WOW - Wiki Aided Organization DraWing

MASTER DISSERTATION

Roberto Milton Gouveia Nunes

MASTER IN INFORMATICS ENGINEERING

UNIVERSIDADE da MADEIRA

A Nossa Universidade

www.uma.pt

december | 2013

WOW - Wiki Aided Organization DraWing

MASTER DISSERTATION

Roberto Milton Gouveia Nunes

MASTER IN INFORMATICS ENGINEERING

SUPERVISOR

David Sardinha Andrade de Aveiro

Acknowledgements

I would like to thank to the following:

1. Prof. Dr. David Aveiro. My supervisor, his critiques and guidance throughout the elaboration of this thesis were extremely invaluable.
2. My colleagues Vítor Nóbrega and Duarte Pinto. The exchange of ideas has proven to be most helpful in many aspects of this dissertation.
3. My family, friends and colleagues who have supported me in many ways through this difficult journey. Special thanks to my father Arsénio Nunes, my mother Susana Nunes and my sister Cláudia Nunes, and to my colleagues Aquilino Viveiros and Artur Vieira who were always there since the beginning of this journey.

Resumo

À medida que o mundo evolui, as organizações tornam-se progressivamente mais complexas, e a necessidade de compreender essa complexidade aumenta de igual forma. Com esta procura, surge a engenharia organizacional, que é uma área que aparece com o intuito de tornar as organizações mais fáceis de compreender, tentando-se colocar em prática o conceito de consciência organizacional, que significa que todos os colaboradores que fazem parte de uma organização precisam de perceber como a mesma funciona e qual o seu papel nela. A metodologia DEMO (Metodologia de Engenharia de Design para Organizações) surgiu com o propósito de representar a consciência organizacional, através da definição e criação de diagramas coerentes e consistentes.

Wikis semânticos têm funcionalidades que podem ajudar na modelação de empresas. O UEAOM (Modelo Adaptativo de Objetos Universal para Empresas) é um modelo que permite a especificação e evolução dinâmica de linguagens, meta-modelos, modelos e as suas representações na forma de diagramas e tabelas.

Neste projeto, foi implementado um Sistema baseado no UEAOM, e no Semantic Media Wiki, que permite a criação e edição gráfica de diagramas. O UEAOM pode ser visto ao nível do meta-modelo, onde uma linguagem é definida, e ao nível do modelo, onde instâncias de classes desse modelo são criadas. O sistema desenvolvido por nós foca-se ao nível do modelo, mas usa como base o projeto que se foca no meta-modelo.

A linguagem DEMO foi utilizada como exemplo para a implementação e testes de um editor gráfico baseado nas tecnologias web e SVG, integrado com o Semantic Media Wiki para permitir a edição de diagramas de organizações e navegação nos mesmos de uma forma intuitiva, coerente e consistente.

Palavras-chave

Engenharia Organizacional, UEAOM, SemanticMediaWiki, DEMO, SVG, Editor de Diagramas Universal

Abstract

As the world evolves, organizations are becoming more and more complex, and the need to understand that complexity is increasing as well. With this demand, arises organizational engineering, which is a subject that emerged with the purpose to make organizations easier to understand, by putting in practice the concept of organizational self-awareness, which means that that the collaborators who are part of an organization, need to understand it and know what their role in it is. The DEMO methodology (Design Engineering Methodology for Organizations), came up with the purpose of representing these organizations' self-awareness, through the definition and creation of consistent and coherent diagrams.

Semantic wikis have features that can help in enterprise modelling. UEAOM (Universal Enterprise Adaptive Organization Model) is a model that allows the specification and dynamic evolution of languages, meta-models, models, and their representations as diagrams and tables.

In this project, it was implemented a system based on UEAOM, and Semantic Media Wiki which allows a graphical creation and edition of diagrams. UEAOM can be divided into the meta-modeling level where a language is defined, and the modelling level where instances of classes of that language are created. The system we developed focuses on the modeling level, but will takes as a basis the project that focuses on meta-modeling.

The DEMO language was used as an example for the implementation and tests of a graphical editor, based in web technologies and SVG, integrated with SemanticMediaWiki to allow an intuitive, coherent and consistent navigation and editing of organization diagrams.

Keywords

Organizational Engineering, UEAOM, SemanticMediaWiki, DEMO, SVG, Universal Diagram Editor

Contents Index

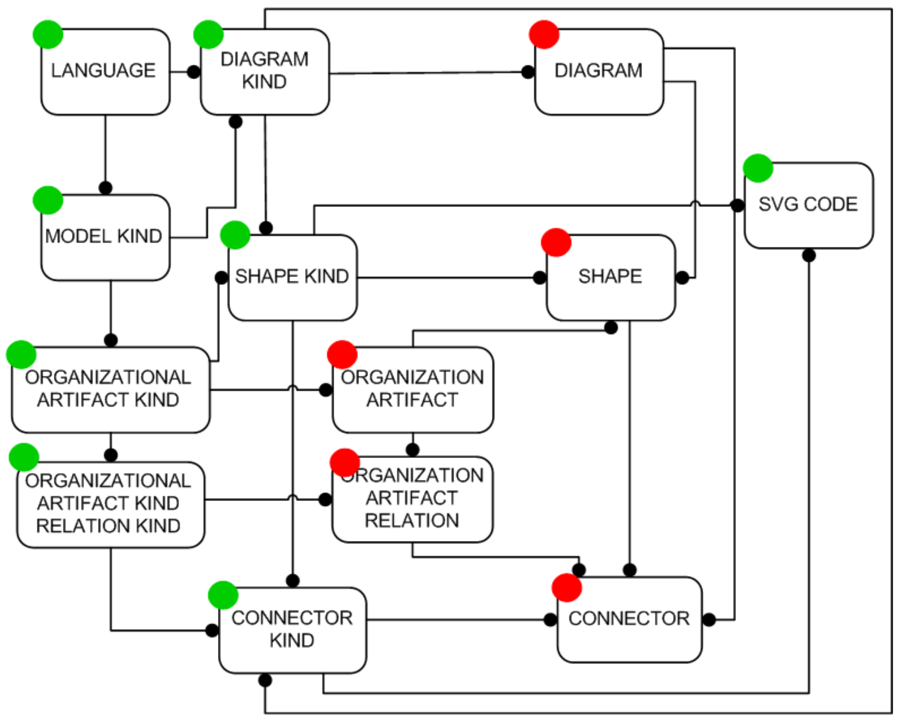
Figures Index

Tables Index

1. Introduction

This project focuses on the development of a web-based diagram editor, named as UDE (Universal Diagram Editor). The initial purpose is to describe how organizations work, and provide simplified views through different kinds of diagrams. This editor should also allow the edition of any type of diagram in the future.

This project consists in the implementation of part of the UEAOM, which is a model that systematizes the integrated management and adaptation of enterprise models, their representations, their underlying meta-models (abstract syntax), and their representation rules (concrete syntax) [1]. To accomplish this goal, we used SMW (Semantic MediaWiki), which is the software used by the well-known Wikipedia, as a basis. Each class of the UEAOM model corresponds to a wiki page or a wiki property on SMW.



Line explanation:

——●	Means instances of the class with the black dot need to refer to instances of the opposite class in order to be created or used
🟢	Page Created in the Meta-editor
🔴	Page Created in the Diagram-editor

Figure 1 - UEAOM (simplified version)

The UEAOM model was being improved at the same time this project was developed. Figure 1, from [31], depicts a simplified version of UEAOM (full version can be found in Figure 15).

It’s also important to say that this project, named as WOW (Wiki Organization draWing), was developed at the same time as the WAMM project (Wiki-Aided Meta Modeling) [31]. WAMM was integrated in this project, and can be seen as a basis to WOW, because it defines classes that will be used by the UDE, as we can view in Figure 1. The classes marked with a red circle correspond to the classes which are created by the UDE, and these depend on the classes that are created by the meta-model editor, marked with a greed circle. As we can observe in the picture, the meta-model editor defines, for instance, the diagram types, shape types, and connector types, and the UDE creates instances of those classes, namely the diagrams, shapes, and connectors, etc, to use in the UDE.

1.1. Motivation

Nowadays, nearly 75% of the computer engineering projects fail to meet the user’s expectations. One of the main causes is an insufficient or inadequate knowledge of the organizational reality to be automated or supported by an information system. The organizational engineering discipline emerged in the 90s and adds concepts and engineering methods applied to organizations with the goal of understanding and representing multiple facets of them, as well as make the analysis and organizational changes easier, regardless of the implementation of the information systems. Semantic wikis are easy to use for people with a low computer knowledge. We want it to be possible that all the employees of a given organization are able to contribute to the creation of a collective awareness of the organizational reality through the use of semantic wikis. This tool enables a distributed and coherent collection of organizational knowledge in the form of elements and semantic relations aligned with the organizational reality, that allow the capture and monitoring of its evolution, as well as a more efficient development of ISs that support such reality.

1.2. Objectives

- Analysis of a MediaWiki and SemanticMediaWiki based prototype and extension of the prototype to allow visualization and incremental creation and edition of organizational diagrams.
- Development of a web-based diagram editor, and integration of the editor in SemanticMediaWiki, taking as a basis the existing editor on the Open Modeling platform, the Modelword diagram editor and other existing solutions, and good design practices implemented in these tools as well as on Microsoft Visio. The technologies used were: JavaScript, AJAX and SVG for a maximum interactivity and flexibility with the user interaction, with functionalities such as auto-complete. This project extends an already existing extension that allows the incremental edition of models based on textual descriptions, forms and semantic web. Application and validation of the extension, using the "EU-Rent case" [23] as an example. The theoretical and methodological basis to use in this project is the DEMO (Design Engineering Methodology for Organizations) methodology, promoted by Enterprise Engineering Institute with headquarters in the Netherlands and with whom M-ITI has a partnership.

1.3. Content

In this sub-section, it is provided a brief description of the content of this document. In chapter 2, «Context», there are some organizational engineering theoretical concepts which are needed and are a good basis for the reader to understand further chapters. Still in chapter 2 there is a description of the UEAOM, which is a model that systematizes the integrated management and adaptation of enterprise models, their representations, their underlying meta-models (abstract syntax), and their representation rules (concrete syntax). In chapter 3, «State-of-the-art» it is describe the research that was done, where we present some already existing web-based diagramming solutions and choose the most adequate one to use in this project. In Chapter 4, «Software and programming languages» we make an introduction and description of all the software and programming languages that were used in this project. In chapter 5, «DEMO Methodology on MediaWiki», we describe how the implementation of this project is done on MediaWiki, i.e., definition of wiki pages, and how forms and templates are used. In chapter 6, «Universal Diagram Editor Implementation», we describe some the implementation of the UDE (Universal Diagram Editor), and all its features. Next we have chapter 7, «Conclusion», and chapter 8, «References», and finally, on chapter 9 - «Appendix», we have the installation manual of this project, and the HTML and JavaScript/JQuery code that was developed.

2. Context

In this section we begin with an introduction to Organizational Engineering. We start with some theoretical concepts, followed by a brief description of the Grammar of WOSL, explained in the next sub-section. Then we proceed to the “Universal Enterprise Adaptive Object Model”, which is developed using this grammar. Both UEAOM and this project (which implements part of the UEAOM), were developed at the same time and influenced one another.

2.1. Enterprise Ontology Theoretical Concepts

A widely adopted definition of ontology is that an 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. An 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 context 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. [2]

2.2. World Ontology Specification Language (WOSL)

2.2.1. Factual Knowledge

2.2.1.1. The Ontological Parallelogram

Factual knowledge is the knowledge about the states and state changes of a world, like knowing that a person, 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 2. It explains how people use signs as representations of objects in order to be able to communicate about these objects in their absence.

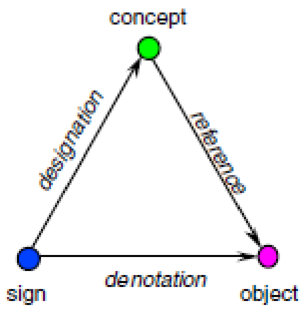


Figure 2 - The meaning triangle

The elementary notions that we 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 are 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 for example or the composite object denoted by "Ludwig Wittgenstein owns car 16-EX-AF". 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. Example of a concrete concepts: the mental picture I have of the person Ludwig Wittgenstein. Examples of an abstract concept: 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, denotation, and reference. **Designation** 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** 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. Examples: the concept Ludwig Wittgenstein refers to a particular person.

A **class** is a collection of objects. By definition a class contains all objects that conform to the associated type. Examples: the class of persons, i.e., the collection of objects that share those properties that make them conform to the type person, the class of cars, i.e., the collection of objects that conform to the type car, and the collection of object pairs <person, car> that share the property that the person owns the car. **Extension** is a relationship between a type and a class. We say that a class is the extension of a type. Examples: the class persons is the extension of the type person; the class cars is the extension of the type car; the class ownerships is the extension of the type owns. The relationships between **individual concepts** and **generic concepts** (types), and consequently between individual objects and classes are depicted in Figure 3. In this figure, which is based on Figure 2, we have deliberately left out the signs (predicate names and proper names) because they are not relevant in ontology. Ontology is about the essence of things, not about how we 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.

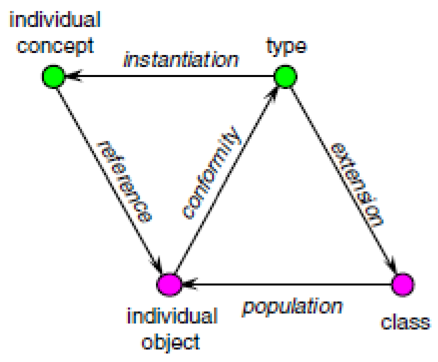


Figure 3 - The ontological parallelogram

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. [12]

2.2.1.2. 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”.

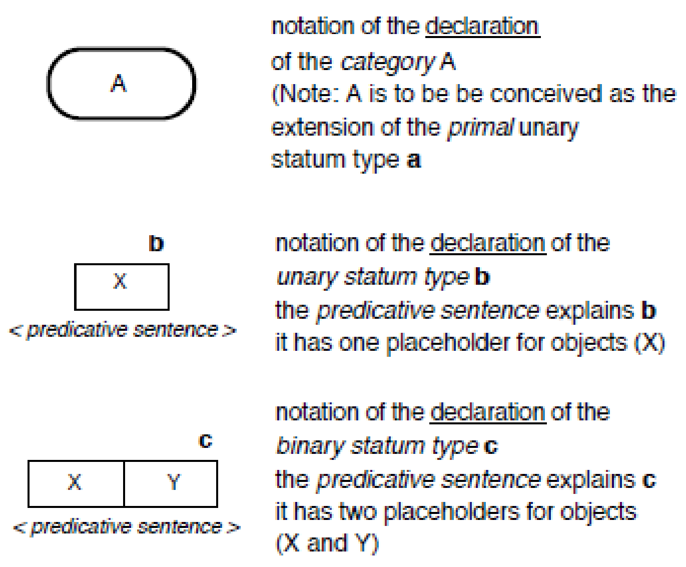
Contrary to a statum, a **factum** is the result or the effect of an act. Example: book title T has been published. The becoming existent of a factum is a transition. Before the occurrence of the transition, it did not exist and after the occurrence it does exist. Facta are subject to occurrence laws. 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”. [12]

2.2.1.3. World Ontology

We are now able to provide a precise definition of the ontology of a world: a 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. [2]

2.2.1.4. 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 4 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][17]



notation of the declaration of the *category* A
(Note: A is to be be conceived as the extension of the *primal* unary statum type **a**

notation of the declaration of the *unary statum type* **b**
the *predicative sentence* explains **b**
it has one placeholder for objects (X)

notation of the declaration of the *binary statum type* **c**
the *predicative sentence* explains **c**
it has two placeholders for objects (X and Y)

Figure 4 - Statum type declarations

Fig 1 and 2 show the specification of existence laws.

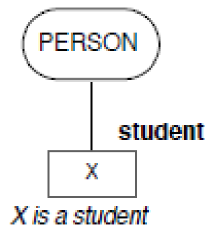


Figure 5 - Example of a reference law

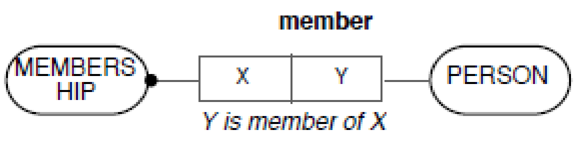


Figure 6 - Example of a dependency law

Figure 7 shows an example of a factum type.

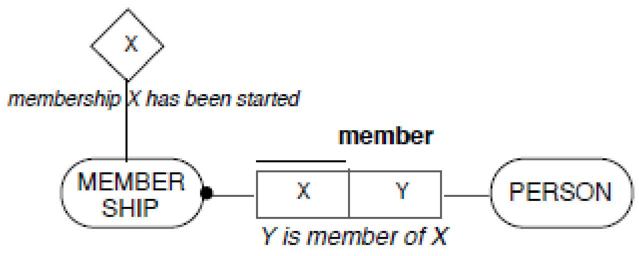


Figure 7 - Example of a factum type

2.3. Basic Ontological Notions

The ontological system definition was adopted from [24] 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 following 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 [24] 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.

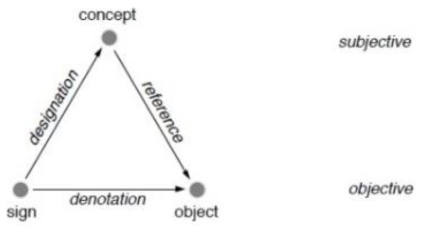


Figure 8 - The meaning triangle

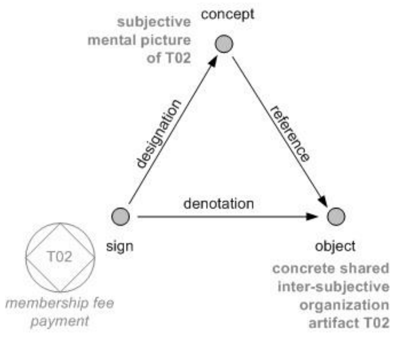


Figure 11 - Meaning triangle applied to a transaction OA

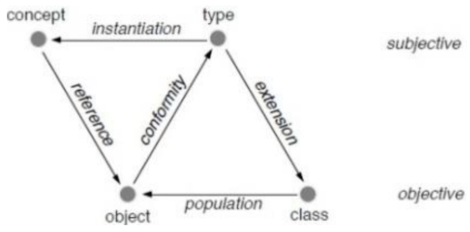


Figure 9 - The ontological parallelogram

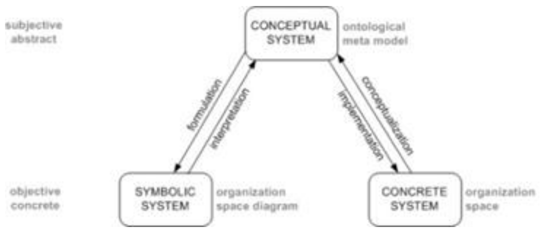


Figure 12 - Model triangle applied to the organization space

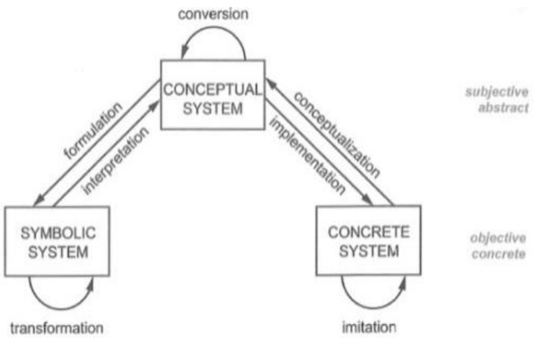


Figure 10 - Model triangle applied to organizations

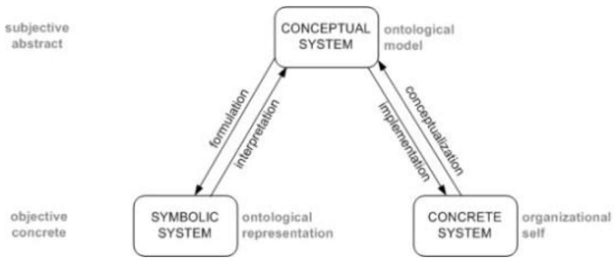


Figure 13 - The model triangle

In Figure 8 and Figure 9, we find, respectively, the meaning triangle and the ontological parallelogram, taken from [25], which explain how (individual) concepts are created in the human mind. We will also base our claims in the model triangle, taken from [26] and presented in Figure 13. 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 – allow 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 10 depicts an adaptation from the model triangle of Figure 13 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 11, 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 (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 [3] 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 UDE (Universal Diagram Editor) that was developed in this project is to be used by the GO of each organization, and other employees other than the ones who belong to the GO, should be able to view the diagrams, in order to be aware of the organizational self.

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) [12] – 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 [27] 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 [26], 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 12, another adaptation from the model triangle. The G.O.D. organization is addressed in detail in [3]. 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.

2.4. 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. 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. The abstract syntax of a model manages the formal structure of the model elements and the relationships amongst them.

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. 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. [4]

2.5. The Universal Enterprise Adaptive Object Model

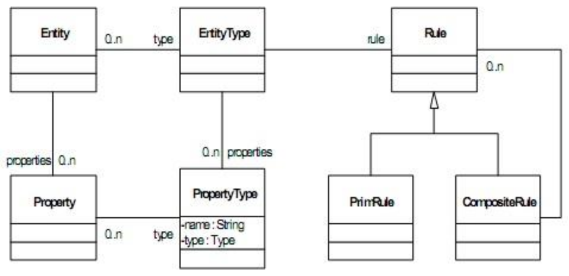


Figure 14 - Type Square

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 15. 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 that are part of the corresponding diagram. These wiki pages also allow dynamic editing of diagrams and underlying models. We also apply the type-square pattern – depicted in Figure 14 – 4 times as to allow run-time dynamic change of: (1) meta-model elements, (2) model elements, (3) shape elements and (4) connector elements. The UEAOM is represented with the World Ontology Specification Language (WOSL). WOSL is based in Object Role Modeling language which is also used as a base for the specification of the anatomy of Archimate, a similar effort to ours. In [29], 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 [29], 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 the initiative in [29] 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. This is why we named our editor UDE (Universal Diagram Editor), which means it can create and edit any kind of diagram. For a better understanding and following the essential and important validation by instantiation principle [27] 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 this proposal.

2.5.1. Abstract Syntax

Relevant classes for the specification of the abstract syntax of any version of any language are presented in Figure 16. 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 (OAKRK). They specify all allowed artifacts (e.g. transaction kind OAK and transaction execution relation OAKRK) 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, Unicity 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, Unicity 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 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>.

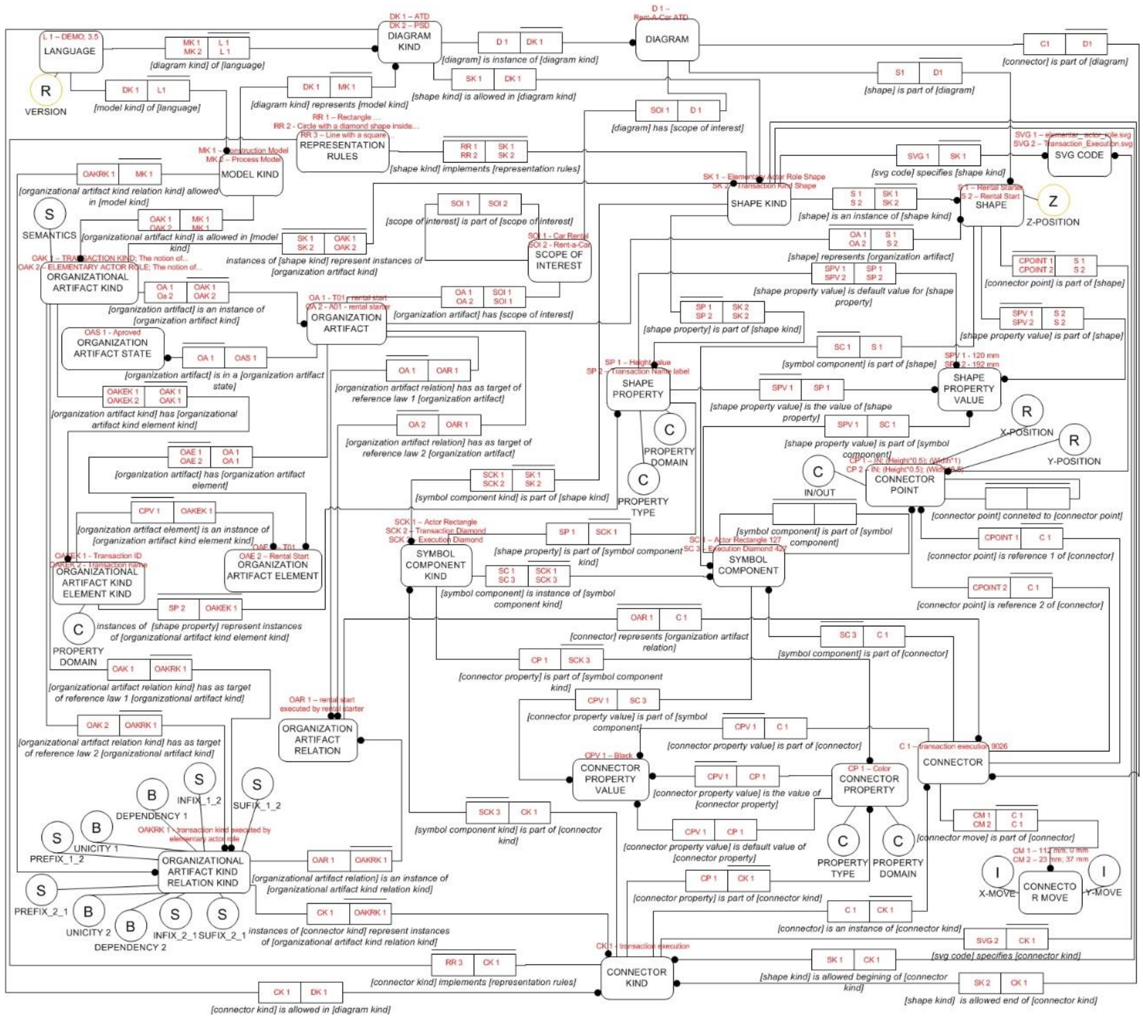


Figure 15 - Universal Enterprise Adaptive Object Model

In Figure 17 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 17 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.

