVG for driving test and other car/road related situations, while it wasn't very helpful, since it's a diagram editor and not a SVG editor, it showed some interesting qualities that helped decide for SVG instead of canvas but not much more than that.

Since there weren't many online editors, another possibility was found, there are a few libraries/frameworks related to SVG and one of them was a SVG-to-Canvas library called Canvg, which is a SVG parser and renderer that helps bypass browsers' inability to save SVG files and PNGs, by first rendering SVG images in an HTML5 Canvas element, which was very useful for the Connection-edit part of this project. Another framework that helps create SVG based applications for the web, was SPARK (SVG Programmers' Application Resource) which gives an API and conventions for a GUI component framework built in SVG.

In the end, the SVG-Edit extension for MediaWiki was still the best solution since it had all the functionalities we needed already, but especially because it was already integrated with MediaWiki and we could change the code as we wish. This SVG editor was used for the creation of shape kinds and was initially thought to create connector kinds but that this idea was left aside since it was easier to create a different type of interface for that.

2.3 Meta modeling

Meta modeling, in this project, is the analysis, construction and development of a Meta-editor based on the Universal Enterprise Adaptive Object Model and theories taken from the same, but before reaching that model, there were some other types of models to have in consideration, like UML and MOF.

The UML is the standard for creating models but it's basically software engineering oriented and doesn't comply with Meta Models. And because of that, the **Meta-Object Facility** (**MOF**) was invented, which is an Object Management Group (OMG) standard for model-driven engineering [5], to respond to the need of Meta Modeling architecture to define the UML. This Meta Modeling architecture is better explained in the context section of this project in 3.3 Theoretical Foundations on Models.

2.4 Meta-Editor

In the initial proposal for this project there was to two platforms, **Open Modeling** and **Modelworld**, to serve as inspiration for the creation of the Meta Editor.

The **Modelworld** platform does not permit to create new types of models but works very well for the models that they support and it’s not considered a Meta-editor but a diagram editor which isn’t what we are looking for but can be helpful for the diagram editor extension of the Meta-Editor, and it’s the official page is www.modelworld.nl.

The **Open modeling** platform, which official page is located at open-modeling.sourceforge.net, is a web-based application for the creation and editing of architecture models and their structures. It permits for the edit of Meta Models which is something we have interest in, and it supports the DEMO methodology so with that in mind and the work done in [8], the general ideas and structure will help for the definition of the structure of our own framework, since it doesn’t comply with the semantic way of thinking that this project has in mind and specially with the usage of the MediaWiki platform. Another important aspect is it’s complex interface which many will find difficult to use, which is something that we hope that with MediaWiki we will try to simplify by using forms/templates and properties.

There isn’t any editor that complies with all our needs. To satisfy our dynamic meta-modeling needs, we needed to propose, a new model, that we called Universal Enterprise Adaptive Object Model (UEAOM). Which was created during this project, and it serves as a basis for the creation of the Meta Editor in the Semantic MediaWiki framework.

This conceptual model, UEAOM, allows us to specify the abstract and concrete syntaxes of any language.

All the process used to create this Meta Editor will be explained in some detail in the implementation chapter.

3 Context

The basis for this project is the **Universal Enterprise Adaptive Object Model** [1], which was being produced during the realization of this project, with several ideas for the final UEAOM based on the discussions that arose during the project meetings. The ideas behind UEAOM are explained in the next sections, fully based on [1], [22] and partially in [10] and [2].

3.1 Enterprise Ontology Theoretical Concepts

3.1.1 Enterprise Ontology

A widely adopted definition of ontology is that ontology is a formal, explicit specification of a shared conceptualization. It regards the conceptualization of (a part of) the world, so it is something in our mind. This conceptualization is supposed to be shared, which is the practical goal of ontologies. This takes also place in communication. Third, it is explicit. Ontology must be explicit and clear, there should be no room for misunderstandings. Fourth, it is specified in a formal way. Natural language is inappropriate for this task, because of its inherent ambiguity and impreciseness. The notion of ontology as applied in this book is the notion of system ontology. Our goal is to understand the essence of the construction and operation of complete systems, more specifically, of enterprises. The goal of enterprise ontologies is to make available the right amount of the right kind of knowledge of the operation of the company in a manner that one is able to look through the distracting and confusing appearance of the enterprise right into its deep kernel. [10]

3.1.2 Factual knowledge

Factual knowledge is the knowledge about the states and state changes of a world. By factual knowledge we mean knowledge about the state and the state changes of a world, like knowing that a person or a car or an insurance policy exists, as well as knowing that the insurance policy of a car started at some date. The basis for understanding factual knowledge is the meaning triangle, as exhibited in Figure 6 .It explains how people use signs as representations of objects in order to be able to communicate about these objects in their absence.

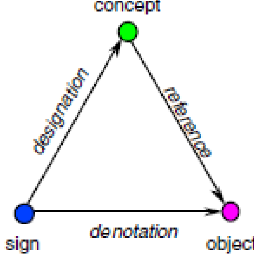


Figure 6 - The meaning triangle

The elementary notions that we will make use of are designated by the words “sign”, “object” and “concept”. The notion of concept is considered to be a subjective notion whereas sign and object are considered to be objective notions. Objective means that it concerns things outside the human mind and subjective means that it concerns things that can only exist inside the human mind. The three notions are elaborated below.

A **sign** is an object that is used as a representation of something else. A well-known class of signs is the symbolic signs, as used in all natural languages. Examples of symbolic signs are: the person name "Ludwig Wittgenstein" or the car license number "16-EX-AF".

An **object** is an observable and identifiable individual thing, for example a person or a car. Only concrete objects are observable by human beings. However, there are many interesting objects that are not observable. The number 3 and the composite object denoted by “Ludwig Wittgenstein owns car 16-EX-AF” are a few of those examples. These objects are called abstract objects.

A **concept** is a subjective individual thing. It is a thought or mental picture of an object that a subject may have in his or her mind. An example of a **concrete concept** is the mental picture a person has of the person Ludwig Wittgenstein while the example for an **abstract concept** is the fact that Ludwig Wittgenstein owns car 1 6-EX-AF.

The basic notions of **sign**, **object** and **concept** are related to each other by three basic notional relationships:

- **Designation -** This is a relationship between a sign and a concept. We say that a sign designates a concept. Example: The name "Ludwig Wittgenstein" designates a particular concept of the type person.
- **Denotation** – This is a relationship between a sign and an object. We say that a sign denotes an object. Examples: the name "Ludwig Wittgenstein" denotes the object person Ludwig Wittgenstein.

- **Reference** is a relationship between a concept and an object, a concept refers to an object. An example of that is the concept Ludwig Wittgenstein which refers to a particular person.

The relationships between individual concepts and generic concepts (types), and consequently between individual objects and classes are depicted in Figure 7 .In this figure, which is based on Figure 6, the signs (predicate names and proper names) are deliberately left out because they are not relevant in ontology. Ontology is about the essence of things, not about how people name them. The resulting figure is called the ontological parallelogram. It explains how (individual) concepts are created in the human mind. The notional relationships instantiation, conformity and population are explained hereafter.

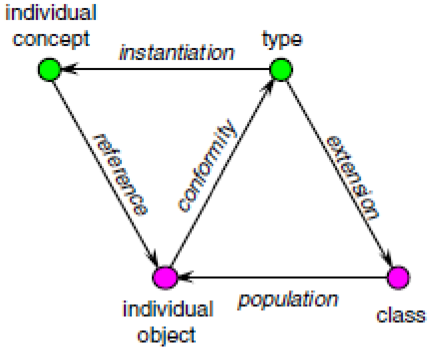


Figure 7 - The ontological parallelogram

Instantiation is a relationship between a concept and a type: every concept is an instantiation of a type. Examples: the person Ludwig Wittgenstein is an instantiation of the type person.

Conformity is a relationship between (the 'form' of) an object and a type. We say that an object conforms to a type. Examples: the object, denoted by the sign "Ludwig Wittgenstein", conforms to the type person; the object, denoted by the sign "16-EXAF", conforms to the type car.

Population is a relationship between an object and a class. We say that a class is a population of objects. A more common way of expressing this is saying that the object is a member of or belongs to the class. Example: the object, denoted by the sign "Ludwig Wittgenstein", belongs to the class person.

3.1.3 Stata and Facta

At any moment a world is in a particular state, which is simply defined as a set of objects. These objects are said to be current during the time that the state prevails. A

state change is called a transition. The occurrence of a transition is called an event. Consequently, a transition can take place several times during the lifetime of a world, events however are unique. An event is caused by an act. In order to understand profoundly what a state of a world is, and what a state transition is, it is necessary to distinguish between two kinds of objects, which we will call stata (singular: statum) and facta (singular: factum).

A **statum** is something that is just the case and that will always be the case; it is constant. Otherwise said, it is an inherent property of a thing or an inherent relationship between things. Example: The author of book title T is A. The existence of these objects is timeless. For example, a particular book title has a particular author. If it is the case at some point in time, it will forever be the case. A derived statum is defined by its derivation rule. The being specified of this rule is the only necessary and sufficient condition for the existence of the derived statum. This marks an important difference between a world and a database system about that world. E.g. the age of a person in some world exists at any moment; however, it has to be computed when it is needed. Stata are subject to existence laws. These laws require or prohibit the coexistence of stata. For example, if the author of some book is "Ludwig Wittgenstein", it cannot also be "John Irving".

Contrary to a statum, a **factum** is the result or the effect of an act. Example: book title T has been published. The becoming existent of a factum is a transition. Before the occurrence of the transition, it did not exist and after the occurrence it does exist. Facta are subject to occurrence laws. These laws allow or prohibit sequences of transitions. For example, sometime after the creation of the factum "loan L has been started", the transition "loan L has been ended" might occur, and in between several other facta may have been created, like "the fine for loan L has been paid".

3.1.4 World Ontology

"We are now able to provide a precise definition of the ontology of a world; world ontology consists of the specification of the state space and the transition space of that world. By the state space is understood the set of allowed or lawful states. It is specified by means of the state base and the existence laws. The state base is the set of statum types of which instances can exist in a state of the world. The existence laws determine the inclusion or exclusion of the coexistence of stata. By the transition space is

understood the set of allowed or lawful sequences of transitions. It is specified by the transition base and the occurrence laws. The transition base is the set of factum types of which instances may occur in the world. Every such instance has a time stamp, which is the event time. The occurrence laws determine the order in time in which facta are allowed to occur." [10]

3.1.5 The Grammar of WOSL

"WOSL is a language for the specification of the ontology of a world. In order to keep the specification of the grammar of WOSL orderly and concise, we present it in a number of figures, exhibited hereafter. Figure 8 exhibits the ways in which Statum types can be declared. By the declaration of a Statum type is understood stating that the Statum type belongs to the state base of the world under consideration. Statum types can be declared intensionally or extensionally. By intensional we mean the notation of the statum type as a unary, binary, ternary etc. concept type. Intensional notations are referred to be a bold small letter (or a string of small letters). Extensional notations are referred to by a capital letter (or a string of capital letters). To understand what a state of a world is, it is necessary to distinguish between two kinds of objects: stata and facta. WOSL language has several graphical pictures to represent these stata and facta."[2][10]

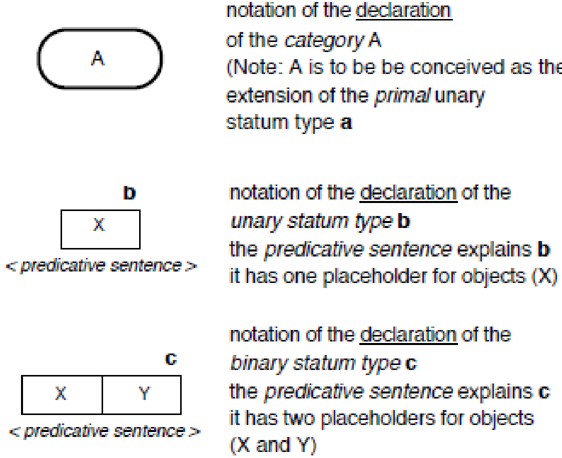


Figure 8 - Statum type declarations

Figure 9 and Figure 10 show the specification of existence laws.

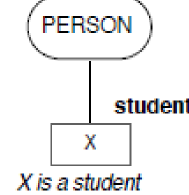


Figure 9 - Example of a reference law

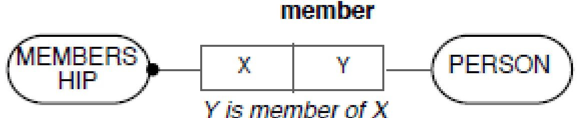


Figure 10 - Example of a dependency law

Figure 11 shows an example of a factum type.

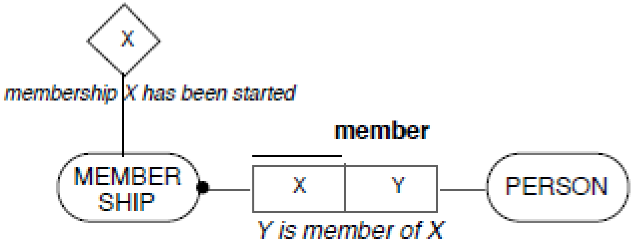


Figure 11 - Example of a factum type

3.2 Basic Ontological Notions

"The ontological system definition was adopted from [14] which concerns the construction and operation of a system. The corresponding type of model is the white-box model, which is a direct conceptualization of the ontological system definition presented next. Something is a system if and only if it has the next properties: (1) composition: a set of elements of some category (physical, biological, social, chemical etc.); (2) environment: a set of elements of the same category, where the composition and the environment are disjoint; (3) structure: a set of influencing bonds among the elements in the composition and between these and the elements in the environment; (4) production: the elements in the composition produce services that are delivered to the elements in the environment. From [14] we find that in the Ψ-theory based DEMO methodology, four aspect models of the complete ontological model of an organization are distinguished. The Construction Model (CM) specifies the construction of the organization: the actor roles in the composition and the environment, as well as the transaction kinds in which they are involved. The Process Model (PM) specifies the state space and the transition space of the coordination world. The State Model (SM) specifies the state space and the transition space of the production world. The Action Model (AM) consists of the action rules that serve as guidelines for the actor roles in the composition of the organization."[1]

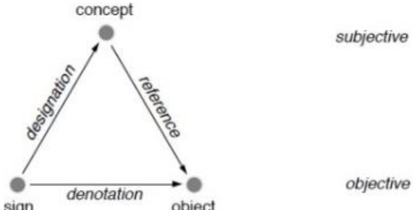


Figure 12 - The meaning triangle 2

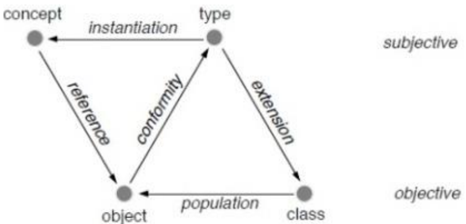


Figure 13 - The ontological parallelogram 2

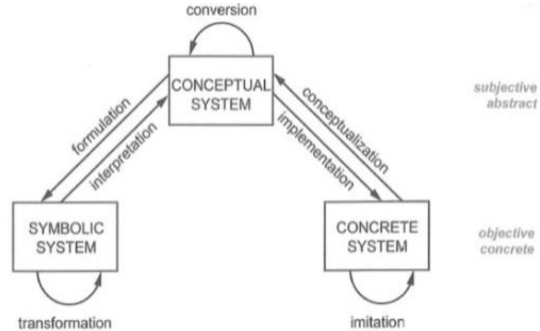


Figure 14 - The model triangle

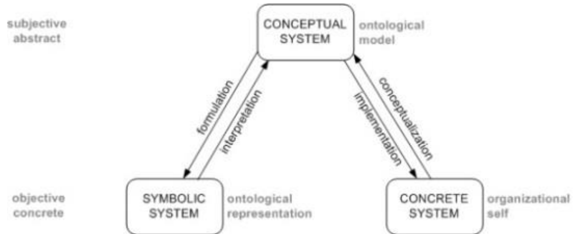


Figure 15 - Model triangle applied to organizations

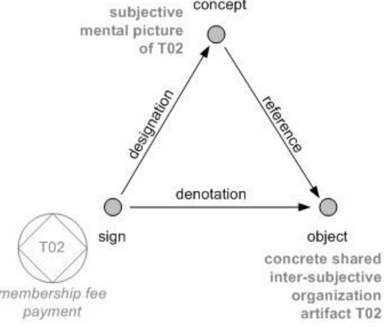


Figure 16 - Meaning triangle applied to a transaction OA

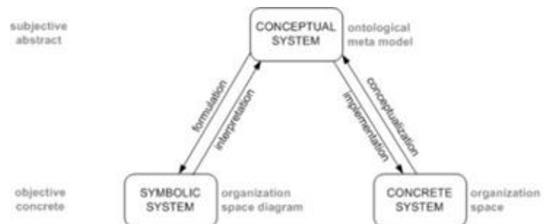


Figure 17 - Model triangle applied to the organization space

"In Figure 12 and Figure 13, we find, respectively, the meaning triangle and the ontological parallelogram, taken from [13] which explain how (individual) concepts are created in the human mind. We will also base our claims in the model triangle, taken from [14] and presented in Figure 14. We find that the model triangle coherently overlaps the meaning triangle. This happens because a set of symbols – like a set of DEMO representations (signs) that constitute a symbolic system – allows the interpretation of a set of concepts – like a set of DEMO aspect models, part of the ontological model, constituting a conceptual system. This conceptual system, in turn, consists in the conceptualization of the "real" inter-subjective organizational self, i.e., the set of OAs constituting the concrete organization system's composition structure and production. Figure 4 is an adaptation from the model triangle of Figure 14 and depicts

our reasoning. We call the set of all DEMO diagrams, tables and lists used to formulate the ontological model as ontological representation.

Now relating with the meaning triangle, we can verify that a particular sign (e.g., a transaction symbol with label membership fee payment), part of an ontological representation (e.g., actor transaction diagram, representing a library's construction model) designates (i.e., allows the interpretation or is the formulation) of the respective concept of the particular transaction part of the respective ontological model (e.g., construction model). This subjective concept, in turn, refers to a concrete object of the shared inter-subjective reality of the organization's human agents (e.g., the particular OA transaction T02). Figure 16, an adaptation from the meaning triangle depicts this other reasoning.

Another example of an OA related with T02 would be the transaction initiation OA, relating T02 with actor role registrar (also designated by A02) and formulated by a line connecting the transaction and actor role symbols of T02 and A02. Actor role registrar is, in turn, another OA of the construction space of the library. Once such role is communicated to all employees of a library, it becomes a “living” abstract object part of the shared inter-subjective reality of the library's human agents. Such objects, along with other OAs of the organizational inter-subjective reality, give human agents a way to conceptualize their organizational responsibilities – in this case, requesting membership fee payments to aspirant members. We name this set of all abstract objects living in the inter-subjective reality of an organization's members as the organizational self.

From these notions we proposed a set of claims presented in more detail in [10] and summarized next. An organization – besides producing a set of products or services for its environment – also produces itself. That is, enclosed in its day-to-day operation, there will be parts of its operation which change the organization system itself, i.e., change the set of OAs that constitute its composition, structure and production. By formally and explicitly specifying these change acts one keeps a definite and updated record of produced OAs. Such a record – the OAs base – constitutes the means for one to always be able to conceptualize the most current and updated ontological model of the organizational self. Thus the continuous production of the organizational self should include the synchronized production of the collective and subjective “picture” (awareness) of the organizational self – the conceptualization that constitutes its ontological model – thanks to the synchronized production of the respective symbolic

system – an ontological representation that allows the interpretation of the ontological model and the conceptualization (awareness) of the organizational self. To separate concerns, we propose that change acts are performed by a (sub-)organization considered to exist in every organization (O) that we call: G.O.D. Organization (GO) – change acts lead to the Generation, Operationalization and Discontinuation of OAs. The GO's production world will contain the current state of O's self as well as its relevant state change history. The GO has the role of continuously realizing and capturing changes of organizational reality. Thus, by implementing the GO pattern in a real organization, in an appropriate manner, providing automatic generation of ontological representations derived from the OAs base, one can achieve OSA. This is possible because one can implement clear rules that, based on the arrangement of OAs of the organizational self, automatically produce the appropriate ontological representation which, in turn, allows the appropriate interpretation of the ontological model, that is, the correct conceptualization of the organizational self.

OAs constituting the organizational self are arranged in a certain manner as to specify all the spaces (state, process, action and structure) of an organization's world, i.e., they have to obey certain rules of arrangement between them. We call the specification of these rules as the ontological Meta model. The ontological meta-model is the conceptualization of the OA space. By OA space we understand the set of allowed OAs. It is specified by the OA base and OA laws. The OA base is the set of OA kinds of which instances, called OAs, may occur in the state base of the GO's world. The OA laws determine the inclusion or exclusion of the coexistence of OAs. The definition of the OA space is quite similar to the definition of state space of an organization's production world – specified in World Ontology Specification Language (WOSL) [13] – and, thus, it is appropriate to use WOSL to express the ontological meta-model in, what we propose to call: the Organization Space Diagram (OSD). DEMO's OSD is currently called as the DEMO Meta Model (DMM), the chosen name for the specification provided in [14] and consisting, in practice, in the OSDs corresponding to the four DEMO aspect models: SM, CM, PM and AM. These diagrams formulate, for each aspect model, the OA kinds out of which instances – OAs – can occur in the organizational self and coexistence rules governing how to arrange these instances. Another reason we propose to use the expression Organization Space Diagram is because we're in fact looking at a Space Diagram which, following the model triangle [13], is a symbolic system which is a formulation of the conceptual system of the

ontological meta model. So, for coherency reasons, one should not use terms “Meta” and “Model” to name those figures but use, instead, the term Organization Space Diagram. The OSD allows the interpretation, in one's mind, of the ontological meta model. The complete set of organization artifact kinds and laws governing the arrangement of their instances constitutes the organization space. The conceptualization of the organization space consists in the ontological meta-model which, in turn, is formulated in what we call the Organization Space Diagram. A depiction of this reasoning is present in Figure 17, another adaptation from the model triangle. The G.O.D. organization is addressed in detail in [1]. This is an evolution of the conceptual model proposed in other works, taking in account state-of-the-art related model theory and concepts described next.” [1]

3.3 Theoretical Foundations on Models

“In a graphical modeling language, the vocabulary is expressed in terms of pictorial signs. Those graphical primitives form the concrete syntax i.e the lexical layer of such language. The abstract syntax, on the other hand, is usually defined in terms of an abstract visual graph or a meta-model specification. A meta-model specification of a language defines the set of grammatically correct models that can be constructed using that language, a vocabulary. The concrete syntax provides a concrete representational system for expressing the elements of that meta-model [19]. In a communication process, besides agreeing on a common vocabulary, the participants need to also share the meaning for the syntactical constructs being communicated so they are able to interpret in a compatible manner the expressions being used. To this end a language's semantics can be constructed in two parts: a semantic domain i.e. the real world entities to which those semantics apply and a semantic mapping from the syntactic vocabulary to such domain that tells us the meaning of each of the language's expressions as an element in that specific domain. In graphical languages, vocabulary, syntax and semantics cannot be clearly separable. A graphical vocabulary of a modeling language may include shapes of differing sizes and colors that often fall into a hierarchical typing that constrains the syntax and informs about the semantics of the system [19]. The abstract syntax of a model manages the formal structure of the model elements and the relationships amongst them [21].

The MetaObject Facility (MOF) Specification is the industry-standard environment where models can be exported from one application, imported into another, transported

across a network, stored in a repository and then retrieved, rendered into different formats like XMI or XML, transformed, and used to generate application code [5]. The Adaptive Object Model (AOM) is a pattern that represents classes, attributes, and relationships as meta-data. It is a model based on instances rather than classes. Users change the meta-data (object model) to reflect changes in the domain. These changes modify the system's behavior. In other words, it stores its Object-Model in a database and interprets it. Consequently, the object model is active, when you change it; the system changes immediately [15]." [1]

3.4 The universal enterprise adaptive object model

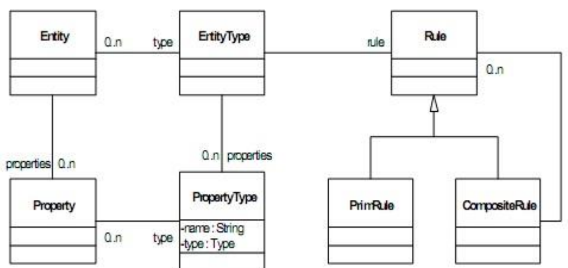


Figure 18 - Type Square

"The long term objective of our research is the development of a wiki-based system that allows an effective integrated enterprise modeling, while allowing dynamic evolution of meta-models, models and their representations, while providing intuitive navigation through their elements and also their semantics, allowing wide-spread model interpretation and distributed model creation and change, reflecting enterprise changes, thus addressing our problem. An essential step in this direction is what we call the Universal Enterprise Adaptive Object Model (UEAOM), depicted in Figure 19. We apply the AOM pattern referred in the previous section so that each page or semantic property of our semantic wiki-based system corresponds to instances of classes of our AOM.

Wiki pages, that are instances of class DIAGRAM, automatically generate SVG diagrams based on shape and connector pages. These pages also allow dynamic editing of diagrams and underlying models. We also apply the type-square pattern [15] – depicted in Figure 18– 4 times as to allow run-time dynamic change of: (1) meta-model elements, (2) model elements, (3) shape elements and (4) connector elements. Our UEAOM is represented with the World Ontology Specification Language (WOSL)[13]. WOSL is based in Object Role Modeling language [16] which is also used as a base for

the specification of the anatomy of Archimate [20], a similar effort to ours. In [18] a relation between Adaptive Object Model pattern and the MOF standard is presented, where run-time instances of the operational level are equivalent to MOF's M0 and knowledge level; classes, attributes, relations and behavior is equivalent to M1, being M2 an equivalent to the models used to define an AOM. As in the work of Ferreira et al., in our UEAOM all these MOF levels are projected as run-time instances. In our prototype system, we have as instances both organization artifacts – i.e., concrete organization models – and organization artifact kinds – i.e., the meta-model specification or, in other words, the abstract syntax. So both M1 and M2 levels of the MOF framework exist and change at run-time. But the MOF and Ferreira's initiative are too software development oriented and too complex for our needs. The main idea is to apply these fundamental theoretical foundations and adapt them to the field of enterprise ontology.