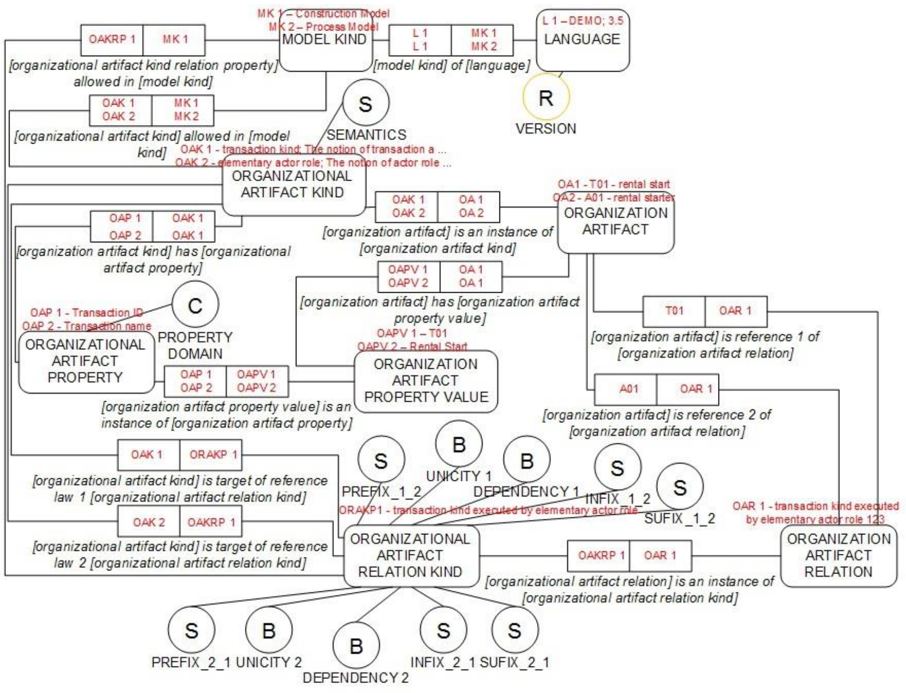


Figure 16 - UEAOM - Abstract Syntax classes

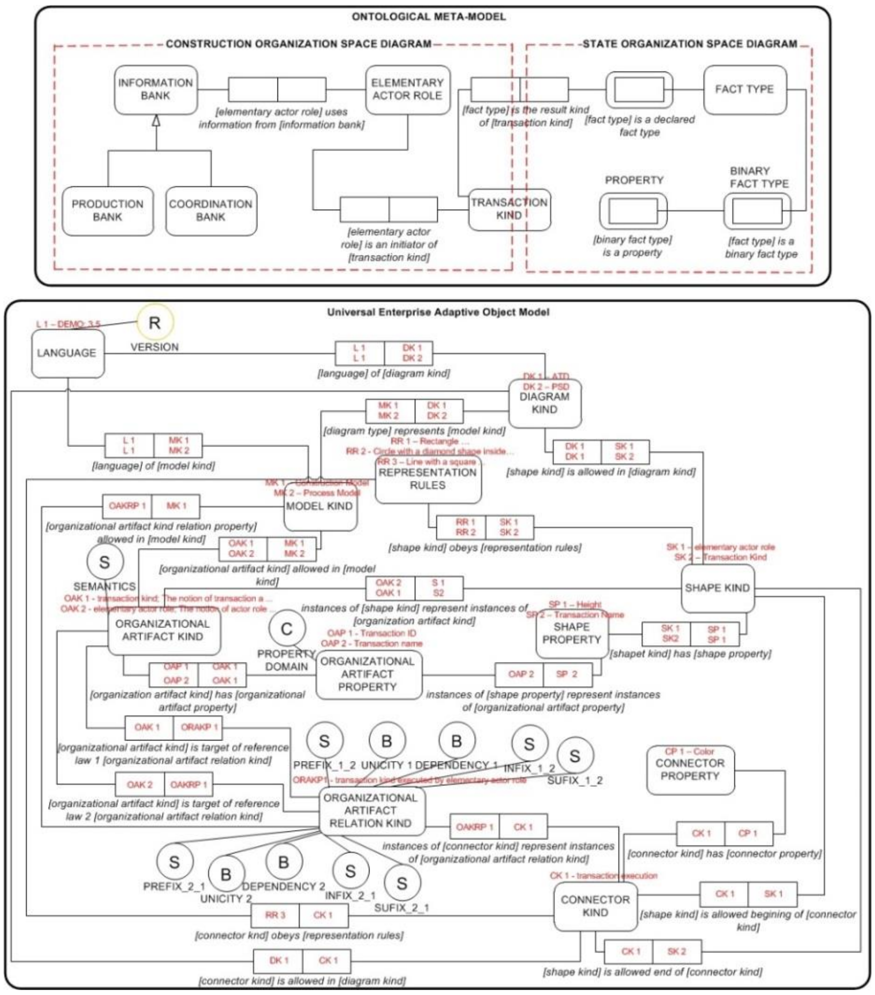


Figure 17 - DEMO Ontological Meta-Model and UEAOM classes used to represent it

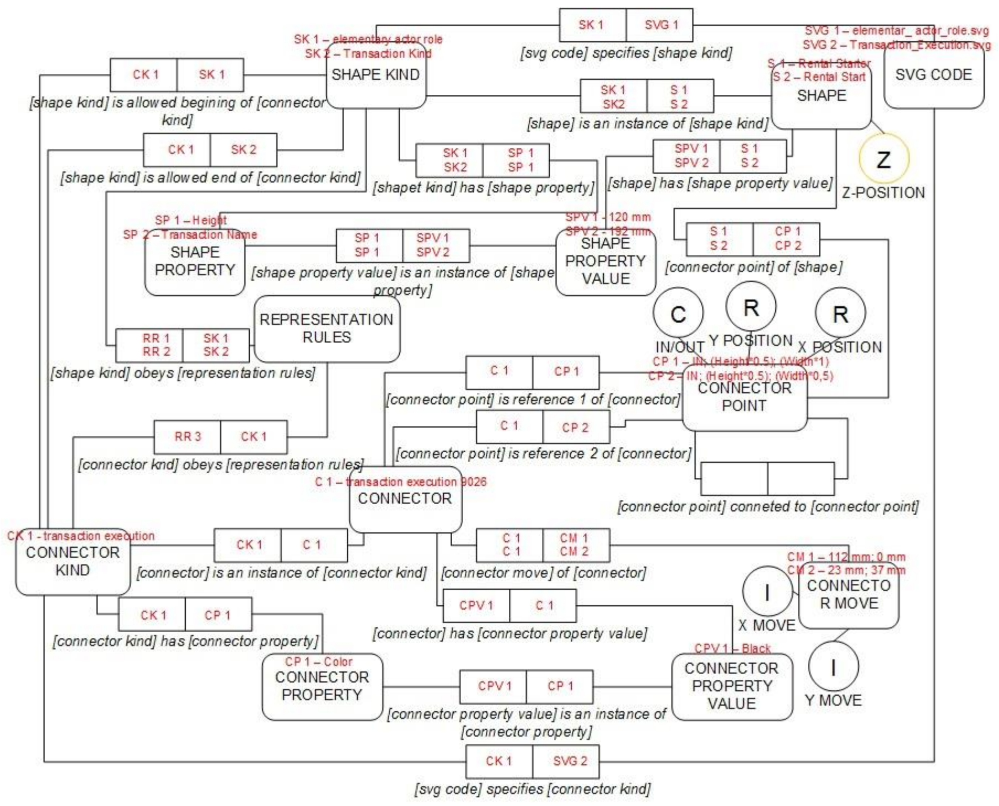

SK 1 – elementary actor role
SK 2 – Transaction Kind
SHAPE KIND
SK 1 | SVG 1
[svg code] specifies [shape kind]
SVG 1 – elementar_ actor_role.svg
SVG 2 – Transaction_Execution.svg
SVG CODE
CK 1 | SK 1
[shape kind] is allowed begining of [connector kind]
SK 1 SK2 | S 1 S 2
[shape] is an instance of [shape kind]
S 1 – Rental Starter
S 2 – Rental Start
SHAPE
CK 1 | SK 2
[shape kind] is allowed end of [connector kind]
SK 1 SK2 | SP 1 SP 1
[shapet kind] has [shape property]
SPV 1 SPV 2 | S 1 S 2
[shape] has [shape property value]
Z
Z-POSITION
SP 1 – Height
SP 2 – Transaction Name
SHAPE PROPERTY
SP 1 SP 1 | SPV 1 SPV 2
[shape property value] is an instance of [shape property]
SPV 1 - 120 mm
SPV 2 - 192 mm
SHAPE PROPERTY VALUE
S 1 S 2 | CP 1 CP 2
[connector point] of [shape]
C
IN/OUT
R
Y POSITION
R
X POSITION
REPRESENTATION RULES
RR 1 RR 2 | SK 1 SK 2
[shape kind] obeys [representation rules]
C 1 | CP 1
[connector point] is reference 1 of [connector]
CP 1 – IN; (Height*0.5); (Width*1)
CP 2 – IN; (Height*0.5); (Width*0,5)
CONNECTOR POINT
RR 3 | CK 1
[connector knd] obeys [representation rules]
C 1 | CP 2
[connector point] is reference 2 of [connector]
C 1 – transaction execution 9026
CONNECTOR
[connector point] conneted to [connector point]
CK 1 - transaction execution
CONNECTOR KIND
CK 1 | C 1
[connector] is an instance of [connector kind]
C 1 C 1 | CM 1 CM 2
[connector move] of [connector]
CM 1 – 112 mm; 0 mm
CM 2 – 23 mm; 37 mm
CONNECTO R MOVE
I
X MOVE
I
Y MOVE
CK 1 | CP 1
[connector kind] has [connector property]
CPV 1 | C 1
[connector] has [connector property value]
CP 1 – Color
CONNECTOR PROPERTY
CPV 1 | CP 1
[connector property value] is an instance of [connector property]
CPV 1 – Black
CONNECTOR PROPERTY VALUE
CK 1 | SVG 2
[svg code] specifies [connector kind]

Figure 18 - UEAOM Concrete syntax

2.5.2. 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 18. 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 OAKRKs. 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 OAKRK 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 2 - The meaning triangle, 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.

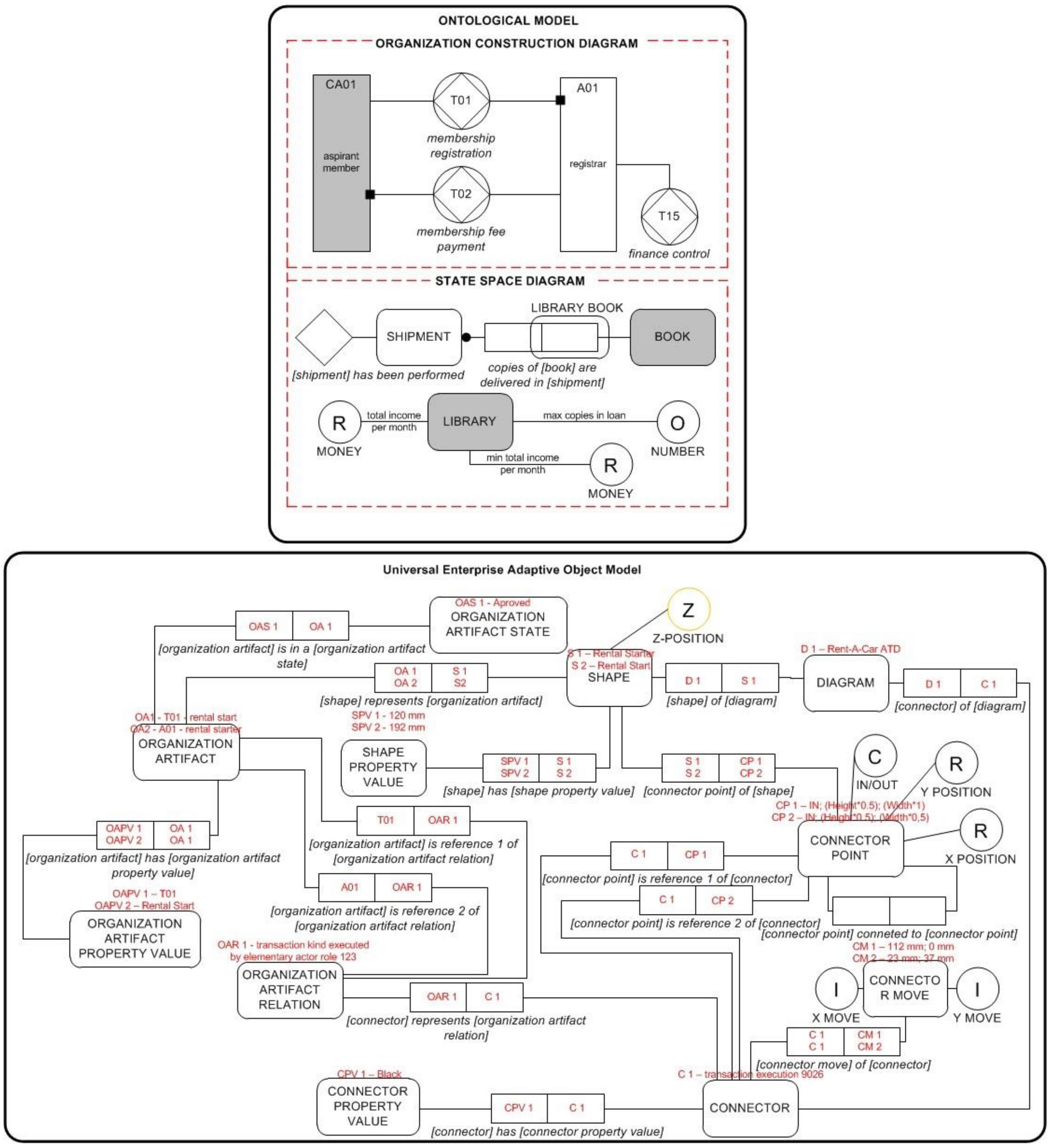

ONTOLOGICAL MODEL
ORGANIZATION CONSTRUCTION DIAGRAM
CA01
aspirant member
T01
membership registration
T02
membership fee payment
A01
registrar
T15
finance control
STATE SPACE DIAGRAM
SHIPMENT
[shipment] has been performed
LIBRARY BOOK
copies of [book] are delivered in [shipment]
BOOK
R
MONEY
total income per month
LIBRARY
max copies in loan
O
NUMBER
min total income per month
R
MONEY
Universal Enterprise Adaptive Object Model
OAS 1
OA 1
OAS 1 - Aproved
ORGANIZATION ARTIFACT STATE
[organization artifact] is in a [organization artifact state]
Z
Z-POSITION
S 1 – Rental Starter
S 2 – Rental Start
SHAPE
D 1 – Rent-A-Car ATD
DIAGRAM
OA 1
OA 2
S 1
S2
[shape] represents [organization artifact]
D 1
S 1
[shape] of [diagram]
D 1
C 1
[connector] of [diagram]
SPV 1 - 120 mm
SPV 2 - 192 mm
OA1 - T01 - rental start
OA2 - A01 - rental starter
ORGANIZATION ARTIFACT
SHAPE PROPERTY VALUE
SPV 1
SPV 2
S 1
S 2
[shape] has [shape property value]
S 1
S 2
CP 1
CP 2
[connector point] of [shape]
C
IN/OUT
R
Y POSITION
CP 1 – IN; (Height*0.5); (Width*1)
CP 2 – IN; (Height*0.5); (Width*0,5)
OAPV 1
OAPV 2
OA 1
OA 1
[organization artifact] has [organization artifact property value]
T01
OAR 1
[organization artifact] is reference 1 of [organization artifact relation]
C 1
CP 1
CONNECTOR POINT
R
X POSITION
[connector point] is reference 1 of [connector]
OAPV 1 – T01
OAPV 2 – Rental Start
ORGANIZATION ARTIFACT PROPERTY VALUE
A01
OAR 1
[organization artifact] is reference 2 of [organization artifact relation]
C 1
CP 2
[connector point] is reference 2 of [connector]
[connector point] conneted to [connector point]
CM 1 – 112 mm; 0 mm
CM 2 – 23 mm; 37 mm
OAR 1 - transaction kind executed by elementary actor role 123
ORGANIZATION ARTIFACT RELATION
OAR 1
C 1
[connector] represents [organization artifact relation]
I
X MOVE
CONNECTOR MOVE
I
Y MOVE
C 1
C 1
CM 1
CM 2
[connector move] of [connector]
C 1 – transaction execution 9026
CPV 1 – Black
CONNECTOR PROPERTY VALUE
CPV 1
C 1
[connector] has [connector property value]
CONNECTOR

Figure 19 - DEMO concrete diagrams example and UEAOM classes used to represent them

In Figure 19 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 19, 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 the 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. State-of-the-art

3.1. Alternative Web-based Diagramming Solutions

In this chapter we make a review of some already existing solutions of web-based diagram editors. We will see how those editors load shapes, how they save information about shapes while working on the editor, and how they export information about shapes when saving the diagram.

3.1.1. Modelworld

Modelworld is an online diagramming tool that runs in a browser, does not require installation or plugins. Users can access the diagrams anywhere. It also allows multi-user access. It has specific support for the following languages: DEMO, Archimate, BPMN, IDEF and UML. Figure 20 illustrates modelworld user interface.

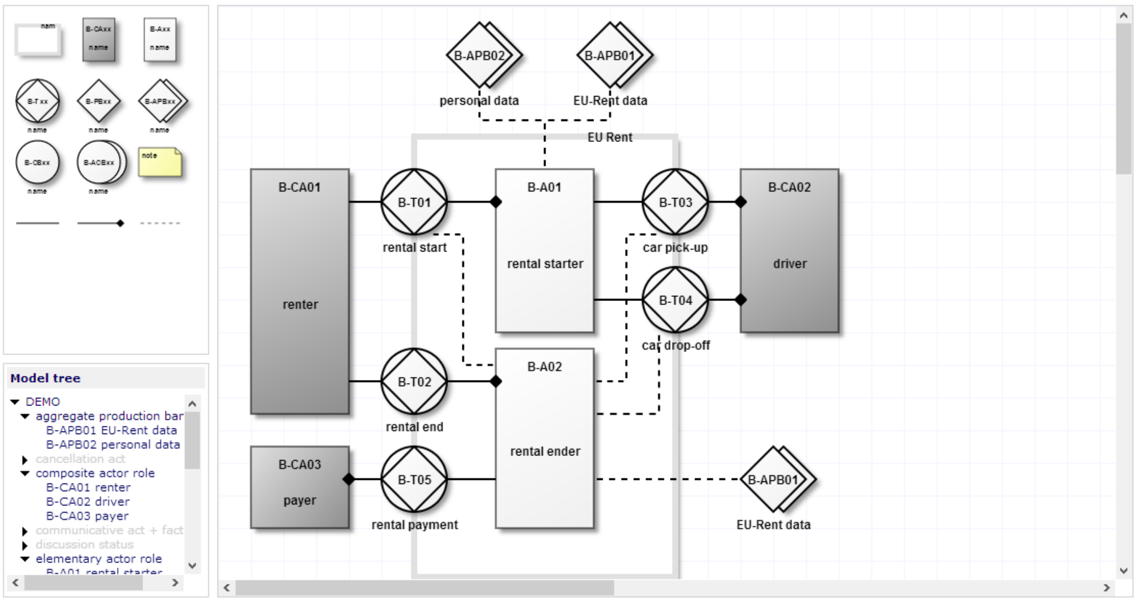


Figure 20 - Modelword User Interface

Modelword uses HTML5 canvas with JavaScript, and it also uses Ajax for asynchronous requests to the server, by the client's machine. The server uses PHP and MySQL. Information about shapes is kept in a MySQL database. Every time the user creates, edits, or deletes a shape, a request is made to the database, and the corresponding database tables of that shape are edited. For internet explorer 7 and 8, that do not support HTML5, it is used JavaScript emulation of HTML5. Figure 21 illustrates modelworld's technical arquitecture layers.

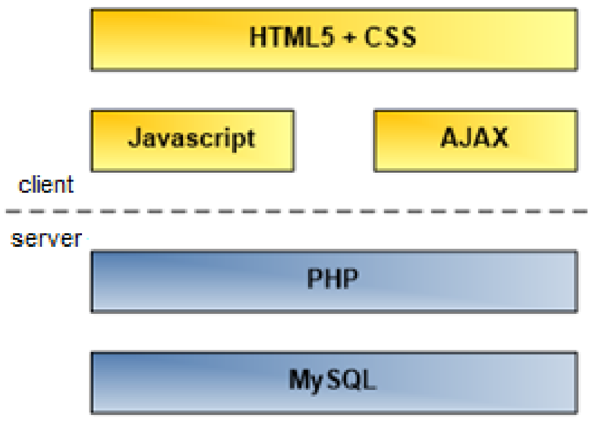


Figure 21 - Modelworld Technical Architecture Layers

When a user wants to open a diagram, an Ajax request is sent to the server, php on the server generates the code to draw the diagram, that code is sent back to the client and executed on the client's machine through the JavaScript eval function, and the diagram is generated. The generated JavaScript code includes functions like "drawIcon" and "drawLine".

A simple example that involves modifying data, would be adding a new shape. To create a new shape, an Ajax request is made from the client machine. Then a php script handles that request by calling a function to create the shape and insert it on the database. Then a query is made to the database to request that shape's properties, and a php function generates the code to create the shape. That code is sent back to the client and executed through the JavaScript eval function. This also helps reducing the server load. [18]

3.1.2. Diagramo

Diagramo is an online diagramming tool implemented with pure HTML5 and JavaScript. No flash java or other plugins. It allow collaborative diagram editing. Diagrams can be exported in three different formats – SVG, GIF and JPEG. It is an open source project. Figure 22 illustrates the user interface of diagramo web-editor.

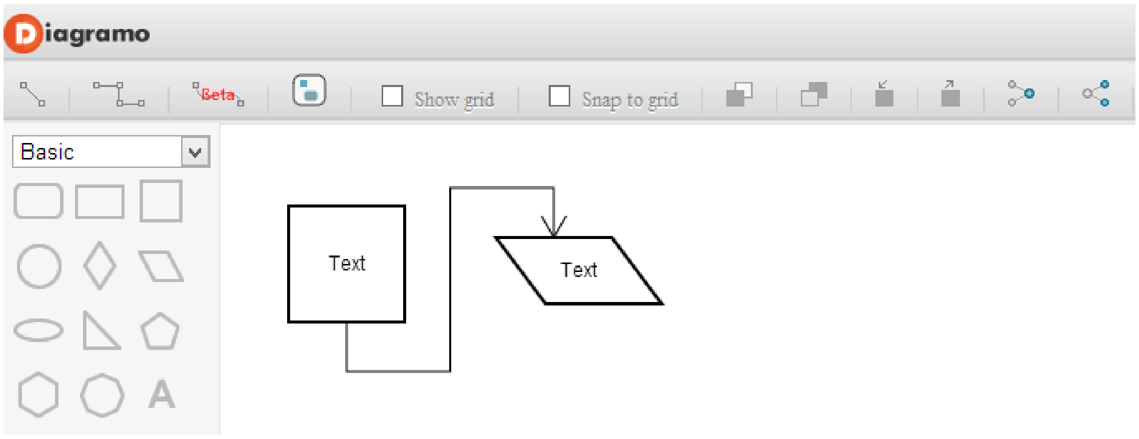


Figure 22 - Diagramo user interface

It has support for basic shapes like rectangles, circles and polygons. It also has a smart connection handler that automatically adjust when the shapes are being moved. Manual editing of the connector line is also supported. Good support for labels, and group selection. It also allows the grouping of symbols. However, this editor's project has too many files and lines of code, for such a simple editor, and the architecture is difficult to understand, at least without having an arquitecture model. It's a good editor, but only for basic shapes. It would be difficult to add more elaborate and complex shapes to the palette with this kind of architecture.

3.1.3. RaphaelJS

RaphaelJS is a small JavaScript library that simplifies work with vector graphics on the web. It has features such has image crop and rotate, or specific chart creation that can be easily achieved through the library API.

RaphaelJS uses the SVG W3C Recommendation and VML (vector markup language) as a base for creating graphics. This means that every graphical object that is created is also a DOM object. These objects can be modified later and it is also possible to add event handlers. RaphaelJS's goal is to provide an adapter that makes drawing vector art cross-browser compatible and easy. RaphaelJS support the following web browsers: Firefox 3.0+, Safari 3.0+, Chrome 5.0+, Opera 9.5+ and Internet Explorer 6.0+. In Table 1 it is possible to see how to create a canvas, add a circle to it, and modify its color and stroke-color. The code is simple and intuitive.

```
// Creates canvas 320 x 200 at 10, 50
var paper = Raphael(10, 50, 320, 200);
```

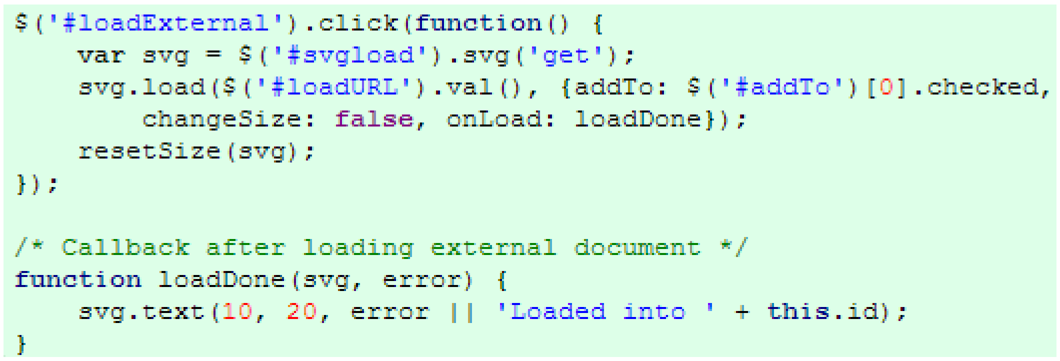
```
// Creates circle at x = 50, y = 40, with radius 10
var circle = paper.circle(50, 40, 10);
// Sets the fill attribute of the circle to red (#f00)
circle.attr("fill", "#f00");

// Sets the stroke attribute of the circle to white
circle.attr("stroke", "#fff");
```

Table 1 - RaphaelJS example

RaphaelJS is a very good library, but we consider it more driven to animations and it still lacks in interactivity, which is what we are trying to achieve. Interactivity is supported, but when trying to create more elaborate examples, the code can be very complex and difficult to understand. [19]

3.1.4. JQuery SVG

JQuery SVG is a JQuery plugin, which allows interaction with an SVG canvas. JQuery itself was initially developed for HTML, and not SVG, although most of the methods work with SVG. JQuery SVG library helps overcoming these problems, such as enabling SVG element selection through its class, just like in HTML. JQuery SVG has some useful functions to work with SVG, such as loading SVG content from an external file to the canvas. Figure 23 illustrates an example.



```
$('#loadExternal').click(function() {
    var svg = $('#svgload').svg('get');
    svg.load($('#loadURL').val(), {addTo: $('#addTo')[0].checked,
        changeSize: false, onLoad: loadDone});
    resetSize(svg);
});

/* Callback after loading external document */
function loadDone(svg, error) {
    svg.text(10, 20, error || 'Loaded into ' + this.id);
}
```

Figure 23 - SVG Load

When the user clicks the button to which the event has been added, the SVG code that is in a certain location, will be loaded to a div, in the case the div with the id “svgload”. This is useful if a shape has to be defined and saved on a file on the server. We can easily “import” that shape to the canvas.

JQuery SVG has also exporting features that allow an SVG element to be exported to a specified div. Figure 24 shows an example of how it can be done. [20]

```
$('#export').click(function() {
    var xml = $('#svgbasics').svg('get').toSVG();
    $('#svgexport').html(xml.replace(/&/g, '&amp;').replace
    (/</g, '&lt;').replace(/>/g, '&gt;'));
});
```

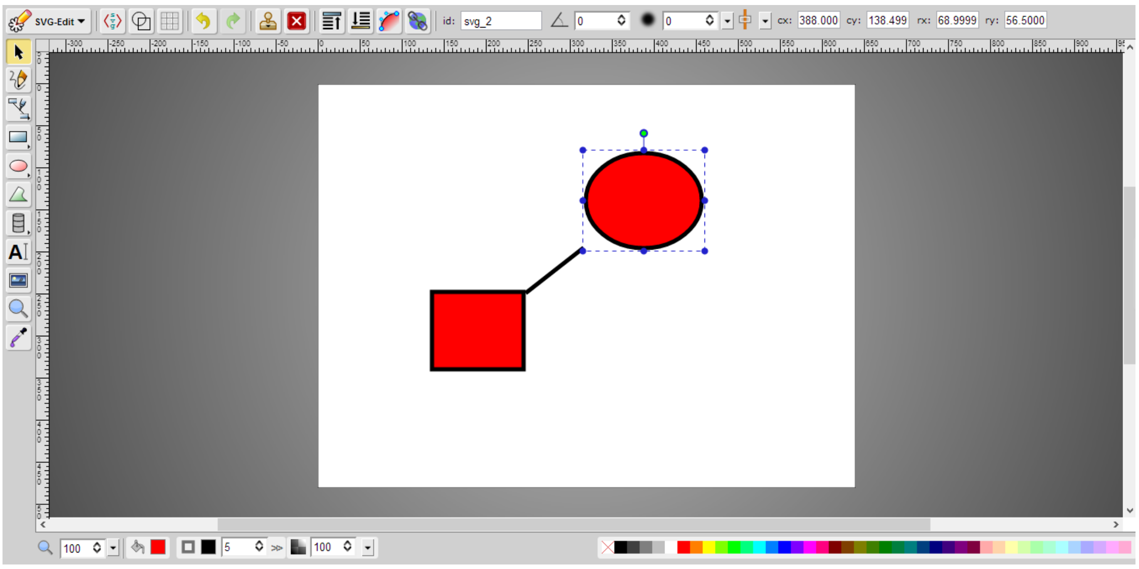
Figure 24 - Jquery SVG Export

Through this function, by clicking on a button, the SVG code that is on a certain div, in this case, “#svgbasis”, can be sent to another div, “#svgexport”. This function could be modified to save the SVG element as a file in a desired location.

In this project, we did not use JQuery SVG, because JQuery itself evolved since the last released version of JQuery SVG, and though JQuery isn’t yet fully compatible with SVG, most of the functions and methods work, and for the tasks we wanted to implement, JQuery alone did the job.

3.1.5. SVG-Edit

SVG-edit is a web-based, open source graphical editor based on SVG and JavaScript. It does not require an installation, only a compatible browser. It runs on the client side. It has browser support for Firefox 1.5+, Opera 9.5+, Safari 4+, Chrome 1+, and Internet Explorer 6+ (with the Chrome Frame plugin). Figure 25 illustrates SVG-edit user interface, with some basic shapes drawn.



id: svg_2 cx: 388.000 cy: 138.499 rx: 68.9999 ry: 56.5000

Figure 25 - SVG-Edit User Interface

3.1.5.1. Components

SVG-edit consists of two major components: svg-editor.js and svgcanvas.js. These components work cooperatively. File svgcanvas.js can be used outside SVG-edit, allowing developers to create alternative interfaces to the canvas. The term canvas is not to be confused with HTML5 canvas element. In SVG-edit, canvas simply refers to a drawing area.

3.1.5.2. Main Features

- Free-hand drawing
- Lines and Polylines
- Rectangles and Squares
- Ellipses and Circles
- Polygons/Curved paths
- Stylable Text
- Raster Images
- Select/move/resize/rotate
- Undo/Redo
- Color/Gradient picker
- Grouping/Ungrouping
- Shape Aligning
- Zoom
- Import SVG into Drawing
- Connector lines and Arrows
- Export Image to PNG

SVG-edit is a very good graphical editor, intuitive, and has many useful features like the ones described before, with good primitives for drawing basic shapes, and also composite and more elaborate shapes. However, SVG-edit is not focused on diagramming, it is more focused in general image drawing. Its support for connectors is still very poor at the moment, as it only allows to draw straight line connectors, and connector points are not implemented. In Figure 25 there is an example of a connector, and as we can observe, the linking to the circle is not perfect, because it is connected to the selection boundary, and not to the circle boundary. But this editor is an interesting solution and has a good margin of progression. [21]

3.1.6. Open Modeling

3.1.6.1. Description

Open modeling is a web-based application to model and publish architecture models, procedures and related structured information. Not only publishing is done on the web, but modeling and maintaining the information (texts and diagrams) is done using the well-known web browser. The modelers of this information can see exactly the same (WYSIWYG) result as those who view the information. Open modeling supports DEMO methodology, Archimate, and other modeling languages.

3.1.6.2. Open Modeling Installation Software

The open modeling editor works under a server which requires the following software:

- **Apache Tomcat** – an open-source software implementation of the Java Servlet and JavaServer Pages technologies. The Java Servlet and JavaServer Pages specifications are developed under the Java Community Process.
- **Java Development Kit** – an Oracle Corporation product aimed Java developers. It is composed by a compiler and libraries.
- **MySQL Essential** – Referred to as Structured Query Language is a programming language designed for managing data in relational database management systems (RDBMS). Is scope includes data insert, query, update and delete, schema creation and modification, and data access control.

3.1.6.3. Usage Scenarios

An example scenario of the use of Open Modeling: a department maintains an organization chart. This department has the authorization to modify this chart. The department “Process Management” uses the department names in the procedure descriptions. The tasks of the department and the functionaries are maintained by the department “Human Resources”. Users who don't work for Human Resource can view this data, if they have the right authorization.

Another example is the maintenance of business objectives. The management team connects the business objectives to business processes. Some users will be allowed to view the objectives, others may not. This usage is easy to implement with Open Modeling. [22]

3.1.6.4. Conclusion

Open modeling tool is a good starting point for creating a web-based diagram editor, taking as a basis the DEMO Methodology, but there are several changes that would have to be made. The database of the open modeling tool is redundant and highly complex with hundreds of tables, and the interface for manipulating the diagrams is not properly user-friendly, since too many clicks have to be made to achieve a certain task, for instance, to create a connector between two elements, the user had to click on one of them, hold shift, and click on the other. Then has to right click on the selected elements, and choose the option “create connector between shapes”. This kind of user interface can still be improved to be more similar intuitive and simple, like clicking on the connector kind, then clicking on the first shape, hold and drag mouse to the second shape, and release the mouse.

3.2. Choosing a diagramming web-technology

During our research and based on the alternative editors we reviewed, we decided that the best solution was to build a web-based diagram editor from scratch, and that there were two main web-technologies that were the most adequate for this kind of project – SVG and HTML5 canvas. In this section, we will provide a comparison between both, state the advantages and disadvantages of both, choose most adequate technology for this project, and support that choice.

3.2.1. SVG

SVG is used to describe Scalable Vector Graphics, is based on XML, and can be easily integrated with HTML. It is built into the document using elements, attributes, and styles. SVG content can be static, dynamic, interactive and animated — it is very flexible. It can also be styled with CSS and add dynamic behavior to it using the SVG DOM. And of course, because the text inside SVG exists in a file, it is relatively accessible too. With

SVG we can do a lot more than simple vector graphics and animations. We can develop highly interactive Web applications with scripting, advanced animation events, filters, and almost anything. [15][16]

3.2.2. Canvas

The HTML5 Canvas specification is a versatile JavaScript API allowing us to code programmatic drawing operations. Canvas, by itself, allows us to define a canvas context object (manifesting as a <canvas> element on your HTML page), which can then be drawn inside. To do the actual drawing, we have two different options:

- A 2D drawing context,
- A 3D drawing context (WebGL).

The former is more established, more widely supported, and is available in all the modern Web, while the latter is in the early process of being defined, having only a handful of experimental implementations. Since 3D graphics are not a requirement for this project, we are just going to look at the 2D context of Canvas. This context provides you with a simple yet powerful API for performing quick drawing operation, on a 2D bitmap surface. There is no file format, and you can only draw using script. You do not have any DOM nodes for the shapes you draw — it is all on the surface, as pixels. [15][16]

3.2.3. Canvas vs SVG Summary

Canvas	**SVG**
Pixel based	Vector based
Single HTML element	Multiple graphical elements, which become part of the DOM
Modified through script only	Modified through script and CSS
Event model/user interaction is granular (x,y)	Event model/user interaction is abstracted (rect, path)
Performance is better with smaller surface, a larger number of objects (>10k), or both	Performance is better with smaller number of objects (<10k), a larger surface, or both

Table 2 - Canvas vs. SVG summary

[15]

Sometimes developer knowledge, skill set, and existing assets play a significant role into the choice of technologies. If a developer has deep knowledge of low level graphic APIs and limited knowledge of web technologies, the likely technology to choose is canvas. Also, performance is absolutely critical on high traffic websites. It is necessary to compare the performance characteristics of the two technologies. This might require the development of accessibility, custom styling, and more granular user interactions that do not come with canvas. It does not mean that canvas, though typically viewed as highly performant, is the obvious choice. The following graphs show the difference between of rendering time between SVG and Canvas objects.

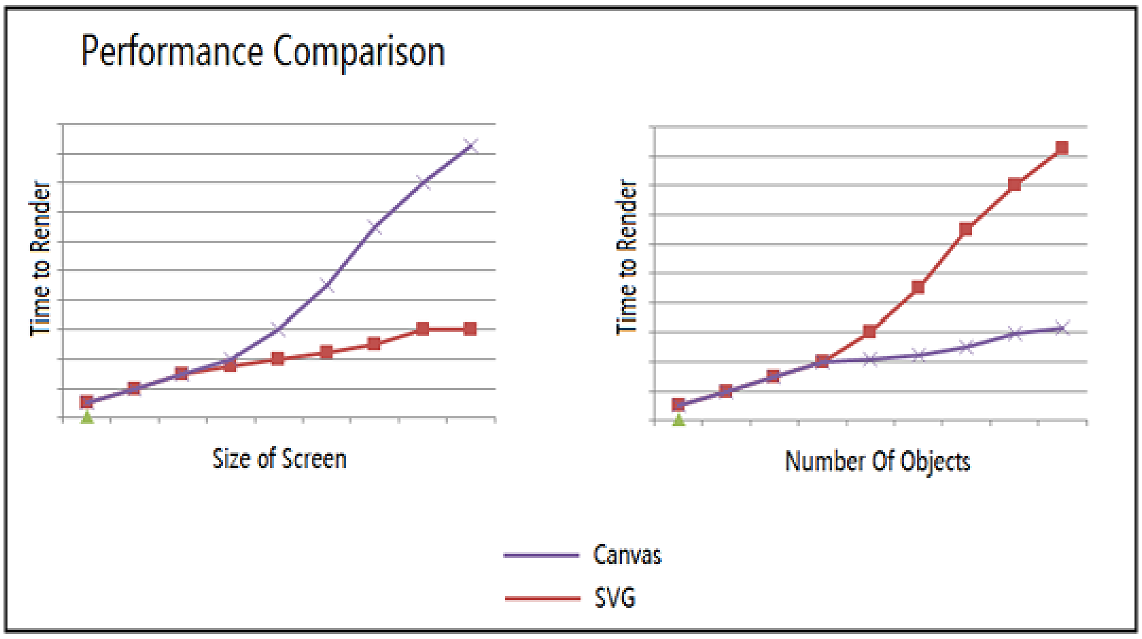


Figure 26 - Performance comparison between SVG and canvas

Generally, as the size of the screen increases, canvas begins to degrade as more pixels need to be drawn. As the number of objects increases on the screen, SVG begins to degrade as we are continually adding them to the DOM. These measurements are not necessarily accurate and can certainly change depending upon implementation and platform, whether fully hardware accelerated graphics are being used, and the speed of the JavaScript engine. [15] Table 1 and table 2 show the advantages and disadvantages of SVG and Canvas.

3.2.4. Advantages

Canvas	SVG
• High performance 2D surface for drawing anything you want. • Constant performance — everything is a pixel. Performance only degrades when the image resolution increases. • You can save the resulting image as a .png or.jpg. • Best suited for generating raster graphics, editing of images, and operations requiring pixel-level manipulation.	• Resolution independence — this makes SVG better suited for cross-platform user interfaces because it allows scaling for any screen resolution. • SVG has very good support for animations. Elements can be animated using a declarative syntax, or via JavaScript. • SVG is an XML file format, which means it is easily accessible through the DOM. • You have full control over each element using the SVG DOM API in JavaScript.

Table 3 - Advantages of SVG and Canvas

3.2.5. Disadvantages

Canvas	SVG
• There are no DOM nodes for anything you draw. It is all pixels. • There's no API for animation. You have to resort to timers and other events to update the Canvas when needed. • Poor text rendering capabilities. • Might not be the best choice for cases where accessibility is crucial.	• Slow rendering when document complexity increases — anything that uses the DOM a lot will be slow. • Not suited for applications like games, due to lack of performance with increased complexity.

• Canvas is not suited for Web site or application user interfaces. This is because user interfaces typically need to be dynamic and interactive, and Canvas requires you to manually redraw each element in the interface.	

Table 4 - Disadvantages of SVG and Canvas

3.2.6. Conclusion

For this project's context, we chose the SVG technology because we think that it is the technology that will best serve our needs. We aim to build an interactive tool, and SVG provides easy access to its elements through its DOM. This interactivity can be easily achieved with JQuery, which provides an API that is particularly useful to access, edit, and remove DOM elements. Thus, we can easily access and edit a SVG shape's properties, such as the position, width, or label (SVG has better support for text rendering than HTML Canvas). SVG is also resolution independent and we can, for instance, zoom in an SVG image, if we want to show it on a bigger monitor, and the image quality will not degrade at all. SVG has also primitives to create basic shapes such as rectangles, circles and lines, and also for creating text elements. More complex shapes can be created with the path element, but we decided not to use this element due to simplicity reasons, but it can be explored in future improvements of this project. SVG has also support for CSS styling, which is another good advantage.

The idea of this project is to create a SVG web-based graphical editor that allows not only the graphical definition of diagrams, but also a textual definition, which means that elements of a diagram such as shapes or connectors, or the diagram itself, can be defined graphically or textually, and to make it consistent, both should be synchronized with each other, i.e., when a graphical object is defined, it should also be defined textually, in an automatic way, and vice-versa. To accomplish this idea, we used the MediaWiki software, and other programming languages, described in chapter 4. We took as a basis the sister project, which is a meta-model editor developed on top of MediaWiki, that allows the definition of organization artifact kinds, like diagram kinds, shapes kinds or connector kinds. In chapter 5 we describe how this meta-model works, and we also describe how our project works on MediaWiki. In chapter 6 we describe how the implementation of Universal Diagram Editor (UDE) was done. In the UDE, at the same time diagrams, shapes and connectors are defined, the corresponding wiki pages are defined on MediaWiki, and vice-versa, although it does not make sense in defining certain properties textually, such as the coordinates of a shape in a diagram. Chapters 5 and 6 are strongly related, and perhaps an incremental reading of both is a good advice.

4. Software and Programming Languages

This section includes all the software that served as a basis for this project. We begin with MediaWiki, which is the software used by the well-known Wikipedia. Then we introduce three extensions for MediaWiki. The first extension is SemanticMediaWiki, which is an extension to make MediaWiki semantic. This was the solution used for representing the organization facts. It was defined that each page should represent a class of the UEAOM model, and a standard nomenclature was created for this purpose. Then we follow up with SemanticForms and SVGEdit extensions. SemanticForms is an extension that facilitates the user interaction with SemanticMediaWiki, because one can create wiki pages and edit properties more easily. SVGEdit extension is an extension for creating organization facts at a meta-mode level. Finally we describe the programming languages we used.

4.1. MediaWiki

MediaWiki is a free and open source software package written in PHP, originally for use on Wikipedia. It is now used by several other projects of the non-profit Wikimedia Foundation and by many other wikis. It is licensed under the GNU General Public License (GPL). The software is optimized to efficiently handle large projects, which can have terabytes of content and hundreds of thousands of hits per second. MediaWiki is an extremely powerful, scalable software and a feature-rich wiki implementation, that uses PHP to process and display data stored in a database, such as MySQL.

MediaWiki pages, also commonly referred to as “wiki pages”, use MediaWiki's wikitext format, so that users without knowledge of XHTML or CSS can edit them easily. When a user submits an edit to a page, MediaWiki writes it to the database, but without deleting the previous versions of the page, thus allowing easy reverts in case of vandalism or spamming. MediaWiki can manage image and multimedia files, too, which are stored in the file system. [5]

4.1.1. MediaWiki API (Application Programming Interface)

MediaWiki has an extensible web API (application programming interface) that provides direct, high-level access to the data contained in the MediaWiki databases. Client programs can use the API to login, get data, and post changes. The API supports thin web-based JavaScript clients and end-user applications (such as vandal-fighting tools). The API can be accessed by the backend of another web site.

In this project, we intend to use MediaWiki API, combined with JQuery's Ajax functions to create, edit and delete wiki pages. These wiki pages will be used as a database for the diagram editor we will be developing. This theme will be discussed with more detail in the following chapters. The API is accessed via URLs such as:

"http://en.wikipedia.org/w/api.php?action=query&list=recentchanges"

In this case, the query would be asking Wikipedia for information relating to the last 10 edits to the site. One of the perceived advantages of the API is its language independence. It listens for HTTP connections from clients and can send a response in a variety of formats, such as XML, serialized PHP, YAML, or JSON. We will be using the XML format because it is easily accessible by JQuery, a JavaScript library which we will be using to build our diagram editor application. [5]

4.1.2. SemanticMediaWiki

Semantic MediaWiki (SMW) is an extension of MediaWiki – the wiki application best known for powering Wikipedia – that helps to search, organize, tag, browse, evaluate, and share the wiki's content. While traditional wikis contain only text which computers can neither understand nor evaluate, SMW adds semantic annotations that allow a wiki to function as a collaborative database. Semantic MediaWiki was first released in 2005, and currently has over ten developers. In addition, a large number of related extensions have been created that extend the ability to edit, display and browse through the data stored by SMW: the term "Semantic MediaWiki" is sometimes used to refer to this entire family of extensions. Semantic MediaWiki has grown a long way from its roots as an academic research project. It is currently in active use in hundreds of sites, in many languages, around the world, including Fortune 500 companies, biomedical projects, government agencies and consumer directories. [6]

4.1.2.1. Categories and Properties

SMW introduces special markup elements which allow users to provide «hints» to computer programs on how to interpret certain pieces of information given in the wiki. Such hints are called semantic annotations and can be viewed as an extension of the existing system of categories in MediaWiki. Categories are an editing feature of MediaWiki, and are used as universal "tags" for articles, describing that the article belongs to a certain group of articles (are a means to classify articles according to certain criteria). For example, by adding [[Category:Cities]] to an article, the page is tagged as describing a city. MediaWiki can use this information to generate a list of all cities in a wiki, and thus help users to browse the information. Categories that already exist should be used, instead of creating new ones. Otherwise, the category's article will be empty, and it is strongly recommended to add a description that explains which articles should go into the category. The MediaWiki approach is to have many categories on each page, to identify all aspects of that page's subject. Semantic MediaWiki was created in part to eliminate the need for categories, by allowing for semantic properties to represent this data. Thus, SMW provides a further means of structuring the wiki. Wiki pages have links and text values in them, but only a human reader knows what the link or text represents. For example, «is the capital of Germany with a population of 3.396.990» means something very different from «plays football for Germany and earns 3.396.990 dollars a year». SMW allows you to annotate any link or text on the page to describe the meaning of the hyperlink or text. This turns links and text into explicit properties of an article. The property «capital of» is different from «on national football team of», just as the property «population» is different from «annual income». This addition enables users to go beyond mere categorization of articles. Properties are used to specify single pieces of information about the topic of some page; the value of a property can either be a standalone value, or the name of a page on the wiki. Every property should be defined on your wiki, with a page in the “Property:” namespace. Properties are used by a simple mark-up, similar to the syntax of links in MediaWiki: [[property name::value]]. This statement defines a «value» for the property of the given «property name». The page where this is used will just show the text for value and not the property assignment. Consider the Wikipedia article on Berlin. This article contains many links to other articles, such as «Germany», «European Union», and «United States». 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 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. Since categories and properties merely emphasize a particular part of an article's content, they are often called (semantic) annotations. Information that was provided in an article is now provided in a formal way accessible to software tools. In this project, for each class of the UEAOM, there will be a category with same name of the class, and all pages that represent instances of a certain class, should belong to the category with the same name. Properties will be useful to establish relations between wiki pages at both model and meta-model levels. [17]

4.1.3. SemanticForms Extension

Semantic Forms is an extension to MediaWiki that allows users to add, edit and query data using forms. It is heavily tied in with the Semantic MediaWiki extension, and is meant to be used for structured data that has semantic markup. Having Semantic MediaWiki installed is a precondition for the Semantic Forms extension.

Very simply, SemanticForms allow a user to add, edit, and query data on SemanticMediaWiki, without any kind of programming. Forms can be created and edited not just by administrators, but by users themselves.

The main components of Semantic Forms functionality are form definition pages, which exist in a new namespace, 'Form:'. These are pages consisting of markup code which gets parsed when a user goes to add or edit data. Since forms are defined strictly through these definition pages, users can themselves create and edit forms, without the need for any actual programming.

The Semantic Forms extension enforces the use of templates in creating semantic data. It does not support direct semantic markup in data pages; instead, all the semantic markup is meant to be stored indirectly through templates. A form allows a user to populate a pre-defined set of templates for a page (behind the scenes, that data is turned into semantic properties once the page is saved).

Semantic Forms also supports auto completion of fields, so users can easily see what the previously-entered values were for a given field. This greatly helps to avoid issues of naming ambiguity, spelling, etc. [7]

4.1.4. SVG-Edit Extension

SVG-edit is intended for users who need to do quick edits to existing SVG images and do not want to use proprietary or open-source software that requires installation. It is a fast, web-based, JavaScript-driven SVG drawing editor that works in any modern browser. The SVG-Edit Extension is an extension that allows SVG-Edit to be integrated in MediaWiki. The editor widget is loaded in an "iframe" within the current page context. This extension was modified in a parallel project, entitled WAMM [31], to allow the creation and editing of shape kinds. Figure 27 depicts SVG-Edit extension, with an example of a shape kind. These shape kinds are loaded to UDE's palette, when a new diagram is created or opened. After that new shapes can be created within the diagram.

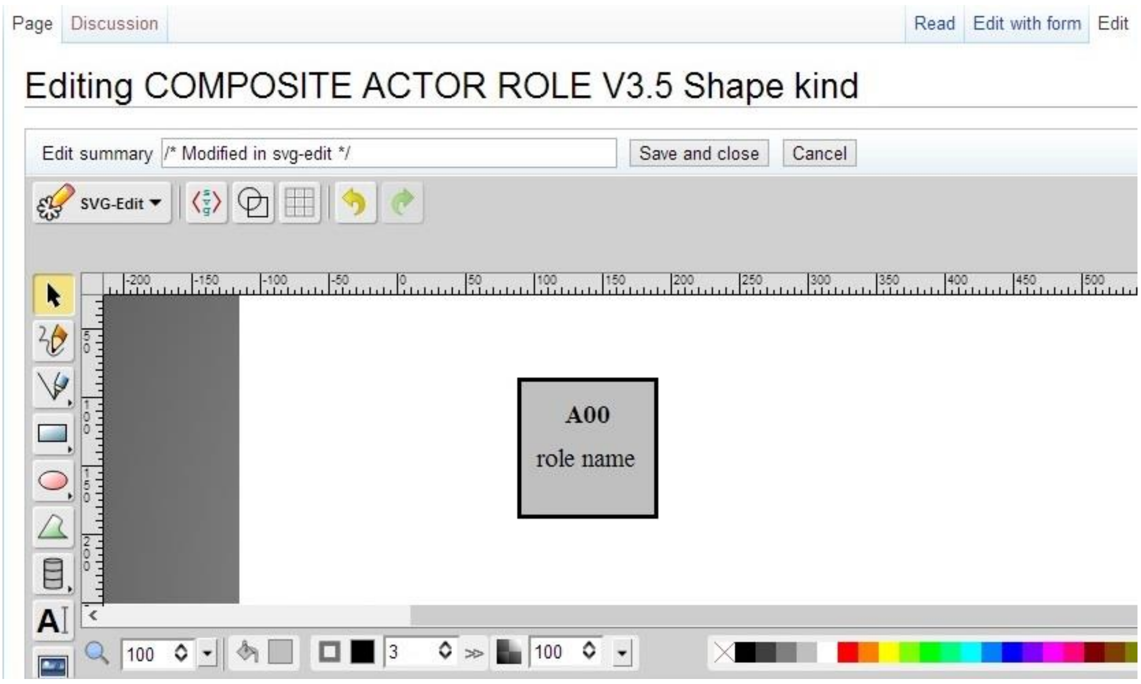


Figure 27 - MediaWiki SVGEdit Extension

4.2. Scalable Vector Graphics (SVG)

Scalable Vector Graphics (SVG) is a family of specifications of an XML-based file format for describing two-dimensional vector graphics, both static and dynamic (i.e. interactive or animated). The SVG specification is an open standard that has been under development by the World Wide Web Consortium (W3C) since 1999.

SVG images and their behaviors are defined in XML text files. This means that they can be searched, indexed, scripted and, if required, compressed. Since they are XML files, SVG images can be created and edited with any text editor, but it is often more convenient to create these types of images with drawing programs such as Inkscape.

All major modern web browsers have at least some degree of support and render SVG markup directly, including Mozilla Firefox, Internet Explorer 9, Google Chrome, Opera and Safari. However, earlier versions of Microsoft Internet Explorer (IE) do not support SVG natively, and Internet Explorer 9 does not run under Windows XP.

The main advantage of using SVG is that we will be able to easily visualize the DEMO diagrams on a web browser from any location, and create, edit and delete diagrams, without the need of installing software. Another advantage is that we can zoom in and zoom out the diagrams and the image quality will not deteriorate. [8]

4.2.1. Functionality

In this section we will cover the functional areas of the SVG specification that will be used in this project:

Basic shapes - Straight-line paths and paths made up of a series of connected straight-line segments (polylines), as well as closed polygons, circles and ellipses can be drawn. Rectangles and round-cornered rectangles are also standard elements. This functional area is the most widely used on this project to draw the DEMO shapes and connectors, which will be taken as an example to our Universal Diagram Editor.

Text - Unicode character text included in an SVG file is expressed as XML character data. Many visual effects are possible, and the SVG specification automatically handles bidirectional text (for composing a combination of English and Arabic text, for example), vertical text and characters along a curved path.

Painting - SVG shapes can be filled and/or outlined (painted with a color, a gradient, or a pattern). Fills can be opaque or have any degree of transparency. "Markers" are line-end features, such as arrowheads, or symbols that can appear at the vertices of a polygon.

Color - Colors can be applied to all visible SVG elements, either directly or via 'fill', 'stroke,' and other properties. Colors are specified in the same way as in CSS2, i.e. using names like black or blue, in hexadecimal such as #2f0 or #22ff00, in decimal like rgb(255,255,127), or as percentages of the form rgb(100%,100%,50%).

Interactivity - SVG images can interact with users in many ways. In addition to hyperlinks as mentioned below, any part of an SVG image can be made receptive to user interface events such as changes in focus, mouse clicks, scrolling or zooming the image and other pointer, keyboard and document events. Event handlers may start, stop or alter animations as well as trigger scripts in response to such events.

Scripting - All aspects of an SVG document can be accessed and manipulated using scripts in a similar way to HTML. The default scripting language is ECMAScript (closely related to JavaScript) and there are defined Document Object Model (DOM) objects for every SVG element and attribute. Scripts are enclosed in <script> elements. They can run in response to pointer events, keyboard events and document events as required. We used Jquery JavaScript library to add interactivity to SVG shapes and connectors. Jquery is especially useful for DOM manipulation.

Fonts - As with HTML and CSS, text in SVG may reference external font files, such as system fonts. If the required font files do not exist on the machine where the SVG file is rendered, the text may not appear as intended. To overcome this limitation, text can be displayed in an 'SVG font', where the required glyphs are defined in SVG as a font that is then referenced from the <text>element. [8]

To finish this subsection, we present two SVG examples. First, we will show a SVG <rect> element example, which represents a rectangle and is a commonly used element to create SVG shapes, namely an elementary actor role. The following code defines the element:

```
<svg xmlns="http://www.w3.org/2000/svg"
   <rect x="10" y="10" height="100" width="100"
      style="stroke:#006600; fill: #00cc00"/>
</svg>
```

Table 5 - SVG rectangle definition

Using this element you can draw rectangles of various width, height, with different stroke (outline) and fill colors, with sharp or rounded corners. The location of the rectangle is determined by the x and y attributes. This location is relative to any enclosing (parent) elements location. The size of the rectangle is determined by the width and height attributes. The style attribute allows to set additional style information like stroke and fill colors, width of the stroke etc. Figure 2 shows the resulting rectangle image.

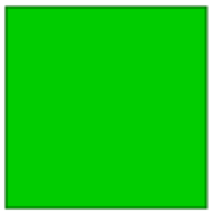


Figure 28 - SVG rectangle

The other example is a SVG <polyline> element that will be used to create lines (connectors). The SVG <polygon> element defines a set of connected straight line segments. Here is a simple example:

```
<svg xmlns="http://www.w3.org/2000/svg">
  <polyline points="0,40 40,40 40,80 80,80 80"
    style="fill:white;stroke:red;stroke-width:4" />
</svg>
```

Table 6 - SVG polyline definition

As you can see, we can define an unlimited number of segments, which is useful to create connector moves to implement the connectors on our diagram editor. In figure 2 we can take a look at how this SVG image renders. [9] [10]

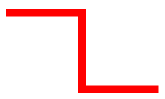


Figure 29 - SVG polyline

4.3. JavaScript

JavaScript (JS) is an interpreted computer programming language. As part of web browsers, implementations allow client-side scripts to interact with the user, control the browser, communicate asynchronously, and alter the content of the document. It has also become common in server-side programming, game development and the creation of desktop applications.

JavaScript is a prototype-based scripting language with dynamic typing and has first-class functions. Its syntax was influenced by C. JavaScript copies many names and naming conventions from Java, but the two languages are otherwise unrelated and have very different semantics. The key design principles within JavaScript are taken from the Self and Scheme programming languages. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. [11]

We intend to use JavaScript along with jQuery (described in the following chapter), to manipulate SVG elements on the DOM, and adding interaction to SVG objects.

4.4. Jquery

JQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across most browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.

Used by over 65% of the 10,000 most visited websites, jQuery is the most popular JavaScript library in use today. JQuery is free, open source software, licensed under the MIT License. JQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications. JQuery also provides capabilities for developers to create plugins on top of the JavaScript library. This enables developers to create abstractions for low-level interaction and animation, advanced effects and high-level, theme-able widgets. The modular approach to the jQuery library allows the creation of powerful dynamic web pages and web applications.

The set of JQuery core features — DOM element selections, traversal and manipulation —, enabled by its selector engine, created a new "programming style", fusing algorithms and DOM-data-structures and influenced the architecture of other JavaScript frameworks. JQuery has also been used in MediaWiki since version 1.16. [13]

4.4.1. JQuery’s Features

- DOM element selections using the multi-browser open source selector engine Sizzle.
- DOM traversal and modification, including support for CSS.
- DOM manipulation (based on CSS selectors) that uses node elements name and node elements attributes (id and class) as criteria to build selectors.
- Events
- Effects and animations
- AJAX
- Extensibility through plug-ins

- Utilities - such as user agent information, feature detection
- Multi-browser support.

4.4.1.1. Browser support

Both version 1.x and 2.x of JQuery support "current-1 version" (meaning the current stable version of the browser and the version that preceded it) of Firefox, Google Chrome, Safari, and Opera. The version 1.x also supports Internet Explorer 6 or higher version. However, jQuery version 2.x dropped Internet Explorer 6–8 support and can run only with IE 9 or higher. Due to this reason, version 1.x is still recommended by jQuery's developers. [13]

At the time we finished this project, version 2.x of Jquery had not yet been released. We decided to keep the version 1.10.2 (which was the version we were using), so we could have support for Internet Explorer 6-8 as well. [13]

4.4.2. Ajax

Ajax is a group of interrelated web development techniques used on the client-side to create asynchronous web applications. With Ajax, web applications can send data to, and retrieve data from, a server asynchronously (in the background) without interfering with the display and behavior of the existing page. Data is usually retrieved using the XMLHttpRequest object. Despite the name, the use of XML is not needed (JSON is often used instead), and the requests do not need to be asynchronous.

For this project, we will use both asynchronous and synchronous requests. The synchronous requests will be particularly useful, for example, in situations we need to query data from MediaWiki and load it immediately on a select box, right after opening a dialog. Asynchronous requests will be used to edit MediaWiki pages on the background, whenever possible, so that the user can continue working normally without the need to wait for the completion of the request.

The DOM is accessed with JavaScript to dynamically display, and to allow the user to interact with the information presented. JavaScript and the XMLHttpRequest object provide a method for exchanging data asynchronously between browser and server to avoid full page reloads.

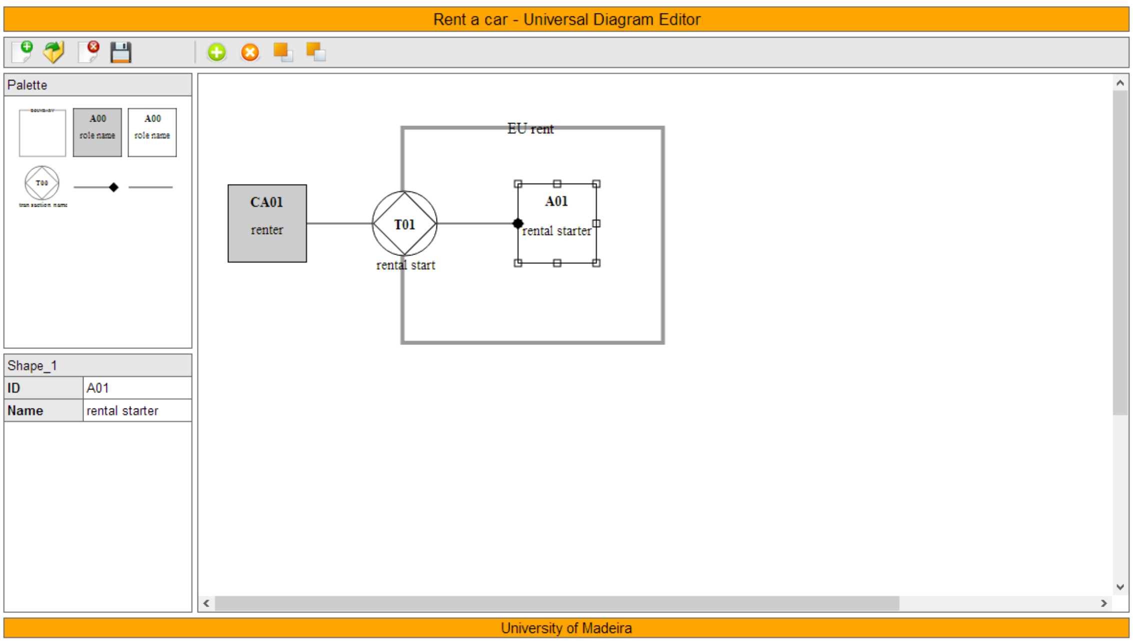
Jquery Ajax function simplifies the process of making and Ajax request. We can easily get the data by specifying the url, choose whether we want a synchronous or asynchronous request, and create a callback function to handle the data that was requested, once the request is completed.