Having the UEAOM contextualized, an explanation of its content is now due. With the UEAOM's classes we are not explicitly specifying syntaxes of particular modeling languages. What we can do, while instantiating these classes, is to specify any syntax of any modeling language, along with particular models of each language, and also their evolution, all this in run-time. For a better understanding and following the essential and important validation by instantiation principle [17] we present, for all elements of our AOM, example instances for the DEMO language, namely a fragment of the EU-rent case's Construction Model and its respective Actor Transaction Diagram. Thus, we can find, in red color expressions, instances of both our classes and fact types of our UEAOM concerning the EU-rent case which allow a better interpretation of our proposal." [1]

3.4.1 Abstract Syntax

"Relevant classes for the specification of the abstract syntax of any version of any language are presented in Figure 20. The main concepts of the abstract syntax specification are expressed in the classes LANGUAGE, MODEL KIND, ORGANIZATIONAL ARTIFACT KIND (OAK) and ORGANIZATIONAL ARTIFACT RELATION KIND (OARK). They specify all allowed artifacts (e.g. transaction kind OAK and transaction execution relation OARK) for different types of models that can exist for different languages. Class ORGANIZATIONAL ARTIFACT RELATION KIND has ten properties that can be divided in two groups of five where

each group specifies one of the two sides of an allowed relation between two OAKs. The ones named prefix, infix and suffix specify the formulation that can be done around the names of the two OAKs being related. Most times, only the infix needs to be specified. With the unicity and dependency properties we specify the cardinality of the relation and which OAKs are mandatory or not to participate in the relation. Reference law fact types specify which two OAKs are allowed to participate in this relation. Practical example of the first set of the referred 5 properties: F Transaction Kind T is initiated by Elementary Actor Role corresponds to a set of Dependency 1, Reference law 1, Unicitiy 1, Infix_1_2 and Reference law 2. F Elementary Actor Role T is initiator of Transaction Kind would be its corresponding Dependency 2, Reference law 2, Unicitiy 2, Infix_2_1 and Reference law 1. Thanks to this part of our UEAOM specification we allow a precise and formal formulation of the abstract syntax of models, already giving considerable semantics thanks to the prefix, infix, suffix and OAK names that can be composed in formulations for each direction of the relation. Instances of class ORGANIZATION ARTIFACT PROPERTY specify intrinsic properties of OAKs, like identifiers and names. The respective property PROPERTY DOMAIN allows us to specify the domain for each intrinsic property of an OAK (e.g., string, number, etc.). Examples of instances are property transaction id with domain T<number> or transaction name with domain <string>."[1]

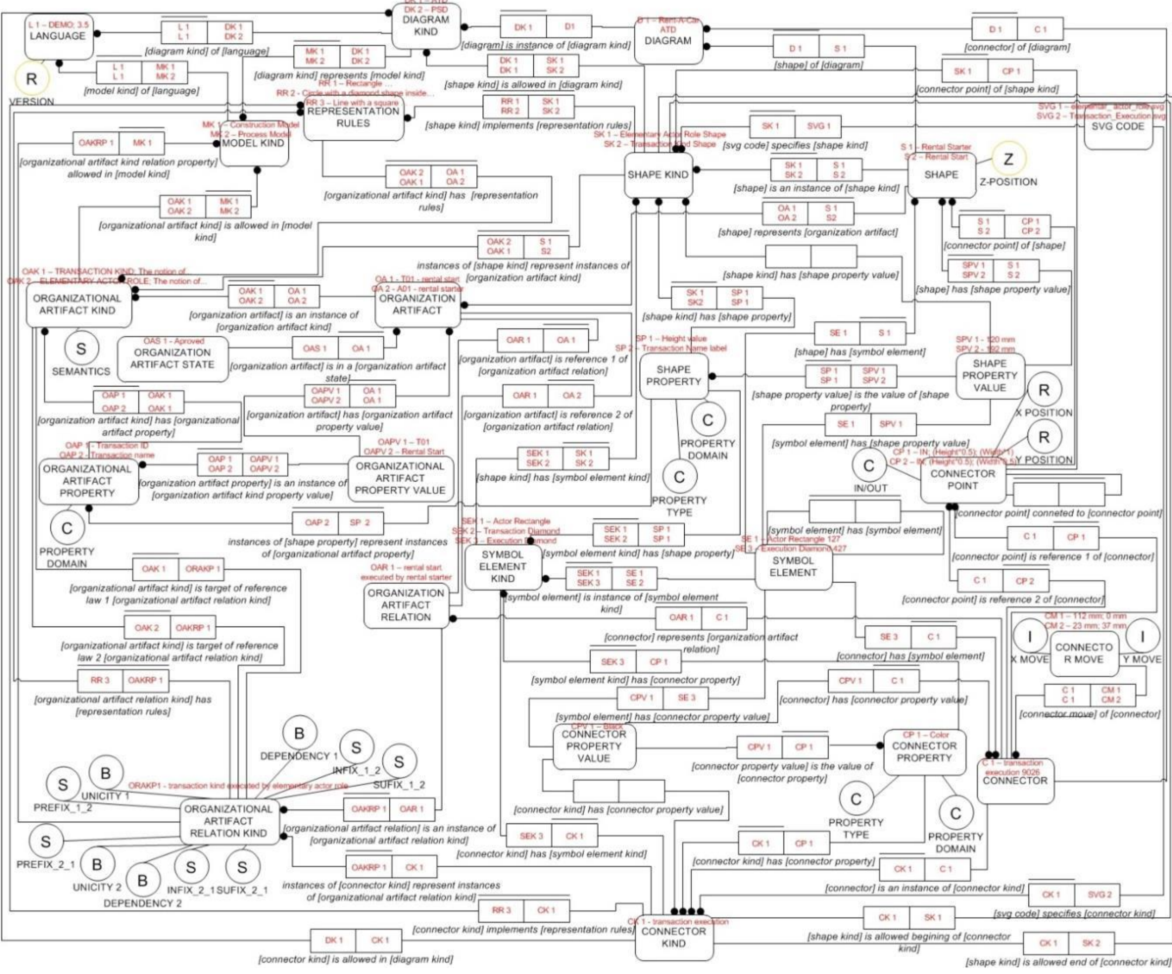


Figure 19 - Universal Enterprise Adaptive Object Model

"In Figure 21 we can see an excerpt of the current DEMO ontological meta-model and the UEAOM classes used to define it. Both these models are the equivalent to the M2 MOF model that, as we have seen, sets the rules for specifying concrete models. All elements of this meta-model can be considered instances of the classes we just have presented. The binary fact type [elementary actor role] is an initiator of [transaction kind] is, in our UEAOM, an instance of ORGANIZATIONAL ARTIFACT RELATION KIND class, with values for the infixes being: initiates and initiated by. There are, however, other classes: DIAGRAM KIND, SHAPE KIND, CONNECTOR KIND, CONNECTOR and SHAPE PROPERTY that are present in this Figure 21 and are part of the meta-model level of the UEAOM but are not part of the abstract syntax, these will be explained in more detail in the next section." [1]

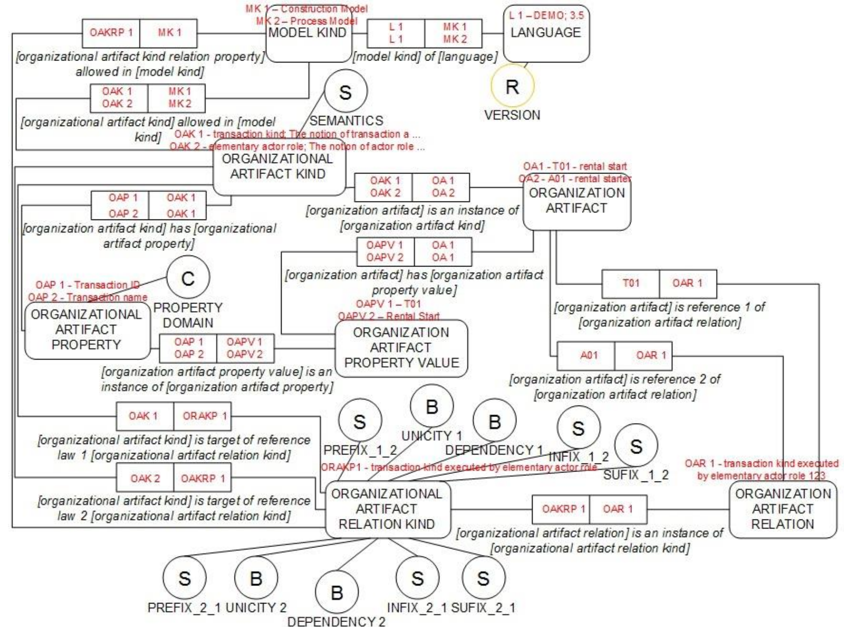


Figure 20 - UEAOM - Abstract Syntax classes

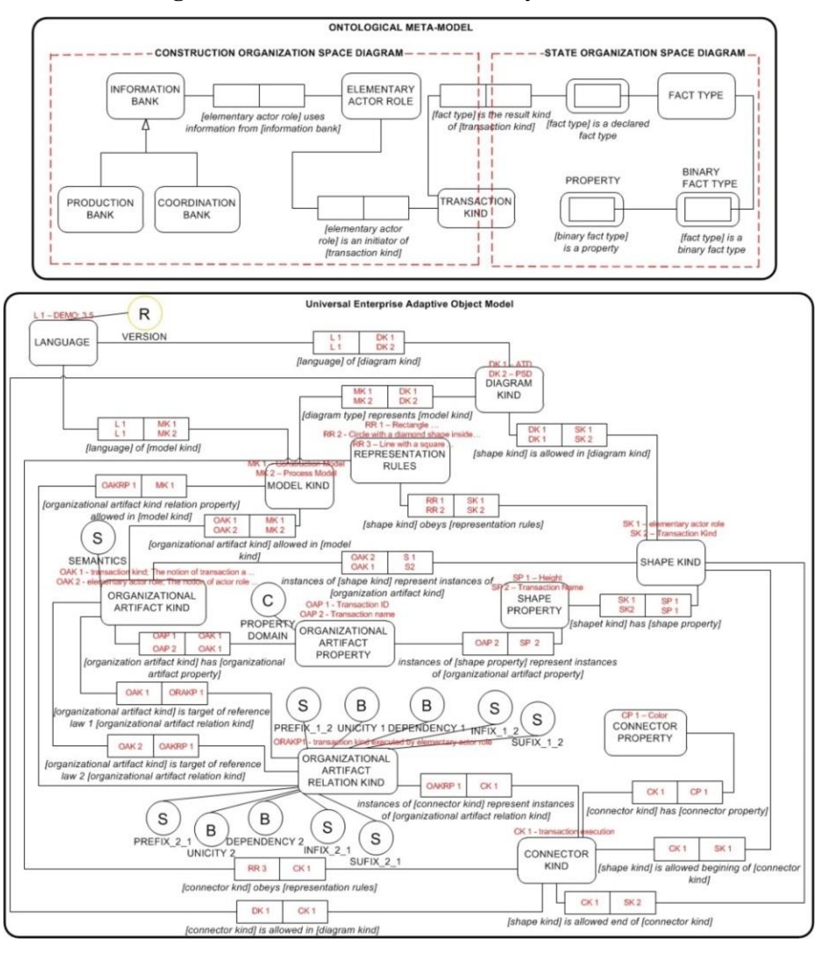


Figure 21 - DEMO Ontological Meta-Model and UAOM classes used to represent it

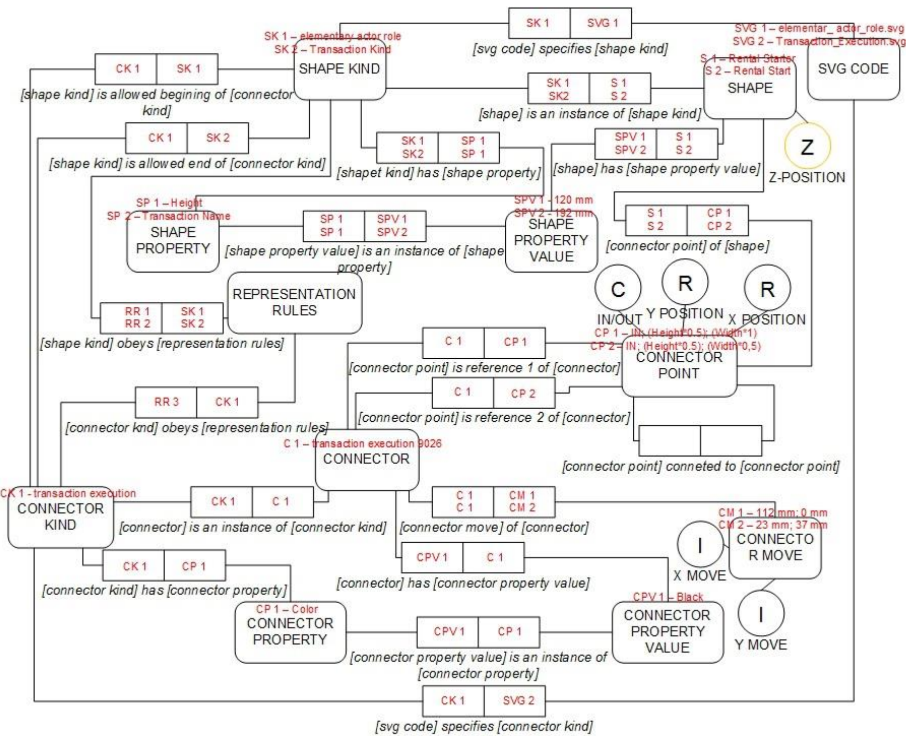


Figure 22 - UEAOM Concrete syntax

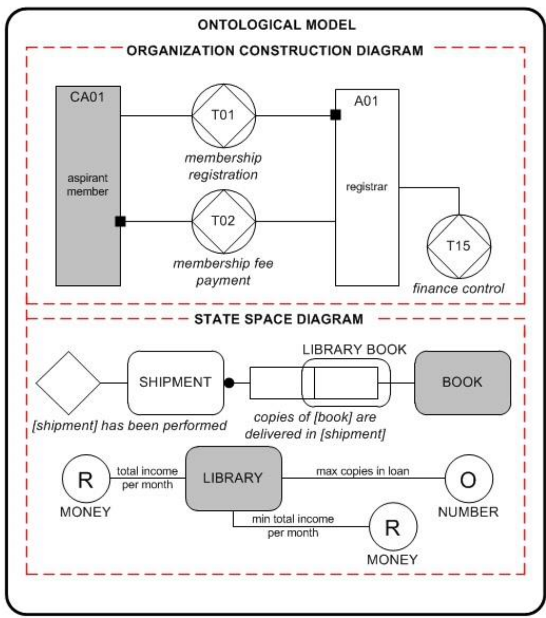
3.4.2 Concrete Syntax

"The UEAOM classes that allow the specification of rules for the concrete representation of models, i.e., the concrete syntax, are presented next. These classes, together with all their inter-relating fact types are present in Figure 22. With the class SHAPE KIND, instances of the types of shapes allowed to be part of diagram kinds representing certain model kinds are specified. These shape kinds are also specifically connected to the OAKs whose instances they will represent. For example, the elementary actor role shape is allowed in diagram kind Actor Transaction Diagram, which represents the construction model of DEMO language. Instances of this shape represent instances of OAK actor role.

With SHAPE PROPERTY, we specify the properties for each shape, e.g., line color and actor id label of actor role shape. Instances of CONNECTOR KIND specify allowed representations for OAKRs, e.g. transaction initiation connector instances represent instances of OAKR transaction initiation. With CONNECTOR PROPERTY, the properties of each connector are specified, e.g., for the just mentioned connector, line color: black and line dashing: continuous.

Instances of REPRESENTATION RULES, class are an informal textual based specification of rules on how ORGANIZATIONAL ARTIFACT KINDS and ORGANIZATIONAL ARTIFACT RELATION KINDS should be represented. These

rules are taken in consideration in either SHAPES or CONNECTORS that represent those OAKs and OARKs. For example, a transaction is a black circle with a black diamond inside. It is also according to the REPRESENTATION RULES that we have a definite answer if an OARK will give origin or not to a connector or if instead it will be represented by the connection of two shape kinds directly. Revisiting the full example from Figure 19, an elementary actor role shape would be an instance of class SHAPE KIND, for the representation of instances of the actor role OAK. Transaction shape would also be an instance of SHAPE KIND for the representation of instances of transaction OAK. So an instance of class CONNECTOR KIND for the representation of this OAKR would be transaction initiator connector, with properties like line type: dashed. Many of the SHAPE KINDs and CONNECTOR KINDs are comprised by multiple symbols that need to be considered individually as having a set of properties. Although in most cases the aggregate of composing symbols are treated as "one" in the diagram drafting, such as a circle and diamond in an actor transaction diagram transaction, that have a fixed size (height and width) and none of them can be altered, there are also cases in which symbols need to be treated and moved in the diagrams in a separate and independent way having their own set of SHAPE PROPERTIES or CONNECTOR PROPERTIES like, for example, in a process step diagram where the diamond inside the transaction can be moved and re-sized according to the needs. As a solution for this, we have classes SYMBOL ELEMENT KIND that specify each symbol element to be present in a shape kind or connector kind and SYMBOL ELEMENT that are instances of SYMBOL ELEMENT KIND and specify concrete representations of SYMBOL ELEMENTS of a specific kind. As an example of this we can consider the actor transaction diagram SHAPE KIND transaction as being composed by the SYMBOL ELEMENT KINDS Transaction Diamond and Transaction Circle." [1]



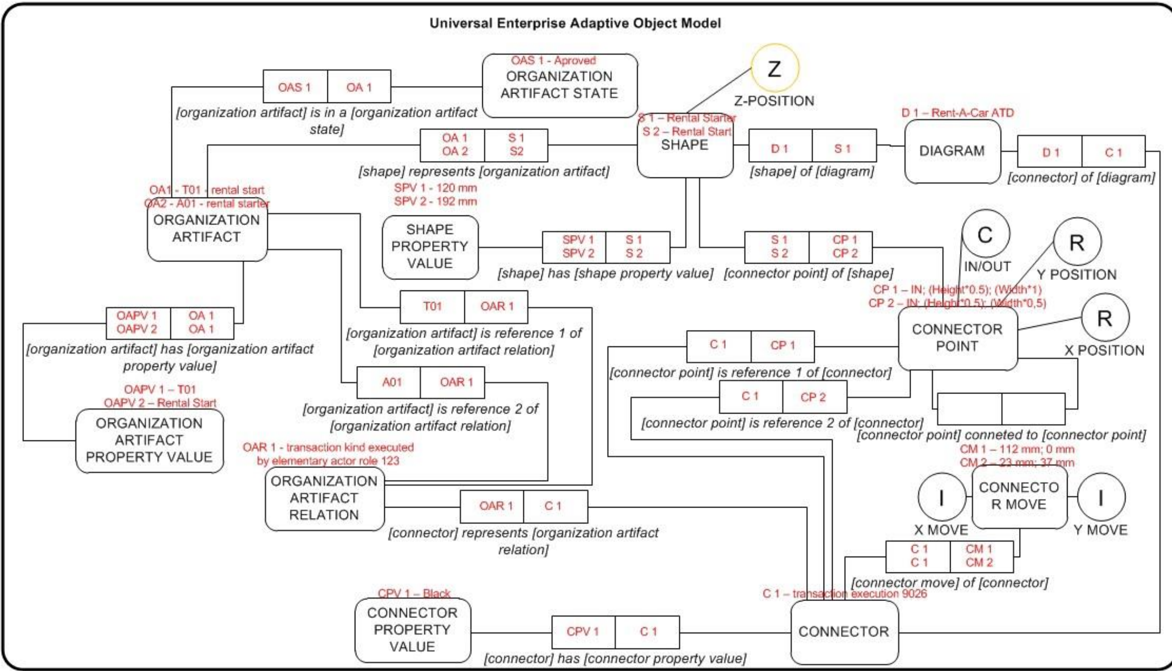


Figure 23 - DEMO concrete diagrams example and UEAOM classes used to represent them

"In Figure 23 we have a partial example of a concrete representation of the DEMO ontological models of an Actor Transaction Diagram and Object Fact Diagram and the corresponding part in the UEAOM. DEMO Ontological models are the equivalent to the M1 level of MOF and instances of their OA's to the MOF's M0 level.

Ontological Models and their representation are covered in the UEAOM by the classes: DIAGRAM, where concrete instances of a certain DIAGRAM KIND are specified; SHAPE, where concrete instances of SHAPE KIND are specified; SHAPE PROPERTY VALUE, where concrete instances of SHAPE PROPERTY are specified; CONNECTOR, where concrete instances of CONNECTOR KIND are specified;

CONNECTOR PROPERTY VALUE, where concrete properties of the CONNECTOR PROPERTY are specified; ORGANIZATIONAL ARTIFACT, where concrete instances of ORGANIZATIONAL ARTIFACT KIND are specified; ORGANIZATIONAL ARTIFACT PROPERTY VALUE, where concrete OA properties are specified and ORGANIZATIONAL ARTIFACT RELATION, where concrete instances of ORGANIZATIONAL ARTIFACT RELATION KIND are specified and all their relating fact types. In this way, also allowing them to be changed in an easy and consistent way in run-time environment.

Again using a concrete example from Figure 23, we have the "CA-01 aspirant member shape", this is an instance of SHAPE (this SHAPE an instance itself of the SHAPE KIND "Composite Actor Role") that represents the ORGANIZATIONAL ARTIFACT "CA-01 aspirant member" (itself an instance of the ORGANIZATIONAL ARTIFACT KIND "Composite Actor Role"); the string "aspirant member" is an instance of SHAPE PROPERTY VALUE (that represents the instance of ORGANIZATIONAL ARTIFACT PROPERTY VALUE "aspirant member") and so is "CA-01". These two strings are VALUES, instances of the SHAPE PROPERTIES "Actor Name" and "Actor ID" respectively (that again represent the instances of ORGANIZATIONAL ARTIFACT PROPERTY VALUE "Composite Actor Name" and "Composite Actor ID").

The DIAGRAM KIND and MODEL KIND classes were not present in the original DEMO Ontological meta-model as all models were specified in this single meta-model. But in our UEAOM, as we have generalized this definition to accommodate any language for organizational modeling, the DIAGRAM KIND and MODEL KIND classes are vital so we can relate to each specific Ontological Model. Actor Transaction Diagram or Process Step Diagram would be examples of instances of this DIAGRAM KIND while the first would be a representation of the MODEL KIND Construction Model and the second a representation of the MODEL KIND Process Model. The LANGUAGE class and its property VERSION is used to define the modeling language being modeled and the version of such language." [1]

3.5 EE tools supporting DEMO

"To generalize the access and awareness of the organizational reality is not a trivial task. Such tool must not only enable the collection of distributed and coherent

organizational knowledge aligned with the organizational reality but also be understandable and of easy use by any of the organization's collaborators. This collection of organizational knowledge should be in an integrated repository of both the conceptual understanding and the symbolic understanding in the form of diagrams and tables. There are some solutions for DEMO modeling like Visio [22] (only diagrams), Xemod [23] and ModelWorld [23] but we found ourselves facing the same issues with all of them. For our objectives Visio would be the less helpful, as it offers no support for anything but diagram specification, and even that support is achieved by custom made stencils that until now have a very limited way of enforcing the rules and/or restrictions of the modeling language. Visio also fails to help us with our needs of generalized access and awareness of organizational reality and facilitating incremental changes to models of organizational reality, as it does not present a way of offering a generalized access to the knowledge nor it facilitates any kind of coherent incremental change. A change in a diagram is exclusively a change in that diagram, it does not propagate to other diagrams that share the same organizational fact.

Xemod is a tool built exclusively for DEMO modeling and as such offers another level of support. This support comes at a cost, as this is also a far more expensive tool than a basic license of Visio. But even though in Xemod we have a set of rules to help us model and support for the whole methodology and not only the diagrams, Xemod also has its issues: (1) it is impossible to change the pre-existing stencils, so if the standards change, you are most likely having to pay once more to upgrade your tool to the newer version; (2) ineffective way offered to propagate knowledge – unless you have Xemod installed on every workstation, the way of sharing your organizational knowledge is by exporting it to a far more complex Access database or Excel spreadsheet; (3) we find another flaw that is shared with Visio, although in Xemod there is some sort of change propagation, as your changes in a Transaction Result Table reflect in all diagrams, it still doesn't provide a full support in the changes as, for instance, changing an Object Fact Diagram class name, does not reflect neither in the name of related fact types nor in the name of related result type name.

ModelWorld is an online modelling and diagramming tool for business architecture models and, unlike the previous two, is free. As it runs on the browser it has the advantages of being platform independent and allowing for collaborative modelling and validations. Just like in Xemod, ModelWorld has a set of rules on how diagram shapes

in DEMO modeling relate, but in the same way it does not allow for the stencils change either, leaving you at the mercy of the updates, in this case however, free of charge.” [1]

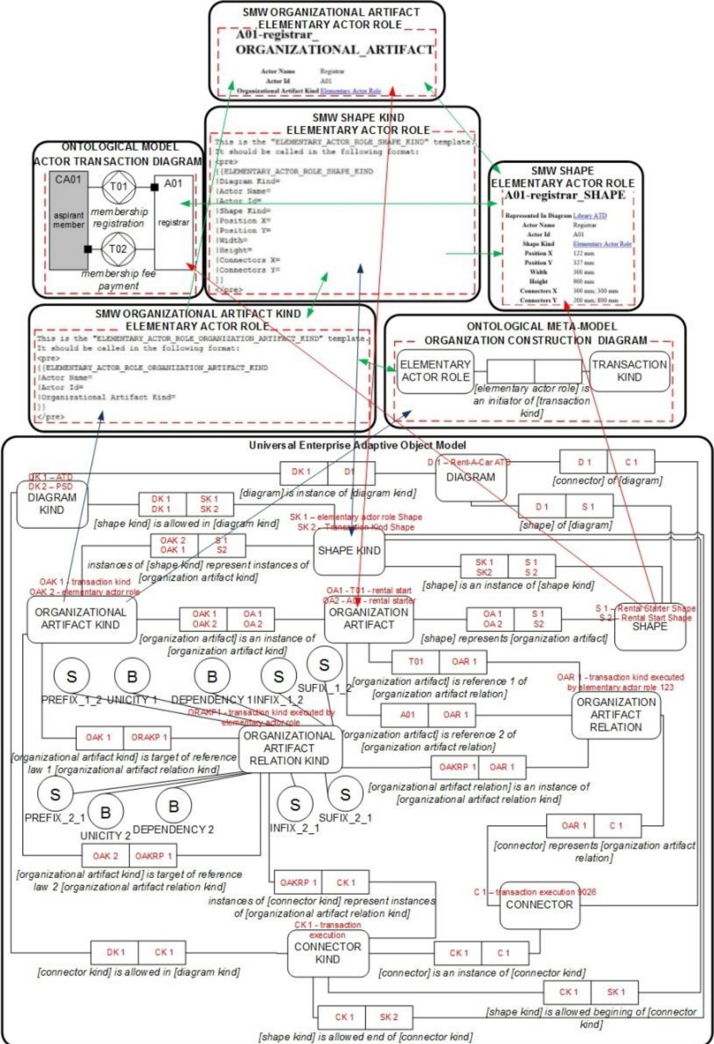


Figure 24 - Overview of the SMW Pages and their Relation with DEMO Models and UEAOM

“But all three tools fail to meet the needs; in facilitating the incremental and integrated changes. ModelWorld too offers very little support in this important aspect and, for a generalized access and awareness, like Visio, ModelWorld only allows exporting the information in diagram form.