The term Ajax has come to represent a broad group of web technologies that can be used to implement a web application that communicates with a server in the background, without interfering with the current state of the page. In the article that coined the term Ajax, Jesse James Garrett explained that the following technologies are incorporated:

- HTML (or XHTML) and CSS for presentation
- The Document Object Model (DOM) for dynamic display of and interaction with data
- XML for the interchange of data, and XSLT for its manipulation
- The XMLHttpRequest object for asynchronous communication
- JavaScript to bring these technologies together [14]

5. DEMO Methodology on MediaWiki

The Universal Diagram Editor (UDE) we developed uses MediaWiki as a database to save the symbols that are created through the editor's graphical interface. Symbols are saved as wiki pages. Each class of the UEAOM correspond to a wiki page, or to a wiki property. This projects context focuses in the concrete syntax but also uses the abstract syntax, as both are interrelated. In Figure 30 we can see the interface of the UDE, with a diagram of the ATD kind already created. Each graphical definition of a symbol in the UDE's canvas, for instance, the symbol "CA01-renter", corresponds to a wiki page on MediaWiki. Each wiki page that defines a symbol, has the graphical properties of that symbol, such as width and height, or x and y position.



Shape_1

ID	A01
Name	rental starter

Figure 30 - Universal Diagram Editor (UDE)

In this chapter we describe how we use MediaWiki, how these wiki pages are maintained, and the structure of these pages, before proceeding to the description of the implementation and functionality of the UDE, on chapter 6. The pages that represent facts at a meta-mode level will also be described. Although this projected is centered on the model level, the meta-model level is required not only to understand how the system works, but also for the system to work, since it depends on it. The following subsections 5.1 and 5.2 are an adaptation from [30] with some modifications on the examples for creating wiki pages, on the nomenclature of the wiki pages of the model level, and other minor modifications.

5.1. Approach for creating wiki pages

This SMW based implementation 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 OAKRKs, 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 «ATD» page specifies that only the presence of SHAPE KINDs «ACTOR ROLE SHAPE» and «TRANSACTION KIND SHAPE» are allowed in instances of this diagram kind. Concrete diagrams of an organization, for instance, the Rent-a-car Actor (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::ATD. 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 export, and also zoom operation without losing resolution quality. The page «Shape 1», which corresponds to «A01-rental starter», is an example of an instance of a SHAPE and the page «Connector 1», which is the page that corresponds to the executor link between «CA02-Driver» and «T03-car_drop-off» is an example of an instance of the UEAOM class CONNECTOR, and semantically associated with the page «EXECUTOR_V3.5», 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, «Is instance of shape kind» and of CONNECTOR PROPERTY, for example, «Is part of diagram». At model level these properties are instantiated as objects instances of classes SHAPE PROPERTY VALUE and CONNECTOR PROPERTY VALUE. For example, «ELEMENTARY ACTOR ROLE

V3.5 Shape kind» and «Diagram_1», 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 this 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 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 JavaScript/JQuery based diagram editor, the UDE, was developed to implement all the functionality deemed convenient like ones present in well-known modelling tools such as Microsoft Visio. This editor is presented in chapter 6. [30]

5.2. Standard nomenclature for pages and properties

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 an 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, for instance «TRANSACTION_KIND_V3.5», where the number 3.5 corresponds to the version of that shape kind. The version was added to the page name because a shape kind can have different versions, but SMW does not allow pages with the same name, thus the addition of the version at the end of the name. A wiki page representing an instance of an ORGANIZATION ARTIFACT is a little different. It consists of a string followed by a number, where words are lower cased. For example, the wiki page Oa 1, which represents the OA A01-rental_starter, is defined by the string Oa (which stands for organization artifact), followed by the number 1. This first letter is upper because

SMW forces pages names to have its first letter as capital. The nomenclature for the definition of wiki pages that represent instances of pages of the meta-model level was defined this way to eliminate two issues that came up if the page names were the same as the artifact they were representing. The first one is that a user may want to represent an organization artifact in two different scopes of interest. The name of the artifact cannot be the same because SMW does not allow pages with the same name, so by adding an incremental number at the ending of the name of the page solves this problem, and we could have for instance the Oa 1 and Oa 2, both representing A01-rental starter, but in different scopes of interest. The other issue was that if a user want to change the name for instance, A01-rental starter. The page name is, for instance used as a semantic property the wiki page “Shape 1”, which is the page that represents the A01-rental starter symbol. The property of this page would be “Represents Organization Artifact::A01-rental starter”. If the page name was edited to A02-rental starter, for instance, the value for this property would no longer be correct, unless it was edited dynamically as well. But we decided that it would be simpler to use wiki page names as ID (for instance Oa 1), and use the organization artifact name as a properties in the Oa 1 page, for instance: “Specific ID::A01”, and “Name::rental starter”. Now if the user wants to edit the name and ID of the artifact, he can simply edit these properties and properties of other pages that depend of the name of this page would no longer be affected.

Properties also have a simple standard nomenclature. Any property consists of lowercase words (except first letter which is uppercase), separated by spaces. Examples of valid properties are: «Specific ID», and «Name». 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.

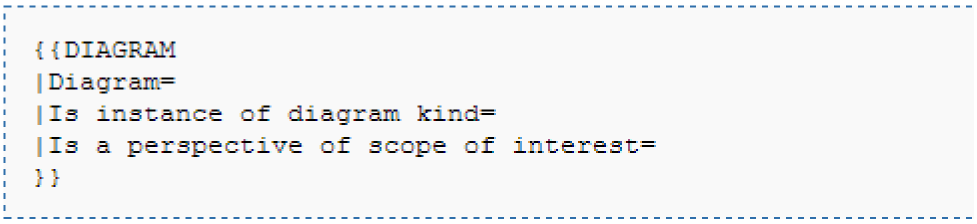
The instances of OAKRK and OAR of the UEAOM are a particular case when it comes to the nomenclature. Here we are dealing with composed OAKs and OAs with multiple elements, and as such this has to be considered in the naming. For example in an Actor Transaction Diagram we have two kinds of OAKRKs, the «TRANSACTION KIND.initiated by.ACTOR ROLE» and « TRANSACTION KIND.executed by.ACTOR ROLE ». As previously, and maintaining the coherence, at the meta-model level the OAKRKs 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 OAKRK. At model level the principle is also the same, the nomenclature used is the names used of the relating OAs (in lower case, except first letter) as previously explained separated with the Oa 1 example. An example of an OAR is «Oar 1». [30]

5.2.1. Semantic Forms

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 Oa 1 (A01-rental starter) on SMW, we would have properties such as «Specific ID» or «Name» as represented in Figure 33 - Oa 1 wiki page.

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, there would be a template on the namespace Template with the name DIAGRAM (Template:DIAGRAM) that would list the «Diagram» (name), the «Is instance of diagram kind», and the "Is a perspective of scope of interest" properties. This template is depicted in Figure 31.



```
{{DIAGRAM
|Diagram=
|Is instance of diagram kind=
|Is a perspective of scope of interest=
}}
```

Figure 31 - DIAGRAM Template

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.

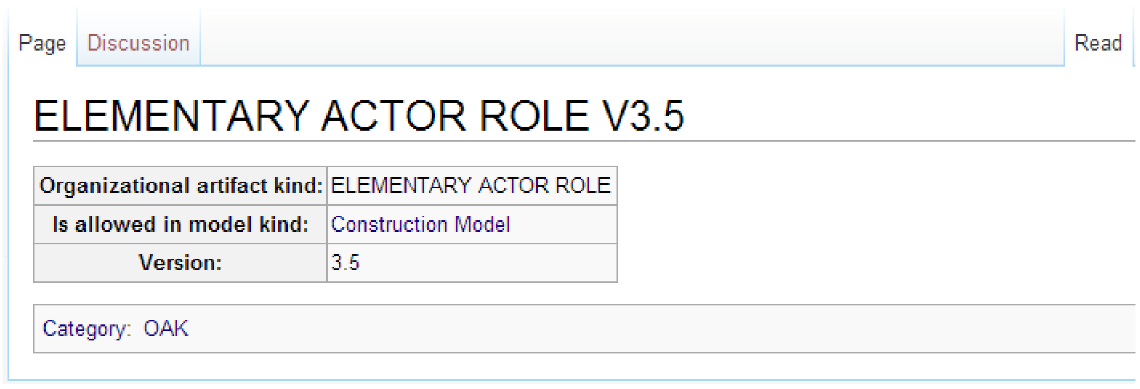
The Forms are the implementations for the templates. For instance in a «DIAGRAM FORM» one would fill all the listed properties in the «DIAGRAM TEMPLATE», but this task is intended to also be allowed by creating a diagram using SVG that would automatically generate the page, leaving no need to use the 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. [30]

5.3. UEAOM implementation on SMW

5.3.1. Organization Artifacts (OAs) and Shapes

One of the main advantages of using MediaWiki in this project, is that organization artifacts can be defined textually on the wiki, for example, the organization artifact A01-rental starter can be created using MediaWiki interface, and we can use that artifact later on our editor to create a shape that will represent it. We can also create OAs through the diagram editor, and they will be available for visualization on MediaWiki. So, we have a system that can be manipulated with both textual and graphical interfaces. We defined that wiki pages that represent fact types at a meta-model level have the title defined with uppercase text, and wiki pages that represent fact types at a model level, have its title defined with lowercase text, except for the first letter, which is automatically forced to uppercase by MediaWiki.

Each wiki page represents a fact type at a meta-model level or a fact at a model level. An example of a fact at a meta-model level is "ELEMENTARY ACTOR ROLE V3.5" and an example of a fact at a model level is "A01-rental starter". "A01-rental starter" is an instance of the organization artifact kind "ELEMENTARY ACTOR ROLE V3.5". Figure 32 - ELEMENTARY ACTOR ROLE V3.5 wiki page and Figure 33 - Oa 1 wiki page show the corresponding wiki pages.

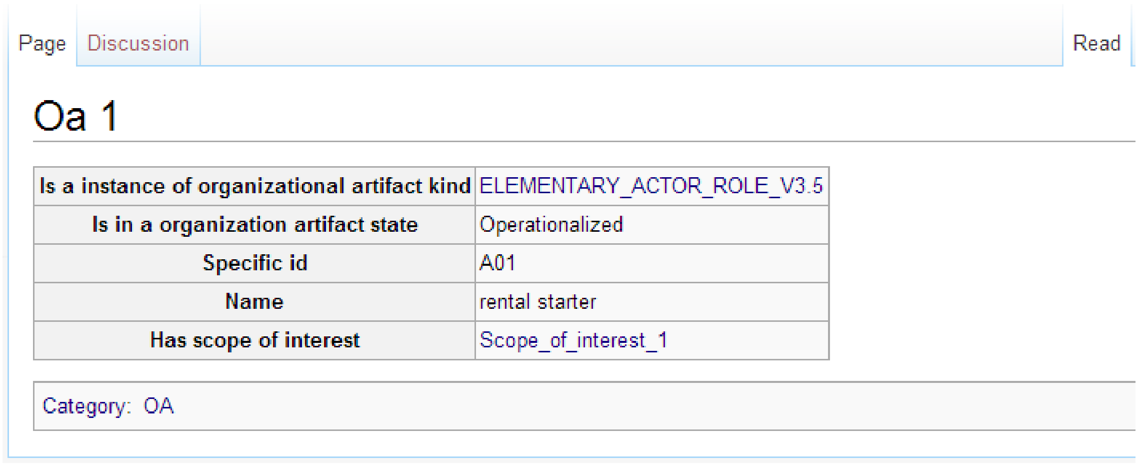

Page Discussion Read

ELEMENTARY ACTOR ROLE V3.5

Organizational artifact kind:	ELEMENTARY ACTOR ROLE
Is allowed in model kind:	Construction Model
Version:	3.5

Category: OAK

Figure 32 - ELEMENTARY ACTOR ROLE V3.5 wiki page


Page Discussion Read

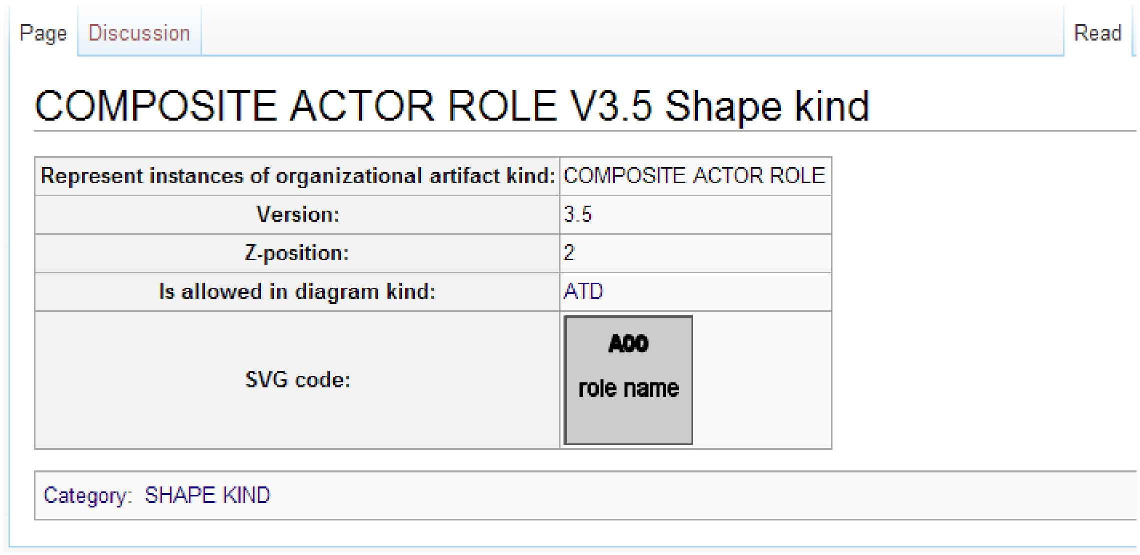
Oa 1

Is a instance of organizational artifact kind	ELEMENTARY_ACTOR_ROLE_V3.5
Is in a organization artifact state	Operationalized
Specific id	A01
Name	rental starter
Has scope of interest	Scope_of_interest_1

Category: OA

Figure 33 - Oa 1 wiki page

As we can see, wiki pages consist of a title, a set of properties and property values, and a category. If you notice, the name of the wiki page A01-rental starter is “Oa 1”, and not rental-starter. Initially it was defined that way, but we decided to change and use the page name as an ID, because it brings us some benefits which we explain shortly. All pages that belong to the model level (instances), have its title defined this way. In order to understand why we did this, we are going to introduce another two more wiki page examples, “COMPOSITE ACTOR ROLE V3.5 Shape kind”, represented in Figure 34, and “Shape 1”, represented in Figure 35 - Shape 1 wiki page.

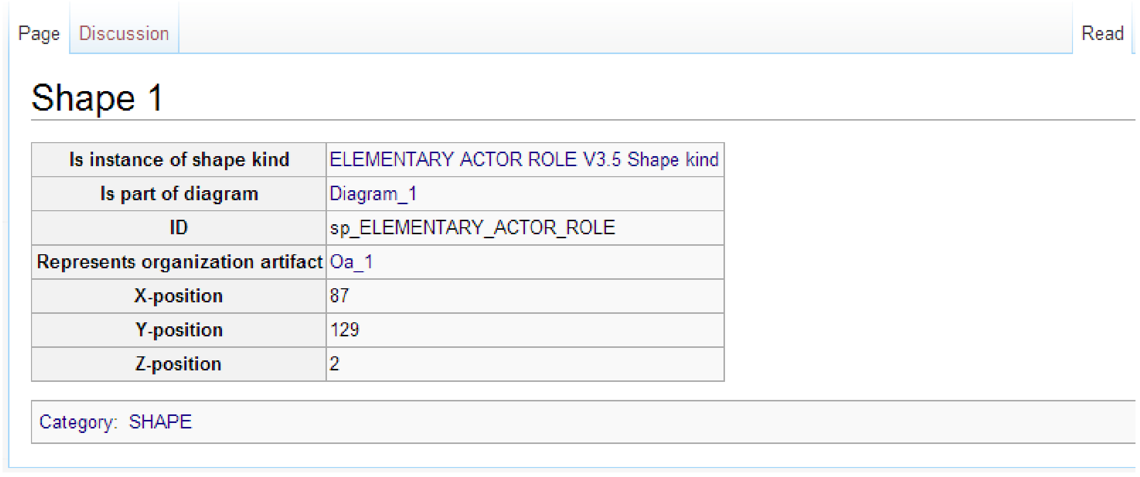

Page | Discussion | Read

COMPOSITE ACTOR ROLE V3.5 Shape kind

Represent instances of organizational artifact kind:	COMPOSITE ACTOR ROLE
Version:	3.5
Z-position:	2
Is allowed in diagram kind:	ATD
SVG code:	A00 role name

Category: SHAPE KIND

Figure 34 - COMPOSITE ACTOR ROLE V3.5 Shape kind wiki page


Page | Discussion | Read

Shape 1

Is instance of shape kind	ELEMENTARY ACTOR ROLE V3.5 Shape kind
Is part of diagram	Diagram_1
ID	sp_ELEMENTARY_ACTOR_ROLE
Represents organization artifact	Oa_1
X-position	87
Y-position	129
Z-position	2

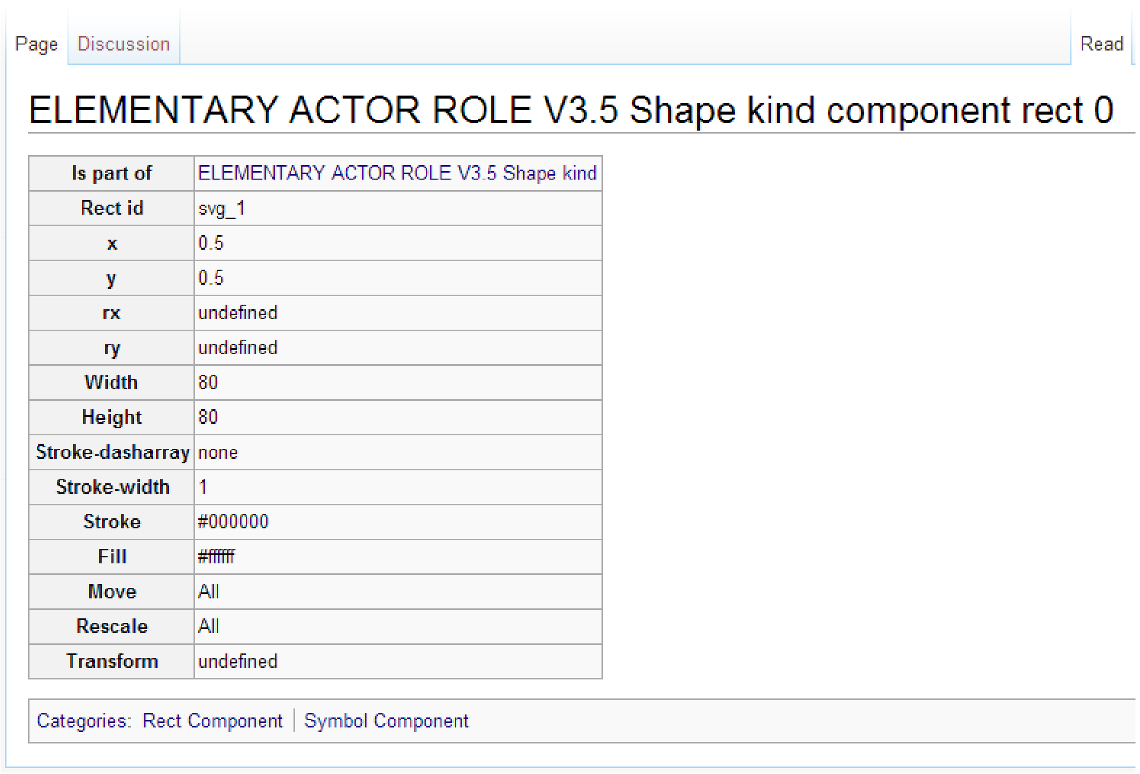
Category: SHAPE

Figure 35 - Shape 1 wiki page

"COMPOSITE ACTOR ROLE V3.5 Shape kind" is a shape kind wiki page. It represents the "COMPOSITE ACTOR ROLE V3.5" OAK, and can be seen as of an extension of it. It represents the same artifact kind, only it has graphical properties such as the SVG code of the corresponding SHAPE KIND, and the pre-defined z-position on the diagram. Shape 1, in Figure 35 - Shape 1 wiki page, is an instance of this shape kind, and represents the organization artifact Oa 1. The differences between Oa 1 and Shape 1 are that Oa 1 represents the fact that there is an organization artifact with the name A01-rental starter on the organization. Shape 1 is a graphical representation of that artifact, and an instance of "COMPOSITE ACTOR ROLE V3.5", and saves information such as the diagram which it belongs, and its coordinates, X-position and Y-position, on the diagram. The symbol that corresponds to Shape 1 (A01-rental starter), can be viewed in Figure 30.

At this moment, we can explain why we decided to use model-level wiki page titles as IDs. Let's suppose that a user of the system wants to change the name of the Oa 1 (A01-admitter). If the name was the page title (and not a property), the user would also need to change the property value of the property "Represents organization artifact" on "Shape 1" page. And this is fundamental property because it's the property that allows us to query the organization artifact that a shape represents. There are scenarios where there would be the need to change more properties with the older approach, in more than one page, namely on the connectors, so this example we just presented is the best case scenario. We think this is a great solution, but it brings up another issue. A user now needs to click and enter on the wiki page to view the name of the corresponding OA. We solved this problem with the ASK API from MediaWiki, which allows us to display lists of wiki pages in a table format, and that allows us to view property values of several pages at once, such as the pages names. These queries will be explained still on this chapter.

Shapes are formed by one or more symbols components. For instance, an actor, from the actor transaction diagram (ATD), has three components: a rectangle and two labels (ID and name), and a transaction (from the same diagram type), is formed by four components: a circle, a diamond shape, and two labels. On the UEAOM, we decided that symbol components should be classes, which means there will be a wiki page for each component of a shape. This makes sense because each component can have its individual properties separately, for instance, x and y position on each components page makes more sense than having all at once in one page. So, again, we will have wiki pages for component classes at a meta-model level, that have pre-defined data about each component type, and we will have wiki pages that represent instances of those components. The example we will present is a rectangle component, which belongs to the shape kind ELEMENTARY ACTOR ROLE V3.5 Shape kind. In Figure 36 - ELEMENTARY ACTOR ROLE V3.5 Shape kind component rect 0 and Figure 37 - Shape 1 component rect 0, we can view both the component kind and an instance of that component.

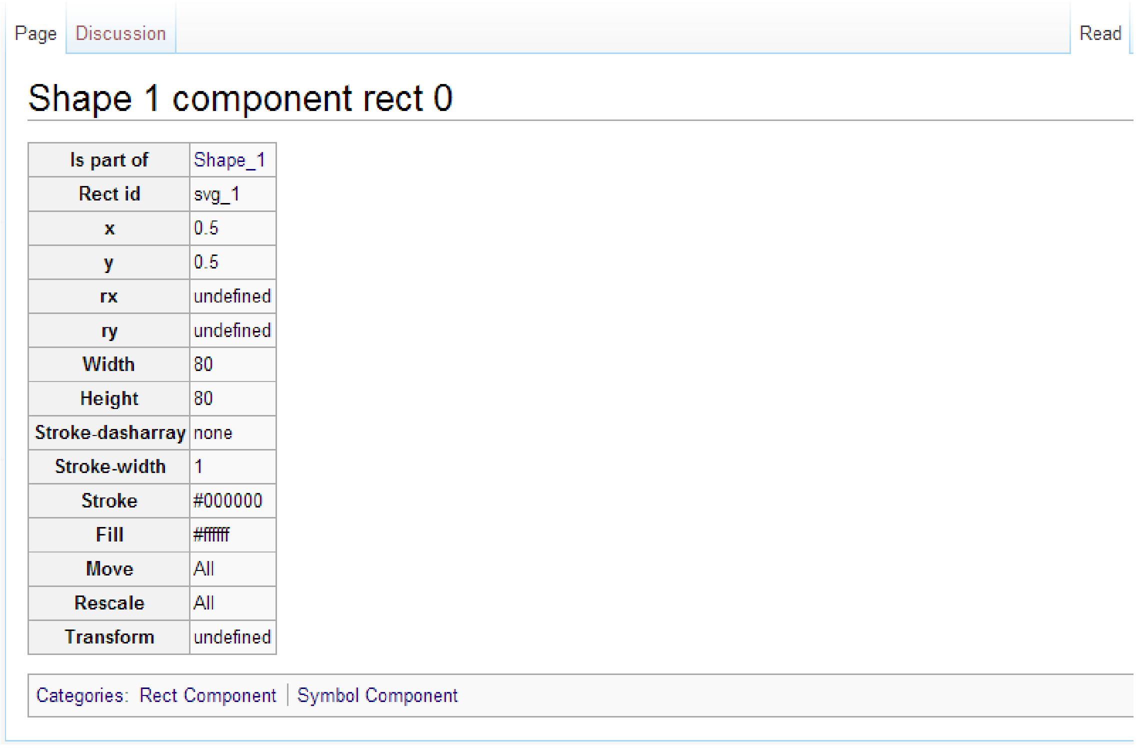


ELEMENTARY ACTOR ROLE V3.5 Shape kind component rect 0

Is part of	ELEMENTARY ACTOR ROLE V3.5 Shape kind
Rect id	svg_1
x	0.5
y	0.5
rx	undefined
ry	undefined
Width	80
Height	80
Stroke-dasharray	none
Stroke-width	1
Stroke	#000000
Fill	#ffffff
Move	All
Rescale	All
Transform	undefined

Categories: Rect Component | Symbol Component

Figure 36 - ELEMENTARY ACTOR ROLE V3.5 Shape kind component rect 0



Shape 1 component rect 0

Is part of	Shape_1
Rect id	svg_1
x	0.5
y	0.5
rx	undefined
ry	undefined
Width	80
Height	80
Stroke-dasharray	none
Stroke-width	1
Stroke	#000000
Fill	#ffffff
Move	All
Rescale	All
Transform	undefined

Categories: Rect Component | Symbol Component

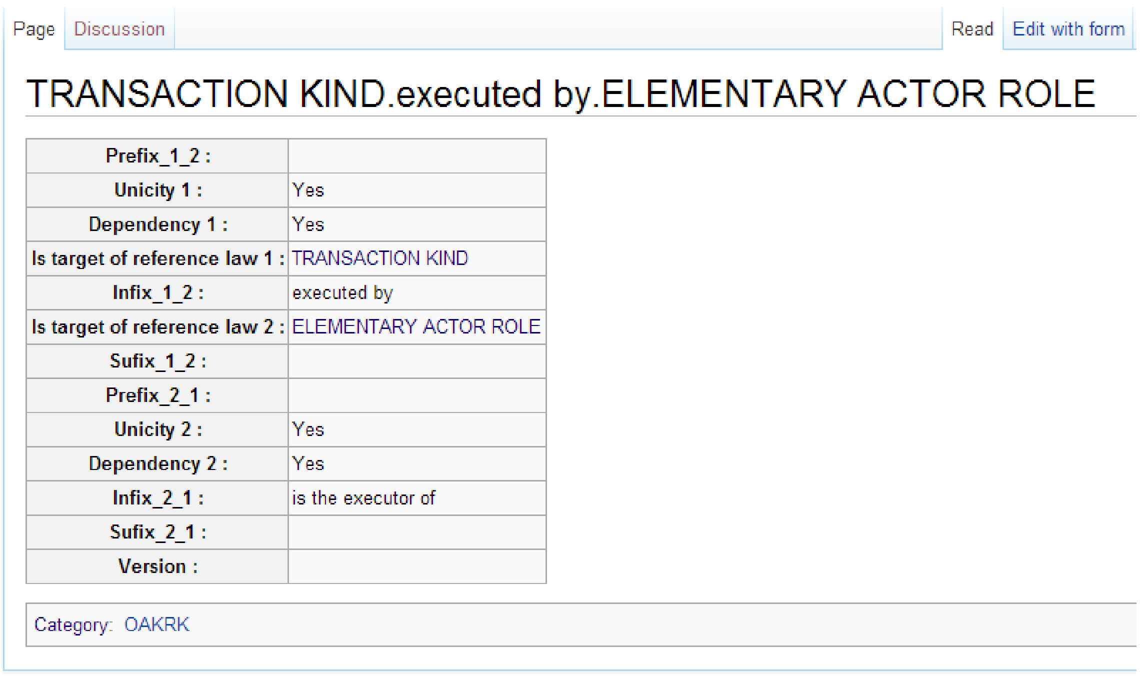
Figure 37 - Shape 1 component rect 0

The component kind page defines the default properties of the component to be created such as width. The component wiki page is an instance of that component kind, with the same properties, but only with custom values for those properties. An example of the symbol that represents this component can be viewed in Figure 30, and it is obviously the rectangle that is part of the symbol of Shape 1 (A01-rental starter). When we create a new shape, on the Universal Diagram Editor (UDE), we must query all components of that shape, and create instances, by creating new pages with the default values, and SVG elements (one for each component), at the same time on the editor. When editing properties of the (shape) component, on the editor, the properties values of the wiki page that represent the instance of the component will also be modified.