These problems encountered in current state-of-the-art tools led us to developing our own tool as to fully support and implement our notion of organizational self-awareness and the evolution of the meta-models governing such awareness.” [1]

3.6 Realizing OSA with a wiki

“The Wikipedia article on Berlin article contains many links to other articles, such as «Germany» and «European Union». However, the link to «Germany» has a special

meaning: it was put there since Berlin is the capital of Germany. To make this knowledge available to computer programs, one would like to «tag» the link [[Germany]] in the article text, identifying it as a link that describes a «capital property». With Semantic MediaWiki (SMW), this is done by putting a «property name» and «::» in front of the link inside the brackets, thus: [[Is capital of::Germany]]. In the article, this text still is displayed as a simple hyperlink to «Germany». The additional text «capital of» is the name of the property that classifies the link to Germany. Information that was provided in an article is now provided in a formal way accessible to software tools. We foresaw that SWM could be an adequate base to develop an EE tool supporting and facilitating OSA. For that we took some implementation decisions described next." [1]

3.6.1 Fundamental patterns used

"As a theoretical base for the realization of OSA with a SMW we have specified in the Universal Enterprise Adaptive Object Model (UEAOM)[1]. This model is represented in a diagram in the World Ontology Specification Language [13], a derivative of the Object Role Modelling (ORM) language [16] Due to the inherent preciseness and first order logic predicate behind ORM, also WOSL is a very adequate language for our goal to specify in a powerful and precise way all aspects of models, the respective meta-models, their representations as well as their evolution. The classes of our UEAOM follow the type square and the adaptive object model patterns [15] usually applied to software engineering to allow dynamic and runtime evolution of a software system's services, but here applied to the enterprise engineering context, precisely to allow dynamic and runtime evolution of not only organization systems, but also of the meta-models governing the structure and instantiation of the elements of organization systems. Following Figure 24, certain wiki pages will be the specification of objects that are instances of the following classes of our UEAOM: ORGANIZATION ARTIFACT KIND (OAK) – for the specification of meta-model elements; SHAPE KIND – whose pages specify shape kinds whose instances will represent instances of OAKs; ORGANIZATION ARTIFACT – whose pages specify OAs – at model level – that are instances of certain OAKs; and SHAPE, whose pages specify particular shapes – at diagram level – representing particular OAs. A similar reasoning is followed for the classes ORGANIZATION ARTIFACT RELATION KIND (OARK) – relation kinds between OAKs; CONNECTOR KIND; ORGANIZATION ARTIFACT RELATION;

and CONNECTOR. An example of a SHAPE that represents – at diagram level – an OA (itself, in turn, at model level) consists in the page titled: «A01-rental_starter_shape». A page called «A01-rental_starter» will be an OA represented by the before-mentioned shape. This OA, in turn, is an instance of the OAK, itself specified by the page «ACTOR ROLE». This page, in turn relates to page «ACTOR ROLE SHAPE KIND» specifying the characteristics of a shape to represent the OAK specified by the page «ACTOR ROLE». As we can see by these examples, one of the advantages of using the adaptive object model (AOM) and type square patterns is that they allow a systematic and precisely organized instantiation at several levels and concerns, namely: AOM classes' level, meta-model, model and representation concerns – while keeping all relevant relationships between objects.

On each page one has to specify semantic wiki properties to provide semantics to the respective object of the UEAOM, relating it with all other relevant objects. In the example, by adding to the page «A01-rental_starter» the link: [[is_instance _of::ACTOR ROLE]], we specify with the property is instance of that A01 is an instance of the meta-model level OAK specified by the page «ACTOR ROLE». All pages that are instances of OAKs or OARKs need to specify the property [[is_instance_of::ORGANIZATION ARTIFACT KIND]] or [[is_instance_of:: ORGANIZATION ARTIFACT RELATION KIND]] respectively. Following this method we can, for example, use the SMW mechanism of semantic queries and automatically obtain a list of all OAs, OARs and also the respective SHAPEs and CONNECTORs that represent them. We can also execute queries to dynamically obtain the current version of the meta-model, specified by the pages instances of OAKs and OARKs." [1]

3.6.2 Creating models and diagrams

"Our implementation based in a SMW serves the purpose of not only formally specifying the meta-model behind models and diagrams, but also of the organizational self and its change and to visualize and edit diagrams automatically and dynamically generated from the pages and their semantic properties. Each ORGANIZATION ARTIFACT page makes no sense by themselves, and need to be contextualized in a user friendly way. This contextualization is achieved in two steps, the first is to establish relations between the OAs by creating ORGANIZATION ARTIFACT RELATIONS and the second is representing such OAs in a DIAGRAM. Just like OAs and OARs are

instances of certain OAKs and OARKs, DIAGRAMS will be instances of a certain DIAGRAM KIND. A page specifying a DIAGRAM KIND, in turn allows the formalization of which SHAPE KINDs are allowed in a certain DIAGRAM. For example the «ACTOR TRANSACTION DIAGRAM KIND» page specifies that only the presence of SHAPE KINDs «ACTOR ROLE SHAPE» and «TRANSACTION KIND SHAPE» is allowed in instances of this diagram kind. Concrete diagrams of an organization like, for example, the EU-Rent Actor Transaction Diagram (ATD), are pages specifying instances of the UEAOM class DIAGRAM and these pages, in turn, have to contain the property is_instance _of_diagram_kind::ACTOR TRANSACTION DIAGRAM KIND. These wiki pages that specify concrete diagrams have a special behavior implemented by an extension to SMW. These pages output a DIAGRAM generated in run time environment using those OA pages and their semantic properties, automatically generating an image implemented in the Scalable Vector Graphics (SVG) format: an open format allowing easy import/export operations and also zoom operation without losing resolution quality. The page «A01-rental_starter_shape» is an example of an instance of a SHAPE and the page «CA02-Driver.is_ the_executor_of.T03-car_drop-off_connector» is an example of an instance of the UEAOM class CONNECTOR, and semantically associated with the page «transaction_execution_connector_kind», itself an instance of CONNECTOR KIND. At meta-model level, instances of classes SHAPE KIND and CONNECTOR KIND will be associated with instances of classes SHAPE PROPERTY like, for example, «actor_id» and of CONNECTOR PROPERTY like, for example, «line_color». At model level these properties are instantiated as objects instances of classes SHAPE PROPERTY VALUE and CONNECTOR PROPERTY VALUE. For example, «A01» and «Black», respectively. These are more examples of the application of the type square pattern, also applied in the case of OAKs and OAKRs furthermore showing the immense power to our approach. Instances of properties and values could have been also implemented as wiki pages but the most appropriate approach was to use the mechanism of semantic properties already present in SMW.

Thus, classes of our UEAOM that include the name property are usually implemented as properties in the respective pages and classes including the term VALUE, as values of the semantic properties themselves in the respective wiki pages. The dynamic power of type square power is kept as we can dynamically change properties that can be associated with kinds by editing the special wiki pages of templates.

To facilitate the process of creation of models and their representations, and also make the changes directly made in SVG diagrams reflect in SMW pages and their properties, a java script based diagram editor is currently under development to implements all the functionality deemed convenient like ones present in well-known modeling tools such as Microsoft Visio.” [1]

3.6.3 Page names and semantic properties

Properties	Values
Height (open)	100 mm
Width (open)	100 mm
Description (open)	Driver that picks up the car at the rental and drops it off.
Connector points (open)	CP 1 (100 mm
Connector points (open)	50 mm)
Connector points (open)	CP 2 (50 mm
Connector points (open)	100 mm)
Connector points (open)	CP 3 (50 mm
Connector points (open)	0 mm)
Connector points (open)	CP 4 (0 mm
Color (open)	Black
Represented organi... (open)	Driver
Is instance of (open)	ACTOR ROLE SHAPE KIND
Symbol elements (open)	Actor Rectangle 2
Actor id (open)	CA-02
Actor name (open)	driver
Diagram (open)	EU-Rent

Figure 25 - Semantic box for CA02-driver_shape

“A standard specification is an explicit set of requirements for an item, material, component, system or service. The need to define a standard nomenclature for wiki pages is crucial to create a homogeneous model and ensure compatibility with other projects that may be developed and integrated with this. A wiki page representing a ORGANIZATION ARTIFACT KIND or ORGANIZATION ARTIFACT RELATION KIND at meta-model level consists of capital letters and words are separated by underscore. For example, the wiki page for representing a «transaction kind» fact type should be «TRANSACTION_KIND». A wiki page representing an instance of a ORGANIZATION ARTIFACT is a little different. It consists of a capital letter followed by an order number, and then a hyphen, followed by the name of the fact, where words are lower cased and separated by underscore. For example, the wiki page «A01-rental_starter» is defined by the capital letter «A» (standing for ELEMENTARY ACTOR ROLE) followed by the number «01», and then a hyphen followed by the actor name “rental_starter”, both of these are ORGANIZATION ARTIFACT PROPERTY

VALUES, used here together to form an identifier of the page. On another example, the wiki page «CA02-driver» is defined by the capital letters «CA» (standing for COMPOSITE ACTOR ROLE) followed by the number «02» and then a hyphen followed by the composite actor name “driver”.

Properties also have a simple standard nomenclature. Any property consists of lowercase words separated by underscore. Examples of valid properties are: «actor_id», and «initiating_actor_role». Every class represented in the UEAOM will have their own set of properties that need to be implemented (given values) for the creation of instances of such class. There is no typical SMW page for the classes of our UEAOM, these are specified in the implementation as templates and forms that we will explain in greater detail in section 3.4. As an example, Figure 25 depicts the semantic box that shows the properties for the wiki page «CA02-driver_shape», an instantiation of the class SHAPE.

The instances of OARK and OAR are a particular case when it comes to the nomenclature. Here we are dealing with composed OAKs and OAs with multiple elements, and as such this has to be considered in the naming. For example in an Actor Transaction Diagram we have two kinds of OARKs, the «ACTOR_ROLE.is_an_initiator_of.TRANSACTION_KIND» and «ACTOR_ROLE. is_the_executor_of.TRANSACTION_KIND». As previously, and maintaining the coherence, at the meta-model level the OARKs consist of capital letters separated by underscore, but here composed with possible a prefix, an infix and/or a suffix, in lower case, also separated by underscore to create the full name of the OARK. At model level the principle is also the same, the nomenclature used is the names used of the relating OAs (in lower case) as previously explained separated by underscore, again with a possible prefix, infix and/or suffix. An example of an OAR is «A01-rental_starter.is_ an_initiator_of.T02-car_drop-off».

To help on the task of remembering all the names of the fact types, the Halo extension used. It is an extension to SMW and has been developed as a part of Project Halo in order to facilitate the use of Semantic Wikis for a large community of users. The focus of the development was to create tools that increase the ease of use of SMW features and advertise the immediate benefits of semantically enriched contents. We decided to use Halo due to its auto-completion feature that is a great help for the task of defining and reusing organization artifacts. This happens as, for example, actor roles and transactions names are frequently changed and Halo extension allows us to prevent inconsistencies in the specification and interpretation of the artifacts and their names.

With Halo it's possible to define properties in a very clear manner to connect pages and create semantic relations between them.” [1]

3.6.4 Semantic Forms

Template:DIAGRAM TEMPLATE

This is the "DIAGRAM_TEMPLATE" template. It should be called in the following format:

```
{{DIAGRAM_TEMPLATE
|Diagram ID=
|Diagram Name=
|Is Instance of=
|Represented Model Kind=
|Diagram Description=
|Represented Shapes=
|Represented Connectors=
}}
```

Figure 26 - DIAGRAM TEMPLATE page view

“Semantic MediaWiki offers us countless extensions, one of them being the Semantic Forms. Semantic Forms are of a substantial value in maintaining the correctness and the structure of the whole wiki pages. Semantic Forms allow for a full structural definition for all the pages of the same kind using three constructs; properties, templates and forms.

Properties are the elementary “construct” of semantic forms, and, for every piece of information in a SMW page, a property should be created. For example in the page «CA02-driver_shape» on SMW, we would have properties such as «actor_name» or «actor_id» as represented in Figure 25.

These properties are then grouped in Templates. Templates are in a basic way structuring the allowed properties for each page. In a concrete implementation of the UEAOM, there is the need for a template for each of the represented classes in the model in order for structured instances of those classes to be created. For example for the Object Class Diagram, a template would list the «diagram_id», the «diagram_name», the «is_instance_of», the «represented_model_kind», the «diagram_description», the «represented_ shapes» and the «represented_connectors» as shown in Figure 26.

One would notice that in Figure 26, the names are not accordingly to our previously defined nomenclature nor the properties just mentioned. This is because the lists in the Template page are not the properties themselves but instead, the label names we decided to give to each of them when creating the template. These labels are useful to maintain the pages user friendly but one could have simply used the same name. In the “edit

page” option in the «DIAGRAM TEMPLATE» page we can find the corresponding properties to each label as shown in Figure 27. Although this Template is used to create instantiations of DIAGRAM, this is not the same thing as the class DIAGRAM KIND, such class still needs to exist as a page and with a template of its own. The template pages for the UEAOM classes can be seen as the SMW page implementation of the classes themselves.

Editing Template:DIAGRAM TEMPLATE

```
{| class="wikitable"
! Diagram ID
| [[diagram_id::{{{Diagram ID|}}}]]
|-
! Diagram Name
| [[diagram_name::{{{Diagram Name|}}}]]
|-
! Is Instance of
| [[is_instance_of::{{{Is Instance of|}}}]]
|-
! Represented Model Kind
| [[represented_model_kind::{{{Represented Model Kind|}}}]]
|-
! Diagram Description
| [[diagram_description::{{{Diagram Description|}}}]]
|-
! Represented Shapes
| {{#arraymap:{{{Represented Shapes|}}}|,|x|[[represented_shapes::x]]}}
|-
! Represented Connectors
| {{#arraymap:{{{Represented Connectors|}}}|,|x|[[represented_connectors::x]]}}
|}
```

Figure 27 - DIAGRAM TEMPLATE edit view

The Forms are the implementations for the templates. For instance in a «DIAGRAM FORM» one would fill all the listed properties in the «DIAGRAM TEMPLATE», but this task is intended to also be allowed by creating a diagram using SVG that would automatically generate the template, leaving no need to use the forms for such creation. Forms however are still useful in an editing perspective, as one can edit the pages using the form instead of the visual editor, while keeping the data structured.” [1]

4 Implementation

For the implementation of UNIVERSAL ENTERPRISE ADAPTIVE OBJECT MODEL (UEAOM), we had the wiki framework MediaWiki and its extension Semantic MediaWiki, which adds the capability to work with semantic data, which leads us to the use of templates, explained in the section Templates.

Basic idea was to follow the UEAOM which was created using the knowledge from Enterprise Ontology (EO), displayed in Figure 19 which is part of the research developed by the thesis advisor.

With UEAOM in focus we had to develop the best way to create the pages, so the essential classes in UEAOM are considered pages (Numeric non decimal values in Figure 29) and by following the fluxogram (Figure 28) it`s possible to create a certain language. Also it`s possible to aid the creation of the pages for diagram editor, which was done in conjunction with this project by another colleague, with whom there was a constant contact for discussing issues and ideas, in parallel with the thesis advisor.

From this Model, it was possible to define the majority of the implementations of the templates for the development of a wiki-based system and that same model can be later improved depending on the evolution and needs of the future. The initial idea is to create Forms which are going to be used to created pages with their associated template.

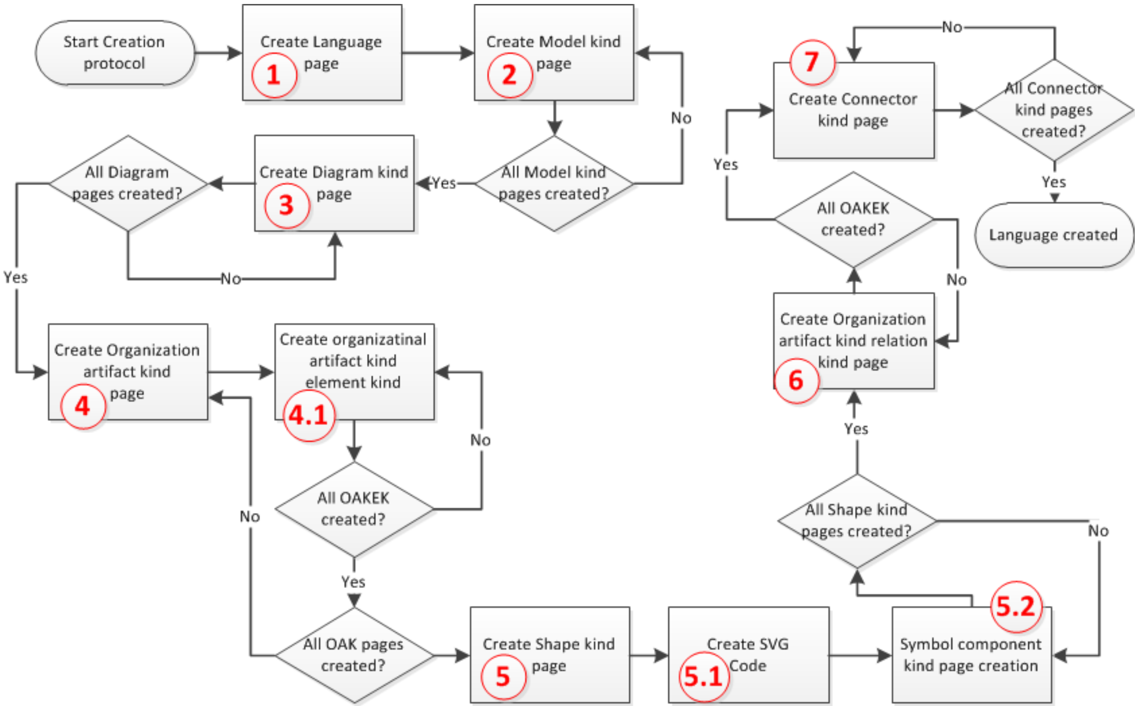


Figure 28 - Fluxogram of the creation protocol

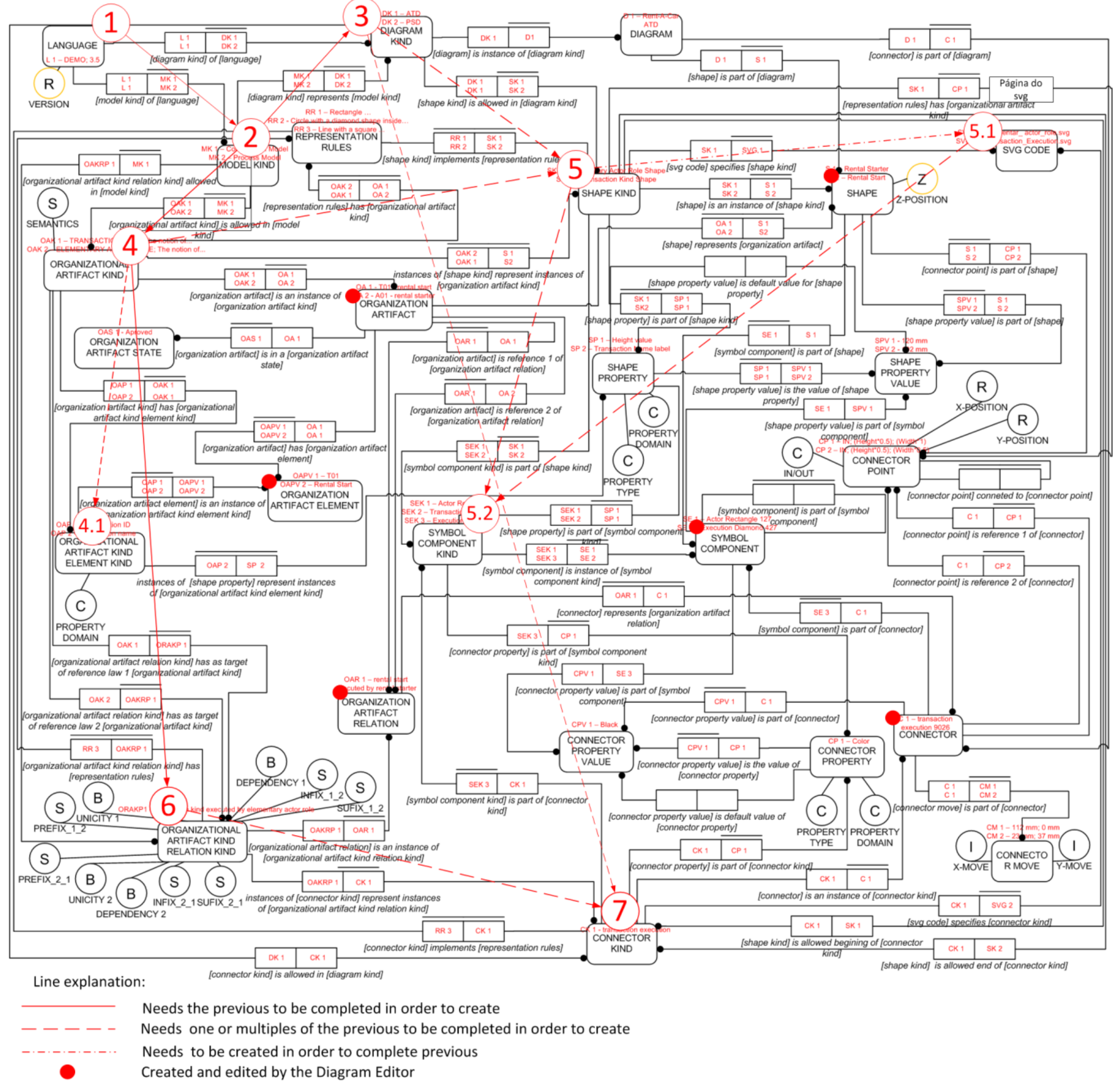


Figure 29 - UEAOM – Page creation protocol to follow and line explanation

Following the rules from EO, the general idea is that for each connection, where there is a black dot in the end, it needs the initial class/page of the start of the connection to be created in order for the class/page, associated with the black dot, to be created. So before the class/page is created, the previous class/page to which it’s connecting needs to be created. Following Figure 29 and the numbers below, we can see the protocol and brief explanation for creating pages for a certain Language (Meta-Modeling):

1. Create Language page, using Language Form.
2. Create one or multiple Model kinds pages, using Model kind Form.
3. Create one or multiple Diagram kinds pages, using Diagram kind Form.
4. Create one or multiple Organizational artifact kind pages, using Organizational artifact kind Form, each can have:

4.1. Multiples Organizational artifact kind element kind, which is not a page but a template, used in conjunction with the previous template in order to complete the Organizational artifact kind.

5. The Shape kind page can be created, after Organizational artifact kind is created, using the shape kind template, which creation is better explained in SVG-Edit MediaWiki toolbar use procedure.
   5.1. Shape kind page has a property called SVG Code, from which the SVG related code is created via SVG-Edit, and a page is created via the SVG-Edit application, with the SVG image, and link via that property.
   5.2. For each shape component (element in SVG) created in SVG-Edit, an element/component page is created with each element having its own specific properties filled with the data from the SVG which varies between numeric and string and mostly dependent on the SVG permissions for each property.
6. The Organizational artifact kind relation kind page can be created, after the Organizational artifact kinds are created, by using the OAKRK Form.
7. The Connector kind can be created, only after the Diagram kind page and Organizational artifact kind relation kind page are created, because each of them is needed in order for their values to be available in the select menu, in the connector kind interface. Those pages are going to be used as a value, in a property, on the connector kind template. This is not created by a form but by an interface, called Connector-Edit, created for this specific page creation, which is better explained in chapter Connector-Edit.

The classes with a red dot are the classes that are used in conjunction with templates to create pages for the diagram editor phase of this project which is being elaborated by another colleague. All of this is explained on the chapters about Templates and properties, but essentially, templates are used on the pages associated with certain classes in a very similar way like we explained before, all of this is explained in chapter 4.4 Defining wiki pages and semantic properties.

4.1 MediaWiki

Media-wiki is considered the most used wiki framework, since it is a free open source software mostly know by being used by Wikipedia to provide knowledge to the world. Any person can download and use it and it has many tutorials how to install and use. Media-wiki has been improving overtime and one of its most important features is the possibility to use extensions which are used to improve or add new types of content or functionalities to the core software.

4.2 Configuring and installing MediaWiki

For the configuration of MediaWiki and installation, there was a need to have PHP installed and MySQL also. The version of the software used is displayed in Figure 30 which is a print screen from the MediaWiki page Special: Version.

Installed software

Product	Version
MediaWiki	1.19.2
PHP	5.3.8 (apache2handler)
MySQL	5.5.16

Figure 30- MediaWiki installed software

4.2.1 Basic Extensions:

For the use of the semantic information, there were a few more applications to install which are called extension because they extend the capability of MediaWiki, which can be visualized in Figure 31.

- **Semantic MediaWiki** extension allows for the use of semantic data inside MediaWiki.
- **Semantic Form** extension allows the use of forms that create pages with a specific template or even multiple templates depending how the form is constructed. And it needs the **Semantic MediaWiki** extension in order to work, since it's basically an extension to that extension.

Installed extensions

Semantic extensions		
Semantic Forms (Version 2.5.1)	Forms for adding and editing semantic data	Yaron Koren, Stephan Gambke and others
Semantic MediaWiki (Version 1.7.1)	Making your wiki more accessible - for machines *and* humans (online documentation)	Markus Krötzsch, Denny Vrandecic and Jeroen De Dauw. Maintained by AIFB Karlsruhe.

Figure 31 - Semantic extensions installed

4.2.2 Parser hooks:

MediaWiki uses parser hooks to add functionalities inside pages, and since there was a need to create two of them which were called connector extension, which is used for the inline addition of the connector-edit page, and special page name OAKRK, which is used to create the page name for OAKRK pages, both displayed in Figure 32.

Parser hooks		
connector extension	Display connector-edit page in iframe	buga based on Minseong Kim and Others
ScriptManager Extension (Version 1.0.0_0)	Organizes javascript libraries.	Kai Kühn. Maintained by Ontoprise.
Special pagename OAKRK (Version 1)	<Special pagename parser function Extension for creating OARK page name>	Vitor N�brega

Figure 32 - Parser hooks installed

4.2.3 Additional extensions

Another addition to MediaWiki was SVG-edit extension which permits for the manipulation of SVG shapes inside MediaWiki and WikiEditor which added an improved interface while editing pages, which we can also in MediaWiki page Special: Version and displayed in Figure 33.

Other		
SVGEdit	In-browser editing of SVG drawings with SVG-Edit	Brion Vibber
Validator (Version 0.4.13)	Provides generic parameter handling support for other extensions	Jeroen De Dauw
WikiEditor (Version 0.3.1)	Provides an extendable wikitext editing interface and many feature-providing modules	Trevor Parscal, Roan Kattouw, Nimish Gautam and Adam Miller

Figure 33 - Other extensions installed

- **SVGEdit** extension adds functionalities that permits for the use of SVG-Edit inside MediaWiki without any
- **Validator** extension like the names says validates Semantic data input, by providing warnings and error outputs. It's considered a small utility extension that helps **Semantic MediaWiki** validate user-provided parameters.
- **WikiEditor extension** provides the user with a more user friendly interface when editing MediaWiki pages, which can be showed in Figure 34, it gives the user a powerful tool especially for a beginner user that don't know anything, on how to edit wiki pages.

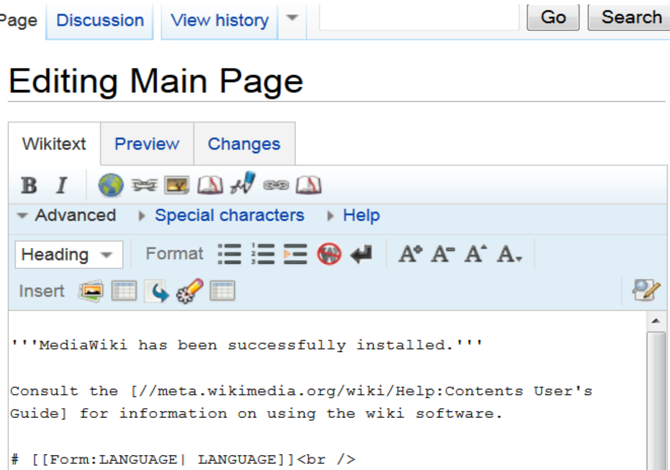


Figure 34 –WikiEditor interface

4.2.4 Parametization

4.2.4.1 SVG technical modifications

For the visualization of SVG images directly inline MediaWiki wasn't possible so we had to use a converter to do so, by using the following commands:

```
$wgFileExtensions[] = 'svg';
$wgSVGConverter = true;
$wgSVGConverters['ImageMagick']=   'C:/xampp/htdocs/ImageMagick-6.8.5-9/convert
background white -geometry $width $input $output';
$wgSVGConverterPath = "C:/xampp/htdocs/ImageMagick-6.8.5-9";
```

These commands convert the SVG image into temporary PNG format, which MediaWiki works with. While maintaining the basic SVG file it creates a temporary PNG file of that SVG file (saved in the thumb directory inside images), which is going to be visualized in the SVG file page, which we can see in Figure 35 (Which contains

some conversion issues, in most cases caused by the attribute translate) and the real version which is visualized in html page and not directly inside MediaWiki, Figure 36.

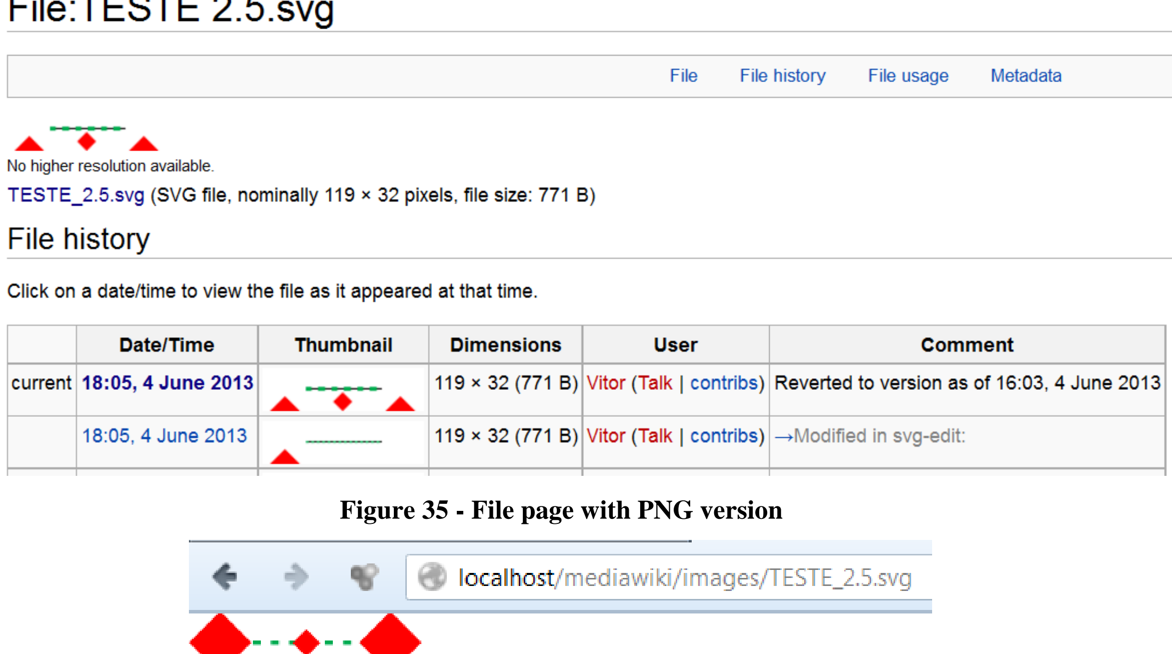


Figure 35 - File page with PNG version

Figure 36 - Original SVG File

4.2.4.2 File system

There was a need to know the location of the SVG file, and there was a problem with the manner that MediaWiki saves the files, which wasn't easy to follow. MediaWiki normally uploads files in safe mode, which creates a complex directory system which isn't easy to reach outside Mediawiki, and because of that it was decided to simply turn it off, by using the following command:

```
$wgHashedUploadDirectory = false;
```

This simply puts all uploaded files into the images directory, which saves us from creating a complex algorithm to find the file, and then creates a simple link like the one seen in Figure 36 (e.g. $IP/images/TESTE_2.5.svg). In the more complex case, the file would be distributed into sub-directories of "$wgUploadDirectory" based on the first two characters of the md5 hash of the filename, which creates, in all cases, a complex and unnecessary file system (e.g. $IP/images/a/ab/TESTE_2.5.svg). There is a good reason for this sort of system, which is that some file systems doesn't perform well with large numbers of files in one folder. While that can be an issue in the future, when we have a large number of images, there isn't a big reason to worry about it, at the moment,

since the previous versions of the files are still maintained in the normal directories format, so the amount of files won't be that large.

4.3 Semantic Web

The word Semantic means the study of meaning, and in this case semantic web means to interconnect the meaning of the words in the web. It's basically a way to both Computer and humans to comprehend themselves, by giving meaning to the content of pages and data, which helps the computer to interconnect information and humans to reach that information in a comprehensive way, which in the long run, means that computers could share global knowledge in an easy and understandable way[6].

4.4 Defining wiki pages and semantic properties

In order to create semantic properties we use the **Semantic MediaWiki** (Version 1.7.1) extension in order to provide semantic data within the wiki page.

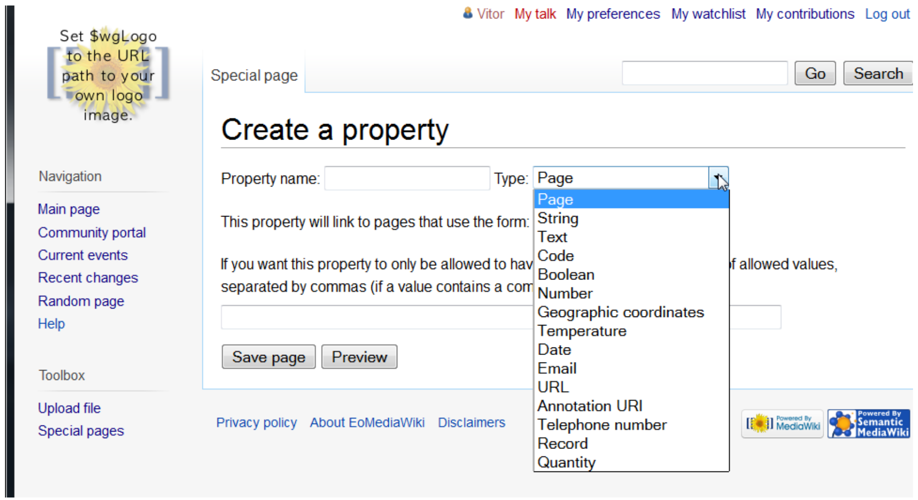


Figure 37 – Menu for creating semantic Property with Type options

One of the issue, well not really an issue but something there was a need to comprehend, is that when a property was created, if the first letter was lowercase or uppercase it didn't matter because if the first letter was lowercase it would be directed to the uppercase one for example, if a property with the name "teste" was created, what would be created would be the Property:Teste, which is something that Semantic MediaWiki does automatically.

To use this property in a page the correct example would be: [[Teste::example]] while if we used [[teste::example]] it would consider this a dynamically created property

different from the one we created, in conclusion properties are case sensitive and there's a need to pay attention to it, in order not to have conflicting properties.
To create a dynamically created property, a property must be used in any page inside MediaWiki different from any manually created properties, for example: [[dynamically::link to page]].

For the manually created the must be a property page created with that name, where it can be added some specific information like the type of property but the dynamically created ones are consider [[Has type::Page]].

While there are a few type of property, like we can see on figure 1, only 4 of them are used like: Page, string, text and number. And in the case of property type: Page, it can be used to link to specific files inside MediaWiki like we can see in figure 2 and to do so, we only need to write something like this example: [[End connector symbol position::File:Inside_shape.svg]]

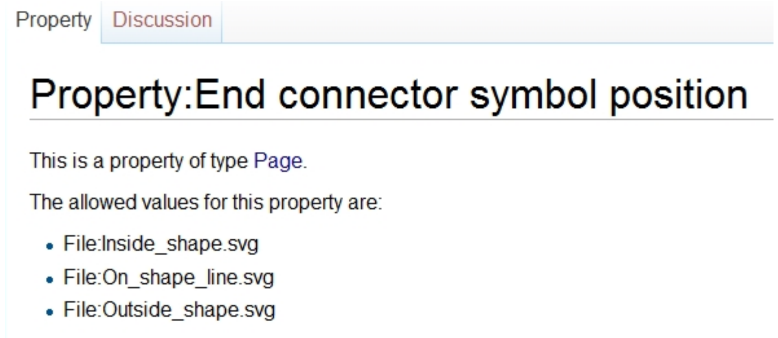


Figure 38 - Overview of page Property: End connector symbol position

In the manually created property we could have the type of property and limit the allowed values for that property when there is a need for it, which can be showed in Figure 38 and the code used on page "Property: End connector symbol position" is written below:

```
This is a property of type [[Has type::Page]].
The allowed values for this property are:
* [[Allows value::File:Inside_shape.svg]]
* [[Allows value::File:On_shape_line.svg]]
* [[Allows value::File:Outside_shape.svg]]
```

It permits for anyone to use 3 specific values for that property which are Inside_shape.svg, On_shape_line.svg and Outside_shape.svg. And any values are case sensitive, so the value has to be exactly what is written in the property page.

Some examples of properties types and allowed values used are displayed in Table 1.

Table 2 - Semantic properties examples

Property	Property Type	Allowed Values
[[Language::]]	[[Has type::String]]	
[[SVG code::]]	[[Has type::Page]]	
[Connector z-position]	[[Has type::Number]]	* [[Allows value::1]] * [[Allows value::2]] * [[Allows value::3]]
[Start connector symbol position name::]	[[Has type::String]]	* [[Allows value::Inside_shape]] * [[Allows value::On_shape_line]] * [[Allows value::Outside_shape]]

4.5 Templates and properties

Templates are used inside MediaWiki to add a structured template used to connect various properties and to use them inside Forms or simply directly in manual editing pages.

For the task of implementing templates to correctly structure the UEAOM into a dependable framework there were created a considerable number of templates and most of them is associated to a Category, as we can see in Figure 39, and we will explain how this templates are created it, further along this section .

Templates

The following templates exist in the wiki.

Showing below up to **23** results starting with **#1**.

View (previous 50 | next 50) (20 | 50 | 100 | 250 | 500)

1. CONNECTOR defines category: CONNECTOR
2. CONNECTOR KIND defines category: CONNECTOR KIND
3. Circle element defines category: Shape Component
4. DIAGRAM defines category: DIAGRAM
5. DIAGRAM KIND defines category: DIAGRAM KIND
6. Ellipse element defines category: Shape Component
7. LANGUAGE defines category: LANGUAGE
8. Line element defines category: Shape Component
9. MODEL KIND defines category: MODEL KIND
10. OA defines category: OA
11. OAE
12. OAK defines category: OAK
13. OAKEK
14. OAKRK defines category: OAKRK
15. OAR defines category: OAR
16. Path element defines category: Shape Component
17. Polygon element defines category: Shape Component
18. Polyline element defines category: Shape Component
19. Rect element defines category: Shape Component
20. SHAPE KIND defines category: SHAPE KIND
21. SHAPE KIND2
22. Shape
23. Text element defines category: Shape Component

Figure 39 - List of Templates created in MediaWiki

For each class in the UEAOM there should be a corresponding template or a property depending on the importance of the class for example 'Connector property value' is not a property but the value of the 'Connector property'. The naming of the template, initially was the complete name of the class, but some names would be larger than 3 words (e.g. organizational artifact kind relation kind) so it was decided to use the acronym (e.g. OAKRK) and since there were a few using the name organizational, those names are all being used in acronym instead of the full name.

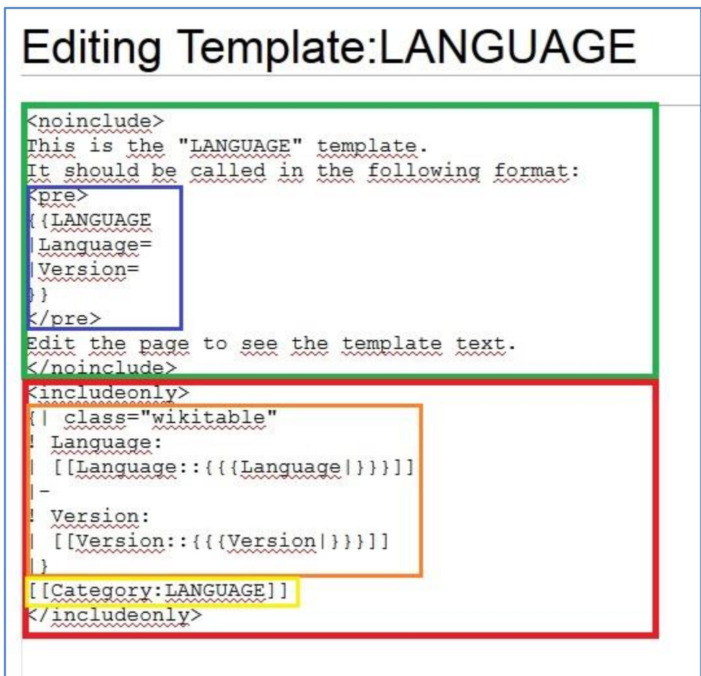


Figure 40 - Template: LANGUAGE page edit

Each template has its own specification and has a few areas we have to pay attention to, the area inside the tag <noinclude> (green area in Figure 40) is displayed in the page and inside the tag <Pre> (blue area in Figure 40) it displays the nomenclature and structure used to define that specific template but it doesn't necessarily mean that it is defined that way because it depends on how the next section is coded because if the template creation is done manually there a possibility of errors which can lead to misspelling of certain property in both areas, <Pre> and< includeonly > areas.

Inside the other tag called < includeonly > , which isn't displayed except when editing the page, has the code definition for the template, it associates the properties with the names used inside the template which can divergence, but in this case are the same and they receive the property type from that association(orange area in Figure 40), we can also can see which class of representation this template will have in the page, which is the class "wikitable", which is the presentation of template in the form of a table. Also a template can have a category associated with the template (yellow area in Figure 40), or even multiple, it's not considered mandatory but in this case we decide to associate a Category to each template, so it's easier to search and identify which template a page is using. Templates and their properties:

Most properties follow the nomenclature used in the UEAOM, follow a very specific formula, for example if we have this fact type between the classes '[diagram kind] is

part of [language]' then the resulting property will be 'is part of language'. Some examples of these properties are displayed in the Table 3.

While most of the properties follow that nomenclature, there are some properties that are created based on their class name and others dependent if in their fact type is included the suffix 'is specified' and an example of that is the "[svg code] specifies [connector kind]", from which additional properties are created, and another divergence is "[svg code] specifies [shape kind]" on which the SVG code is the SVG file and the connection between them is done in the shape kind template by the property "SVG CODE".

During the realization of this project some infix were altered, or because it didn't really correctly specify what it should, but the idea behind the resulting property maintained the same, except for the infix used for the resulting property

Table 3 - Template properties specification examples

Class 1	Class 2	Fact type	Resulting property
LANGUAGE			[Language::]
MODEL KIND			[Model Kind::]
MODEL KIND	LANGUAGE	[model kind] is part of [language]	[Is part of language::]
DIAGRAM KIND			[Diagram Kind::]
......	......	......	......
CONNECTOR KIND			[CONNECTOR KIND::]
SVG CODE	CONNECTOR KIND	[svg code] specifies [connector kind]	[[Start connector symbol position name::]] [[Start connector symbol name::]] [[Start connector symbol position::]] [[Mid connector symbol::]] [[Mid connector symbol name::]] [[End connector symbol::]] [[End connector symbol name::]] [[End connector symbol position::]] [[Start connection rule::]] [[End connection rule::]] [[Connection line width::]] [[Connection line color::]] [[Connection line dasharray::]] [[Connection line path::]] [[Connector z-position:: 3]]
SVG CODE	SHAPE KIND	[svg code] specifies [shape kind]	[[SVG CODE::]]
ORGANIZATIONAL ARTIFACT KIND	MODEL KIND	[organizational artifact kind] allowed in [model kind]	[allowed in model kind::]
......	......	......	......
ORGANIZATIONAL ARTIFACT KIND RELATION KIND	ORGANIZATIONAL ARTIFACT KIND	[organizational artifact kind relation kind] has as target of reference law 1 [organizational artifact kind]	[has as target of reference law 1 organizational artifact kind::]

ORGANIZATIONAL ARTIFACT KIND RELATION KIND	ORGANIZATIONAL ARTIFACT KIND	[organizational artifact kind relation kind] has as target of reference law 2 [organizational artifact kind]	[has as target of reference law 1 organizational artifact kind::]
PREFIX_1_2	ORGANIZATIONAL ARTIFACT KIND RELATION KIND	[prefix_1_2] of [organizational artifact kind relation kind]	[Prefix_1_2::]
UNICITY 1	ORGANIZATIONAL ARTIFACT KIND RELATION KIND	[unicity 1] of [organizational artifact kind relation kind]	[Unicity 1::]
......	......	......	......
SUFIX_2_1	ORGANIZATIONAL ARTIFACT KIND RELATION KIND	[sufix_2_1] of [organizational artifact kind relation kind]	[Sufix_2_1::]

For the creation of the properties, a spreadsheet was created in Google docs, from where everyone involved could look up the properties and see which specific rule was being implemented, and also there is another type of data included in that spreadsheet, but wasn't needed for this implementation. That spreadsheet specifies the backbone of the Model in an EO context.

4.5.1 Templates examples

Template:OAKRK

This is the "OAKRK" template. It should be called in the following format:

```
{{OAKRK
|Prefix_1_2=
|Unicity 1=
|Dependency 1=
|Is target of reference law 1=
|Infix_1_2=
|Is target of reference law 2=
|Sufix_1_2=
|Prefix_2_1=
|Unicity 2=
|Dependency 2=
|Infix_2_1=
|Sufix_2_1=
|Version=
}}
```

Figure 41 - Template Structure for Organizational artifact kind relation kind

Template:OAK

This is the "OAK" template. It should be called in the following format:

```
{{OAK
|Organizational artifact kind=
|Is allowed in model kind=
|Version=
}}
```

Figure 42 - Template Structure for Organizational artifact kind

Template:OAKEK

This is the "OAKEK" template. It should be called in the following format:

```
{{OAKEK
|Has organizational artifact kind element kind=
}}
```

Figure 43 - Template Structure for Organizational artifact kind element kind

Template:DIAGRAM KIND

This is the "DIAGRAM KIND" template. It should be called in the following format:

```
{{DIAGRAM KIND
|Diagram kind=
|Represents model kind=
|Is part of language=
}}
```

Figure 44 - Template Structure for Diagram kind

Template:CONNECTOR

This is the "CONNECTOR" template. It should be called in the following format:

```
{{CONNECTOR
|Represents organizational artifact relation=
|Is part of diagram=
|Is instance of connector kind=
|Version=
|Connector id=
|Connector moves=
|Connector beggining shape=
|Connector beggining shape point=
|Connector mid shape=
|Connector mid shape point=
|Connector ending shape=
|Connector ending shape point=
}}
```

Figure 45 - Template structure for Connector

The template for the Connector (Figure 45) had multiples changes made, during this project because of changes made to the diagram editor, which isn't part of this project but directly connected, especially on how to save the information about the shapes used and how the connector line would be implemented and saved.
It's possible to see an aggregation of all the templates created, with their category and with their associated class, in Figure 46. That connection between their class and templates is easily identified.

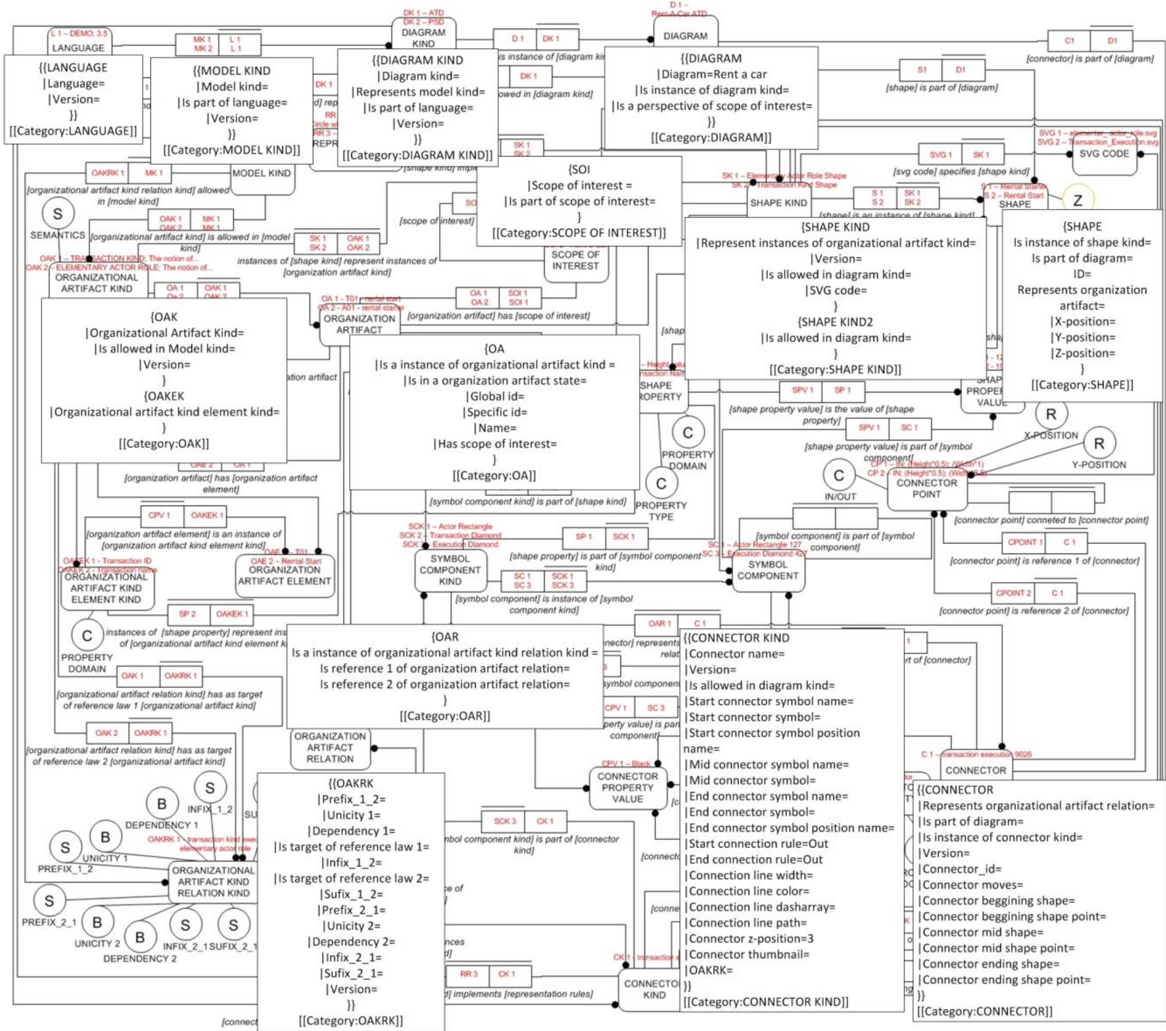


Figure 46 - UEAOM with the template definition for each class

4.6 Templates and categories:

Each template corresponds to a Category and the name of that Category corresponds to the names used for the template, but in some cases like the templates for the components/elements the Category will be generic for all of them, with the Category

name: COMPONENT KIND, and for each component/element, there will be an extra Category based on the name of the element template. There are some templates that are used with others and because of that they don't use a category. The templates used and category of each and displayed in Table 4.

Table 4 - Templates and category for each

Template	Category
CONNECTOR	CONNECTOR
CONNECTOR KIND	CONNECTOR KIND
Circle element	Shape Component Circle Component
DIAGRAM	DIAGRAM
DIAGRAM KIND	DIAGRAM KIND
Ellipse element	Ellipse Component Shape Component
LANGUAGE	LANGUAGE
Line element	Line Component Shape Component
MODEL KIND	MODEL KIND
OA	OA
OAE	Used in conjunction with OAK so no category associated with it
OAK	OAK
OAKEK	Used in conjunction with OAKRK so no category associated with it
OAKRK	OAKRK
OAR	OAR
Path element	Path Component Shape Component
Polygon element	Polygon Component Shape Component
Polyline element	Polyline Component Shape Component
Rect element	Rect Component Shape Component
SHAPE KIND	SHAPE KIND
SHAPE KIND2	Used in conjunction with SHAPE KIND so no category associated with it
Text element	Text Component Shape Component

The reason why we use categories is because they facilitate for queries and give information about which templates is being used, in each pages, and since some templates properties are very similar to each other, like for example, the ellipse and the circle, that only diverge in 3 very similar properties, 'r' property for ellipse and 'rx' and 'ry' properties for circle, which we can see in Table 5, and it`s the only way to have the certainty of which template a page is using.

4.7 Template for elements

The SVG offers a multitude of functionalities but for this project, the focus was in the Graphic elements that contain all the shapes elements needed for the implementation of the shapes and another element that isn`t considered a shape elements called <text>, which is considered a Text content element (there are a few other). Some graphical elements weren't used since they didn't have any utility for what we were creating, like <image>, <use> (Structural element) and <textPath> (Text content element). While text path element is interesting, there isn't any real use in defining text elements that way, so the <text> was used instead.

Below we can see the basic shapes, the shape elements and graphic elements that exist in the SVG code [4].

Basic shapes:

- <circle>, <ellipse>, <line>, <polygon>, <polyline>, <rect>.

Shape elements:

- <circle>, <ellipse>, <line>, <path>, <polygon>, <polyline>, <rect> .

Graphics elements:

- <circle>, <ellipse>, <image>, <line>, <path>, <polygon>, <polyline>, <rect>, <text>, <use>, <textPath> .

The definition of the elements attributes that will be permitted to be used by the templates for each of the shapes elements and the text element are displayed in the Table 5. The Table 5 provides the basis for the component templates created.

Table 5 - Elements and their attributes

Attribute \ Element	<circle>	<ellipse>	<line>	<path>	<polygon>	<polyline>	<rect>	<text>
id	X	X	X	X	X	X	X	X
x							X	X
y							X	X
font-size								X
font-style								X
font-weigh								X
text-anchor								X
font-family								X
stroke-dasharray	X	X	X	X	X	X	X	X
stroke-width	X	X	X	X	X	X	X	X

stroke	X	X	X	X	X	X	X	X
fill	X	X						X
x1			X					
y1			X					
x2			X					
y2			X					
cx	X	X						
cy	X	X						
r	X							
rx		X					X	
ry		X					X	
width							X	
height							X	
D				X				
points					X	X		
Possible added attribute to element								
Rescale	X	X	X	X	X	X	X	X
Move	X	X	X	X	X	X	X	X
~~Z position~~	~~X~~	~~X~~	~~X~~	~~X~~	~~X~~	~~X~~	~~X~~	~~X~~
Position								X
Added property for text template								
~~Text~~								~~X~~

For the text templates we added the property called “Text” for the value of the text Element but after some discussions that ideas was left behind and the value is obtained by grabbing the value out of the OA- Organizational Artifact page.

Each component is associated to a shape (by the property “Is part of shape”) and each shape is associated to a shape kind (by the property “Is instance of shape kind”) and that association is done by the name of the page of each.