5.3.2. Organization Artifact Relations (OARs) and Connectors

There are two types of symbols that can be drawn on the canvas – shapes and connectors. In the previous section we described the shapes and the artifacts whose shapes represent, and now, in this section, we will make a brief description of the OARs and the connectors. Note: canvas is not to be confused with HTML5 canvas. What we have is canvas that we created based on SVG and where we draw SVG shapes and connectors.

Let’s begin with Figure 38 - TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE wiki page, which is an example of an OAKRK (organization artifact kind relation kind). In this example, the properties “is target of reference law 1” and “is target of reference law 2” mean that only organization artifacts kinds “TRANSACTION KIND” and “ELEMENTARY ACTOR ROLE” are allowed participate in this relation. The “Infix_1_2” and “Infix_2_1” specify how the targets relate, either from target one to target two, or vice-versa.



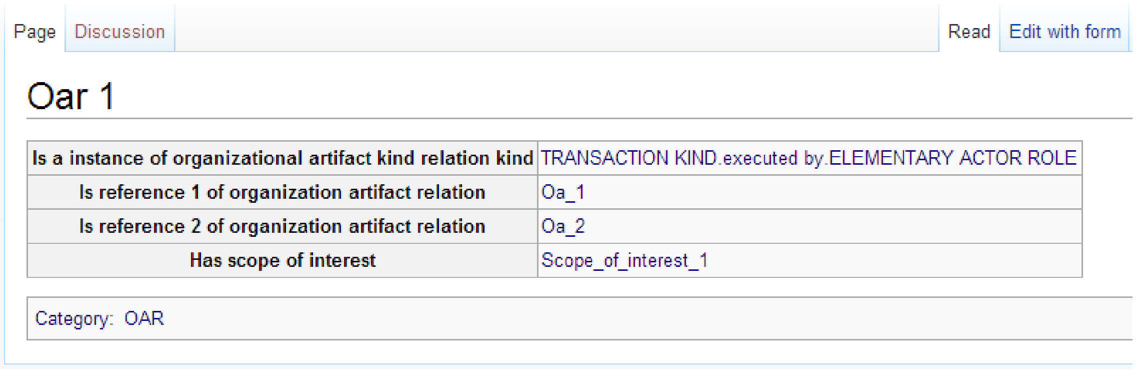
TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE

Prefix_1_2 :	
Unicity 1 :	Yes
Dependency 1 :	Yes
Is target of reference law 1 :	TRANSACTION KIND
Infix_1_2 :	executed by
Is target of reference law 2 :	ELEMENTARY ACTOR ROLE
Sufix_1_2 :	
Prefix_2_1 :	
Unicity 2 :	Yes
Dependency 2 :	Yes
Infix_2_1 :	is the executor of
Sufix_2_1 :	
Version :	

Category: OAKRK

Figure 38 - TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE wiki page

Figure 39 - Oar 1 wiki page is the instance that represents the OAKRK in Figure 39 - Oar 1 wiki page. As we can see, we have the target references that participate on the relation, and the scope of interest to which this OAR belongs.



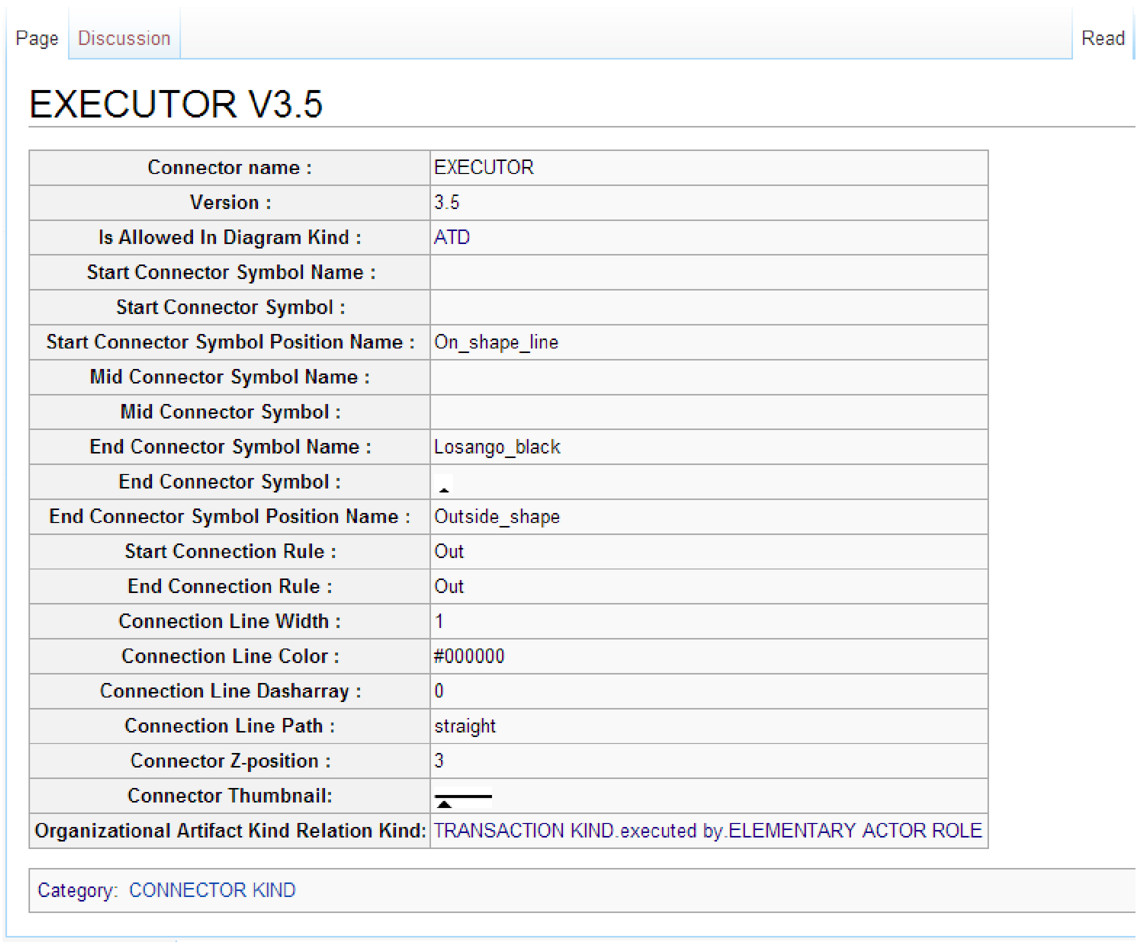
Oar 1

Is a instance of organizational artifact kind relation kind	TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE
Is reference 1 of organization artifact relation	Oa_1
Is reference 2 of organization artifact relation	Oa_2
Has scope of interest	Scope_of_interest_1

Category: OAR

Figure 39 - Oar 1 wiki page

The wiki page in Figure 40 - EXECUTOR V3.5 wiki page represents an example of a connector kind and the wiki page in Figure 41 - Connector 1 wiki page, Connector 1, is an instance of this connector kind.


Page Discussion Read

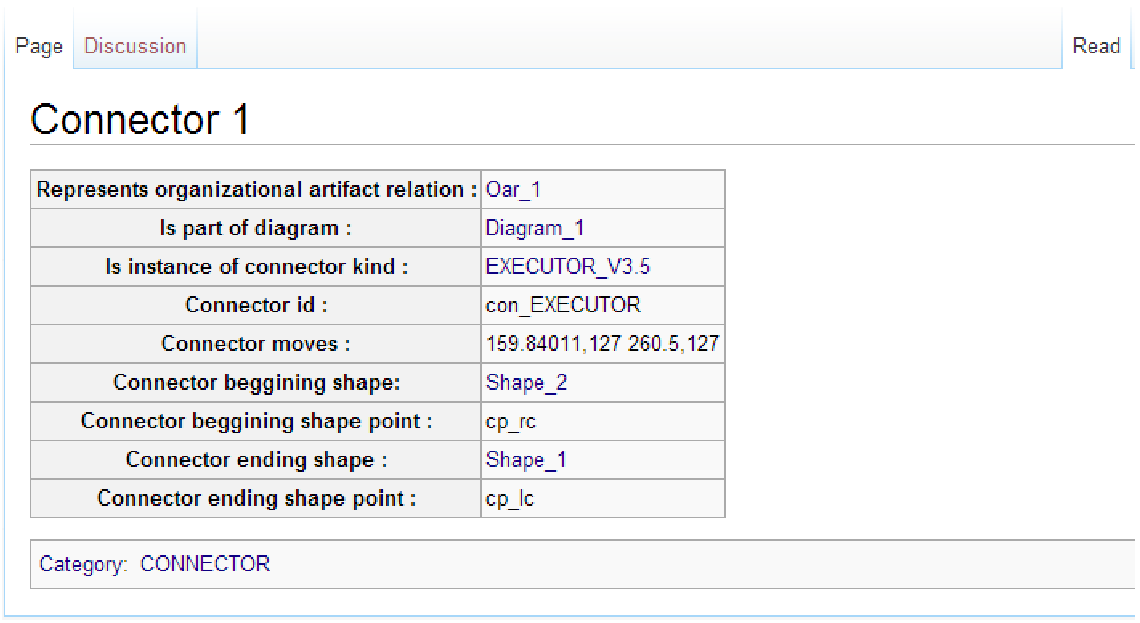
EXECUTOR V3.5

Connector name :	EXECUTOR
Version :	3.5
Is Allowed In Diagram Kind :	ATD
Start Connector Symbol Name :	
Start Connector Symbol :	
Start Connector Symbol Position Name :	On_shape_line
Mid Connector Symbol Name :	
Mid Connector Symbol :	
End Connector Symbol Name :	Losango_black
End Connector Symbol :	▲
End Connector Symbol Position Name :	Outside_shape
Start Connection Rule :	Out
End Connection Rule :	Out
Connection Line Width :	1
Connection Line Color :	#000000
Connection Line Dasharray :	0
Connection Line Path :	straight
Connector Z-position :	3
Connector Thumbnail:	▲
Organizational Artifact Kind Relation Kind:	TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE

Category: CONNECTOR KIND

Figure 40 - EXECUTOR V3.5 wiki page

An example of “Connector 1” is depicted in Figure 30, and it is the line that links the symbols “CA01-renter” and “T01-rental start”. The “connector moves” property in “Connector 1” wiki page has information about how many moves the connector line has made. Each pair of values represent a point on the connector line. The properties “Connector beginning shape” and “Connector beginning shape point” represent both the shape where the connector starts, and the connector point of that shape where the line of the connector is connected. An analogous reasoning can be made for “Connector ending shape” and “Connector ending shape point” properties.


Page Discussion Read

Connector 1

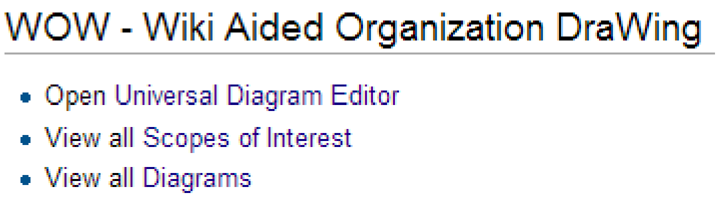
Represents organizational artifact relation :	Oar_1
Is part of diagram :	Diagram_1
Is instance of connector kind :	EXECUTOR_V3.5
Connector id :	con_EXECUTOR
Connector moves :	159.84011,127 260.5,127
Connector beggining shape:	Shape_2
Connector beggining shape point :	cp_rc
Connector ending shape :	Shape_1
Connector ending shape point :	cp_lc

Category: CONNECTOR

Figure 41 - Connector 1 wiki page

5.3.3. Scopes of Interest, Diagrams, and Queries

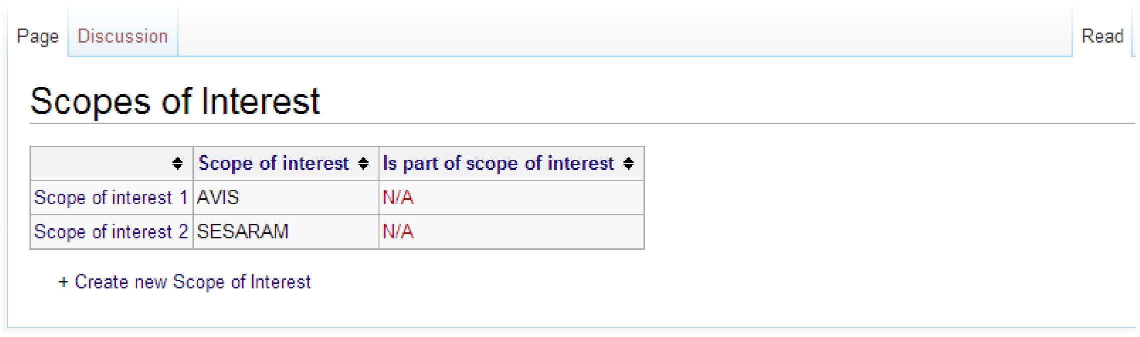
To access the user interface on MediaWiki, the user must enter the wiki page named "WOW". From there, the user has three options: open the universal diagram editor, whose functionality will be explained further ahead, view all scopes of interest, or view all diagrams. Scope of interest is a context where the diagrams, OAs, and OARs belong. In Figure 42 - WOW main menu we can take a look at WOW main menu.


WOW - Wiki Aided Organization DraWing

- Open Universal Diagram Editor
- View all Scopes of Interest
- View all Diagrams

Figure 42 - WOW main menu

If a user chooses the "View all Scopes of Interest" option, he will be presented with a list of all the currently created scopes of interest, and from there, he can find a certain diagram. But we also created the option "View all Diagrams" because a user may want to see all the diagrams, apart from the scopes of interest. Figure 43 depicts the "Scopes of Interest" wiki page with two scopes of interest currently created, namely "AVIS" and "SESARAM". It was created a query based on the SemanticMediaWiki ASK API, to get list scopes of interest when the user enters the page.


Page Discussion Read

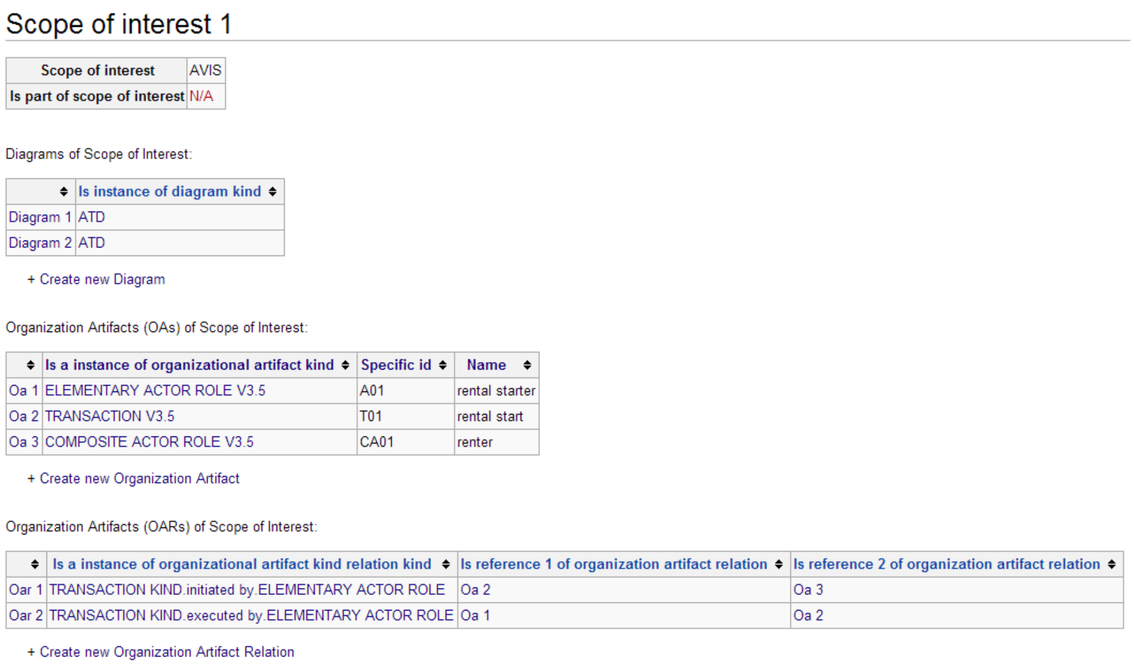
Scopes of Interest

	Scope of interest	Is part of scope of interest
Scope of interest 1	AVIS	N/A
Scope of interest 2	SESARAM	N/A

+ Create new Scope of Interest

Figure 43 - Scopes of Interest wiki page

If the user wishes to enter the page, he simply needs to click on the corresponding links on the left column, which are the page titles. Without these queries, there would be no way of a user to know which scope of interest he was dealing with, without entering the page and looking at the property "Scope of interest", which is the name of the scope of interest. With these queries, a user can easily find the desired scope of interest only by looking at the second column of the table. If the user, for instance, clicks on "Scope of interest 1", he is presented with the page in Figure 44 - Scope of interest 1 wiki page.


Scope of interest 1

Scope of interest	AVIS
Is part of scope of interest	N/A

Diagrams of Scope of Interest:

	Is instance of diagram kind
Diagram 1	ATD
Diagram 2	ATD

+ Create new Diagram

Organization Artifacts (OAs) of Scope of Interest:

	Is a instance of organizational artifact kind	Specific id	Name
Oa 1	ELEMENTARY ACTOR ROLE V3.5	A01	rental starter
Oa 2	TRANSACTION V3.5	T01	rental start
Oa 3	COMPOSITE ACTOR ROLE V3.5	CA01	renter

+ Create new Organization Artifact

Organization Artifacts (OARs) of Scope of Interest:

	Is a instance of organizational artifact kind relation kind	Is reference 1 of organization artifact relation	Is reference 2 of organization artifact relation
Oar 1	TRANSACTION KIND.initiated by.ELEMENTARY ACTOR ROLE	Oa 2	Oa 3
Oar 2	TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE	Oa 1	Oa 2

+ Create new Organization Artifact Relation

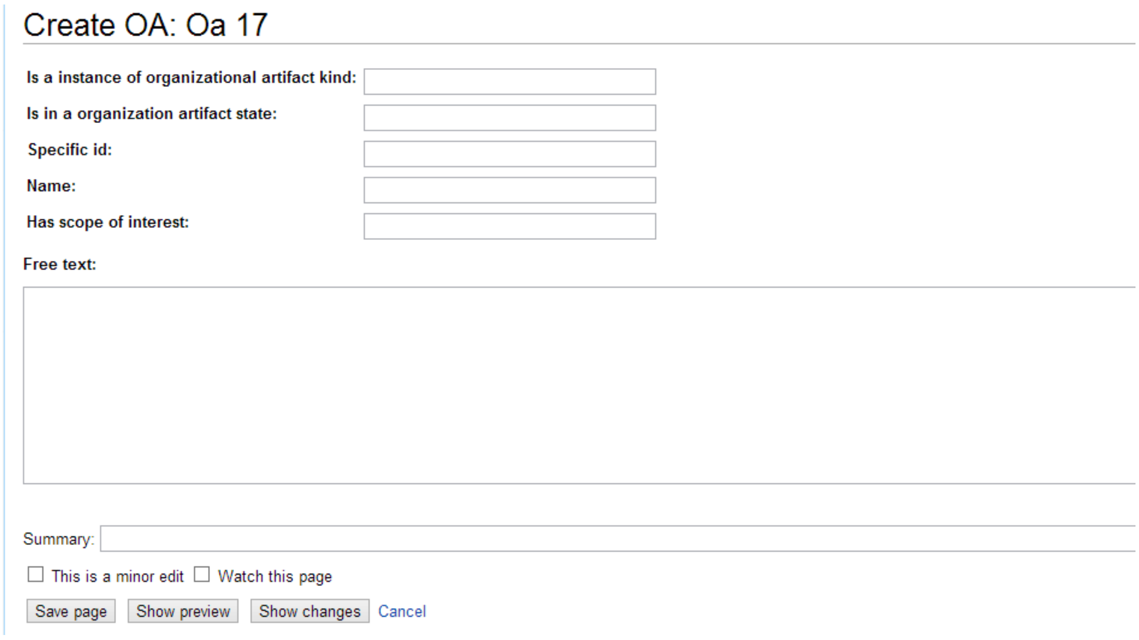
Figure 44 - Scope of interest 1 wiki page

This wiki page is the scope of interest 1 page and besides the properties of the page, it also shows all the diagrams, OAs and OARs that belong to that scope of interest. This is achieved through a SemanticMediaWiki ASK API query, created dynamically when the scope of interest was created on the Universal Diagram Editor. Basically, it asks for all the pages whose properties "Is a perspective of scope of interest", for the diagrams, and

"Has scope of interest", for the OAs and OARs have the value equal to the name of the scope of interest page, which in this case is "Scope of interest 1".

5.3.4. Templates, Forms, and Properties

Through the "Scope of interest" page, we can also create new pages. Under each category of pages, there is a link that allows us to create new pages. It redirects the user to a form, where the page of the corresponding category can be created. For instance, let's click on "+ Create New Organization Artifact". The form to create a new OA can be viewed in Figure 45 - Create new OA page through form. Note: It is also possible to edit a wiki page with a form by clicking the "Edit with form" button.


Create OA: Oa 17

Is a instance of organizational artifact kind:	
Is in a organization artifact state:	
Specific id:	
Name:	
Has scope of interest:	

Free text:

Summary:

☐ This is a minor edit ☐ Watch this page

Figure 45 - Create new OA page through form

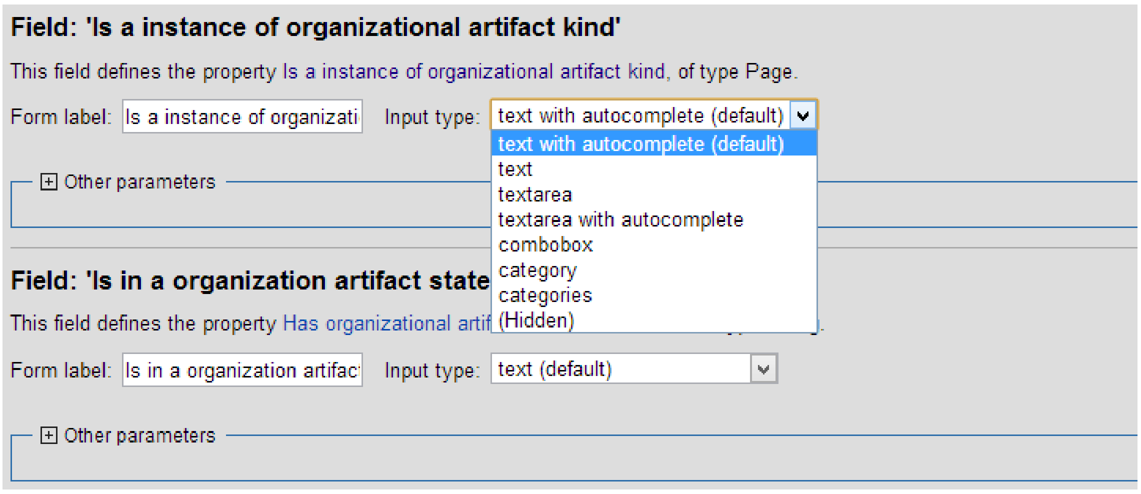
When a user enters a page from the category diagram, he is presented with the diagram page properties, and also with two queries that show both the shapes and connectors that belong to that diagram. This query is also created dynamically at the time of creation of the diagram, and the result is similar to the page "Scope of interest 1", in Figure 44 - Scope of interest 1 wiki page.

We created templates and forms for all wiki pages that need to be created/modified, not also because it's helpful to the user, but also because when creating or editing wiki pages through the SMW sfautoedit API, a form and a template are required. The templates define the structure of the pages, i.e., the properties, and the forms allow users to create and edit pages. In Figure 46 - OA template we can look at how the OA template is defined.

```
{{OA
|Is a instance of organizational artifact kind=
|Is in a organization artifact state=
|Specific id=
|Name=
|Has scope of interest=
}}
```

Figure 46 - OA template

In Figure 47 – OA form, we can see the Form template. To create a form, it is necessary to specify a template from which the form will be created. It takes the properties defined on the template, as labels, and then we can define the input type. We usually leave it at default option (text with autocomplete). Please take a look at Figure 47 – OA form to view an example.



Field: 'Is a instance of organizational artifact kind'

This field defines the property Is a instance of organizational artifact kind, of type Page.

Form label: Is a instance of organizati Input type: text with autocomplete (default)

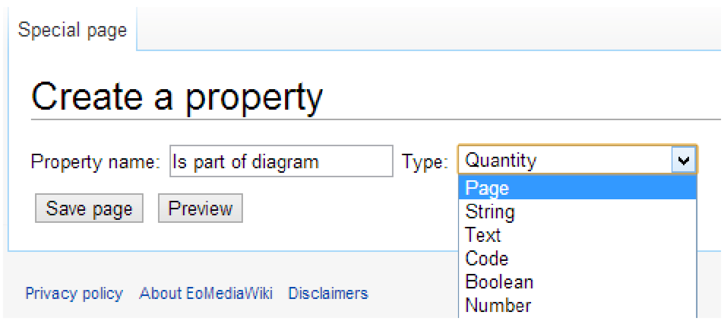
Field: 'Is in a organization artifact state

This field defines the property Has organizational artif

Form label: Is in a organization artifac Input type: text (default)

Figure 47 – OA form

We also need to define the properties that are used on the wiki pages. There is over a dozen of property types, but we only needed to use three types: String, Number and Page. It is a good idea to define the properties because if a value type does not match the defined type, a warning is shown to the user. Also, we define many of the properties with the type "Page", because it is particularly useful when we want the values of the properties to be links to pages that have the same name as the value. In Figure 48 - "Is part of diagram" property creation, we define the property "Is part of diagram" with the type "Page". This helps a lot in navigation between pages.


Special page

Create a property

Property name: Is part of diagram Type: Quantity

Figure 48 - “Is part of diagram” property creation

5.3.5. Implementation Overview

In this subsection we present an overview of how wiki pages are interrelated between each other, and how properties are connected between pages. For that, we will use a part of the “Rent a car” ATD diagram example, depicted in Figure 49, inside the red rectangle. This example includes the shapes «T01-rental start» and «A02-rental starter», and the connector between them, which is of the “EXECUTOR” kind.

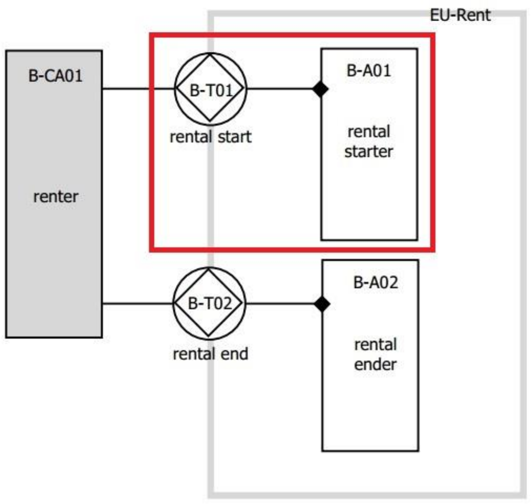
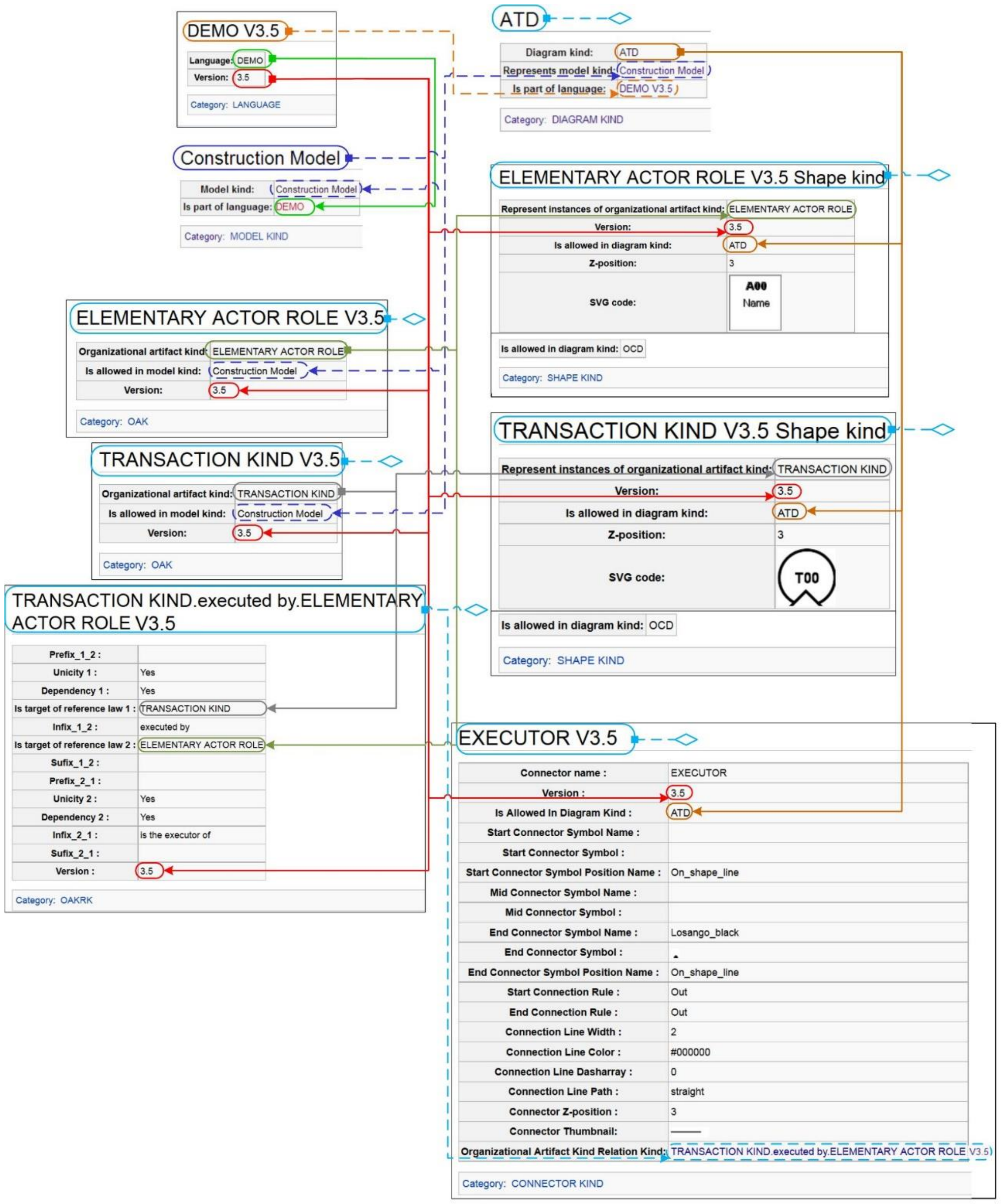


Figure 49 - Part of the “Rent a car” ATD diagram

In Figure 50, we can see the definition of the pages that were used to create this example. These pages were created through the meta-model editor, and they define the format and properties of the pages that are created by the UDE. The wiki pages which were created by the UDE, for this example, are depicted in Figure 51. These two pictures were taken from [31].