The initial idea about components was alter altered in order for both the symbols from connector kind and shape kind to use the same template since creating another 8 templates just to accommodate for the components from connector kind and just alter the property name “Is part of shape” to “Is part of Connector Kind” was a bit excessive, and in order to simplify that idea, those properties where merged into one, more embracing property, which was given a simplified name from both of them “Is part of”, while it was not quite accepted by all, this implementation is currently being used.

The attributes x-position, y-position, and z-position are added to the template: SHAPE (Figure 47) in order to be used in the translate function for positioning of the shapes in the diagram editor, while the basic positioning for each element will be done by their respective positioning attributes and can only be altered, if one of the elements moves separately from the rest of the elements inside a shape.

Template:SHAPE

This is the "SHAPE" template. It should be called in the following format:

```
{{SHAPE
|Is instance of shape kind=
|Is part of diagram=
|ID=
|Represents organization artifact=
|X-position=
|Y-position=
|Z-position=
}}
```

Figure 47 - Template: SHAPE

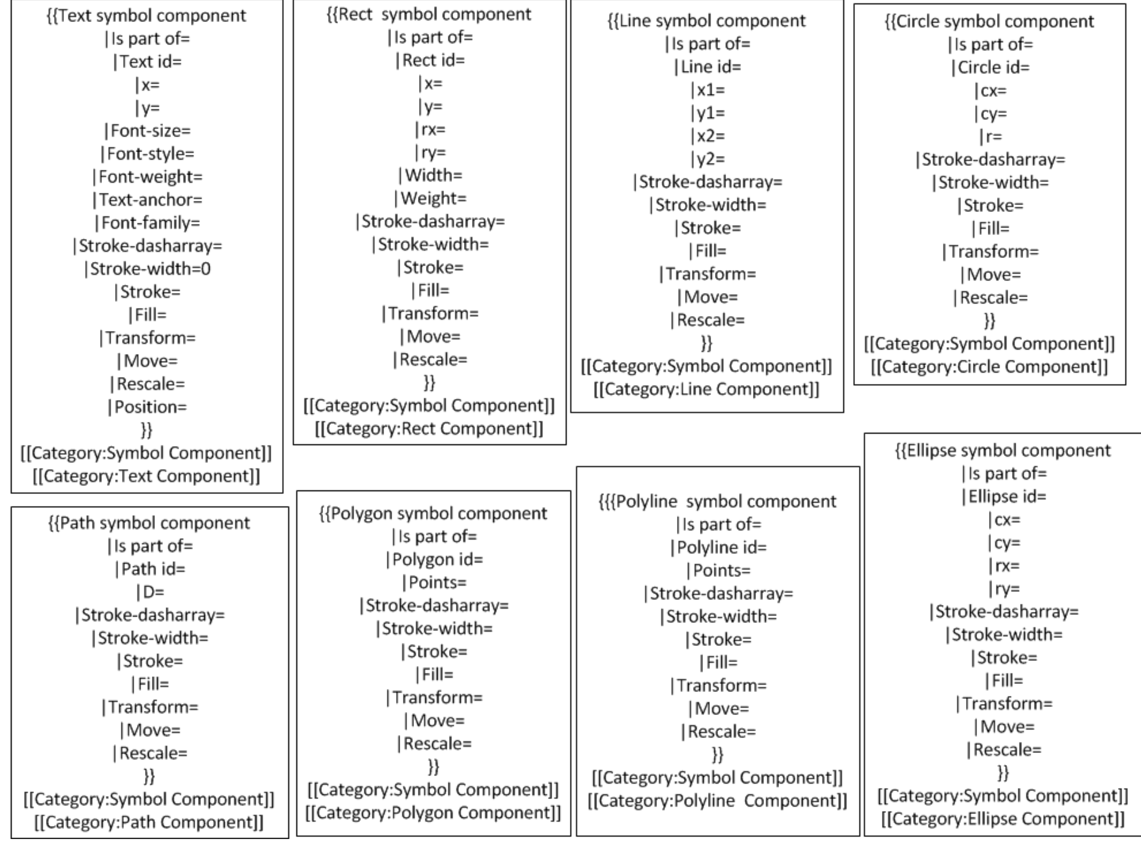


Figure 48 – Component Templates with their associated Categories

While there are a few other attributes for each SVG element, those attribute aren't really needed since they aren't used and there no ideas of them being used in the diagram editor, so there's no need to add them to the template but they can be added later if needed.

An example for the definition of an element/component template is the text element which will use the attributes selected on Table 5 and the name of the template will be the name of the element in combination with "symbol component", example "Text symbol component" resulting in the template in Figure 48. Each of the elements has the property "Is part of shape" and a id which is a combination of the element name with id and the rest of the properties are defined in the Table 5.

4.8 Forms and templates

Forms are used inside MediaWiki to add, edit and query data. It's supplied by the semantic forms extension. Each Form can be used to fill various properties in various templates at the same time. It adds a visual interface for filling templates without doing it directly in the template, which is basically manually filling the template in the page, which means knowing each properties and the name of the template we wish to fill, which in all cases isn't advisable since a little mistake in naming the template or property can result in incorrect filling.

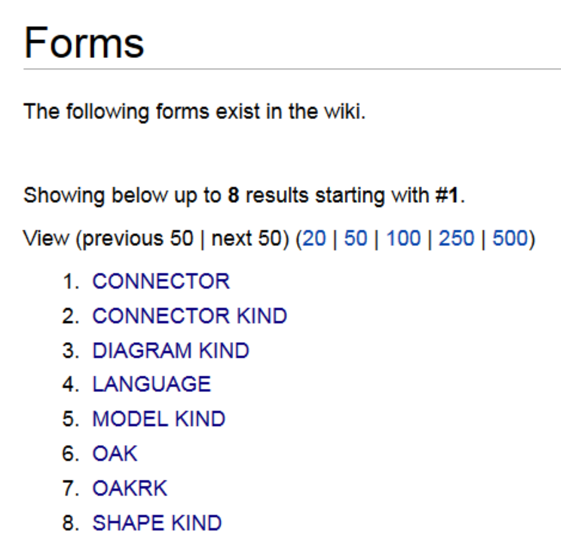


Figure 49 - List of Forms that exist in MediaWiki

While the forms in Figure 49 are basically the essential forms for this project, there was a need to create forms for every single template created, because the diagram editor

project was using a functionality that permitted to edit property values and create pages using the properties and the templates names, which wasn't used in this project.

For each Form there's at least one template related, and in the, is displayed the connection between each class and their corresponding templates. It's also possible to see an example of the Form OAK which is a combination of two the templates OAK and OAKEK: That combination is exemplified in the Section: Form: OAK-Organizational Artifact Kind.

4.8.1 Form: OAKRK

Create OAKRK

Prefix_1_2:

Unicity 1:

Dependency 1:

Is target of reference law 1:

Infix_1_2:

Is target of reference law 2:

Sufix_1_2:

Prefix_2_1:

Unicity 2:

Dependency 2:

Infix_2_1:

Sufix_2_1:

Description:

Summary:

This is a minor edit Watch this page

Save this page Show preview Show changes Cancel

Figure 50 - Form: OAKRK page creator

For the creation of the OAKRK form there was a need to create the template called Organizational artifact relation kind (Figure 5), and after that add some input alterations for some field like 'Is target of reference law 1' on which we need to use the parser:

values from property and select values from all the values of that certain property that are used in the entire MediaWiki page, for creating a combo box with those values, instead of manually writing the values. Code used below:

```
{{{for template|OAKRK}}}
{| class="formtable"
! Prefix_1_2:
| {{{field|Prefix_1_2}}}
|-
! Unicity 1:
| {{{field|Unicity 1}}}
|-
! Dependency 1:
| {{{field|Dependency 1}}}
|-
! Is target of reference law 1:
| {{{field|Is target of reference law 1|input type=combobox|values
from property=Shape }}}
|-
! Infix_1_2:
| {{{field|Infix_1_2}}}
|-
! Is target of reference law 2:
| {{{field|Is target of reference law 2|input type=combobox|values
from property=Shape }}}
|-
! Sufix_1_2:
| {{{field|Sufix_1_2}}}
|-
! Prefix_2_1:
| {{{field|Prefix_2_1}}}
|-
! Unicity 2:
| {{{field|Unicity 2}}}
|-
! Dependency 2:
| {{{field|Dependency 2}}}
|-
! Infix_2_1:
| {{{field|Infix_2_1}}}
|-
! Sufix_2_1:
| {{{field|Sufix_2_1}}}
|}
{{{end template}}}
```

When normally using a Form to create a page, the page name must be filled, and then the page is created, and only after that, the form appear and it is possible to fill the properties values and save the filled form, which isn't something we wanted, so we had to diverge from that using the following code;

```
{{#formlink:form=OAKRK|link text=Create OAKRK Page |link type=button}}
```

This uses a Parser function hooks called "formlink" to do so, it link to a temporary form, with no page name defined or created, meaning that the page name will be created after the form is completed but to do so we have to add a new tag to the form definition,

called ‘info’ tag with the parameters “page name” in order to create name the name for the pages, and there we can add multiples values from the field by writing something like `<OAKRK[Prefix_1_2]>`, which give us the value from the field Prefix_1_2 and we can also add static strings between them, creating something like the following code:

```
{{{info|pagename=<OAKRK[Prefix_1_2]>string value<OAKRK[Infix_1_2]> }}}
```

While it will help for most of the page creations, in this case we had create a Parser function called 'Specialpagename' , which PHP code will be explained and available at the MediaWiki extension section, and in order to use that parser we added the following code:

```
{{{info|page      name={{#Specialpagename:<OAKRK[Prefix_1_2]>|<OAKRK[Is
target  of  reference  law  1]>|<OAKRK[Infix_1_2]>|<OAKRK[Is  target  of
reference law 2]>|<OAKRK[Sufix_1_2]>}} }}}
```

In order to use that parser we had to open brackets “{{“after the page name tag, use the parser name with “#” at the start and “:” at the end of the parser name (“#Specialpagename:”) and then we added the five fields we wanted to be used in that parser, separated between “|”, and end with closing the initial brackets with “}}”.

So when clicking on the save button, the values from those specific fields will be used to create the page name using a formula created inside the parser function, which adds ‘.’ between each selected field, but only the populated fields will be used for that formula.

The creation of this Specialpagename parser function Extension, which has the name “Special pagename OAKRK”, was needed because semantic MediaWiki didn’t allow for that sort of formula to be used, and it was even advised, by one of the semantic MediaWiki forum administrator, in creating that parser instead of trying to figure out a complex way of creating that formula by using the pre-existing parsers.

4.8.2 Form: OAK-Organizational Artifact Kind

For the creation of the OAK (Organizational artifact kind) form, like the previous forms, a template was created, but in this case, two were created, one template called OAK (Figure 42), for the basic values and another template called OAKEK (Figure 43), for the multiple values of the Organizational artifact kind element kind which both can be observed in the template section. This permits for multiple values of the same property in the same page, and for that the only addition needed was to had a new template definition into the form, below the initial template definition, and inside that,

add a new Tag called “multiple” and also add a new button to permit for that, in this case:”add button text=Add Values for OAKEK”. The specific code can be is displayed below:

```
{{{for template|OAKEK|multiple|label=Has organizational artifact kind
element kind|add button text=Add Values for OAKEK}}}
'''Organizational artifact kind element kind:'''
{{{field|Organizational artifact kind element kind|mandatory}}}
{{{end template}}}
```

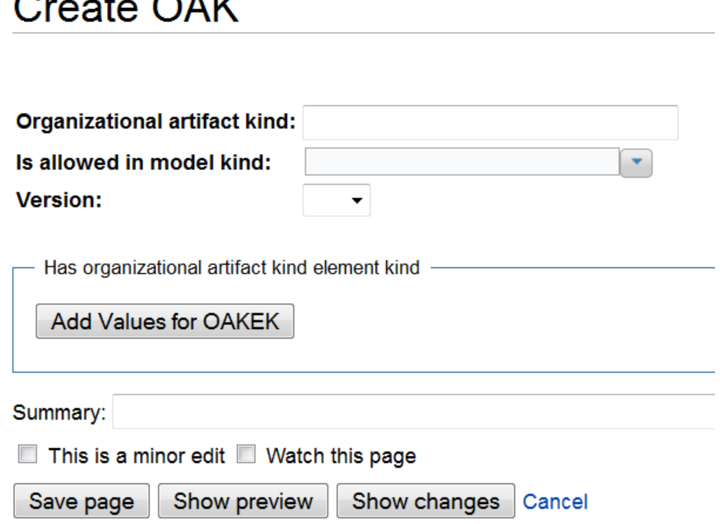


Figure 51 - Form: OAK page creator

4.8.3 How to use the Forms and edit pages

Normally, to use forms, someone needs to click on the Special pages, in the toolbox, and there go to the list of pages and click on the option Form (Figure 52) which will result in the view of the Figure 49 example, and select which form we want to use. Another way is to create a link in a certain page, to the form creation page and create the pages from there (Figure 53).

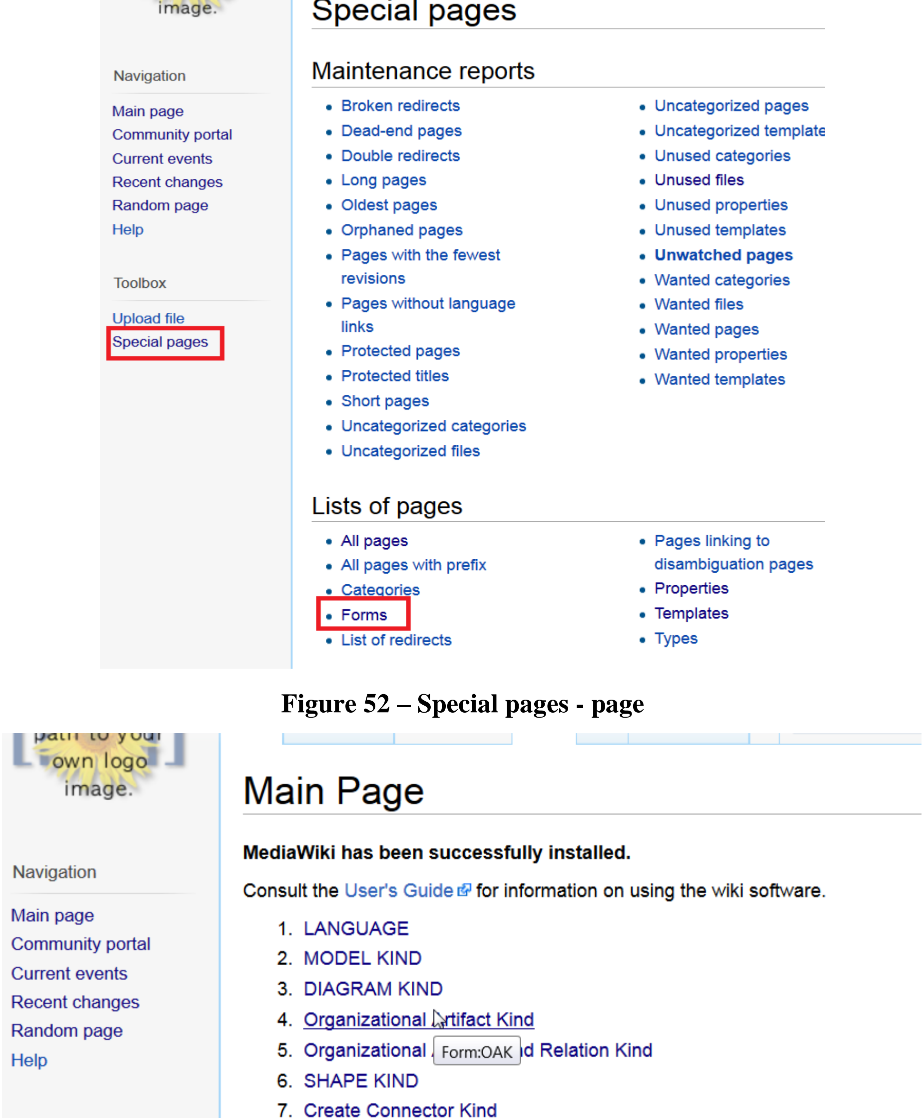


Figure 52 – Special pages - page

Figure 53 - Form link page example

There's a possibility to edit pages using the form used for that, but that can lead to issues with the page names since the page name is created based on the field values used in the form, if some of those values are altered it means that it will have an inconsistent page name that will lead to an inconsistent database page system, but for some cases it's more than reasonable to do so. For example, when creating a diagram kind, a person as to give the name of the diagram kind and select which model kind it's associated to and which language, and those types of selections can be altered since they aren't pertinent to the name of that page, so if there a need to do so, it's possible but only some field (e. g. Figure 54). For other forms, where the field name is incorrect and it is used in the

page name, it's necessarily to delete that page and create it using the form again so the correct page name is created.

Edit DIAGRAM KIND: OCD

Diagram kind: OCD
Represents model kind: Construction Model
Is part of language: DEMO

Figure 54 - Form Edit for OCD page

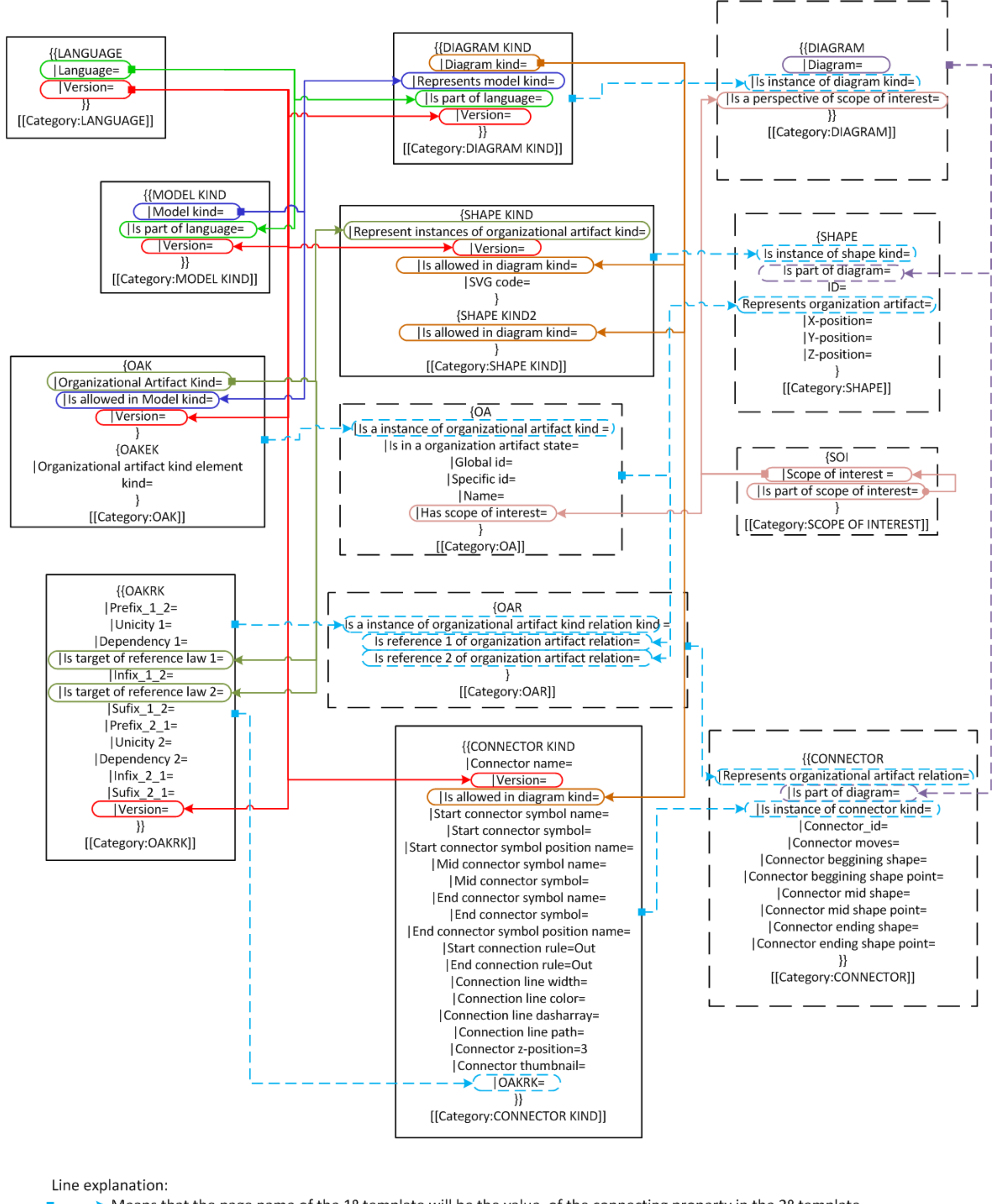
This form and associated template were altered, in the last phase of the project, by adding the version to its template and page name, but the same basics apply but with the diagram kind name and version that can't be altered and if any of those want to be altered a new page must be created.

4.9 Page names, templates and properties connections

In order to understand the previous implementation steps, a person needs to understand how the connections between pages are made, by that we mean the global picture of the connection between pages and properties inside templates.

In Figure 55, it`s possible to see all the connections between pages, more specifically between properties and also between page names and properties. For example, if there is a value in the property «Language», in the «LANGUAGE» template, called "DEMO" then that property value can be used in the property "Is_part_of_language", used in both DIAGRAM_KIND and MODEL_KIND templates. Using the initial Figure 29 in conjunction with Figure 55 it`s easier to understand why there are some pages that need to be created first than others. For a practical overview of the system, an example was created in Figure 57 (page names created in Meta-Editor) and Figure 58 (page names created in Diagram Editor), which is basically a implementation of what we see in Figure 55 , using the EO example case called "Rent a car" , Figure 56, but only the selected part of that example.

In the Diagram Editor section, the properties associated to the pages created in the Meta Editor section will receive the names of the pages and not the value of the primary property like for example, the property «is instance of diagram kind» will receive the page name «ATD V3.5» instead of the value of the property «Diagram kind» which is «ATD». And the same goes for the connections between pages created in the Diagram Editor.



Line explanation:

Means that the page name of the 1º template will be the value, of the connecting property in the 2º template.

Means that the value of that property, of the 1º template, will used in another property in the 2º template and needs to be created first in order to do so. (colored to differentiate each specific property)

Templates used by the Meta-editor

Templates used by the Diagram Editor

Means that property connects to another page of the same template kind

Figure 55 - Pages names, templates and properties connections

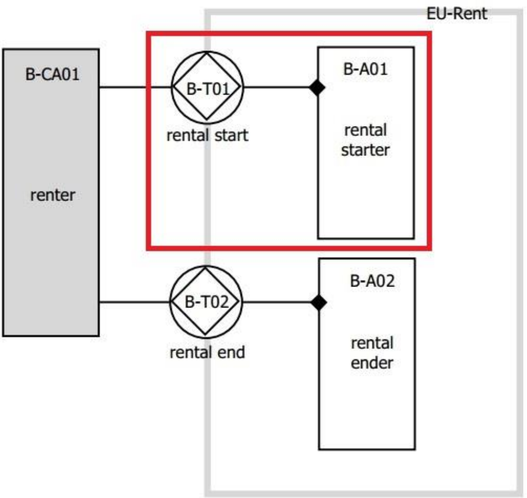


Figure 56 - Rent a car, ATD example with the selected part used for example

The page names created in the Meta-Editor can be identified in Figure 55, by the rectangular shape with black and uninterrupted line, and corresponding page names examples can be visualized in Figure 57. For this page names, all the names are based in one simply rules that is to use the value of the primary property and add capital letter «V» (standing for Version) followed by the value of the property «Version», except for the Organizational artifact kind relation kind, or to simplify, the OAKRK category which uses values of multiple properties to create it's page name but still adds the version value to the last part of the name. For example in Figure 57, it's possible to view an example of that category, the page name «TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE V3.5» uses the value of property «Is target of reference law 1» with the value of the property «Infix_1_2» and the value of the property «Is target of reference law 2» in conjunction capital letter «V» (standing for Version) followed by the value of the version «3.5», and this kind of page name can have additional values of other properties, which is explained in the section 4.8.1 . Another example for the page names is «DEMO V3.5» («LANGUAGE» Category) which uses the value of primary property in conjunction with the defined version protocol used for every page name, being that for the «Shape kind» Category it`s added "Shape kind" after the versions section, example «TRANSACTION KIND V3.5 Shape kind».

The page names created in the Diagram Editor can be identified in Figure 55, by the rectangular shape with black and interrupted line, , and corresponding page names examples can be visualized in Figure 58. The page names are simpler and use the

«Category» name in conjunction with a numeric value, but that combination needs to create a unique page name, in MediaWiki.

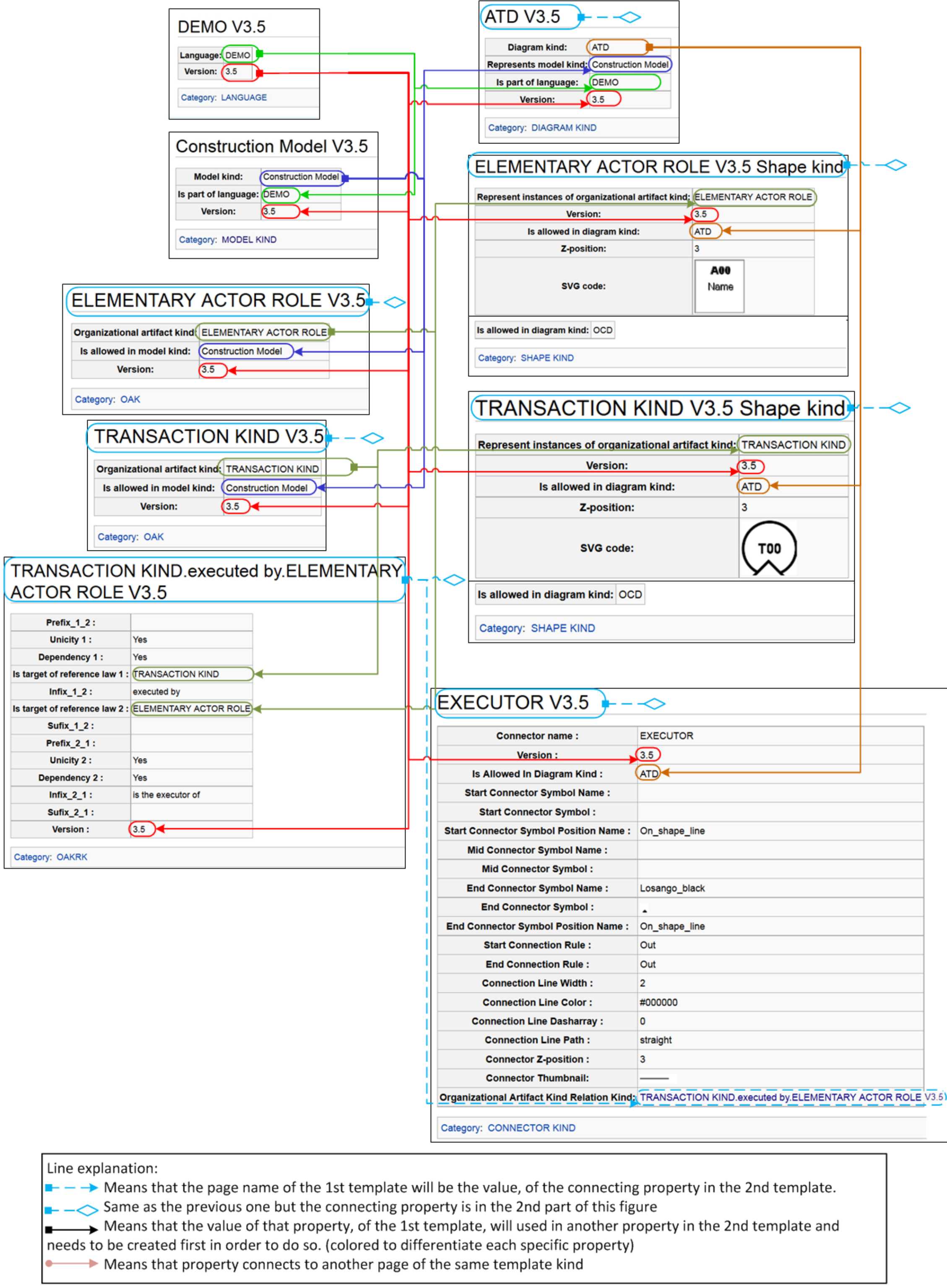


Figure 57 – Pages created for Rent a car example and interconnection between pages-1st part

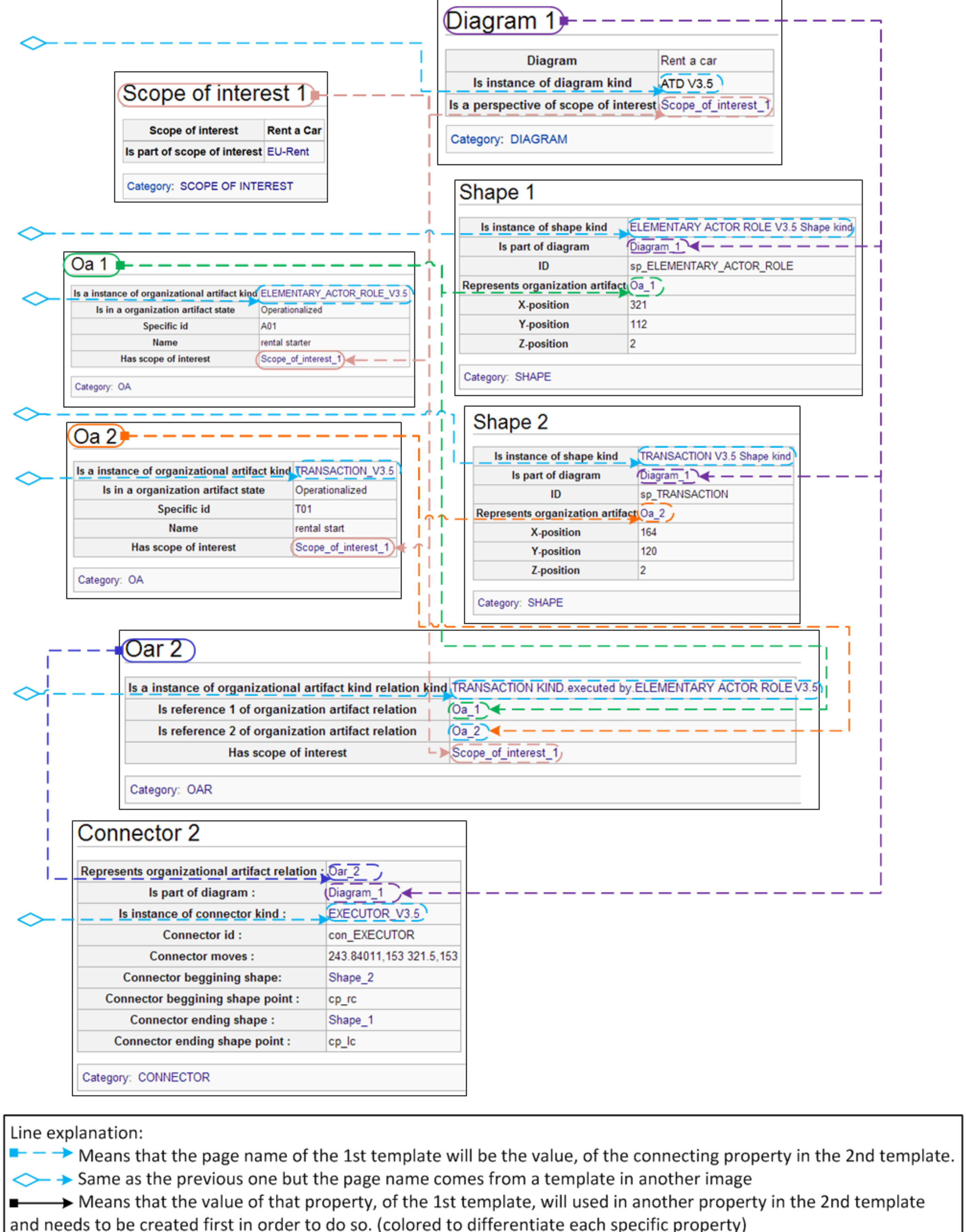


Figure 58 - Pages created for Rent a car example and interconnection between pages- 2nd part

The initial ideas for page names for the Diagram Editor part, discussed in section 3.6.3 Page names and semantic properties, were a bit different from the current ones used, because the initial ones didn`t work well with the constant changes done to the pages, especially when a value of a property, is imbued in the page name.

In the Diagram Editor part, in the pages created, there is no specific property just for the version, since it`s possible to know which version it is by the property values that

receive the names of the pages created in the Meta-Editor, since those page names have the version integrated into them and consequently in the version property in that page. But that doesn`t mean that in the diagram editor we can`t specify the version, it just means that the version is integrated into the properties values that are associated to meta-editor created pages, and when choosing a determined version in the diagram editor, only the pages with that specific version will be used to populate the interface options.

4.10 Creating MediaWiki extensions

For many of the implementation, inside MediaWiki, there was a need to create parser hooks, and for that, it's needed a MediaWiki extension.

For creating MediaWiki extension, there were two choices: create based on existing extensions or simply create them from scratch, and what was learned was that it's best to find a similar extension, alter the basic data, which all extensions have and then add the specific code to the that code. And for the extension that were it isn't possible to find similar, create them from scratch.

When creating an extension, first of all create a PHP file then add the basic information, which we can see an example of the Special page name OAKRK parser, with some explanation of how it is organized:

```
$wgExtensionCredits['parserhook'][] = array(
   'path' => __FILE__,
   'name' => 'Special pagename OAKRK',
   'description' => 'Special pagename parser function extension',
   'descriptionmsg' => 'Special pagename parser function Extension for creating OAKRK page name',
   'version' => 1,
    'author' => 'Vitor Nóbrega',
   'url' => 'https://www.mediawiki.org/wiki/Manual:Parser_functions',
);
```

Then specify the function that will initialize the parser function:

```
$wgHooks['ParserFirstCallInit'][] = 'SpecialpagenameExtensionSetupParserFunction';
```

Allow translation of the parser function name in case there's more than one language which wasn't really used but was created just in case:

```
$wgExtensionMessagesFiles['SpecialpagenameExtension'] = dirname(__FILE__ ) . '/SpecialpagenameExtension.i18n.php';
```

Tell MediaWiki that the parser function exists and give it a magic word for it to recognize inside MediaWiki:

```
function SpecialpagenameExtensionSetupParserFunction( &$parser ) {
```

```
  $parser->setFunctionHook('Specialpagename',
'SpecialpagenameExtensionRenderParserFunction' );
  return true;
}
```

And to end, create the output of the parser function.

```
function SpecialpagenameExtensionRenderParserFunction( $parser, $param1= '',$param2=
'',$param3= '',$param4= '',$param5= '') {
….
return $parser->insertStripItem( $output, $parser->mStripState ); }
```

The other extension created, **Connectorweb** extension, which is based on another extension called “**Anysite”**, was a bit easier since it required just a few alterations. This extension adds the same basic information but the next steps are a bit easier.

Create a magic word for it to recognize inside MediaWiki and associate it to a function.

```
function connector() {
      global $wgParser;
      $wgParser->setHook('con_web', 'connectorweb');
}
```

In the end create the basic function that receives some arguments, which aren’t directly define at start, which gives a bit more freedom and then return a output, which in this case was the html page for Connector-Edit.

```
function connectorweb($input, $argv) {
          if (isset($argv['mywidth'])) {
            $width = $argv['mywidth'];
      } else {
            $width = 700;
      }
      if (isset($argv['myheight'])) {
            $height = $argv['myheight'];
      } else {
            $height = 800;
      }
                  $link="http://localhost/connector-edit/maininterface_7.html";
      $output= '<iframe name="anyweb" src="'.htmlspecialchars($link)
            .'" width="'.$width.'" height="'.$height.'" frameborder="0">'.'</iframe>';
      return $output;
}
```

4.11 SVG-edit

In the search for the best solution to creation and editing of SVG based symbols it was decided to use SVG-Edit[11], in special the extension used by media wiki[12] which provided us with the basic SVG editing options and permitted us to change the code to

fit your needs, but we also needed to adapt the code to serve us with some additional functionalities.

In order for SVG-edit to permit the addition of new functionalities there was a need to study the way the code was implemented and try to figure out the best way to change it without completely changing it, and since the SVG-edit is open source, the way it is implemented permits to do it and if we follow the way they implement their code, it's easy after some careful research and code reading to add new code without corrupting the existent code or creating an extension, but in this case it was needed to change the code because there isn't any implementation of the attributes we wanted to use in the SVG code.

4.11.1 SVG-Edit – Connector point extension

In the last versions of SVG-edit, there were implemented extensions to permit people to create and add new content, without the need to know the complete code, which can be used to create the connector point part of the symbols used.

Issues with that type of connector point is that there is nothing in SVG to create it using SVG code, at this moment, there is already a draft for two new elements, connector and point, where they define the mechanism to visually and logically connect two elements (http://dev.w3.org/SVG/modules/connector/SVGConnector.html), which both would help immensely if they were implemented. But since they aren't, a small extension was made for the creation of a connector point, which is basically, use a foreignObject element and inside that element, is created an element called connector point, which gives the x and y measures that is needed to create a connector point. Besides the new element, we use a symbol that SVG doesn't read, but we can see in the SVG-Edit so when it's saved, it's contained inside the SVG image but not visible outside SVG-Edit. While it's not ideal, it can be used as a workaround while that type of element isn't implemented. While that type of element would help a lot it would mean for a lot of changes in both the basic ideas behind our ideas, but for the best, since everything would be already in the SVG code, we would only need to use it, especially for the implementation of the diagram graphical editor.

While this option can still be helpful, it's not the best solution, the follow up project should try to improve this solution specially how it's implemented, specified or even start from scratch if a better solution is found.

4.11.2 SVG-Edit MediaWiki toolbar

For the creation of any shape kind SVG we used the SVG Edit extension of MediaWiki, with some changes, which is accessed in the edit page toolbar section, via a button called SVG drawing (Figure 61).The use of this toolbar was the easiest solution to create the SVG while inside the page we wanted to associate to the SVG and basically what it does, is open the SVG-Edit and we create the shape kind with the SVG elements we want, with some attention to the new attributes created, especially for this project and follow some basic instruction on how to create the SVG.

While Semantic MediaWiki is a pretty good platform it could need some improvement on how it works with native SVG because while saving the SVG file, the page created for that image doesn’t show the SVG but a PNG version of that file, and it sometimes doesn’t convert correctly, and especially when the attribute transform is used, a lot of time was lost in order to fix this problem, many solutions been tried but to no avail. This is one small issue since the SVG code is correctly saved, and the file used for the Diagram Editor Project is the SVG version and not the PNG one, which initially was thought to be used for the shape kind buttons.

4.11.3 SVG-Edit MediaWiki toolbar use procedure

In order to use this toolbar we have to follow a procedure, whose steps are described in this section.

The Initially step in this procedure is to go to the shape kind Form page (Figure 59) and fill all the values needed to create the page and since there are some chances that there might be more than one diagram kind associated with that shape kind, an additional template was added in order to permit adding extra values, since the forms don’t permit to add multiple values inside a template, for the same property. Which is something that the people behind Semantic Forms should look at it because it’s a pretty usual functionality, but the best solution for that issue was achieved since it’s the only way to make it work, And in terms of semantic search it works just fine because it searches for pages and properties and not specific properties inside specific templates.

For each new value, a new template will be create in the page, but the primary value should still be in the shape kind template, so if any person decides to delete that value they shouldn’t leave it blank, they should exchange it for another value, but this issue is just in terms of visual, and not functional since it won’t affect any kind of semantic search.

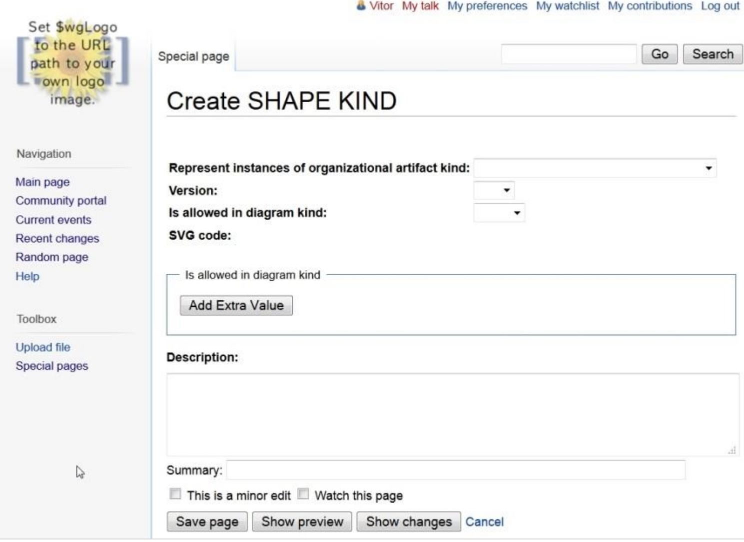


Figure 59 - Form: Shape kind

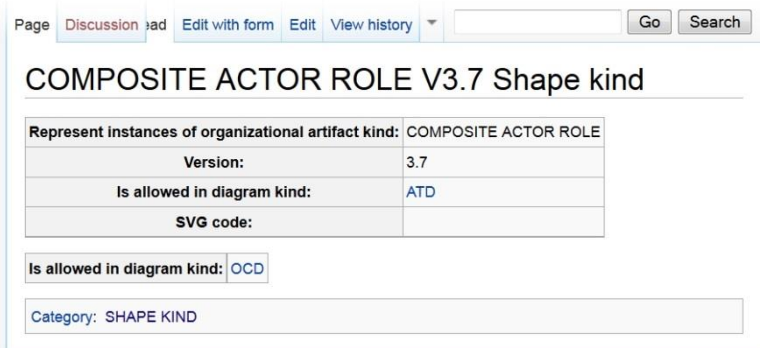


Figure 60 - Shape kind page example

After the creation of the page, Figure 60, a person needs to click on the edit button, and in the edit page we have to select the two "}" from shape kind template, Figure 61, which close the template, then click on the SVG-Edit button, in order to open the SVG-Edit application (Figure 62) and then create the SVG associated with that shape kind, with all the attribute values well defined using the specification needed and necessary for the correct implementation of a shape kind (4.11.5-SVG-Edit procedures for creating shape kind).

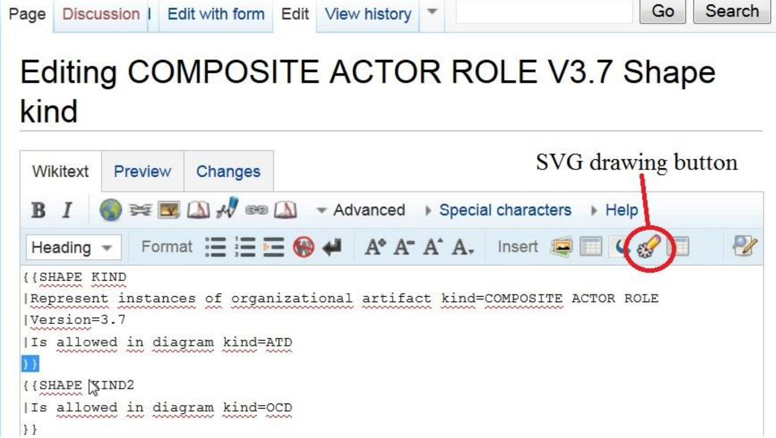


Figure 61 - Shape kind edit page example

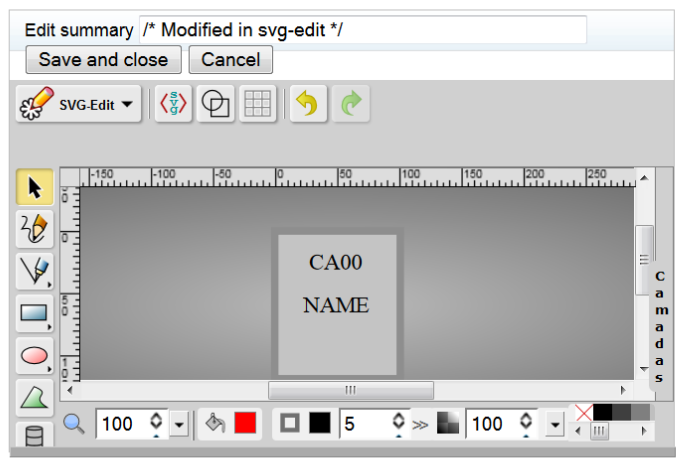
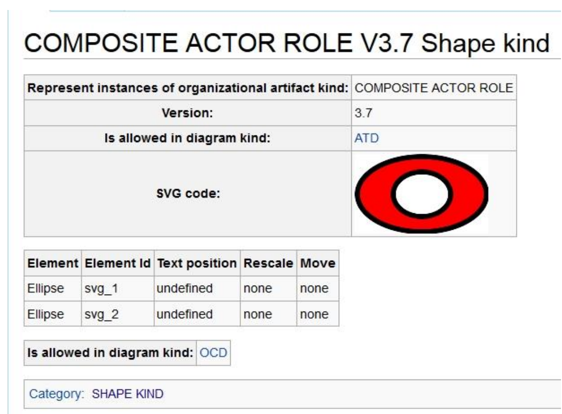


Figure 62 - Shape kind page with SVG-edit application open

Then a person needs to click on the save button (Figure 62), and in the background, PHP code creates components pages associated with shape kind and the JavaScript code populates the page, specially the property SVG_code (Figure 63), which didn't exist previously and a table that displays the special attribute and their values, which basically gives us some information about the SVG elements used in the shape kind and also gives a confirmation that the PHP code was activated for each of those elements and implemented. These elements are considered components in this project because of the UEAOM implementation.

In the end, the user needs to click on the save button to save the page, resulting in the final page, example Figure 64.

Figure 63 - Shape kind edit page – example code created


COMPOSITE ACTOR ROLE V3.7 Shape kind

Represent instances of organizational artifact kind:	COMPOSITE ACTOR ROLE
Version:	3.7
Is allowed in diagram kind:	ATD
SVG code:	

Element	Element Id	Text position	Rescale	Move
Ellipse	svg_1	undefined	none	none
Ellipse	svg_2	undefined	none	none

Is allowed in diagram kind:	OCD

Category: SHAPE KIND

Figure 64 - Shape kind page end result example

- (diff | hist) . . COMPOSITE ACTOR ROLE V3.7 Shape kind; 15:52 . . (+246) . . Vitor (Talk | contribs | block) [rollback]
- (diff | hist) . . **N** COMPOSITE ACTOR ROLE V3.7 Shape kind component ellipse 1; 15:52 . . (+249) . . Vitor (Talk | contribs | block) *(COMPOSITE_ACTOR_ROLE_V3.7_Shape_kind_component_ellipse_1)*
- (diff | hist) . . **N** COMPOSITE ACTOR ROLE V3.7 Shape kind component ellipse 0; 15:52 . . (+249) . . Vitor (Talk | contribs | block) *(COMPOSITE_ACTOR_ROLE_V3.7_Shape_kind_component_ellipse_0)*
- (Upload log); 15:52 . . Vitor (Talk | contribs | block) uploaded "File:COMPOSITE ACTOR ROLE V3.7 Shape kind.svg" *(Modified in svg-edit:)*
- (diff | hist) . . **N** COMPOSITE ACTOR ROLE V3.7 Shape kind; 15:46 . . (+183) . . Vitor (Talk | contribs | block) *(Created page with "{{SHAPE KIND |Represent instances of organizational artifact kind=COMPOSITE ACTOR ROLE |Version=3.7 |Is allowed in diagram kind=ATD }} {{SHAPE KIND2 |Is allowed in diagram kin...")*

Figure 65- Page creation information from Recent changes page

In the Figure 65, there an example of which pages are created during all this interactions, specially the part of the component pages created, which is done in the PHP page:' templatecreate.php'.This PHP code is used to create pages for the components in the shape kind (e.g ellipse component) and there are 8 types of components which are discussed in the Section about templates for elements.

4.11.4 SVG-Edit attributes

There are a few attributes that didn’t exist and where created and added in the SVG code in order for some specifications that will be used for the diagram editor. We can see an example in Figure 66, where we can view a ‘rect’ element and ‘text’ element with all the new attributes highlighted. The specification of this new attributes will be discussed in the next sections.

```
<svg width="281" height="353" xmlns="http://www.w3.org/2000/svg">
 <!-- Created with SVG-edit - http://svg-edit.googlecode.com/ -->

 <g>
  <title>Layer 1</title>
  <rect id="Boundary" height="346.99999" width="275.00001" y="2" x="2"
move="none" rescale="All" stroke-linecap="butt" stroke-linejoin="miter" stroke-
dasharray="none" stroke-width="6" stroke="#8e8e8e" fill="#c4c4c4"/>
  <text Z-Position="2" text_pos="topcenter" xml:space="preserve"
text-anchor="middle" font-family="Fantasy" font-size="18" id="Boundary name"
y="16.80078" x="140.5" move="none" rescale="All" stroke-linecap="butt" stroke-
linejoin="miter" stroke-dasharray="none" stroke-width="0" stroke="#8e8e8e"
fill="#000000">NAME</text>
 </g>
</svg>
```

Figure 66 - SVG-EDIT special attribute code example

4.11.4.1 Rescale:

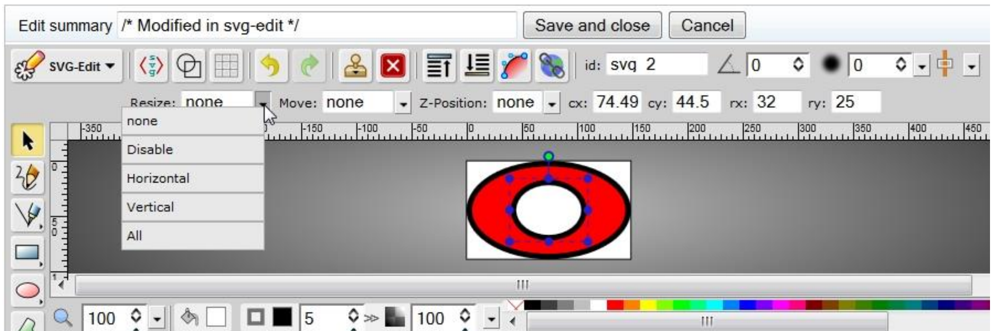


Figure 67 - Rescale attribute selection

The Rescale attribute is attribute created, especially for this project, which defines if a shape can be resized of the shape and how, with 4 types of possible values for that attribute: ‘horizontal’, which means it can be reshaped in the horizontal side and not in vertical side, ‘vertical’, which is the opposite of the previous one, disable, which means that the shape can’t be resized and the last value is ‘all’ which permits resize in any direction we want but there is still a generic value used by SVG-edit called ‘none’ which means the value doesn’t have nothing and does the same as ‘all’.

One of the reasons why rescale was used instead of resize was that SVG-edit already has an attribute called resize that's used at start, so it was decided to use rescale instead, but in the interface we still use the initial name Resize.

4.11.4.2 Move:

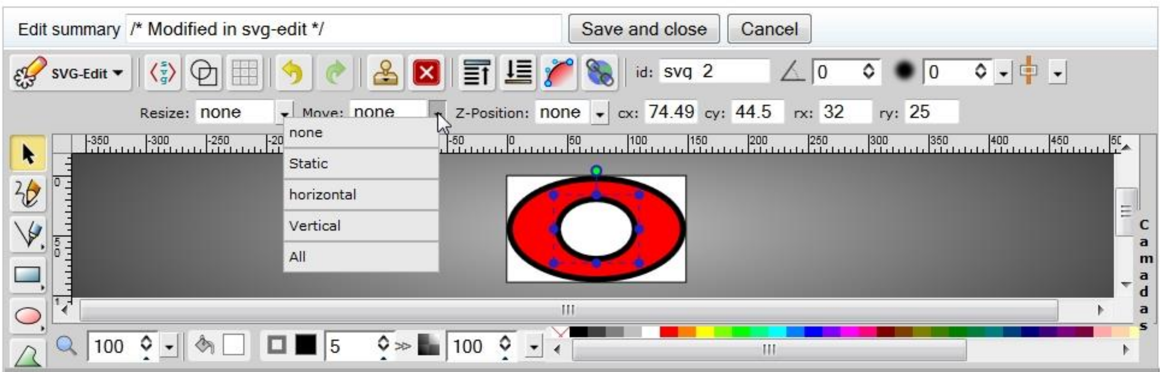


Figure 68 - Move attribute selection

For the editing of the diagrams we decide that there should be a mechanism for movement inside an object/Shape and when there's a combination of shapes used for example to create banana shape for ….., there's a need to permit a shape to move inside the banana shape and not move outside the shape, this attribute is called '**move**' which is short for movement, and the general idea is to permit a shape to move inside another shape without leaving it.

4.11.4.3 Position:

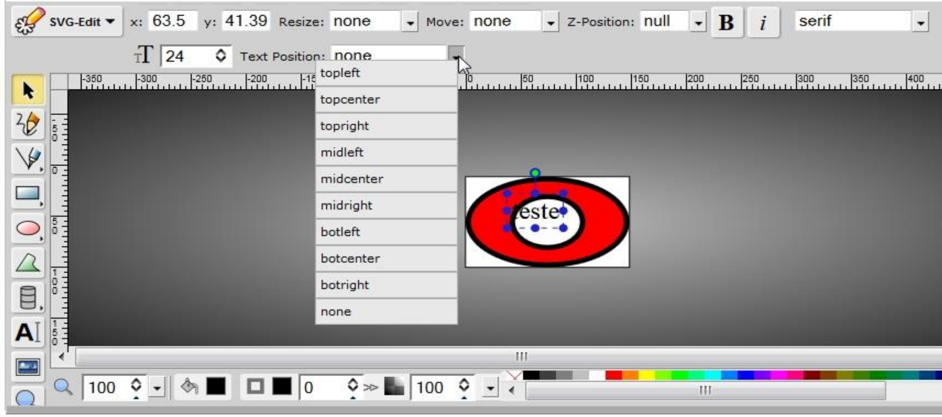


Figure 69 - Text position attribute selection

For text types there is a different type of attribute that works more or less the same, as the "move" attribute, but a lot more restricted, but only for text elements, called **Position**, which has a lot more options: topleft, topcenter (which was changed from topmid after deciding that having the horizontal and vertical using the same

nomenclature , mid, could provide for error and confusion, so following a functionality, in SVG-Edit, that handles this type of issues, the idea was followed their nomenclature which was mid for vertical and center for horizontal), “topright”, “midleft”, “midcenter” (which was changed from “midmid”), “midright”, “botleft”, “botcenter” (which was changed from “botmid”), “botright” and another option when there is no real position for it, called “none” which permits any position.