Through the scheme of these two pictures, we can easily view which properties, in which pages, have values that correspond to names of existing wiki pages. This is represented by dashed lines, where the in the beginning of the line there is the page, and on the ending of the line, there are the properties which use that page name as a value. A good example would be the “Diagram 1” page, from Figure 51, which is used as the value of the property “is part of diagram”, in “Shape

1”, “Shape 2”, and “Connector 1”, meaning that these symbols belong to that diagram. Through the solid lines, we can observe how a certain property will propagate to other pages, for instance, the value of the property “Version”, from “DEMO V3.5” page, will be the same in all the pages of the of the type “OAK”, “OAKRK”, “SHAPE”, and “CONNECTOR” that are allowed in the “DIAGRAM KIND” (ATD) which is part of that “LANGUAGE” (DEMO V3.5).



DEMO V3.5

Language:	DEMO
Version:	3.5

Category: LANGUAGE

Construction Model

Model kind:	Construction Model
Is part of language:	DEMO

Category: MODEL KIND

ELEMENTARY ACTOR ROLE V3.5

Organizational artifact kind:	ELEMENTARY ACTOR ROLE
Is allowed in model kind:	Construction Model
Version:	3.5

Category: OAK

TRANSACTION KIND V3.5

Organizational artifact kind:	TRANSACTION KIND
Is allowed in model kind:	Construction Model
Version:	3.5

Category: OAK

TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE V3.5

Prefix_1_2 :	
Unicity 1 :	Yes
Dependency 1 :	Yes
Is target of reference law 1 :	TRANSACTION KIND
Infix_1_2 :	executed by
Is target of reference law 2 :	ELEMENTARY ACTOR ROLE
Sufix_1_2 :	
Prefix_2_1 :	
Unicity 2 :	Yes
Dependency 2 :	Yes
Infix_2_1 :	is the executor of
Sufix_2_1 :	
Version :	3.5

Category: OAKRK

ATD

Diagram kind:	ATD
Represents model kind:	Construction Model
Is part of language:	DEMO V3.5

Category: DIAGRAM KIND

ELEMENTARY ACTOR ROLE V3.5 Shape kind

Represent instances of organizational artifact kind:	ELEMENTARY ACTOR ROLE
Version:	3.5
Is allowed in diagram kind:	ATD
Z-position:	3
SVG code:	A00 Name

Is allowed in diagram kind:	OCD

Category: SHAPE KIND

TRANSACTION KIND V3.5 Shape kind

Represent instances of organizational artifact kind:	TRANSACTION KIND
Version:	3.5
Is allowed in diagram kind:	ATD
Z-position:	3
SVG code:	T00

Is allowed in diagram kind:	OCD

Category: SHAPE KIND

EXECUTOR V3.5

Connector name :	EXECUTOR
Version :	3.5
Is Allowed In Diagram Kind :	ATD
Start Connector Symbol Name :	
Start Connector Symbol :	
Start Connector Symbol Position Name :	On_shape_line
Mid Connector Symbol Name :	
Mid Connector Symbol :	
End Connector Symbol Name :	Losango_black
End Connector Symbol :	▲
End Connector Symbol Position Name :	On_shape_line
Start Connection Rule :	Out
End Connection Rule :	Out
Connection Line Width :	2
Connection Line Color :	#000000
Connection Line Dasharray :	0
Connection Line Path :	straight
Connector Z-position :	3
Connector Thumbnail:	——
Organizational Artifact Kind Relation Kind:	TRANSACTION KIND.executed by.ELEMENTARY ACTOR ROLE V3.5

Category: CONNECTOR KIND

Labels:
■- - -➔ Means that the page name of the 1st template will be the value, of the connecting property in the 2nd template.
■- -◇ Same as the previous one but the connecting property is in the 2nd part of this figure
■——➔ Means that the value of that property, of the 1st template, will used in another property in the 2nd template and needs to be created first, in order to do so. (colors differentiate connection of the same type)

Figure 50 - Pages created by the meta-model editor

We can also observe that each page has a category. This category corresponds to the template which defines the properties of each page. This means that the templates are basically the same as the pages, except the properties in the pages are already set with custom values.

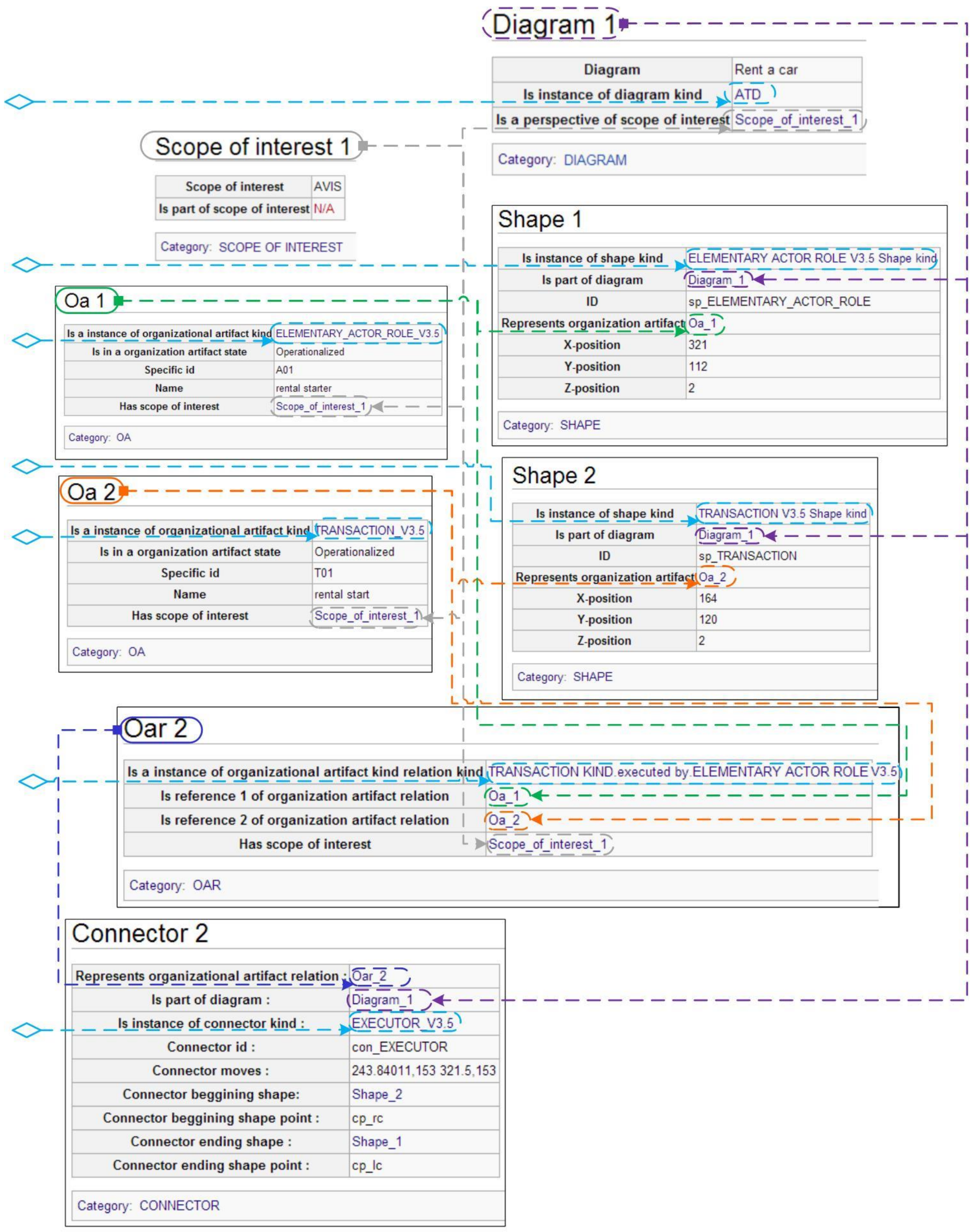


Diagram 1

Diagram	Rent a car
Is instance of diagram kind	ATD
Is a perspective of scope of interest	Scope_of_interest_1

Category: DIAGRAM

Scope of interest 1

Scope of interest	AVIS
Is part of scope of interest	N/A

Category: SCOPE OF INTEREST

Shape 1

Is instance of shape kind	ELEMENTARY ACTOR ROLE V3.5 Shape kind
Is part of diagram	Diagram_1
ID	sp_ELEMENTARY_ACTOR_ROLE
Represents organization artifact	Oa_1
X-position	321
Y-position	112
Z-position	2

Category: SHAPE

Oa 1

Is a instance of organizational artifact kind	ELEMENTARY_ACTOR_ROLE_V3.5
Is in a organization artifact state	Operationalized
Specific id	A01
Name	rental starter
Has scope of interest	Scope_of_interest_1

Category: OA

Oa 2

Is a instance of organizational artifact kind	TRANSACTION_V3.5
Is in a organization artifact state	Operationalized
Specific id	T01
Name	rental start
Has scope of interest	Scope_of_interest_1

Category: OA

Shape 2

Is instance of shape kind	TRANSACTION V3.5 Shape kind
Is part of diagram	Diagram_1
ID	sp_TRANSACTION
Represents organization artifact	Oa_2
X-position	164
Y-position	120
Z-position	2

Category: SHAPE

Oar 2

Is a instance of organizational artifact kind relation kind	TRANSACTION KIND executed by ELEMENTARY ACTOR ROLE V3.5
Is reference 1 of organization artifact relation	Oa_1
Is reference 2 of organization artifact relation	Oa_2
Has scope of interest	Scope_of_interest_1

Category: OAR

Connector 2

Represents organizational artifact relation :	Oar_2
Is part of diagram :	Diagram_1
Is instance of connector kind :	EXECUTOR_V3.5
Connector id :	con_EXECUTOR
Connector moves :	243.84011,153 321.5,153
Connector beggining shape:	Shape_2
Connector beggining shape point :	cp_rc
Connector ending shape :	Shape_1
Connector ending shape point :	cp_lc

Category: CONNECTOR

Labels:
■ – – ➝ Means that the page name of the 1st template will be the value, of the connecting property in the 2nd template.
■ – –◇ Same as the previous one but the connecting property is in the 2nd part of this figure

Figure 51 – Pages created by the UDE (Universal Diagram Editor)

6. Universal Diagram Editor Implementation

In this chapter we describe how the implementation of Universal Diagram Editor was done. In Figure 52 - Universal Diagram Editor (UDE), we can take a look at the final interface of Universal Diagram Editor, with an example already created – the diagram "Rent a car". From now on we will refer to Universal Diagram Editor as the shorthand UDE. To open the UDE the user has to open "Universal Diagram Editor" wiki page on MediaWiki. The UDE is embedded in this wiki page through an iframe. A simple MediaWiki extension was created for this purpose. All the files used to create this web editor are located in "c:\xampp\htodcs\editor" folder.

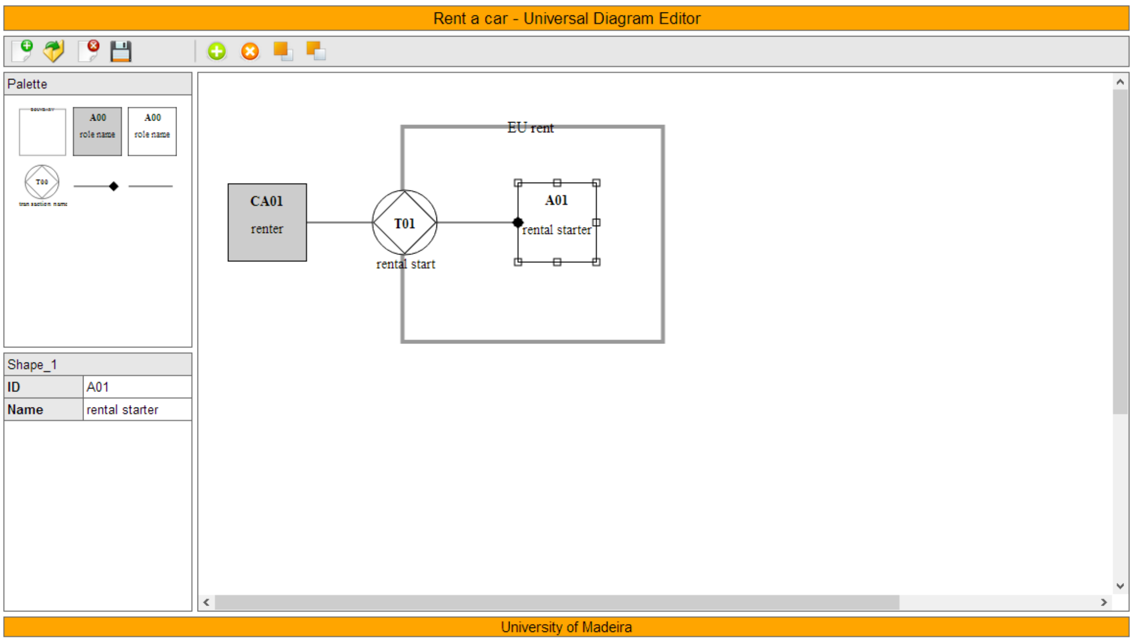


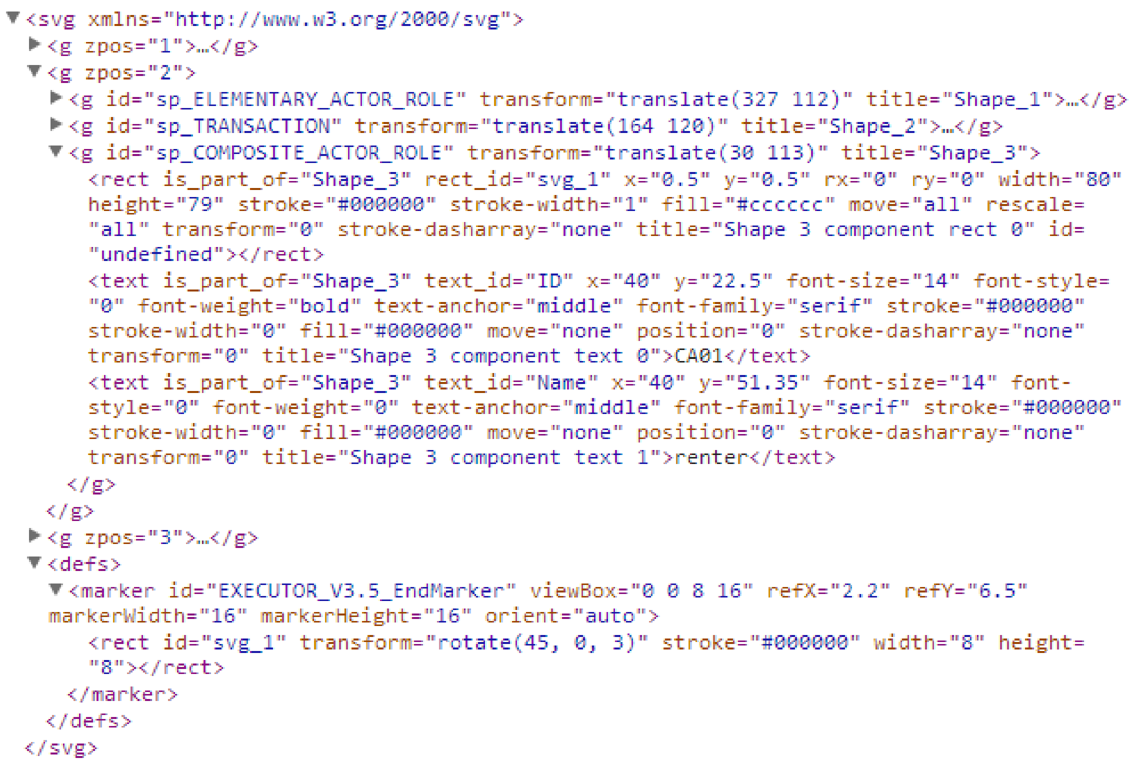
Figure 52 - Universal Diagram Editor (UDE)

The file "editor.html" includes the structure of the document. It is composed by the HTML code and inside there is an inline SVG element, which is the canvas depicted in Figure 52. It is inside this element that more SVG elements can be dynamically added, namely the shapes and connectors. It is on the file "editor.js" that all the handlers and functions to perform the SVG manipulation are included. We decided to use JQuery because it works especially well with the DOM. It is very easy to access, and to create new elements, without the need to reload the page.

Now, looking back at Figure 15 - Universal Enterprise Adaptive Object Model, we can establish a connection between the UDE and UEAOM. This project's context corresponds to the implementation of the concrete syntax, which mean that we will create instances of the classes of the UEAOM model that correspond to the concrete syntax. For instance, the canvas in Figure 52 - Universal Diagram Editor (UDE), is an instance of the class DIAGRAM, which is named “Rent-a-car”. The symbol A01-rental starter is an instance of the class SHAPE and the rectangle that is part of this symbol, is an instance of the class SYMBOL COMPONENT. The connector linking the shapes T01-rental start and A01-rental starter, is an instance of the class CONNECTOR. Also, by looking at Figure 10 - Model triangle applied to organizations, we can say that this diagram is a symbolic system, which is an interpretation of a part of a conceptual system, namely a fragment of the EU-rent case’s construction model and its respective Actor Transaction Diagram. The palette on the top left corner of the UDE, depicted in Figure 52 correspond as well to instances of concrete classes of the UEAOM, namely instances of the classes SHAPE KIND, and CONNECTOR KIND. Also, the shape data section, in the bottom left corner of Figure 52, there is the shape data division. The first column corresponds to instances of the concrete classes “SHAPE PROPERTY” or “CONNECTOR PROPERTY”, depending on which symbol is selected at the moment (shape or connector), and the second column represents instances of the classes “SHAPE PROPERTY VALUE” and “CONNECTOR PROPERTY VALUE” and display the values for the corresponding properties of the selected symbols.

6.1. SVG DOM Structure

Figure 29 shows the DOM structure of the SVG document that corresponds to the diagram “Rent a car”, of the Figure 52. Inside the main <svg> element, there are three main <g> elements, being each <g>, a group that can have one or more children. Each of these three groups define a z-position, meaning that the elements that are inside will have the corresponding z-position. SVG does not have a built-in z-position functionality that allows the user to specify the z-position of a symbol with an integer value. The way that the z-position is defined, is by the order of the elements on the DOM, being the first element the one that is on the back, and the last element the one that is on top. These three groups were created to implement the pre-defined z-positions of the symbol kinds. When a symbol kind is created, the z-position should be defined on moment of the creation, as a property of the symbol. Now, when instances of those symbol kinds are dropped on the canvas, they are automatically inserted in the right group. For example, a symbol whose symbol kind z-position is 2, will be inserted on the group with z-position 2. When we say symbol, we are referring both to shapes and connectors. On the ATD (Actor Transaction Diagram), we defined that the boundary shape kind should have the z-position 1, because it should always stay on the back. The other shape kinds have the z-position value equal to 2, because they should be on the middle, and finally connector kinds have z-position equal to 3, because connectors should always stay on top. When the user drags and drops a symbol from the palette to the canvas, it is automatically placed in the correct z-position. At a DOM perspective, it is placed inside the correct <g>. The advantage of this feature is that the step of defining the z-position is now done automatically and can be skipped by the user, making the interface of the UDE more user-friendly.



```
▼<svg xmlns="http://www.w3.org/2000/svg">
  ▶<g zpos="1">…</g>
  ▼<g zpos="2">
    ▶<g id="sp_ELEMENTARY_ACTOR_ROLE" transform="translate(327 112)" title="Shape_1">…</g>
    ▶<g id="sp_TRANSACTION" transform="translate(164 120)" title="Shape_2">…</g>
    ▼<g id="sp_COMPOSITE_ACTOR_ROLE" transform="translate(30 113)" title="Shape_3">
      <rect is_part_of="Shape_3" rect_id="svg_1" x="0.5" y="0.5" rx="0" ry="0" width="80"
      height="79" stroke="#000000" stroke-width="1" fill="#cccccc" move="all" rescale=
      "all" transform="0" stroke-dasharray="none" title="Shape 3 component rect 0" id=
      "undefined"></rect>
      <text is_part_of="Shape_3" text_id="ID" x="40" y="22.5" font-size="14" font-style=
      "0" font-weight="bold" text-anchor="middle" font-family="serif" stroke="#000000"
      stroke-width="0" fill="#000000" move="none" position="0" stroke-dasharray="none"
      transform="0" title="Shape 3 component text 0">CA01</text>
      <text is_part_of="Shape_3" text_id="Name" x="40" y="51.35" font-size="14" font-
      style="0" font-weight="0" text-anchor="middle" font-family="serif" stroke="#000000"
      stroke-width="0" fill="#000000" move="none" position="0" stroke-dasharray="none"
      transform="0" title="Shape 3 component text 1">renter</text>
    </g>
  </g>
  ▶<g zpos="3">…</g>
  ▼<defs>
    ▼<marker id="EXECUTOR_V3.5_EndMarker" viewBox="0 0 8 16" refX="2.2" refY="6.5"
    markerWidth="16" markerHeight="16" orient="auto">
      <rect id="svg_1" transform="rotate(45, 0, 3)" stroke="#000000" width="8" height=
      "8"></rect>
    </marker>
  </defs>
</svg>
```

Figure 53 - “Rent a car” diagram SVG DOM structure

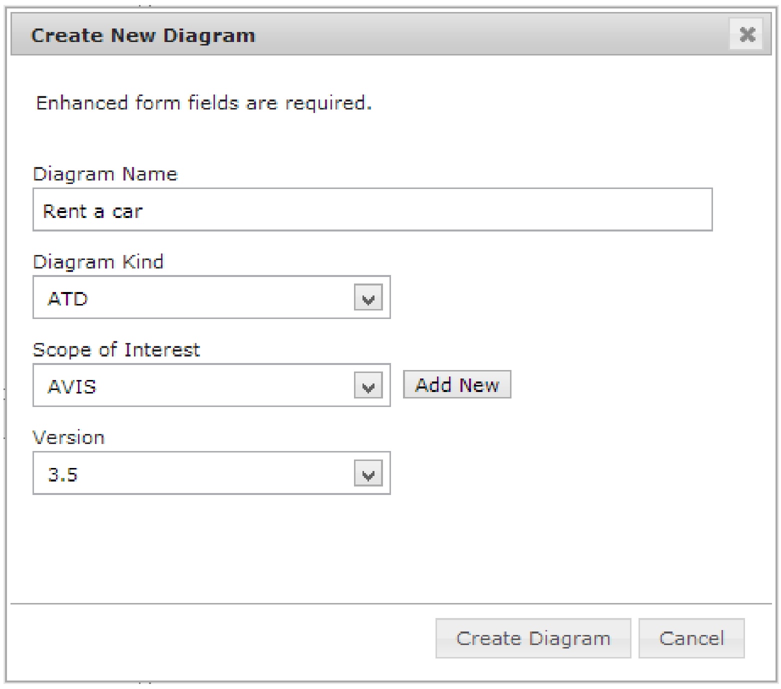
Inside each <g> that defines the z-position, we have symbols. Each symbol is also a group that contains components. For instance, inside the element <g zpos=”2”> there are three groups, each one corresponding to a shape. A group (symbol) has a global x and y position, and when that group is moved, all elements within that group are also move. This is done with the “transform” attribute, that allows a translation, i.e., move the symbol to a desired position. Example: transform=”translate(50 100)”. The first value is the x position the second value is the y position. It is also possible to move components of elements through the x and y attributes, but this position will be relative to the position of the main element, which we call the absolute position. Each element is formed by a group and its children are the components. For instance, the first child of the last shape inside <g zpos=”2”> is a rectangle component that corresponds to a wiki page of the same type of the one shown in Figure 37. There is also the <defs> element on the DOM. This element is used to define shapes that are part of the connectors, for example, the diamond that is part of the executor connector kind. This feature can be achieved with the SVG built-n markers and with be explained in section.

6.2. Features

In this section we describe the features of the UDE, and also explain how the implementation was done, some trade-offs we were faced with, the decisions we had to make, and why we made them.

6.2.1. Create New Diagram

A user can create a new diagram by clicking the “Create New Diagram” button, and the corresponding dialog pops up. We used intuitive icons for the buttons, but if the user is not sure about what each button does, he can place the mouse cursor over the buttons, and a label for each button will be displayed, telling what it does. An example of how to create a diagram is shown on Figure 54.


Create New Diagram

Enhanced form fields are required.

Diagram Name: Rent a car

Diagram Kind: ATD

Scope of Interest: AVIS

Version: 3.5

Figure 54 - Create New Diagram Dialog Form

When the dialog opens, a query is made to SMW (SemanticMediaWiki) to find all diagram kinds that currently exist on SMW. This is done with SMW ASK API and JQuery AJAX function. A query is made to the category diagram kind and once it is completed, the diagram kind names are loaded into the diagram field. The same process is analogous for the “Scope of interest” field. As for the Version field, it is a little different. Initially there are no values for the version field. They only show up once the user selects the diagram kind. Then a query is made to check what versions of shape kinds currently exist for the selected diagram kind. Every time the user changes the Diagram kind field value, a new query is made and the version field values are updated dynamically. Once the user clicks the button “Create Diagram”, the diagram wiki page is created with properties and values of the form, except for the version that is not a property of the diagram. The title of the page is generated automatically. It will always have the prefix “Diagram_”, followed by an integer. At the same time the diagram is created, the palette is dynamically loaded, based on which diagram kind and version were chosen. The thumbnails are created based on the SVG code that was generated, when the shape kind was defined. We used the html logo class, which is a class that can transform an SVG image into a thumbnail dynamically.

6.2.2. Create New Scope of Interest

Scopes of interest define a context where diagrams, OAs and OARs can be associated. A new scope of interest can be created by clicking the “Add New” button, in the “New Diagram Dialog” (Figure 54). A new dialog form will pop up and once the form is submitted, the corresponding wiki page will be created with the property values previously inserted on the form.

6.2.3. Open Diagram

A user can open a diagram by clicking the “Open Diagram button”. A dialog form with list of all the diagrams that currently exist on SMW will pop up. Then, the user has to select the desired diagram and click the button “Open Diagram”. When a diagram is opened, the palette is also loaded, according to the diagram kind, and the diagram version that was chosen at the diagram creation time.

6.2.3.1. Shapes

After the palette is loaded, the shapes are loaded. When loading, SVG elements need to be recreated, because they are not saved as SVG code. The only place where data about symbols is saved, is on wiki pages. The SVG code could be saved into a file on the server, but if a user modified a property of an OA, for instance the name, which corresponds to the shape name label, the next time the diagram was opened, the label would not be correct. So instead we decided to load the shapes dynamically, according to the properties on the wiki pages. First, the main element is created, and then a query to all components of that shape is performed, and SVG components are created. Each property and property value of the component wiki page correspond to an attribute value of the SVG element. The labels of shape are retrieved from the OA whose shape represents.

6.2.3.2. Connectors

The next step is to load the connectors. When loading a connector to the canvas, a query to all connectors that are part of the diagram we are opening, is made. Then the main <g> element is created, taking as attributes names and values, some properties and property values of the connector page, like the connector beginning shape and the connector ending shape. Inside the <g> element, a <polyline> element is created, which is the element that defines the connector line, and also the properties of the page, like the connector moves are retrieved, in order to create the element. An example of a connector page is in Figure 41. The z-position of the main element and the line width and line color are retrieved from the connector kind page. An example of a connector kind page is in Figure 40.