4.11.4.4 Z-Position:

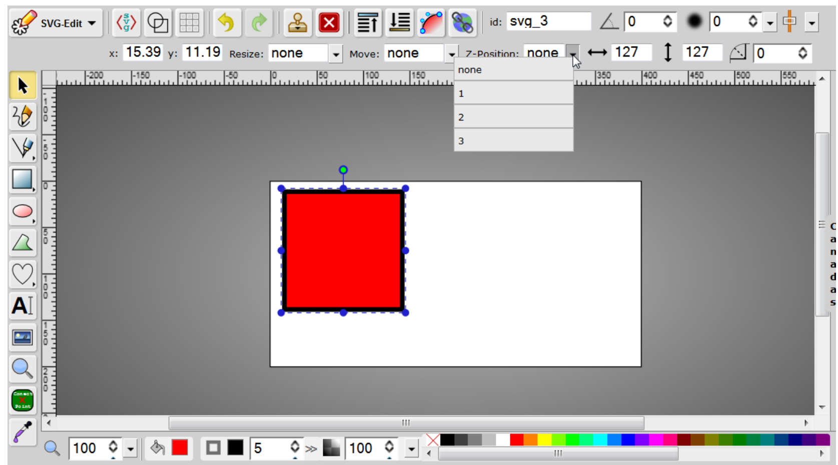


Figure 70 - Z-Position attribute selection

For some shapes there’s a need to define an attribute called z- position which defines on which layer the element is at. The possible values for that attribute are: none, 1, 2 and 3, being that the higher value means it’s on top of the other layers.

Closer to the end of this project, the implementation of this attribute was a little left behind since the Z-position property was added directly in the shape kind template before creating any SVG file. So this property doesn’t exist in the component templates. The reason why this attribute isn’t as effective as initially imagined is that SVG already has its own way position it’s elements based on the which element, in the code, comes first because even if the diagram editor has those 3 values, if there is 2 or more shapes or elements in a determined z-position, how will the editor know which, in that z-position, comes first than the other, when saving to the MediaWiki pages. Because of this there might be a new property just for that situation but only to add to the shape template and to the components template, in order for the diagram editor to know the order of the shapes or elements, in case the diagram editor gets the data directly from the pages and not from the SVG file.

4.11.5 SVG-Edit procedures for creating shape kind

In order for all the component of this project to work when creating a shape kind we need to pay attention to a few details like:

- **Positioning** – the boundary elements need to be positioned to the leftmost side and to the top so there is no blank space when we save it and the general solution is to use a small function in SVG Edit for the SVG properties (Figure 71), that adjust the dimensions of the SVG to the existing Elements (Figure 72) to the boundary elements, which is something that is taken in mind for the rest of the code created.

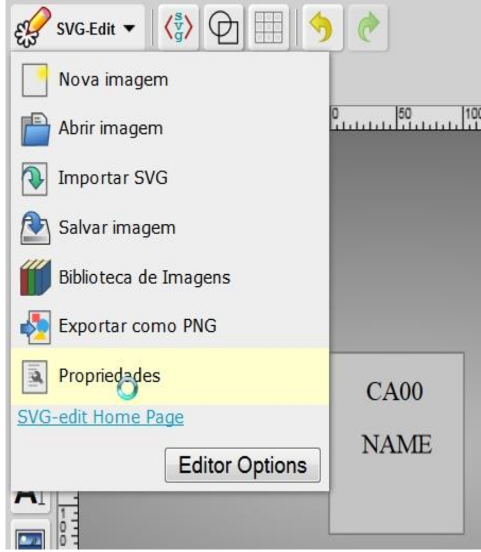


Figure 71 - SVG Properties

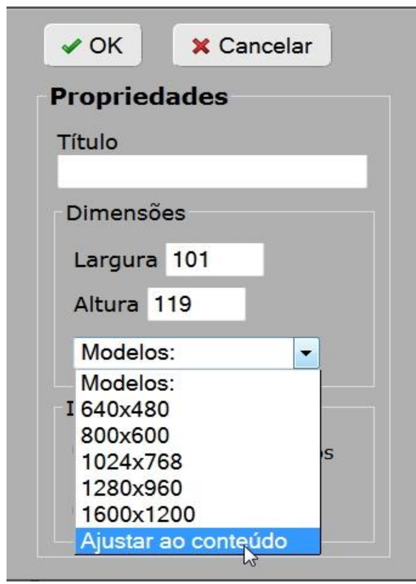


Figure 72 - SVG Dimension definition

- **Text positioning** – if there some text element that is out of some basic shape element do not include it until after the positioning of those basic element and the previous step, just for the height and width of the shape kind to be only based on the other components and not because of the text component because of the positioning in the diagram editor, since the diagram editor works with that assumption.
- **New attribute selection** – use them correctly because the diagram editor uses them, if there isn't any position that you like for text positioning try using a combination of rescale and move attributes, but pay close attention how they work.
- **Attributes not used** – there some attributes not save in the templates, for example the line 'stroke-linejoin' and 'stroke-linecap', so even if you use them,

they won’t be saved in the templates but will be saved in the SVG file, so their use isn’t efficient and not advised. So try to alter only the properties used in the templates for each element/components.

4.11.6 SVG-Edit modifications

In order to have the additions of the new attributes, some changes had to be made to the SVG-EDIT code in order to have the additional menus in the interface and for SVG-Edit to permit for those attributes to be accepted in the SVG code, which is basically xml code but with a certain specifications. It wasn’t easy to adapt the code since there was a lot of code, but after many hours, most of the pertinent sections of the code needed to make this sort of changes were found and altered but while they are working, any changes made to SVG-Edit, in newer versions, might make this alterations not work properly but hopefully they will still work since any update in SVG-Edit will permit for better utilization of the same. The majority of the changes and files associated with them are displayed in Table 6.

Table 6 - SVG-Edit file names, description and changes done

Filename	Resumed Description	Major changes
svg-editor.html	Main file for SVG-edit, where all the interaction happens, a visual overview of the software	Added all the new options for the elements attributes to the html code, and the elements they can interact with
history.js	JavaScript file with all the function to permit the save the changes in order to use the Back/Forward buttons to navigate between the previously content and the most recent one	None were made
units.js	JavaScript file with all the function to change the unit used for the numeric attributes like width and height which can use millimeters or pixel	None were made
touch.js	JavaScript file with all the functions to work with mouse handler touch options	None were made
svgutils.js	JavaScript file with all the function to change and clean attributes with incorrect values and remove attributes with values that are predefined	Since we wanted the attribute stroke-width to show up even if the value was 1, which is the predefined, that value had to be removed from that function.

svgtransformlist.js	JavaScript file with all the function to mostly work with the transform attributes: matrix, translate, scale and rotate	None were made
svg-editor.js	JavaScript file with all the functions that work around the updating and adding new values to attributes inside elements	Added new functions to work with the new attributes, like Rescale, Move, Z-position and Position
svg-editor.css	CSS file with the style sheet used for describing the presentation semantics	None were made
svgcanvas.js	JavaScript file with all the functions that add the initial element and values for a different shape(SVG element) and all the interactions with all the other JavaScript's	Added to the SVG elements, the new attributes that could be used, like Rescale, Move, Z-position and Position with their basic value
select.js	JavaScript file with all the functions to create and select objects(SVG elements)	None were made
sanitize.js	JavaScript file with all the functions to work with attributes permission for each element	Added to the SVG elements the new attributes that could be used, like Rescale, Move, Z-position and Position
path.js	JavaScript file with all the functions to work with path creations, curves and so on	None were made
math.js	JavaScript file with all the function to work with the positioning and movement of objects	None were made
draw.js	JavaScript file with all the function to work with layers and the drawing of the elements and opacity layers	None were made
contextmenu.js	JavaScript file with all the function to work with menus used in the html	None were made
browser.js	JavaScript file with all the functions to work around different browsers	None were made

Most of those changes are added in Appendix - **A.5.**

4.12 Connector-Edit

For the creation and editing of Connector kinds, there was a need to create a separate interface from what we normally use in MediaWiki, which is forms, but the data received from that interface needs to be saved in a template. Each Connector kind can have multiple diagram kinds associated to it and multiple properties, so there was a need to define how to differentiate them, so a template was created inside MediaWiki with specific properties which we can be seen in Figure 73.

Template:CONNECTOR KIND

This is the "CONNECTOR KIND" template. It should be called in the following format:

```
{{CONNECTOR KIND
|Connector name=
|Version=
|Is allowed in diagram kind=
|Start connector symbol name=
|Start connector symbol=
|Start connector symbol position name=
|Mid connector symbol name=
|Mid connector symbol=
|End connector symbol name=
|End connector symbol=
|End connector symbol position name=
|Start connection rule=Out
|End connection rule=Out
|Connection line width=
|Connection line color=
|Connection line dasharray=
|Connection line path=
|Connector z-position=3
|Connector thumbnail=
|OAKRK=
}}
```

Figure 73 - Connector kind template

Defining the Connector Kind basic data/properties:

- **Connector name -** the name given to the Connector Kind**,** which can be filled in the input text type with the label Connector name, in the Figure 74 - Connector Edit Interface.
- **Version -** the version associated to this Connector Kind**,** which can be selected in the input select box with the label Version, in the Figure 74 - Connector Edit Interface.
- **Is allowed in diagram kind -** the diagram kind associated to this Connector Kind**,** which can be selected in the input select box with the label Diagram Type, in the Figure 74 - Connector Edit Interface.
- **OAKRK** – gives the value to the Organizational Artifact Kind Relation Kind which is dynamically obtained from the MediaWiki pages and displayed in the options for

the input select box with the label Organizational Artifact Kind Relation Kind, in the Figure 74 - Connector Edit Interface.

- **Connector z-position** – gives the z-position to the connector kind, which is 3 for every newly created Connector Kind. This value can vary in the connector template.
- **Connector thumbnail** – gives SVG visual representation link to the SVG thumbnail which is only created when the connector is saved in the Connector Edit interface.

Defining the line data:

- **Connection line path –** there are 3 option which are: curved, right_angle and straight, which can be selected in the input select box with the label Line path type, in the Figure 74 - Connector Edit Interface.
- **Connection line dasharray –** the value given to the type of line which can be, which can be selected in the input select box with the label Stroke-dasharray, in the Figure 74 - Connector Edit Interface
- **Connection line color -** any type of color defined by its hex number,
- **Connection line width** – the value given to the line, which is currently pixels based and can be changed to inches or even millimeters but not at the moment,

Defining the Symbols used in the connection:

- **Start connector symbol name** and **Start connector symbol** - gives the name of the SVG shape and its SVG visual representation link, which can be selected below the label Select Start Connection Symbol, in the Figure 74 - Connector Edit Interface.
- **Start connector symbol position name** — this property selects how the symbol is connected to the shape, and those are:
  - Outside_Shape, which means that symbols is connected to the border of that shape,
  - Inside_Shape, which means that the symbols is all inside the shape leaving the line outside of it,

- On_Shape_Line, means that the symbols is positioned on top of the shape border line leaving half of it inside the shape and the other half outside,

This can be selected in the select box with the label Select Start Symbol Position, in the Figure 74 - Connector Edit Interface.

- **Mid connector symbol name** and **Mid connector symbol** – gives the name of the SVG shape and its SVG visual representation link, which can be selected below the label Select Mid Connection Symbol, in the Figure 74 - Connector Edit Interface.
- **End Connector Symbol name** and **End Connector Symbol** - gives the name of the SVG shape and its SVG visual representation link, which can be selected below the label Select End Connection Symbol, in the Figure 74 - Connector Edit Interface.
- **End Connector Symbol position name –** this property follows the same values as **Start connector symbol position name** but is selected in the select box with the label Select End Symbol Position, in the Figure 74 - Connector Edit Interface.

The basic template properties are defined directly in MediaWiki and the specification and information about this connector kind template property are displayed in Table 7.

Table 7 - Connector kind template properties specification

Property	Property Type	Allowed Values and added information
[Connector name::]	[[Has type::String]]	Any value is permitted so it's user dependent
[Version::]	[[Has type::Number]]	Depends in the values that where created in the Form Language, in the property:Version
[Start connector symbol name::]	[[Has type::String]]	Depends in the Symbols that exist in the folder: Symbols
[Start connector symbol::]	[[Has type::Page]]	Depends in the Symbols that exist in the folder: Symbols
[Start connector symbol position name::]	[[Has type::String]]	* [[Allows value::Inside_shape]] * [[Allows value::On_shape_line]] * [[Allows value::Outside_shape]]
~~[Start connector symbol position::]~~	~~[[Has type::Page]]~~	~~* [[Allows value:: File:Inside_shape.svg]]~~ ~~* [[Allows value::File:On_shape_line.svg]]~~ ~~* [[Allows value:: File:Outside_shape.svg]]~~
[Mid connector symbol name::]	[[Has type::String]]	Depends in the Symbols that exist in the folder: Symbols
[Mid connector symbol::]	[[Has type::Page]]	Depends in the Symbols that exist in the

		folder: Symbols
[End connector symbol::]	[[Has type::Page]]	Depends in the Symbols that exist in the folder: Symbols
[End connector symbol name::]	[[Has type::String]]	Depends in the Symbols that exist in the folder: Symbols
[End connector symbol position name::]	[[Has type::String]]	* [[Allows value::Inside_shape]] * [[Allows value::On_shape_line]] * [[Allows value::Outside_shape]]
~~[End connector symbol position::]~~	~~[[Has type::Page]]~~	~~* [[Allows value:: File:Inside_shape.svg]]~~ ~~* [[Allows value::File:On_shape_line.svg]]~~ ~~* [[Allows value:: File:Outside_shape.svg]]~~
[Start connection rule::]	[[Has type::String]]	* [[Allows value::Out]] * [[Allows value::In]] * [[Allows value::In/Out]] Predefined value =Out
[End connection rule::]	[[Has type::String]]	* [[Allows value::Out]] * [[Allows value::In]] * [[Allows value::In/Out]] Predefined value =Out
[Connection line width::]	[[Has type::Number]]	Any numeric value is permitted (user dependent) but has to be a reasonable value
[Connection line color::]	[[Has type::String]]	It's a numeric value but with a # up front and permits any Hex Color Code(user dependent)
[Connection line dasharray::]	[[Has type::String]]	It's a string value with six possibilities for the time being. New ones can be added later but be added to select menu • 0 ; • 2 ; • 2, 5, 2, 5; • 4, 5, 4, 5; • 5, 4, 2, 4; • 5, 4, 2, 4, 2;
[Connector z-position::]	[[Has type::Number]]	* [[Allows value::1]] * [[Allows value::2]] * [[Allows value::3]] Predefined value =3
[Connector Thumbnail::]	[[Has type::Page]]	Created File link when the page is created in the Connector Edit page
[OAKRK::]	[[Has type::Page]]	Depends in the OAKRK page name for the created Organizational Artifact Kind Relation Kind

Label for symbol inside table:

* - Means that it is imbued in the property.

—— - Line in top of the property means it was removed during the project.

4.12.1 Connector-edit implementation

For the creation of the SG-Edit interface many technologies where used, html, CSS, Canvas, SVG, PHP and in this section we will talk on how it was implemented and how we reached the current interface - Figure 74.

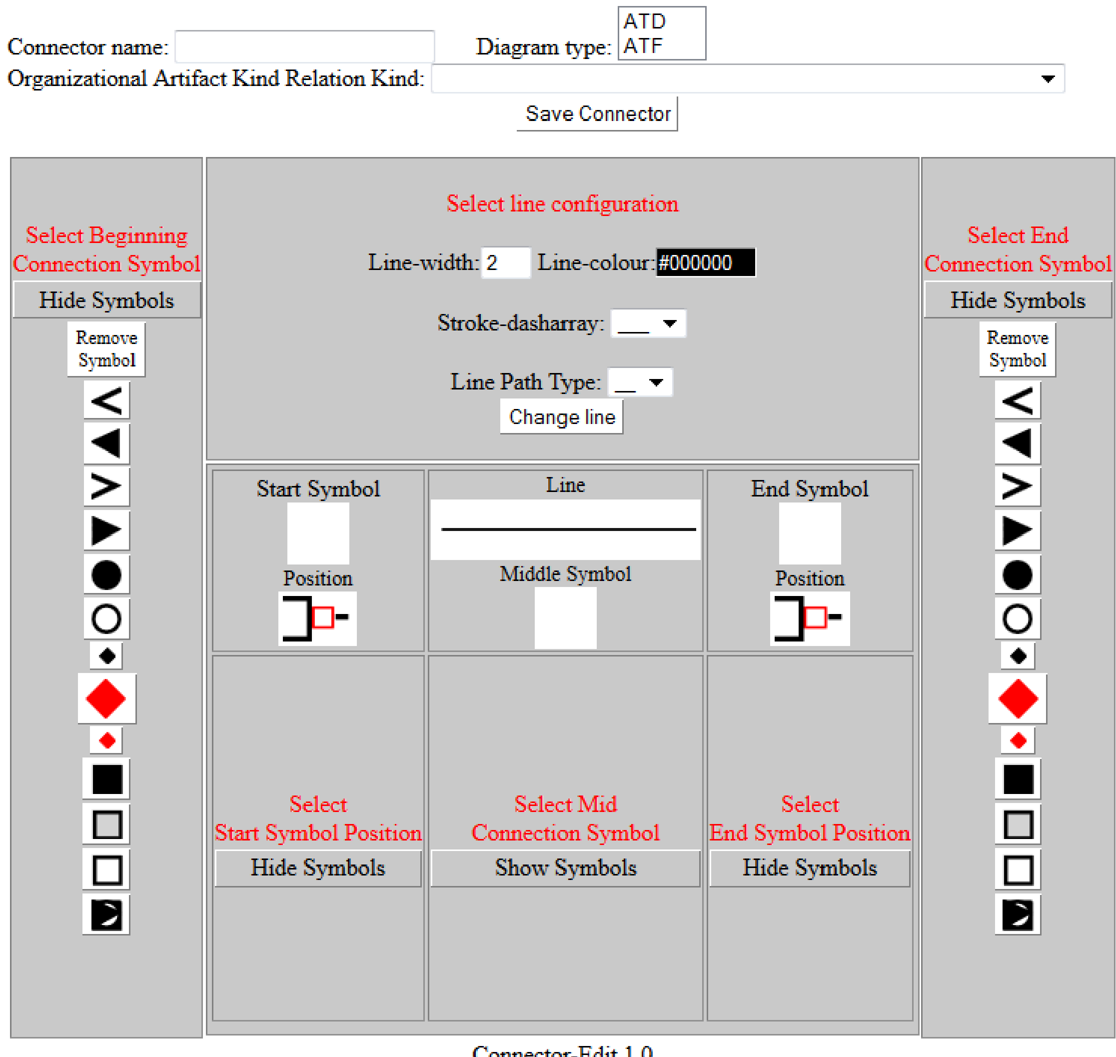


Figure 74 - Connector Edit Interface

For most of this interaction with canvas, the application **canvg** was used**,** which is a JavaScript SVG parser and renderer on Canvas, which permitted to add SVG into a Canvas element and grant some functionality and some parameters that permitted me to

add new SVG elements to canvas without worrying about it being displayed or not, which off course would be. The parameters used where:

- ignoreMouse: true => ignore mouse events
- ignoreAnimation: true => ignore animations
- ignoreDimensions: true => does not try to resize canvas
- ignoreClear: true => does not clear canvas

And there were some other parameters but since they weren't useful they were left aside.

The way it's used is to load the images on windows load or add them when needed, which in this case we used the function canvg, an example of that is that in order to clear a canvas element we had to reset the data inside the canvas, so this code was used in order to do so:

```
canvg(document.getElementById('canvas1'),'<svg><rect height="20" width="20" y="2.5" x="2.5" stroke-width="3" stroke="#ffffff" fill="#ffffff"/></svg>',{ ignoreMouse: true, ignoreAnimation: true,ignoreClear:false});
```

And for the adding of SVG Symbols we used a similar function but instead of using a SVG code directly in the function, it loaded the SVG code directly from a file that represented the symbol.

```
canvg('canvas1', 'http://localhost/connector-edit/Symbols/arrow_left_clear.svg',{ ignoreMouse: true, ignoreAnimation: true});
```

For the creation of the LOGO we sent the symbols and the lines data to a PHP file (saveimage.php) with an algorithm that could handle the variations of SVG elements and the resulting positioning of the combination between the start, mid and end symbols, basically the variations of the width and height of the Symbols in combination with the line elements. Which lead to changes in both x and y in the elements but in order to minimize the variations and the simplification of the code, it was decided to create around each Symbol, in case they exist, a G element is created and the Symbol Elements is added to its child elements, inside that <g> it is added the attribute transform in conjunction with the SVG transformation "translate" to manipulate that symbol position, and the positioning of the line is inside another the element g, below we see an example on how the transform attribute is used, by using two variables for the offset of each axis.

```
<g transform="translate ('.$offsetx.','.$offsety.')"'>
```

In the end of this project this last part of the creation was changed, because of some alterations on how the connector would be implemented in the diagram editor, so it was decided to improve this in order to facilitate the adding of this sort of connector by using markers with the nomenclature used for using three types of markers in lines are: marker-start, marker-mid and marker-end.

For the creation of the line, a polyline element is used and added those markers attributes, which can see an example of how it's implemented below:

```
<polyline points = "20,20 40,20 60,20" style="stroke:black; stroke-width:2; stroke-dasharray:0" marker-start = "url(#StartMarker)" marker-mid = "url(#MidMarker)" marker-end = "url(#EndMarker)"/>
```

Now that the line code it defined, a marker attribute is needed to define the start symbol, mid symbol and the end symbol following the same nomenclature used, for example the marker for the start symbol, the marker-start, is implemented in this form:

1. The markers with the id used in the marker attribute in polyline are created.
2. The SVG code for each element, used in the chosen symbol, is added to its specific marker, as its child elements. This is done for all the three types of symbols.
3. For each marker the values of the viewbox are altered, as the attributes
4. The viewbox attribute, in the marker, changes its values for width and height, depending on the values of the SVG width and height. Also the "markerwidth" uses half the value of the SVG width and the "markerheight" also change its value by half.
5. In order for symbol to be exactly in top of the line, the center of the symbol must be selected and the values for the refx and refy must be altered, in the same way as the "markerheight" and "markerwidth", but depending on the position of the start and end symbol the value for refx diverges between, 1 for the "inside_shape", half the width for the "on_line_shape" and width plus 1 for the "outside_shape".

Resulting in a code similar to this one:

```
<marker id = "StartMarker" viewBox = "0 0 13 13" refX = "12" refY = "6.5" markerUnits = "strokeWidth" markerWidth = "6.5" markerHeight = "6.5"  orient = "auto">
    <rect transform="rotate(45, 6.64258, 6.64372)" fill="#FF0000" stroke="#FF0000" x="2.29941" y="2.30055" width="8.68584" height="8.68584"/>
</marker>
```

After this alteration, the code would continue to work essentially the same but with this improvement to the SVG code.

After that algorithm that string is added to a file inside the folder of the application and uploaded to Media Wiki with the initial selected name file and the format SVG. And in the end, a page is created with the all the data for the connection type (Symbols and line data) using the PHP code created for that, which basically is a variation of the previous code used for creation of pages, and a similar code was used for the uploading also.
The resulting file, besides working has a logo for the connector kind button, in the diagram editor, it also can work directly with the implementation of the diagram editor, in a way, when adding a connector to a diagram, the code used in the SVG can be used directly for the connection between two shapes. Basically the SVG code will help for the creation of the connector code used in the diagram editor, instead of just for the logo in the connector kind button.

In order to follow the UEAOM, it was necessary to create the components pages associated to each symbol used, except for the line symbol which is detailed in the connector kind template, and for that a new PHP file was created, which essentially reads the SVG file of each Symbol and creates the components pages associated with that specific symbol, those components pages use the Templates that are also used by the shape kind to decompose each Symbol into their basic components. The only divergence is that the property "Is part of", instead of linking to the shape king page, it links to the SVG file page, which in this case is the same value used in Connector kind page in one of three possible properties: Start connector symbol, Mid connector symbol, End connector symbol.

Table 8 - Connector-Edit, files and usage description

Filename	Description
Connector-Edit.html	Main page, interface for all the selections
Saveimage.php	Creation of the SVG logo and upload, creating the MediaWiki page for the connector.
savesymbols.php	Creation of the MediaWiki pages, for each components used in each symbols
Connector-edit.php	Extension for MediaWiki in order to use a hook to call the page: maininterface.html which will be called Connector-Edit.html.

The hardest part of the implementation was the connection to the MediaWiki, since for every interaction with MediaWiki; a user has to be logged in order to work. Many example of code related to that sort of interaction exist but they simply didn't work, initially there wasn't any obvious reason, but almost in the end of the project, it was found, it was related to some issue inside XAMPP that didn't permit cookies to work properly, which couldn't be resolved. Since the code for that type of interaction was already working, by using a small fix implemented just for this problem, which was basically an adaption of those examples but going around that issue by not directly using cookies.

The code related to the fix wasn't easy to create because there was a need to understand the MediaWiki API, on how it receives the data, what kind of data it sends back, if there are changes on that specific data for any special case between other factors but in the end, the code was working correctly. In that code a small PHP application was used called "Snoopy", which is a PHP class that simulates a web browser and automates the task of retrieving web page content, which was very helpful with this sort of interaction with the MediaWiki API.

4.12.2 Adapting extension Anywebsite for Connector-Edit

In order to create the connector-edit inside a page, some research was done and a extension called **Anywebsite** was found, which basically gives a parser hook that worked like an iframe, where is added the http link of the page. In order to make it specifically for connector edit instead of grabbing the link inside the parser hook, the code was altered in order for it to create a specific parser hook for the extension: **Connector-Edit,** and not grab a page link in the parser hook but use a static one leading to the main interface page: Connector-Edit.html. At this moment the Parser extension tag used is "con_web" and this sort of implementation can be used for future extensions.

5 General Arquitecture

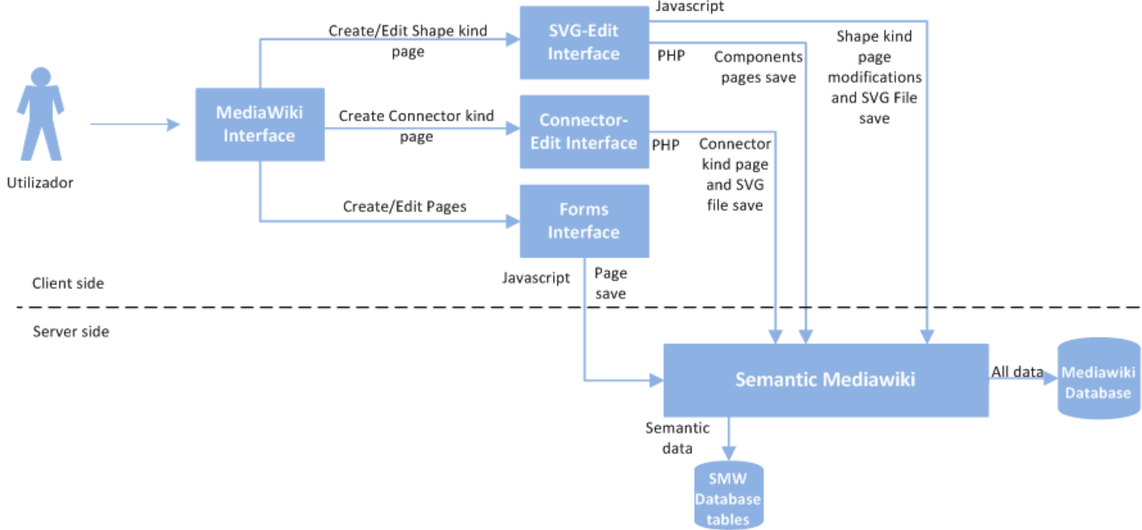


Figure 75 - Meta-Editor arquitecture

Any user that wants to use this Editor will have to use the mediawiki interface. In Figure 75, we can see the basic arquitecture of this Editor, where we can use three diferent interfaces:

1. Create/Edit shape kind pages by using the SVG-Edit interface but before using the SVG-edit the basic page must be created by using the Forms interface.
   - Saving of the resulting SVG file and creation of the specific page is done via a javascript.
   - Creation of the componenet pages is done via PHP.
2. Create connector kind page by using the connector-Edit interface.
   - Creation and saving of the resulting SVG file of the connector and creation of the page is done via PHP.
3. Create and edit pages can be done using the Forms interface. All pages can be altered manually, even the two previous, but it`s not advised if a person doesn`t know what his doing.
   - Saving of the pages is done by javascript.

All the data is saved in Mediawiki batabase and the semantic data is saved in the SMW tables.

6 Conclusion

Working with MediaWiki wasn't always easy, because during the creation of pages using an external page wasn't working like it should, some examples used cookies and it wasn't possible to make it to work. So it was needed to adapt what was found in order to make it work, which took some weeks, perfecting the PHP code for that, because for some examples it worked perfectly but for other it completely messed up, but it lead to a better understanding of the web MediaWiki API.

During the initial phases of adapting SVG-Edit, there were a few problems, because it's not always easy to read other peoples code, but after a few weeks, most of the sections of that code that were needed to be altered were found, so it was possible to add the additional code and without it completely damaging the structure of the existing code and adding bugs in the mixture. Another issue that appeared with SVG-Edit was that when the browsers updated some of their SVG code, the application just gave wrong positioning to the SVG elements created and initially it was not known that it was because of the browser or it was some mistake done in the altered code. After some research it was found out that it wasn't anything to do with the altered code and implemented a fix, based on the solution given in the SVG-Edit forums, which were very helpful since this bug appeared two times because of two different updates but the last one only for Firefox, patch 23.0.1.

One usual difficulty with working with all this different technologies and platforms was that, after a few week, of not working with code associated with a specific software like SVG-Edit, it wasn't possible to recall the precise way, a certain code was created or why, so the comments added to the code, needed to be more precise so that wouldn't happen anymore.

During the implementation of the pages there was a need to pay close attention to how my other colleague was creating the Diagram Editor, since he could change the way he was implementing some features, at that could lead to certain changes in both template or page connections. During this phase the communication had to work, or else we would have incompatible MediaWiki pages, templates and properties and specially the way my other colleague was creating and saving all the shapes, connections. Since this project was creating the database for him to put his data, we had to reach certain agreements, on how it would be saved.

One of the major contributions, done during this thesis, was the UEAOM (Universal Enterprise Adaptive Object Model) and how to adapt it to the MediaWiki Universe, leading to the creation of WAMM (Wiki Aided Meta Modeling) platform and the creation of a DEMO language example by using that platform.

While this prototype is not perfect, if followed instruction, it can create all the pages needed for the addition and alteration of types of Model/Diagrams, their symbols, connectors and their respective grammar. It still needs some improvement like the majority of prototypes, but it's still a good step forward to a good solution.

There can be some improvements, like a better specification of how to manage the versions of the language, the creation of a way of permitting for the migration of models to a newer version, a definition of the specification of the state changes that can occur with all artifacts, a slightly better use of SVG-Edit in the shape kind definition, the implementation of the connector point, which can happen if the existing proposal for the connector point reaches the final stages and is implemented in SVG basic functions and finally the addition of a way of creating the symbols for the connector kind, start, mid and end symbols, so the Connector-Edit can have user created Symbols, instead of only using the basic symbols already created. Another thing that can always be improved is the optimization of the code, which in the most recent code created was done but not in the older code, like the PHP code used for all the interaction with the MediaWiki, which is reutilized all over the code with some minor alterations.

7 Bibliography

[1] Aveiro, D., Pinto, D. “Universal Enterprise Adaptive Object Model”, KEOD 2013, Vilamoura, Portugal, 2013

[2] Capela, J., “Semantic MediaWiki adaptation to support Organizational Engineering”, Universidade da Madeira, 2012

[3] How to Choose Between SVG and Canvas, http://msdn.microsoft.com/en-us/library/ie/gg193983%28v=vs.85%29.aspx, 2013

[4] “Scalable Vector Graphics (SVG)”, http://www.w3.org/Graphics/SVG/, 2013

[5] OMG, 2012. “OMG’s MetaObject Facility (MOF) Home Page”, Available at: http://www.omg.org/mof/ , 2013

[6] “Semantic wiki”, http://en.wikipedia.org/wiki/Semantic_wiki, 2013

[7] “Demo Methodology”, http://www.demo.nl/methodology, 2013

[8] Freitas, E., “Integrating Semantic MediaWiki and Open Modeling Platforms for Production and collaborative Evolution of Organizational Models”, Universidade da Madeira, Funchal, Portugal, Setembro 2012

[9]J. Dietz, “Enterprise Ontology: Theory and Methodology”, Germany: Springer-Verlag Berlin Heidelberg, 2006.

[10] D. Aveiro, “G.O.D. (Generation, Operationalization & Discontinuation) and Control (sub) organizations: a DEMO-based approach for continuous real-time management of organizational change caused by exceptions”, Instituto Superior Técnico, 2010.

[11] svg-edit A complete vector graphics editor in the browser (in JavaScript), http://code.google.com/p/svg-edit/ , 2013

[12] Extension:SVGEdit, http://www.mediawiki.org/wiki/Extension:SVGEdit, 2013

[13] Dietz, J.L.G., 2005. A World Ontology Specification Language. Em S. B. / Heidelberg, ed. On the Move to Meaningful Internet Systems 2005: OTM Workshops. pp 688–699. Available at: http://dx.doi.org/10.1007/11575863_88.

[14] Dietz, J.L.G., 2006. ENTERPRISE ONTOLOGY - UNDERSTANDING THE ESSENCE OF ORGANIZATIONAL OPERATION. In C.-S. Chen et al., eds. Enterprise Information Systems VII. Springer Netherlands, pp 19–30. Available at: http://link.springer.com/chapter/10.1007/978-1-4020-5347-4_3 [Accessed February 18, 2013].

[14] Dietz, J.L.G., 2008. On the Nature of Business Rules. Advances in Enterprise Engineering I, pp.1–15.

[15] Yoder, J.W., Balaguer, F. & Johnson, R., 2001. Architecture and design of adaptive object-models. SIGPLAN Not., 36(12), pp.50–60.

[16] Halpin, T., 1998. Object-Role Modeling: an overview. Available at http://www.orm.net/pdf/ORMwhitePaper.pdf.

[17] Dietz, J. L. G., 2009. Is it PHI TAO PSI or Bullshit? Em The enterprise engineering series. Methodologies for Enterprise Engineering symposium. Delft: TU Delft, Faculteit Elektrotechniek, Wiskunde en Informatica.

[18] Ferreira, H. S., Correia, F. F. & Welicki, L., 2008. Patterns for data and metadata evolution in adaptive object-models. Em Proceedings of the 15th Conference on Pattern Languages of Programs. PLoP '08. New York, NY, USA: ACM, pp 5:1–5:9. Available at: http:// doi.acm.org/ 10.1145/ 1753196.1753203.

[19] Guizzardi, G., 2005. Ontological foundations for structural conceptual models. Available at: http://doc.utwente.nl/50826/.

[20] Lankhorst, M. M., Proper, H. A. & Jonkers, H., 2010. The Anatomy of the ArchiMate Language. International Journal of Information System Modeling and Design, 1(1), pp.1–32.

[21] La Rosa, M. et al., 2011. Managing Process Model Complexity Via Abstract Syntax Modifications. IEEE Transactions on Industrial Informatics, 7(4), pp.614–629.

[22] Aveiro, D., Pinto, D. (2013). "IMPLEMENTING ORGANIZATIONAL SELF AWARENESS, A Semantic MediaWiki based Enterprise Ontology Management approach", KEOD 2013, Vilamoura, Portugal, 2013

[23] Microsoft, 2010. Visio 2010, Microsoft, Available at: http://www.microsoft.com

[24] MPRISE, 2010. Xemod, Available at: http://www.mprise.eu/

[25] Hommes, B.-J., 2013. ModelWorld, Available at www.modelworld.nl

8 Appendix

A.1 – Installation manual

The following guide is a partial transcription from J. Capela's master thesis [2] entitled "Semantic MediaWiki adaptation to support Organizational Engineering" with some additions and alterations.

The OS used for this installation was Windows 7, but the install steps should be the same for all Windows OS. To take advantage of the semantic features and the use of the SVG-Edit extension the recommended browser is Firefox or Chrome (other browser should work but they weren't used).

All files are included in the project CD, within a xamp installation that only needs to be added to «c:\\», resulting in «c:\xampp\» folder.

Installation steps used:

1. Install «xampp-win32-2.5-installer.exe» (or higher version) as Administrator
   a) The installation dir should e «c:\», and the files will be automatically placed inside «c:\xampp\» folder
2. Open your browser and run phpMyAdmin
   a) Create the user «wikiuser», password «12345» with all permissions
3. Extract « mediawiki-1.19.2.tar.gz» (or higher version) files to «htdocs» folder on xampp server
4. Rename the extracted folder to «wiki»
5. Change the «Só de leitura» property to allow writing in «wiki» folder (do the same to «config» folder)
6. Browse «http://localhost/wiki» to start the installation process
7. The form should be filled with the following information:
   a) Wiki name: EOMediaWiki
   b) Admin username: Vitor
   c) Password: password
   d) Database name: wiki20122013
   e) DB username: wikiuser
   f) DB password: 12345
   g) Database table prefix: wikiextra
   h) Storage Engine: InnoDB
   i) Database caracters : UTF-8

8. After completion of the installation process, copy «LocalSettings.php» from «Config» folder to «wiki» folder root

9. Rename «config» folder (i.e. «old_config»)

10. Extract «smw-1.7.1.zip» files to wiki «extensions» folder

11 .Browse «http://localhost/wiki» and login as Administrator:

a) Username: Vitor

b) Password: password

12. Browse «http://localhost/mediawiki/index.php/Special:SMWAdmin»

13. Click the «Initialise or upgrade tables» button

14. Click the «Start updating data» button

15. Browse «http://localhost/ mediawiki /index.php/Special:SMWAdmin» and wait for the completion of the installation process (refresh the page) which, in Figure 76, is possible to see a print screen of the original installation used for this project.

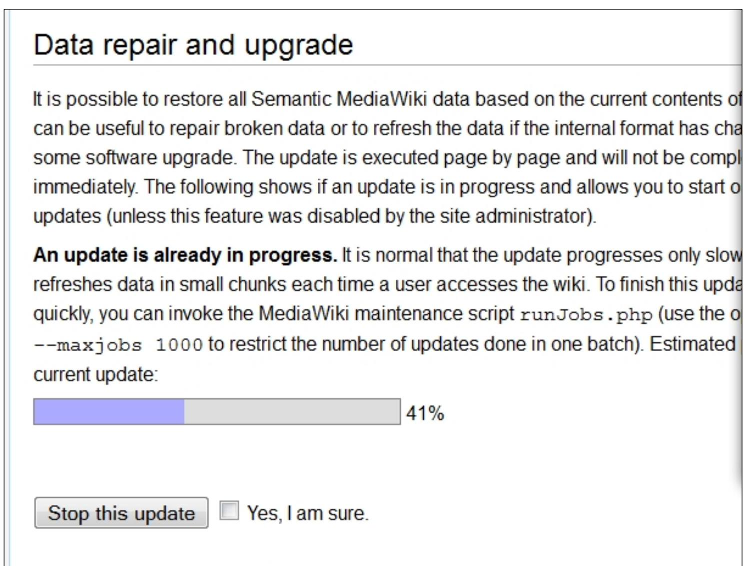


Figure 76 - Update process for the data in SMWAdmin

16. Browse «http://localhost/wiki/index.php/Special:Version» and check if it was properly installed

17. Extract «scriptmanager-1.0.0_0.zip» files to wiki «extensions» folder

18. Extract «Wikieditor-0.3.1.zip» file to wiki «extensions» folder .

19. Extract «semantic_forms_2.5.1.zip» file to wiki «extensions» folder.

20. Extract «SVG-Edit_altered_edition.zip» file to the htdocs folder.

21. Extract «extension: SVG-Edit_altered_version.zip» file to wiki «extensions» folder.

22. Extract « Specialpagenameextension.php » file to wiki «extensions» folder.

23. Extract «connector-edit.php » file to wiki «extensions» folder.

24. Extract «ImageMagick-6.8.5-9.zip » file to the «htdocs» folder and install it there.

25. Add the following lines at the end of «LocalSettings.php» file :

```
##permit uploads
$wgEnableUploads  = true;
##Remove hashed upload
$wgHashedUploadDirectory = false;
## semanticmediamiki
include_once("$IP/extensions/SemanticMediaWiki/SemanticMediaWiki.php")
;
enableSemantics('localhost');
## Semmanticforms
include_once("$IP/extensions/SemanticForms/SemanticForms.php");
require_once("extensions/ScriptManager/SM_Initialize.php");
$phpInterpreter="C:\xampp\php\php.exe";
//$wgDefaultSkin = "ontoskin2";*/
## SVGEDit
require_once("$IP/extensions/SVGEdit/SVGEdit.php");
##external svgedit
$wgSVGEditEditor = 'http://localhost/svg-edit/svg-editor.html';
##wikieditor
require_once( "$IP/extensions/WikiEditor/WikiEditor.php" );
##show options in wikieditor
$wgDefaultUserOptions['usebetatoolbar'] = 1;
$wgDefaultUserOptions['usebetatoolbar-cgd'] = 1;
$wgDefaultUserOptions['wikieditor-preview'] = 1;
##SVG
$wgFileExtensions[] = 'svg';
$wgAllowTitlesInSVG = true;
$wgSVGConverter = true;
$wgSVGConverter = 'ImageMagick';
$wgUseImageMagick = true;
$wgSVGConverters['ImageMagick']= 'C:/xampp/htdocs/ImageMagick-6.8.5-
9/convert background white -geometry $width $input $output';
##to show detailed debugging information.
$wgShowExceptionDetails = true;
$wgUseAjax=true;
##extension for connector edit and
include("$IP\extensions\connector-edit.php");
include("$IP\extensions\Specialpagenameextension.php");
//permit external access - API
$wgAllowCopyUploads = true; //allow for uploads using pages
$wgEnableAPI=true;
$wgEnableWriteAPI=true;
######edit with html elements
//$wgRawHtml = true;
##permit autocomplete in forms
$sfgAutocompleteOnAllChars = true;
###########
```

26.Open a command-line interface and change dir to «C:\xampp\htdocs\mediawiki\extensions\SemanticMediaWiki\maintenance\»

27. Run the script «SMW_setup.php» to initialize the database tables: «C:\xampp\htdocs\mediawiki\extensions\SemanticMediaWiki\maintenance>c:\xampp\php\php.exe SMW_setup.php»

A.1.1 - Testing the Installation:

1. Browse «http://localhost/mediawiki/index.php/Special:Version» and you should see something like Figure 31, Figure 32, Figure 33.

2. Now browse «« http://localhost/mediawiki» and test it by trying to follow some of the examples done in the implementation section.

A.2 – Mediawiki extensions code

- **«Special pagename OAKRK parser » extension** – PHP -specialpagenameextension.php

```
<?php
//error_reporting(0);
// Take credit for your work.
$wgExtensionCredits['parserhook'][] = array(
   // The full path and filename of the file. This allows MediaWiki
   // to display the Subversion revision number on Special:Version.
   'path' => __FILE__,
   // The name of the extension, which will appear on Special:Version.
   'name' => 'Special pagename OAKRK',
   // A description of the extension, which will appear on Special:Version.
   'description' => 'Special pagename parser function extension',
   // Alternatively, you can specify a message key for the description.
   'descriptionmsg' => 'Special pagename parser function Extension for
creating OAKRK page name',
   // The version of the extension, which will appear on Special:Version.
   // This can be a number or a string.
   'version' => 1,

   // Your name, which will appear on Special:Version.
   'author' => 'Vitor N□ga',

   // The URL to a wiki page/web page with information about the extension,
   // which will appear on Special:Version.
   'url' => 'https://www.mediawiki.org/wiki/Manual:Parser_functions',

);
// Specify the function that will initialize the parser function.
$wgHooks['ParserFirstCallInit'][] =
'SpecialpagenameExtensionSetupParserFunction';

// Allow translation of the parser function name
$wgExtensionMessagesFiles['SpecialpagenameExtension'] = dirname( __FILE__ ) .
'/SpecialpagenameExtension.i18n.php';

// Tell MediaWiki that the parser function exists.
function SpecialpagenameExtensionSetupParserFunction( &$parser ) {

   // Create a function hook associating the "example" magic word with the
   // ExampleExtensionRenderParserFunction() function. See: the section
   // 'setFunctionHook' below for details.
   $parser->setFunctionHook( 'Specialpagename',
'SpecialpagenameExtensionRenderParserFunction' );

   // Return true so that MediaWiki continues to load extensions.
   return true;
}

// Render the output of the parser function.
function SpecialpagenameExtensionRenderParserFunction( $parser, $param1=
'',$param2= '',$param3= '',$param4= '',$param5= '') {

   // The input parameters are wikitext with templates expanded.
   // The output should be wikitext too.
     $output="";
    if(trim($param1) != '')
```

```
        {
    $output = $param1;
            if((trim($param2) != '')
            or(trim($param3) != '')
            or(trim($param4) != '')
            or(trim($param5) != ''))
            {
            $output = $output.".";
            }
        }
        if($param2!=""&&$param2!=" "&&$param2!="  ")
        {
    $output = $output.$param2;
            if((trim($param3) != '')
            or(trim($param4) != '')
            or(trim($param5) != ''))
            {
            $output = $output.".";
            }
        }
    if(trim($param3) != '')
        {
    $output = $output.$param3;
            if((trim($param4) != '')
            or(trim($param5) != ''))
            {
            $output = $output.".";
            }
        }
    if(trim($param4) != '')
        {
    $output = $output.$param4;
        if(trim($param5) != '')
            {
            $output = $output.".";
            }
        }
    if(trim($param5) != '')
        {
        $output = $output.$param5;
        }
   //$output = "param1 is $param1 and param2 is $param2";
   return $parser->insertStripItem( $output, $parser->mStripState );
   //return $output;
}
```

- **«Connectorweb extension » extension** – PHP – Connector-edit.php

```
<?php
# buga Extension
#
# Tag :
#   <con_web></con_web>
# Ex :
#   <con_web mywidth="800" myheight="600"></con_web>
#
# Enjoy !
$wgExtensionFunctions[] = 'connector';
$wgExtensionCredits['parserhook'][] = array(
        'name' => 'connector extension',
        'description' => 'Display connector-edit page in iframe',
        'author' => 'buga based on Minseong Kim and Others',
        'url' => 'https://www.mediawiki.org/wiki/Manual:Hooks'
);

function connector() {
        global $wgParser;
        $wgParser->setHook('con_web', 'connectorweb');
```

```
}

# The callback function for converting the input text to HTML output
function connectorweb($input, $argv) {
        if (isset($argv['mywidth'])) {
                $width = $argv['mywidth'];
        } else {
                $width = 100000;
        }
        if (isset($argv['myheight'])) {
                $height = $argv['myheight'];
        } else {
                $height = 700;
        }
        $link="http://localhost/connector-edit/maininterface_7.0.html";
        $output= '<iframe name="anyweb" src="'.htmlspecialchars($link)
                 .'" width="'.$width.'" height="'.$height.'"
frameborder="0">'.'</iframe>';
        return $output;
}
```

A.3 – Connector-Edit code

- **Maininterface.html** – HTML – Interface
- **saveimage.php** – PHP – Saving of the SVG generated image
- **savesymbols.php** – PHP – creation of the corresponding component pages of each symbol used for a connector kind.

A.4 - SVG-Edit extension Code

- **ext.svgedit.toolbar.js**- JavaScript – Toolbar code alterations done
- **deletepages.php** - PHP – Removes any pages that aren`t related correspond to the current shape kind (removel of any SVG element leading to the removal of that corresponding page).

A.5 - SVG-Edit code alteration:

- **Svg-editor.html-** Added all the new options for the elements attributes to the html code, and the elements they can interact with.
    - **General element section:**

```
<div class="toolset" id="tool_rescale">
                    <label title="Change rescale option"
id="tool_rescale">Resize:
                           <!--  Rescale-->
                    <input id="rescale" type="text" title="Change rescale"
size="6" class="attr_changer" data-attr="rescale"/> <!---->
                    </label>
                    <div id="rescale_dropdown" class="dropdown">
                           <button></button>
                           <ul>
                                  <li >none</li>
                                  <li >Disable</li>
                                  <li >Horizontal</li>
                                  <li >Vertical</li>
```

```
                                <li >All</li>
                          </ul>
                    </div>
              </div>
              <div class="toolset" id="tool_move">
                    <label title="Change move option">Move:
                          <!-- Move -->
                    <input id="move" type="text" title="Change move" size="6"
class="attr_changer" data-attr="move"/>
                    </label>
                    <div id="move_dropdown" class="dropdown">
                          <button></button>
                          <ul>
                                <li >none</li>
                                <li >Static</li>
                                <li >horizontal</li>
                                <li >Vertical</li>
                                <li >All</li>
                          </ul>
                    </div>
              </div>
              <div class="toolset" id="tool_Z-Position">
                    <label title="Change move option">Z-Position:
                          <!-- Layer -->
                    <input id="Z-Position" type="text" title="Change Special
Layer" size="3" class="attr_changer" data-attr="Z-Position"/>
                    </label>
                    <div id="Z-Position_dropdown" class="dropdown">
                          <button></button>
                          <ul>
                                <li >none</li>
                                <li >1</li>
                                <li >2</li>
                                <li >3</li>
                          </ul>
                    </div>
              </div>
```

- Text section:

```
<div class="toolset" id="tool_text_pos">
                    <label title="Change text position option">Text Position:
                          <!-- text position alterei-->
                    <input id="text_pos" type="text" title="Change Text
Position" size="12"/>
                    </label>
                    <div id="text_pos_dropdown" class="dropdown">
                          <button></button>
                          <ul>
                          <li style="text_pos:topleft">topleft</li>
                          <li style="text_pos:topcenter">topcenter</li>
                          <li style="text_pos:topright">topright</li>
                          <li style="text_pos:midleft">midleft</li>
                          <li style="text_pos:midcenter">midcenter</li>
                          <li style="text_pos:midright">midright</li>
                          <li style="text_pos:botleft">botleft</li>
                          <li style="text_pos:botcenter">botcenter</li>
                          <li style="text_pos:botright">botright</li>
                          <li style="text_pos:none">none</li>
                          </ul>
                    </div>
              </div>
```

- **svg-editor.js -**Added new functions to work with the new attributes, like Rescale, Move, Z-position and Position

Normal elements, multiples uses in code:

```
$('#rescale').val(selectedElement.getAttribute("rescale") ||"none" );
$('#Z-Position').val(selectedElement.getAttribute("Z-Position") ||"none" );
$('#move').val(selectedElement.getAttribute("move") ||"none" );
….
```

Text section

```
…….

$('#rescale').val(elem.getAttribute("rescale"));

       $('#Z-Position').val(elem.getAttribute("Z-Position"));

       $('#move').val(elem.getAttribute("move"));

       $('#text_pos').val(elem.getAttribute("text_pos"));

….
```

Permit section

```
…….
if (attr !== "id") {//change to permit for new atributes
                    if (attr === "move") {}
                    else   if (attr === "rescale")
                    {}
                    else   if (attr === "Z-Position")
                    {}
                    else   if (isNaN(val)) {
………..
```

Activate code for dropdown menus in the html section

```
…..
Editor.addDropDown('#rescale_dropdown', function() {
                    var resc = $(this).text();
       $('#rescale').val($(this).text()).change();
                    });
Editor.addDropDown('#move_dropdown', function() {
                           var resc2 = $(this).text();
                           $('#move').val($(this).text()).change();
                    });
Editor.addDropDown('#Z-Position_dropdown', function() {
                           var Z_Position = $(this).text();
                           $('#Z-Position').val($(this).text()).change();
                    });
Editor.addDropDown('#text_pos_dropdown', function() {
                           var resc3 = $(this).text();
                           $('#text_pos').val($(this).text()).change();
              if(resc3=="topleft"){svgCanvas.alignSelectedElements("t",
'page');svgCanvas.alignSelectedElements("l", 'page');};

       if(resc3=="topcenter"){svgCanvas.alignSelectedElements("t",
'page');svgCanvas.alignSelectedElements("c", 'page');};

       if(resc3=="topright"){svgCanvas.alignSelectedElements("t",
'page');svgCanvas.alignSelectedElements("r", 'page');};

       if(resc3=="midleft"){svgCanvas.alignSelectedElements("m",
'page');svgCanvas.alignSelectedElements("l", 'page');};

       if(resc3=="midcenter"){svgCanvas.alignSelectedElements("m",
'page');svgCanvas.alignSelectedElements("c", 'page');};

       if(resc3=="midright"){svgCanvas.alignSelectedElements("m",
'page');svgCanvas.alignSelectedElements("r", 'page');};

       if(resc3=="botleft"){svgCanvas.alignSelectedElements("b",
'page');svgCanvas.alignSelectedElements("l", 'page');};
```

```
        if(resc3=="botcenter"){svgCanvas.alignSelectedElements("b",
'page');svgCanvas.alignSelectedElements("c", 'page');};

        if(resc3=="botright"){svgCanvas.alignSelectedElements("b",
'page');svgCanvas.alignSelectedElements("r", 'page');};
                    });
…..
```

- **svgcanvas.js**- Added to the SVG elements the new attributes that could be used, like Rescale, Move, Z-position and Position with their basic value

```
…..
svgedit.utilities.assignAttributes(shape, {
                    "fill": cur_shape.fill,
                    "stroke": cur_shape.stroke,
                    "stroke-width": cur_shape.stroke_width,
                    "stroke-dasharray": cur_shape.stroke_dasharray,
                    "stroke-linejoin": cur_shape.stroke_linejoin,
                    "stroke-linecap": cur_shape.stroke_linecap,
                    "stroke-opacity": cur_shape.stroke_opacity,
                    "fill-opacity": cur_shape.fill_opacity,
                    "opacity": cur_shape.opacity / 2,
                    //code added
                    "rescale":cur_shape.rescale,
                    "Z-Position":cur_shape.Z_Position,
                    "move":cur_shape.move,
                   //
                    "style": "pointer-events:inherit"
             }
…..
```

- **sanitize.js** -Added to the SVG elements the new attributes that could be used, like Rescale, Move, Z-position and Position