6.2.3.3. Markers

Let's begin with an explanation of what a marker is. A marker is a way to add shapes regularly along elements such as a polyline. For this project, we only use markers on polylines. An example of a marker can be viewed at the bottom of Figure 53. Inside the marker, we define the shape that we want to add to the polyline. It could be an arrowhead, or a diamond shape, like the one represented in the connector on Figure 52. We use two types of markers: marker-start and marker-end, meaning that we can add a shape at the beginning or at the ending of a polyline. To add a marker to a polyline, the attribute marker-start or marker-end has to be specified, with the URL of the marker. Example: marker-end="url(#EXECUTOR_V3.5_EndMarker)". One of the great advantages of using markers is that every time we move the polyline on the canvas, the shape defined by the marker, follows the line automatically. We don't need to worry about moving the shape, just the polyline. The first time we tried to implement the connector endings, we did it without markers, and some problems came up, like calculating the position of the shape and moving the shape along with the connector. For each shape, we needed a different formula to calculate the center of the shape and add it to the connector in the correct position. The markers solved all of these problems. Another great advantage of the markers is that we can reuse code. The marker for a given connector kind only needs to be defined once, for any number of connectors. All the connectors will use the same marker, since they are of the same kind. When we load each connector, we only need to define the marker for the first connector. The following connectors of the same kind will use the same marker, the only thing that needs to be defined are the attributes marker-start and/or marker-end, on the polyline element, so that the connector "knows" what marker to use and in what position.

6.2.4. Delete Diagram

This is a basic feature that simply will delete the diagram that is currently opened. It will clear the canvas, the palette, and the title of the diagram. It also deletes the corresponding diagram wiki page, and all the symbol wiki pages (connectors and shapes) that are part of the diagram. It obviously also deletes all the symbol component pages of each symbol.

6.2.5. Delete Symbol

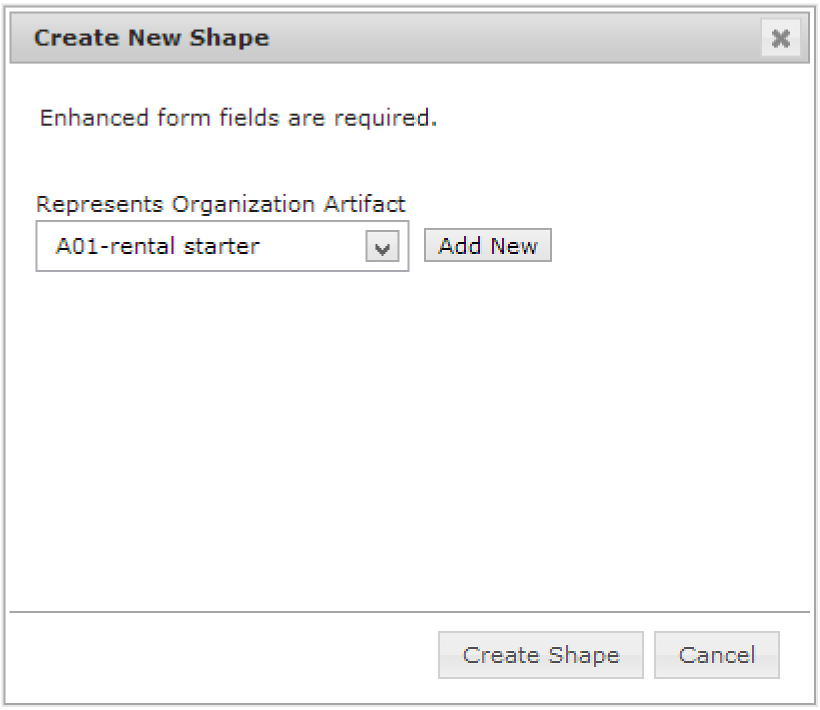
The “Delete Symbol” feature allows a user do delete shapes or connectors. When a symbol is selected and the user presses the delete key, the UDE deletes the SVG element from the DOM and the wiki page(s) associated with that symbol.

6.2.6. Export Image

This another basic feature that allows the user to download the current diagram as a file with SVG extension, to the client machine, through the browser “save as” dialog. This could be useful if the user wants to keep the diagram to view it later when offline, or for any other purpose. Before saving the diagram, we check whether a symbol is selected or not, and deselect it in the affirmative case. Because the selection handles and connector points are also implement in SVG, they need to be removed, otherwise they will count as if they were part of the image.

6.2.7. Create New Shape / Organization Artifact

To create a new shape, a user has to use the “drag and drop” function that was also part of the implementation. It is done by clicking on the desired shape kind, and dragging it and dropping it to the canvas. The “New Shape” dialog form in Figure 55 should pop up.


Create New Shape

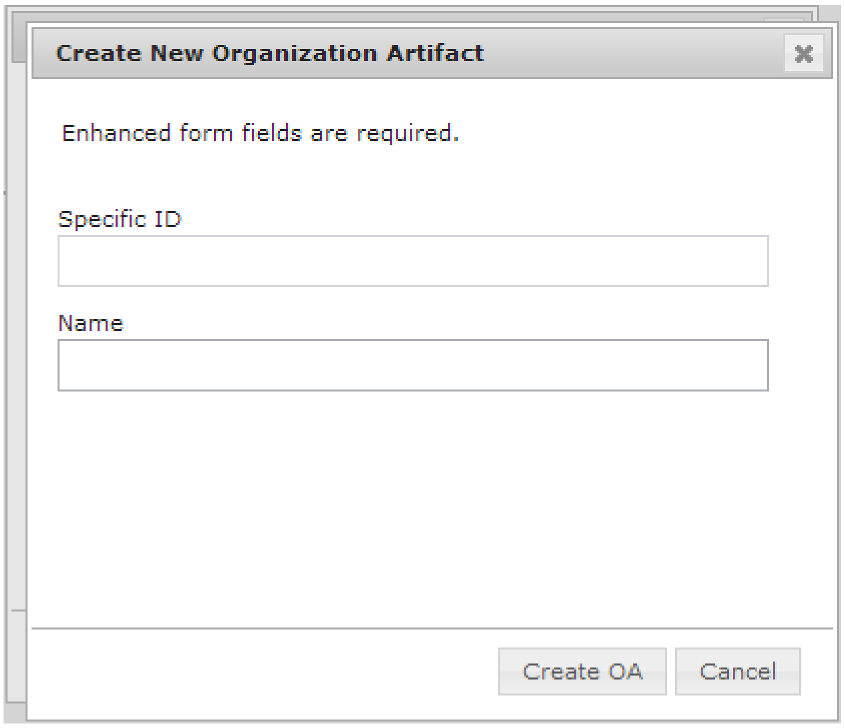
Enhanced form fields are required.

Represents Organization Artifact: A01-rental starter

Figure 55 - Create New Shape dialog form

The UEAOM defines that a shape represents an OA, so, before creating a shape, the OA whose shape will represent, has to be specified. An already existing OA can be specified from the list of OAs that was loaded from MediaWiki, when the dialog opened, or a new OA can be created. It is important to say that the OAs that were loaded to the dialog form are all the instances of the OAK that corresponds to the chosen shape kind, and at the same have the scope of interest of the current diagram.

To create a new OA, the user has to click the "Add New" button and another dialog should pop-up on top of the current one, like it is shown on Figure 56. When the dialog opens, it can be noticed that the "Name" field is enhanced when compared to the "Specific ID" field. This means that the "Name" field is a required field, and the "Specific ID" field is optional. If the user fills in the form and clicks the "Create OA" button, a new wiki page will be created for that OA, and the user will get back to the previous form, where the "Represents Organization Artifact Field" will be updated with the OA that the user had just created a moment ago. The user can now select the OA from which he wants to create the new shape, and click the "Create Shape" button. The form will close, and the corresponding SVG element is added to the canvas, on the position where the user triggered the "mouse up" event. The way the shape is created is similar to the process of loading a shape, when opening a diagram, which was explained on the "Open Diagram" subsection. The only difference is that, instead of reading values from the existing pages to generate the SVG, the main pages and symbol component pages are created based on the data submitted on the form, and the SVG element is created based on the same data and added to the canvas.



Enhanced form fields are required.

Specific ID	
Name	

Figure 56 - New OA form dialog

6.2.8. Create New Connector / Organization Artifact Relation

To create a new connector, the user needs to click on the desired connector kind, on the palette, and immediately the connector points of all shapes are shown, as we can see on Figure 57. These connector points are not more than svg <circle> elements that are added to the shape main element. In the case of the composite actor role, for instance, they are added relatively to the rectangle. The position where to add the connector points has to be calculated, so, for the top three connector points, we specify the position of the circles as 35%, 50% and 75% of the width of the rectangle. Please note that the top left corner of the rectangle is its reference point. If the rectangle is resized, the positions of the connector points will adjust in a proportional way. A similar reasoning can be done for the other connector points, and for the other shapes.

To draw the connector, the user has to click on the connector point of a shape, drag it and release it on the top of a connector point of another shape. The connector SVG element is created and its corresponding wiki page as well. An example of the created wiki page can be viewed in Figure 41. The OAR wiki page is also created, if it does not already exists. An example of this page can be seen on Figure 39. The user does not need to worry about centering the connector on the connector point of the shape, because the editor automatically adjusts the position correctly. If when drawing the connector, the mouse is release somewhere on the canvas, and not over a shape connector point, the connector is not draw. We assumed that it was not possible to have connectors on the canvas that did not have both ends connected to shapes. We have already seen how a connector looks like after being created. The "Rent a car" diagram on Figure 52, has two examples of connectors - an initiator and an executor.

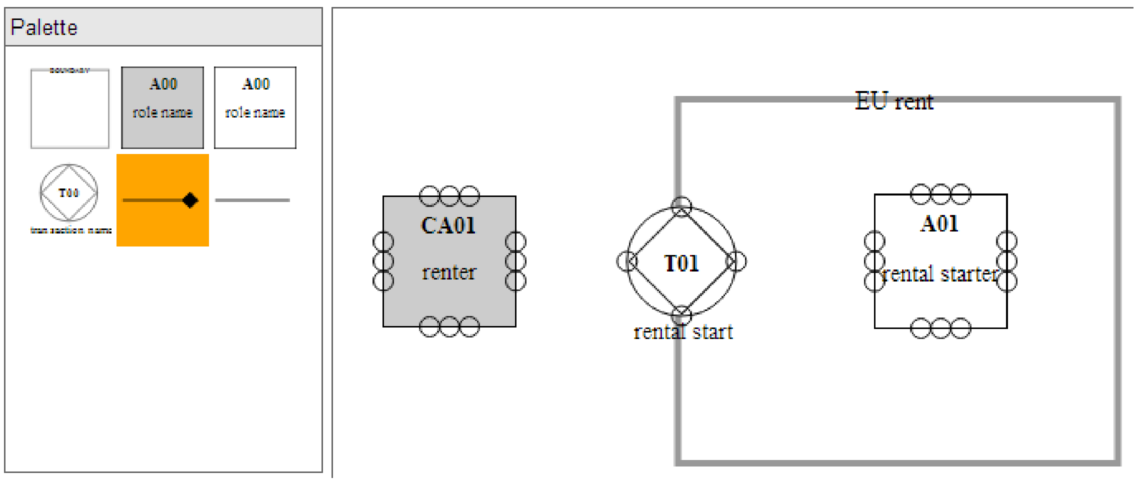


Figure 57 - Shape connector points

6.2.9. Select Symbol

When a shape is selected, the resize handles are placed on the shape's boundary. They indicate that the shape is selected and also enable the shape to be resized. Resize handles are little rectangles, created with the SVG <rect> element. As for the connectors, when selected, they show the connector moves, and not resize handles as in the shapes, because connectors do not need to be resized. The resize of the connector is done in the moment of creation, or when the shapes whose connector is linked to are moved or resized. When a shape is selected, the "Shape Data" division on the editor shows all the labels and label value of that shape. In the case of the connector, the beginning and ending shape properties of the connector are shown. The title is also shown for both shapes and connectors and corresponds to the shape wiki page of the symbol.

6.2.10. Resize Shape

A shape can be resized through the resize handles. It can be resized horizontally and vertically, or both at the same time. Every time a shape is resized, the coordinates of that shape are updated in the corresponding wiki page as well. It was not possible to create a general function to do the shape resizing. Each SVG element type, such as <rect> or <circle> need to have its own resizing function, because they are different types of shapes and the resize is done differently. For instance, in the case of a rectangle, it is done through width and height, and in the case of a circle it done through the radius. Figure 58 illustrates a shape being resized. It can be noted that the connector linked to the shape, moved along with the shape, because the position of the connector point to which that connected was linked to, changed as well.

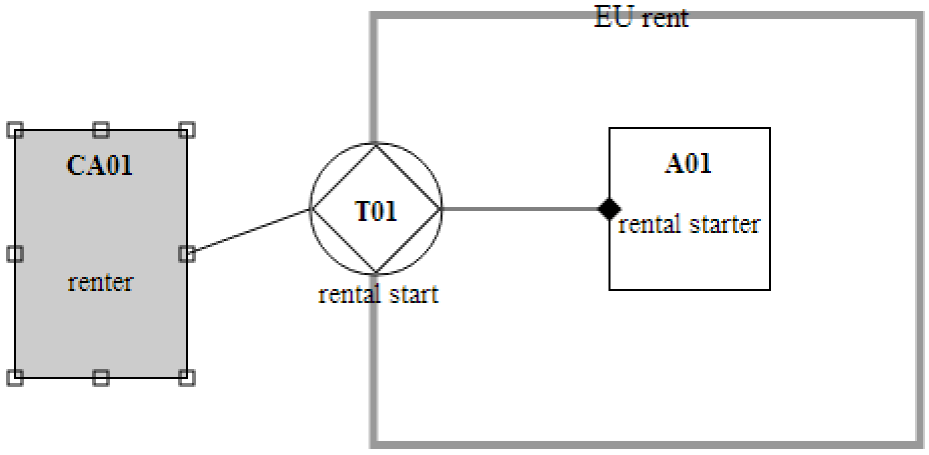


Figure 58 - Shape resizing

Restriction were added when rescale a shape. If the “resize” property is defined to horizontal, or vertical, the resizing can only be done in those directions. If the “resize” property is defined to “none” value, the resizing is disabled. These restrictions are defined at the moment of creation of the shape kind.

6.2.11. Move Shape

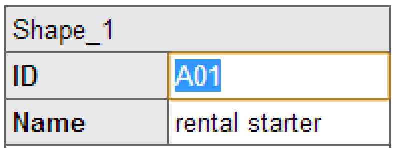
A shape is moved, by moving the group of elements that compose that shape. In SVG, if a group is moved, all elements within that group move as well. Restrictions also apply. When a shape kind is created, the “move” property can be defined to horizontal, vertical, or “none”, meaning that the movement is disabled. When a shape is moved, the connectors linked to that shape will also move along with the shape.

6.2.12. Edit Connector Points and Connector Moves

Connector points are the two points on the beginning and ending of a connector which are used to link a connector to a shape. If a user wants to edit a connector, when clicking and holding on one of the two connector points, all the connector points of all shapes will show up, and the user has to drag the connector point to a connector point of a shape (can be the same shape or other different shape), to perform the edit. If no shape connector point is detected, the action is undone. Connector moves are the inner points that define a connector. These points do not allow connections (that’s why they are called connector moves), and are used only to adjust the direction of the line segments of a connector, by clicking on them and dragging them to the desired position. When editing a connector move, the editor checks if its x position is “close” to the x position of the previous or next connector move, and if so, the x position of the connector move that is being edited, is set to the same x position of the other connector move, so that the line segment between the two connectors moves can define a vertical line. The same process is done to the y position of the connector move, so that we can have horizontal lines. When the both scenarios happen at the same time, we get a right angle somewhere on the connector. This is a great solution to automatically align the connectors.

6.2.13. Edit Labels

When a shape is selected, all the labels and label values of that shape are displayed in the “Shape Data” division, under the palette. An example is depicted in Figure 59.



Shape_1	
ID	A01
Name	rental starter

Figure 59 - Edit Shape Labels

The editing can be done on the right column, which corresponds to the property values. When a shape’s label is edited, the corresponding property value is edited on the OA wiki page whose shape represents. Labels are part of the OA page, and not of the shape page, because the shape page inherits the labels from the OA page.

6.2.14. Add/Remove Connector Moves

Connector moves the points that define the line of the connector. A connector, by default, is created with two connector moves, which correspond to the connector points of that connector, and also correspond to the beginning and ending of the connector line. To add connector moves to a connector, the connector has to be selected, and the "Add CM" button has to be clicked. Figure 60 illustrates a connector with three connector moves.

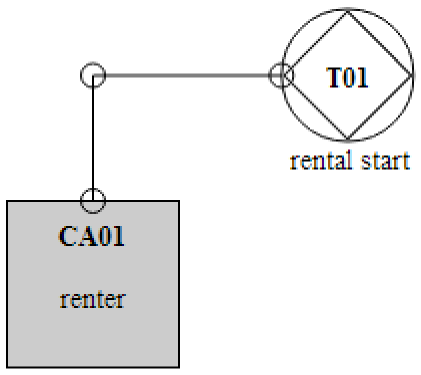


Figure 60 - Connector moves

If the angles made by the line of the connector are close to 90 degrees, the connector moves in the middle are adjusted so that the line can form a right angle. The deletion of connector moves is done through the "Remove CM" button.

6.2.15. Bring to Front / Send to Back

When a symbol kind is created, a default z-position is defined and all instances of that shape kind should have that same z-position. However, a user may still want to send a shape to back or bring it to front, within the group of shapes that share the same z-position. That is why the buttons "Bring to Front" and "Send to Back" were added to the UDE. What they do is simply alter the order of the SVG elements on the DOM. When a user wants to send a symbol to the back, the corresponding SVG element is moved to the first position, otherwise if the user wants to bring a symbol to the front, the corresponding SVG element is sent to the last position. As said previously, what defines the z-position of SVG elements, is their order on the DOM.

6.3. Actions, Pages, and Properties Overview

In this section, we present two tables with an overview of which properties are created for each page, and how their values are defined, and also show which pages and properties of each page are updated, per each different type of user action in the UDE.

In Table 7 we have all types of pages that can be created through the UDE. In the first column we have all the possible page kinds, and an example of each page kind (in bold). In the "Page Properties" column, we have all the properties for each page, and in the column "Page Property Values" we have some page examples, taken from the example used in this chapter, which is depicted in Figure 52. The last 2 columns specify which property values are defined by the user, and which property values that are automatically defined. The properties values of the wiki pages of type SYMBOL COMPONENT are all set automatically, and are not shown in this table for simplicity reasons.

New Page (kind and example)	**Page Properties**	**Page Property Values (examples)**	**User-defined**	**Auto-defined**
SCOPE OF INTEREST **(Scope of interest 1)**	Scope of interest	AVIS MADEIRA	•	
	Is part of scope of interest	AVIS	•	•
DIAGRAM **(Diagram 1)**	Diagram	Rent a car	•	
	Is instance of diagram kind	ATD	•	
	Is a perspective of scope of interest	Scope_of_interest_1	•	
OA **(Oa 1)**	Is an instance of organizational artifact kind	ELEMENTARY_ACTOR_ ROLE_V3.5		•
	Is in an organization artifact state	Operationalized		•
	Specific id	A01	•	
	Name	rental starter	•	
	Has scope of interest	Scope_of_interest_1		•
SHAPE **(Shape 1)**	Is instance of shape kind	ELEMENTARY ACTOR ROLE V3.5 Shape kind		•
	Is part of diagram	Diagram_1		•
	ID	sp_ELEMENTARY_ACT OR_ROLE		•
	Represents organization artifact	Oa_1	•	

	X-position	321		•
	Y-position	112		•
	Z-position	2		•
OAR **(Oar 1)**	Is an instance of organizational artifact kind relation kind	TRANSACTION KIND.initiated by.ELEMENTARY ACTOR ROLE		•
	Is reference 1 of organization artifact relation	Oa_2		•
	Is reference 2 of organization artifact relation	Oa_3		•
	Has scope of interest	Scope_of_interest_1		•
CONNECTOR **(Connector 1)**	Represents organizational artifact relation	Oar_1		•
	Is part of diagram	Diagram_1		•
	Is instance of connector kind	INITIATOR_V3.5		•
	Connector id	con_INITIATOR		•
	Connector moves	113.5,153 177.84011,153		•
	Connector beginning shape	Shape_3		•
	Connector beginning shape point	cp_rc		•
	Connector ending shape	Shape_2		•
	Connector ending shape point	cp_lc		•

Table 7 – User-defined and auto-defined property values

Table 7 shows which properties are created for each page, and how their values are set. Table 8 shows, for each user action, which kinds of pages are edited, and the properties whose values should be updated, for each page.

User Action	**Edited Page Kinds**	**Edited Properties**
Resize Shape	SHAPE	X-position Y-position
	RECT SYMBOL COMPONENT	Width

		Height
	CIRCLE SYMBOL COMPONENT	r
	TEXT SYMBOL COMPONENT	x y
	CONNECTOR	Connector moves
Move Shape	SHAPE	X-position Y-position
	CONNECTOR	Connector moves
Edit ID Label	TEXT SYMBOL COMPONENT	ID
Edit Name Label	TEXT SYMBOL COMPONENT	Name
Add/Edit/Remove Connector Move	CONNECTOR	Connector moves
Edit Connector Point	CONNECTOR	Connector moves Connector beginning shape Connector beginning shape point Connector ending shape Connector ending shape point

Table 8 – Updated pages and properties by action

6.4. System Architecture

In Figure 61, we can view what the system does when the user edits a symbol. On the client side, a JQuery mouse-click handler function fires to update the corresponding symbol(s)' SVG code on the DOM. At the same time, using JQuery Ajax and SMW API combined, a query is made to SMW, and the database tables of the wiki page(s) that correspond to the edited symbols, are updated.

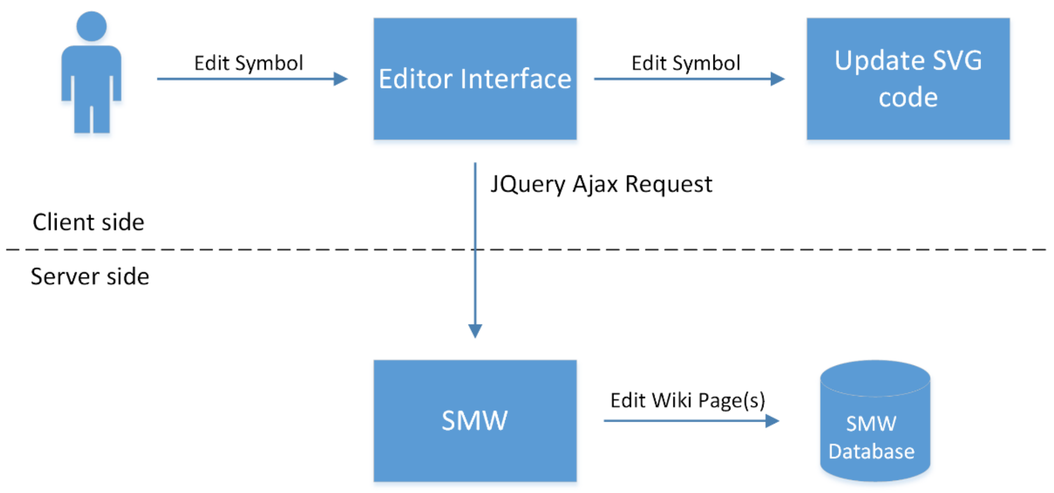


Figure 61 – System Architecture

7. Conclusions

7.1. Technologies and Implementation

SVG and JQuery proved to be an efficient technology combination for the implementation of the UDE (Universal Diagram Editor), because it provided smooth symbol editing, with easy data accessibility through the DOM. However, when synchronizing the UDE with SemantiMediaWiki, performance decreased a little, since wiki pages were being created/edited simultaneously through Ajax requests, as the user performed changes in the graphical user interface. But we assume this is normal, since synchronous Ajax requests also had to be made to SemanticMediaWiki, and not only the default asynchronous. Synchronous Ajax requests mean that the requests are performed sequentially, and this is useful when we need to make a request using data from the previous request.

SemanticMediaWiki was a good choice for the implementation of this project, because it has good support for page navigation, and page editing. Its ASK API was also very useful for inline queries to wiki pages in a table format, allowing the user to view each page's properties without the need to visit the page. The ASK API was also very useful for querying wiki pages through Ajax requests. The idea of saving data about SVG components in wiki pages was a good idea in terms of organization, but in terms of performance, not the best solution, since several pages have to be created when a shape is created, and several pages have to be read when a shape is opened on the editor.

One of the main innovative contributions of this project is to allow dynamic loading of new palettes, whose symbol kinds are also created dynamically by the user through the meta-editor. The connectors are working well with shapes whose boundary is a rectangle or a circle, but the editor can be extended support connections between more elaborate shapes.

The usage of markers revealed to be an excellent solution for the implementation of the connector endings, because it provides reusability of code, and the connector endings move automatically along with the connector, once they are defined and assigned to that connector. But there is still one problem, SVG support for middle markers is not great because if there is more than one connector move in the middle of a connector, the middle marker shows up more than one time, when the desired number would be one.

The final product of the integration of this project with the project dealing with the meta-model editor went reasonably well, because both developers of each project discussed their ideas during weekly meetings along with their supervisor, and for the most cases, a solution to the problems that came up, was agreed between both.

7.2. Future Work

This work is to be taken and improved in the future, and in this sub-section, we describe some aspects of the UDE which can be improved, and leave some suggestions for these improvements, to be considered by future developer(s).

7.2.1. Multiple Symbol Selection/Moving

The final version of the UDE which was implemented, does not allow multiple selection/moving of symbols, only single selection/moving. Single selection is simpler to implement because we can use the JQuery mouse click event handler, which automatically detects which symbol was clicked, and from there the shape can be selected. However, when trying to select multiple symbols, a higher level of complexity is added, because JQuery event handler can't do this alone. To do this, a rectangle had to be drawn upon user selection, and all symbols' positions and boundaries had to be checked if they were inside the boundaries of the drawn rectangle, and if so, all these symbols could be selected. An easier to implement approach, which would be better for selecting specific symbols, but less practical for selecting a group of symbols close to each other, would be holding a key, for instance, the shift key, and the using the JQuery mouse click event handler to select each specific symbol. After the implementation of the multiple symbol selection, the multiple symbol moving would be easier and could be done by editing the global coordinates of each selected symbol.

7.2.2. Deletion of OAs and OARs

When a new shape is created, it will represent an OA (organization artifact). This OA can be an existing one on SMW database, or can be created by the user. However, when deleting a shape, the OA whose shape represents is not deleted, because there might be other shapes in other diagrams representing that OA. The deletion of OAs is a problem that can be addressed in future work. A possible solution would be, when deleting a shape, to check if that shape is the only representation of that OA, and if so (and if the user agreed), the OA would be deleted as well. Another issue is that, currently, it is possible to delete any OA through SMW interface, and it shouldn't be allowed to delete OAs that have shapes representing them. Restrictions could be created to solve this problem, or another solution would be to ask the user if he also wants to delete all shapes which represent the OA. As for the connectors, when they are created, the editor checks if there is any existing OAR (organization artifact relation) that the connector can represent, and if not, a new OAR is automatically created. The problem of OARs deletion is analogous to the problem of OAs deletion.

7.2.3. Connectors and OARs

Connectors can only be created when the beginning and ending shapes which the connector will link, are already created. So, when deleting a shape that is linked to a connector, there is problem of deleting (or not) that connector. Currently, the connector is not deleted, but its "beginning shape" or "ending shape" property will still have the value of the deleted shape. This value could be updated to null, or the connector could be completely deleted to enforce the idea that there shouldn't be allowed connectors with "unlinked" endings.

Another issue about connectors that should be addressed, is the scenario where we have several connectors representing the same OAR, in different diagrams. If we edit a connector in one of the diagrams, the corresponding OAR has to change as well, and it wouldn't make sense in editing all connectors in the other diagrams. A possible solution would be to create a new OAR (or check for an existing one) for the connector that was edited, and the other connectors in the other diagrams would still represent the "old" OAR.

8. References

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

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

[3] 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.

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

[5] “MediaWiki”, Available at: http://en.wikipedia.org/wiki/MediaWiki, 2013

[6] “Introduction to Semantic Media Wiki”, Available at: http://semantic-mediawiki.org/wiki/Help:Introduction_to_Semantic_MediaWiki, 2013

[7] “Semantic Forms”, Available at: http://www.mediawiki.org/wiki/Extension:Semantic_Forms, 2013

[8] “Scalable Vector Graphics”, Available at: http://en.wikipedia.org/wiki/Scalable_Vector_Graphics, 2013

[9] “A Rect Example”, Available at: http://tutorials.jenkov.com/svg/rect-element.html#rect-example, 2012

[10] “SVG polyline”, Available at: http://www.w3schools.com/svg/svg_polyline.asp, 2012

[11] “JavaScript”, Available at: http://en.wikipedia.org/wiki/JavaScript, 2013

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

[13] “JQuery”, Available at: http://en.wikipedia.org/wiki/JQuery, 2013

[14] “Ajax”, Available at: http://en.wikipedia.org/wiki/Ajax (programming), 2013

[15] “How to choose between SVG and canvas”, Available at: http://msdn.microsoft.com/en- us/library/ie/gg193983(v=vs.85).aspx, 2013

[16] “SVG or Canvas? Choosing between the two”, Available at: http://dev.opera.com/articles/view/svg-or-canvas-choosing-between-the-two/, 2010

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

[18] Hommes, Bart. Jan, “Introduction to Modelworld”, 2012

[19] Baranovskiy, D., “Raphael Javascript Library”, Available at: raphaelJS.com, 2013

[20] “Jquery SVG”, Available at: http://keith-wood.name/svg.html, 2012

[21] “SVG-edit”, Available at: https://code.google.com/p/svg-edit/, 2013

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

[23] J. Dietz, “DEMO-3 Way of Working”, 2009.

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

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

[26] Dietz, J.L.G., Enterprise ontology: theory and methodology. Springer-Verlag New York, Inc. Secaucus, NJ, USA. (2006).

[27] Dietz, J. L. G. 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, 2009

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

[29] Ferreira, H. S., Correia, F. F. & Welicki, L. 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, 2008. pp 5:1–5:9. Available at: http:// doi.acm.org/ 10.1145/ 1753196.1753203.

[30] Aveiro, D., Pinto, D. “IMPLEMENTING ORGANIZATIONAL SELF AWARENESS, A Semantic MediaWiki based Enterprise Ontology Management approach”, KEOD 2013, Vilamoura, Portugal, 2013

[31] Nóbrega, V., “Wiki-Aided Meta Modeling”, Universidade da Madeira, Funchal, Portugal, Setembro 2013

9. Appendix

A.1 Installation manual

The following guide is a partial transcription from J. Capela's master thesis [17] entitled "Semantic MediaWiki adaptation to support Organizational Engineering", with some additions and modifications.

The OS used for this installation was Windows 8, 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 browsers are Google Chrome and Mozilla Firefox (other browsers should work but they were not tested).

If you do not wish to make a fresh installation, you can copy the xampp folder to the root of your hard drive. Then you double-click that folder, find xampp control panel, open it and start the services apache and MySQL. Then open your browser and type "localhost/mediawiki/" and you should have access to MediaWiki main page.

However, if you prefer to make a fresh install, please proceed to the following steps:

1. Install «xampp-win32-2.5-installer.exe» (or higher version) as Administrator
    a) The installation dir. should be «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 «read only» 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

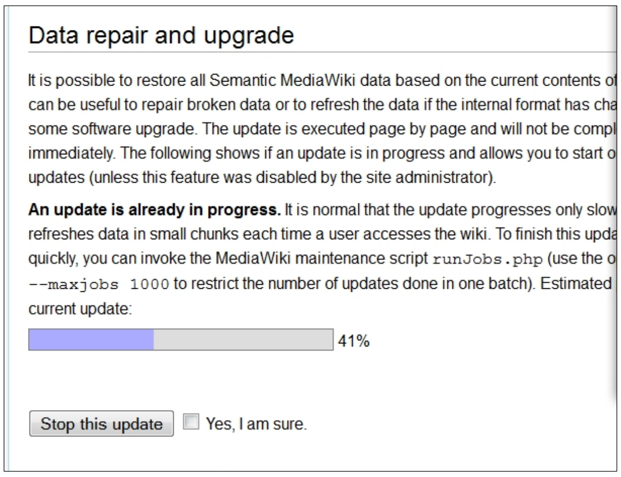
b) Password: password

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

13. Click the «Initialize 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 62 is possible to see a screen print of the original installation used for this project.



Data repair and upgrade

It is possible to restore all Semantic MediaWiki data based on the current contents of
can be useful to repair broken data or to refresh the data if the internal format has cha
some software upgrade. The update is executed page by page and will not be compl
immediately. The following shows if an update is in progress and allows you to start o
updates (unless this feature was disabled by the site administrator).

An update is already in progress. It is normal that the update progresses only slow
refreshes data in small chunks each time a user accesses the wiki. To finish this upda
quickly, you can invoke the MediaWiki maintenance script `runJobs.php` (use the o
`--maxjobs 1000` to restrict the number of updates done in one batch). Estimated
current update:

41%

Figure 62 - 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 «semantic_forms_2.5.1.zip» file to wiki «extensions» folder.

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

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

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

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

25. Extract «ImageMagick-6.8.5-9.zip » file to the htdocs folder.

26. Copy editor folder to xampp/htdocs folder

27. 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("$IP\extensions\diagram-editor.php");
include("$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;
###########
```

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

29. 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 check if the installed software and installed extensions are listed in the page.

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

A.2 UDE (Universal Diagram Editor) Code

A.2.1 HTML code (editor.html)

```
<html xmlns="http://www.w3.org/1999/xhtml">

<head>
      <title>Universal Diagram Editor</title>
      <script src="js/jquery-1.10.2.min.js"></script>
      <script src="js/editor.js"></script>
      <script src="js/jquery.generateFile.js"></script>
      <link rel="stylesheet" type="text/css" href="css/editor.css">
      <link rel="stylesheet" href="jquery-
ui/themes/base/jquery.ui.all.css">
      <script src="jquery-ui/ui/jquery.ui.core.js"></script>
      <script src="jquery-ui/ui/jquery.ui.widget.js"></script>
      <script src="jquery-ui/ui/jquery.ui.mouse.js"></script>
      <script src="jquery-ui/ui/jquery.ui.button.js"></script>
      <script src="jquery-ui/ui/jquery.ui.draggable.js"></script>
      <script src="jquery-ui/ui/jquery.ui.position.js"></script>
      <script src="jquery-ui/ui/jquery.ui.resizable.js"></script>
      <script src="jquery-ui/ui/jquery.ui.button.js"></script>
      <script src="jquery-ui/ui/jquery.ui.dialog.js"></script>
      <script src="jquery-ui/ui/jquery.ui.effect.js"></script>
</head>

<body>

<div id="container">

      <div id="header"><h1 id="mainTitle">Universal Diagram
Editor</h1></div>

      <div id="toolbar">
            <img id="newDiagram" class="icon"
src="images/newDiagram.png" title="New Diagram">
            <img id="openDiagram" class="icon"
src="images/openDiagram.png" title="Open Diagram">
            <img id="deleteDiagram" class="icon"
src="images/deleteDiagram.png" title="Delete Diagram">
            <img id="exportImage" class="icon"
src="images/exportImage.png" title="Export Image">
            <img class="toolbarSeparator"
src="images/toolbarSeparator.png">
            <img id="addCM" class="icon" src="images/addCM.png"
title="Add Connector Move">
            <img id="remCM" class="icon" src="images/remCM.png"
title="Remove Connector Move">
            <img id="moveForward" class="icon"
src="images/moveForward.png" title="Bring to Front">
            <img id="moveBackward" class="icon"
src="images/moveBackward.png" title="Send to Back">
      </div>

      <div id="editor">

      <svg xmlns="http://www.w3.org/2000/svg">

            <g zPos="1"></g>
            <g zPos="2"></g>
            <g zPos="3"></g>
            <defs></defs>

      </svg>

      </div>

      <div id="palette">
            <div class="title">Palette</div>
            <div id="thumbnails"></div>
      </div>

      <div id="shapeData">
            <div class="title">Shape Data</div>
            <table>
            </table>
      </div>

      <div id="footer">University of Madeira</div>

</div>

<div id="newDiagramDialog" title="Create New Diagram">
      <p class="validateTips">Enhanced form fields are required.</p>

      <form>
      <fieldset>

      <label>Diagram Name</label>
      <input id="ndName" type="text"/>

      <label>Diagram Kind</label>
      <select id="ndType"></select><br />

      <label>Scope of Interest</label>
      <select id="ndSOI"></select> <button id="addSOI"
type="button">Add New</button>

      <label>Version</label>
      <select id="ndVersion"></select>

      </fieldset>
      </form>
</div>

<div id="newSOIDialog" title="Create New Scope of Interest">
      <p class="validateTips">Enhanced form fields are required.</p>

      <form>
      <fieldset>

      <label>Scope of Interest</label>
      <input id="nsName" type="text"/>

      <label>Is part of Scope of Interest</label>
      <select id="nsPartOf" class="optional"></select>

      </fieldset>
      </form>
</div>
```

```html
<div id="openDiagramDialog" title="Open Diagram">
      <p class="validateTips">Enhanced form fields are required.</p>

      <form>
      <fieldset>

      <label>Diagram Name</label>
      <select id="odName"></select>

      </fieldset>
      </form>
</div>

<div id="newShapeDialog" title="Create New Shape">
      <p class="validateTips">Enhanced form fields are required.</p>

      <form>
      <fieldset>

      <label>Represents Organization Artifact</label>
      <select id="nsOA"></select> <button id="addOA"
type="button">Add New</button>

      </fieldset>
      </form>
</div>

<div id="newOADialog" title="Create New Organization Artifact">
      <p class="validateTips">Enhanced form fields are required.</p>

      <form>
      <fieldset>

      <label>Specific ID</label>
      <input id="noaID" type="text" class="optional"/>

      <label>Name</label>
      <input id="noaName" type="text"/>

      </fieldset>
      </form>
</div>

<div id="delDiagramDialog" title="Delete Diagram">
      <p id="confirmMsg"></p>
</div>

</body>
</html>
```

A.2.2 JQuery code (editor.js)

```
$(document).ready(function(){

    var selected = 'null';
    var atdShapes = "[id^='sp_']";
    var connectors = "[id^='con_']";
    var shapeConPoint = "[id^='cp_']";
    var mousemove, curDiagram, curScope, curVersion, newCon,
pltElem, pltMouseUp = [];

    // handler for shapes
    $("svg").on("mousedown", atdShapes, main).on("mouseover",
atdShapes, moveCursor).on("mouseleave", atdShapes,
autoCursor);

    function main(e) {

        e.preventDefault();
        mousemove = false;
        var sel = e.target; if (typeof $(sel).attr("id") ==
'undefined') $(sel).attr("id", "undefined");
        x0 = e.pageX;
        y0 = e.pageY;

        var shapeTitle = $(this).attr("title");
        // list of connectors beggining on selected shape
        begConList = $("[id^='con_'][begshape=" + shapeTitle +
"]");
        // list of connectors ending on selected shape
        endConList = $("[id^='con_'][endshape=" + shapeTitle +
"]");
        // all connectors connected to selected shape
        allConList = $.merge($.merge([],begConList),
endConList);

        // connnection point clicked
        if ($(sel).is(shapeConPoint)) {

            var begPoint = getCP(sel);
            startCon(newCon, sel, begPoint);
        }

        else {

            // select shape
            if (selected != 'null') deselectAll();
            selected = $(this);
            showSelHandles(selected);
            updateShapeData(selected);

            // remove connector points from all shapes
            $(shapeConPoint).remove();
            $("#thumbnails img").css("background-color", "");

        }
        $(document).on("mousemove", function(e) {

            mousemove = true;
            document.body.style.cursor = "auto";

            dx = e.pageX - x0;
            dy = e.pageY - y0;

            // if current element is a selection handle,
resize shape
            if ($(sel).is("rect[id^='sh_']")) {
                resizeShape(selected, sel);
            }

            else if ($(sel).is(shapeConPoint)) {

                var point = getCM(newCon, 1);

                point[0] = point[0] + dx;
                point[1] = point[1] + dy;

                updateCP(newCon, point, "last");
            }

            // if current element is not a selection handle,
move shape
            else if (!$(sel).is("rect[id^='sh_']")) {

                translation =
$(selected).attr("transform").slice(10,-1).split(' ');
                xpos = parseFloat(translation[0]);
                ypos = parseFloat(translation[1]);

                var move =
$(selected).children().first().attr("move");
                if (move == "none") return;
                else if (move == "Horizontal") dy = 0;
                else if (move == "Vertical") dx = 0;

                $(selected).attr("transform", "translate(" +
(xpos + dx) + " " + (ypos + dy) + ")");

                updateLinkedCon(selected);
                document.body.style.cursor = "move";
            }

            x0 = e.pageX;
            y0 = e.pageY;

        }).one("mouseup", "svg", function(e) {
            $(document).off("mousemove");
            $("svg").on("mouseover", atdShapes,
moveCursor).on("mouseleave", atdShapes, autoCursor);

            var sel = e.target;
            // if attemptimg to create a connector
            if ($(sel).is("#conPoint")) {

                var point = new Array(0,0);
                updateCP(newCon, point, "last");

                var sel = document.elementFromPoint(e.pageX,
e.pageY);

                // if mouse up on shape connector point,
create connector
                if ($(sel).is(shapeConPoint)) {

                    var endPoint = getCP(sel);
                    updateCP(newCon, endPoint, "last");
                    var endshape =
$(sel).parent().attr("title");
                    var pointID = $(sel).attr("id");
                    $(newCon).attr("endshape", endshape);
                    $(newCon).attr("endPoint", pointID);
                    updateShapeData(newCon);
                    $(shapeConPoint).remove();
                    $("#thumbnails img").css("background-
color", "");

                    // create connector wiki pages
                    var OAKRK = newCon.OAKRK,
                        CK = $(pltElem).attr("title"),
                        conID = $(newCon).attr("id"),
                        conMoves =
$(newCon).children("polyline").attr("points"),
                        begShapeTitle =
$(newCon).attr("begShape"),
                        endShapeTitle =
$(newCon).attr("endShape"),
                        begShapePoint =
$(newCon).attr("begPoint"),
                        endShapePoint =
$(newCon).attr("endPoint"),
                        begSpContent = getPage(begShapeTitle),
                        endSpContent = getPage(endShapeTitle),
                        begSpPropList =
getPropList(begSpContent),
                        endSpPropList =
getPropList(endSpContent),
                        begOATitle =
begSpPropList["represents_organization_artifact"],
                        endOATitle =
endSpPropList["represents_organization_artifact"],
                        conTitle = genTitle("CONNECTOR");

                    var OAR = ask("[[Is a instance of
organizational artifact kind relation kind::"+OAKRK+"]] [[Is
reference 1 of organization artifact
relation::"+endOATitle+"]] [[Is reference 2 of organization
artifact relation::"+begOATitle+"]]");
                    // if oar page does not exist, create it
                    var oarTitle;
                    if (OAR.length == 0) {
                        oarTitle = genTitle("OAR");
                        editPropVal(oarTitle, "OAR", [
                            ["Is a instance of organizational
artifact kind relation kind", OAKRK],
                            ["Is reference 1 of organization
artifact relation", endOATitle],
                            ["Is reference 2 of organization
artifact relation", begOATitle],
                            ["Has scope of interest",
curScope]
                        ]);
                    }
                    else {
                        oarTitle = $(OAR).prop("tagName");
                    }

                    // create connector page
                    editPropVal(conTitle, "CONNECTOR", [
                        ["Represents organizational artifact
relation", oarTitle],
                        ["Is part of diagram", curDiagram],
                        ["Is instance of connector kind", CK],
                        ["Connector id", conID],
                        ["Connector moves", conMoves],
                        ["Connector beggining shape",
begShapeTitle],
                        ["Connector beggining shape point",
begShapePoint],
                        ["Connector ending shape",
endShapeTitle],
                        ["Connector ending shape point",
endShapePoint]
                    ]);

                    $(newCon).attr("title", conTitle);
                    selected = newCon;

                }

                // remove connector
                else $(newCon).remove();
            }

            else if (mousemove == true) {

                // update shape global coordinates
                var shapeTitle = $(selected).attr("title");
                var translation =
$(selected).attr("transform").slice(10,-1).split(' ');
                var x = translation[0];
                var y = translation[1];
                editPropVal(shapeTitle, "SHAPE", [["X-
position", x], ["Y-position", y]]);

                // update width and height
                var container =
$(selected).children().first();
                var ctnTitle = $(container).attr("title");

                if (container.is("rect")) {
                    var width = $(container).attr("width");
                    var height = $(container).attr("height");
                    editPropVal(ctnTitle, "Rect symbol
component", [["Width", width], ["Height", height]]);
                }
                else if (container.is("circle")) {
                    var r = $(container).attr("r");
                    editPropVal(ctnTitle, "Circle symbol
component", [["r", r]]);
                }

                // update all labels position
                var textElems = $(selected).children("text");
                $.each(textElems, function(index, txtElem) {

                    var textTitle = $(txtElem).attr("title");
                    var x = $(txtElem).attr("x");
                    var y = $(txtElem).attr("y");

                    editPropVal(textTitle, "Text symbol
component", [["x", x], ["y", y]]);
                });

                // update connector pages
                $.each(allConList, function(index, connector)
{
                    updateConPage(connector);
                });

            }

        });
    }

    // handler for connectors
    $("svg").on("mousedown", connectors,
conFn).on("mouseover", connectors,
moveCursor).on("mouseleave", connectors, autoCursor);

    function conFn(e) {

        e.preventDefault();
        var sel = e.target;
        var index = $(sel).index();
        x0 = e.pageX;
        y0 = e.pageY;
        // select connector
        if (selected != 'null') deselectAll();
        selected = $(this);
        showCCPs(selected);
        updateShapeData(selected);

        if ($(sel).is("#conPoint") && ($(sel).is(":first-
child") || $(sel).is(":last-child"))) showSCPs();
        else $(shapeConPoint).remove();

        $(document).on("mousemove", function(e) {

            dx = e.pageX - x0;
            dy = e.pageY - y0;

            // check if current element is a connector point
            if ($(sel).is("#conPoint")) {

                var point = getCM(selected, index);

                point[0] = point[0] + dx;
                point[1] = point[1] + dy;

                updateCP(selected, point, index);
            }

            x0 = e.pageX;
            y0 = e.pageY;

        }).one("mouseup", "svg", function(e) {
            $(document).off("mousemove");
            $("svg").on("mouseover", connectors,
moveCursor).on("mouseleave", connectors, autoCursor);

            var sel = e.target;
            var index = $(sel).index();

            // if attempting to edit first or last connector
move (connector point)
            if ($(sel).is("#conPoint") && ($(sel).is(":first-
child") || $(sel).is(":last-child"))) {

                var point = new Array(0,0);
                updateCP(selected, point, index);

                var sel = document.elementFromPoint(e.pageX,
e.pageY);

                // if mouse up on shape connector point, move
connector
                if ($(sel).is(shapeConPoint)) {

                    var selPoint = getCP(sel);
                    updateCP(selected, selPoint, index);
                    var selShape =
$(sel).parent().attr("title");
                    var pointID = $(sel).attr("id");

                    if (index == 0) {
                        $(selected).attr("begshape",
selShape);
                        $(selected).attr("begPoint", pointID);
                    }
                    else {
                        $(selected).attr("endshape",
selShape);
                        $(selected).attr("endPoint", pointID);
                    }

                    updateShapeData(selected);

                }

                // else move connector back to its last point
                else {

                    if (index == 0) {
                        var shape =
$(selected).attr("begshape");
                        var point =
$(selected).attr("begPoint");
                    }
                    else {
                        var shape =
$(selected).attr("endshape");
                        var point =
$(selected).attr("endpoint");
                    }
                    showSCPs();
                    point = $("g[title='" + shape +
"']").find("#"+point);
                    var selPoint = getCP(point);
                    updateCP(selected, selPoint, index);
                }

                $(shapeConPoint).remove();
            }

            // else if attempting to edit an inner connector
move (excludes first and last)
            else if ($(sel).is("#conPoint") &&
($(sel).not(":first-child") || $(sel).not(":last-child"))) {
                alignCP(selected, index);
            }

            updateConPage(selected);

        });
    }
    // create new diagram
    $("#newDiagram").click(function() {
        $("#newDiagramDialog").dialog("open");
    });

    $("#newDiagramDialog").dialog({
        autoOpen: false,
        height: 400,
        width: 450,
        modal: true,
        open: function(event, ui) {

            // load diagram type field values
            var elem = $('<option/>');
            $("#ndType").append(elem);
            var diagramTypes =
ask("[[Category:DIAGRAM_KIND]]");
            $.each(diagramTypes, function(index, value) {
                elem = $('<option/>', {'value':value.tagName,
'text':value.tagName});
                $("#ndType").append(elem);
            });

            // load scope of interest field values
            loadSOI("#ndSOI");

            // upon diagram type field value change
            $("#ndType").change(function() {
                // load version field values
                $("#ndVersion").empty();
                if ($("#ndType").val() != "") {
                    var versions =
ask("[[Category:SHAPE_KIND]] [[Is allowed in diagram kind::" +
$("#ndType").val() + "]]|?Version");
                    var ver, seen = [];
                    $('<option/>').appendTo("#ndVersion");
                    $.each(versions, function(index, value) {
                        ver = $(value).text();
                        if (!seen[ver]) {
                            elem = $('<option/>',
{'value':ver, 'text':ver});
                            $("#ndVersion").append(elem);
                            seen[ver] = true;
                        }
                    });
                }

            });

        },
        buttons: {
            "Create Diagram": function() {

                if (valFields(this)) {

                    resetEditor();
```

```
                        // create new diagram
                        dgName = $("#ndName").val();
                        dgName = capitalizeFirstLetter(dgName);
                        curDiagram = genTitle("DIAGRAM");
                        curScope = $("#ndSOI").val();
                        curVersion = $("#ndVersion").val();
                        $("#mainTitle").text(dgName + " - 
Universal Diagram Editor");

                        // diagram properties and values
                        var content = "\n{{DIAGRAM";
                        content += "\n|Diagram=" + dgName;
                        content += "\n|Is instance of diagram 
kind=" + $("#ndType").val();
                        content += "\n|Is a perspective of scope 
of interest=" + curScope;
                        content += "\n}}";

                        // diagram shapes query
                        content += "Shapes of Diagram:";
                        content += "\n{{#ask: [[Category:SHAPE]] 
[[Is part of diagram::"+curDiagram+"]]";
                        content += "\n| ?Represents organization 
artifact";
                        content += "\n| ?Is instance of shape 
kind";
                        content += "\n| ?X-position";
                        content += "\n| ?Y-position";
                        content += "\n| ?Z-position";
                        content += "\n| format=table";
                        content += "\n}}";

                        // diagram connectors query
                        content += "Connectors of Diagram:";
                        content += "\n{{#ask: 
[[Category:CONNECTOR]] [[Is part of 
diagram::"+curDiagram+"]]";
                        content += "\n| ?Represents organizational 
artifact relation";
                        content += "\n| ?Is instance of connector 
kind";
                        content += "\n| ?Connector moves";
                        content += "\n| ?Connector beggining 
shape";
                        content += "\n| ?Connector beggining shape 
point";
                        content += "\n| ?Connector ending shape";
                        content += "\n| ?Connector ending shape 
point";
                        content += "\n| format=table";
                        content += "\n}}";

                        // create diagram page
                        editPage(curDiagram, content);
```

```
                    // load pallete
                    loadPalette($("#ndType").val(),
$("#ndVersion").val());

                    $(this).dialog("close");
                }
            },
            Cancel: function() { $(this).dialog("close"); }
        },
        close: function() { resetForm(this); }
    });

    // open diagram
    $("#openDiagram").click(function() {
        $("#openDiagramDialog").dialog("open");
    });

    $("#openDiagramDialog").dialog({
        autoOpen: false,
        height: 400,
        width: 450,
        modal: true,
        open: function(event, ui) {

            // get diagrams list
            var results = ask("[[Category:DIAGRAM]]");
            $('<option/>').appendTo("#odName");
            $.each(results, function(index, value) {
                var pgTitle = $(value).prop("tagName");
                var pgContent = getPage(pgTitle);
                var propList = getPropList(pgContent);
                var name = propList["diagram"];
                var elem = $('<option/>', {'value':pgTitle,
'text':name});
                $("#odName").append(elem);
            });

        },
        buttons: {
            "Open Diagram": function() {

                if (valFields(this)) {

                    var dgName = $("#odName
option:selected").text();
                    curDiagram = $("#odName").val();

                    var dgContent = getPage(curDiagram);
                    var dgPropList = getPropList(dgContent);

                    curVersion = "3.5";
                    curScope =
dgPropList["is_a_perspective_of_scope_of_interest"];

                    resetEditor();
                    $("#mainTitle").text(dgName + " -
Universal Diagram Editor");

                    // load palette
                    var diagramType =
dgPropList["is_instance_of_diagram_kind"];
                    loadPalette(diagramType, "3.5");

                    // load shapes
                    var shapes = ask("[[Category:SHAPE]] [[Is
part of diagram::" + curDiagram + "]]");
                    $.each(shapes, function(index, pgTitle) {

                        var spTitle =
$(pgTitle).prop("tagName");
                        var spContent = getPage(spTitle);
                        var spPropList =
getPropList(spContent);

                        // load SVG shape main element
                        var mainElem = makeSVG('g', {id:
spPropList["id"], transform: "translate(" + spPropList["x-
position"] + " " + spPropList["y-position"] + ")",
title:spTitle});

                        var components =
ask("[[Category:Symbol Component]] [[Is part of::" + spTitle +
"]]");
                        $.each(components, function(index,
component) {

                            // load SVG shape components
                            var cpnTitle =
$(component).attr("fulltext");
                            var tag = cpnTitle.split(' ');
                            tag = tag[tag.length-2];
                            var pgContent = getPage(cpnTitle);
                            var cpnPropList =
getPropList(pgContent);
                            var cpnElem = makeSVG(tag,
cpnPropList);
                            $(cpnElem).attr("title",
cpnTitle);
                            $(cpnElem).appendTo(mainElem);

                        });

                        var oa =
spPropList["represents_organization_artifact"];
                        var oaContent = getPage(oa);
                        var oaPropList =
getPropList(oaContent);
                        var id = oaPropList["specific_id"];
                        var name = oaPropList["name"];

                        // update id and name
```

```
$(mainElem).children("text[text_id='ID']").text(id);

$(mainElem).children("text[text_id='Name']").text(name);

$(mainElem).appendTo("g[zPos='"+spPropList["z-
position"]+"']");

                    });

                    // load connectors
                    var connectors =
ask("[[Category:CONNECTOR]] [[Is part of diagram::" +
curDiagram + "]]");
                    $.each(connectors, function(index,
connector) {

                        var conTitle =
$(connector).prop("tagName");
                        var conContent = getPage(conTitle);
                        var conPropList =
getPropList(conContent);

                        var ckTitle =
conPropList["is_instance_of_connector_kind"];
                        var ckContent = getPage(ckTitle);
                        var ckPropList =
getPropList(ckContent);

                        // get connector SVG code
                        $.ajax({ url: "../mediawiki/images/" +
ckTitle + ".svg"
                        }).done(function(data) {

                            var connector = makeSVG('g', {
                                id:
conPropList["connector_id"],
                                begshape:
conPropList["connector_beggining_shape"],
                                begpoint:
conPropList["connector_beggining_shape_point"],
                                endshape:
conPropList["connector_ending_shape"],
                                endpoint:
conPropList["connector_ending_shape_point"],
                                title: conTitle
                            });

                            var line =
$(data).find("polyline").appendTo(connector);

                            $(line).attr("points",
conPropList["connector_moves"]);
                            $(line).attr("fill", "none");
                            $(line).attr("stroke-width",
ckPropList["connection_line_width"]);
```

```
                            $(line).attr("stroke",
ckPropList["connection_line_color"]);

                            $(connector).appendTo("g[zPos='" +
ckPropList["connector_z-position"] + "']");

                            // load markers
                            var mStartID =
$(line).attr("marker-start");
                            mStartID =
mStartID.substring(mStartID.indexOf('#')+1, mStartID.length-
1);
                            if
($("marker[id='"+mStartID+"']").length == 0)
$(data).find("marker[id='"+mStartID+"']").appendTo("defs");

                            var mEndID = $(line).attr("marker-
end");
                            mEndID =
mEndID.substring(mEndID.indexOf('#')+1, mEndID.length-1);
                            if
($("marker[id='"+mEndID+"']").length == 0)
$(data).find("marker[id='"+mEndID+"']").appendTo("defs");

                        });

                    });

                    $(this).dialog("close");

                }
            },
            Cancel: function() { $(this).dialog("close"); }
        },
        close: function() { resetForm(this); }
    });

    $("#newShapeDialog").dialog({
        autoOpen: false,
        height: 350,
        width: 400,
        modal: true,
        open: function(event, ui) {

            // load all instances of the OAK that corresponds
to the chosen SHAPE KIND
            var OAK = $(pltElem).attr("title").slice(0, -11);
            var OAs = ask("[[Category:OA]] [[Is a instance of
organizational artifact kind::" + OAK + "]] [[Has scope of
interest::" + curScope + "]]");

            $('<option/>').appendTo("#nsOA");
            $.each(OAs, function(index, value) {

                var pgTitle = $(value).prop("tagName");
                var pgContent = getPage(pgTitle);
```

```
                    var propList = getPropList(pgContent);
                    var OA;
                    if (typeof propList["specific_id"] ==
'undefined') OA = propList["name"];
                    else OA = propList["specific_id"] + '-' +
propList["name"];
                    var elem = $('<option/>', {'value':pgTitle,
'text':OA});
                    $("#nsOA").append(elem);

               });

          },
          buttons: {
               "Create Shape": function() {

               if (valFields(this)) {

                    var SK =
$(pltElem).attr("title").replace(/_/g,' '),
                         OA = $("#nsOA").val(),
                         oaText = $("#nsOA
option:selected").text();
                         S = genTitle("SHAPE");

                    var textName, textID;
                    if (oaText.indexOf('-') != -1) {
                         text = oaText.split('-');
                         textID = text[0];
                         textName = text[1];
                    }
                    else {
                         textName = oaText;
                    }

                    var gType = $(pltElem).attr("type"),
                         gSubType = $(pltElem).attr("subType"),
                         id = gType + '_' + gSubType;

                    var shapePos = [pltMouseUp[0]-260,
pltMouseUp[1]-125];
                    var skContent = getPage(SK);
                    var skPropList = getPropList(skContent);
                    var zPos = skPropList["z-position"];

                    // create main svg element
                    var mainElem = makeSVG('g', {id: id,
transform: "translate(" + shapePos[0] + " " + shapePos[1] +
")", title:S});

                    // create main wiki page
                    editPropVal(S, "SHAPE", [
                         ["Is instance of shape kind", SK],
                         ["Is part of diagram", curDiagram],
                         ["ID", id],
                         ["Represents organization artifact", OA],
                    ["X-position", shapePos[0]],
                    ["Y-position", shapePos[1]],
                    ["Z-position", zPos],
                ]);

                var components = ask("[[Category:Symbol
Component]] [[Is part of::" + SK + "]]");

                $.each(components, function(index, component)
{

                    var cpnTitle =
$(component).prop("tagName");
                    var cpnIndex =
cpnTitle.indexOf("component_");
                    var cpnTagNr =
cpnTitle.substring(cpnIndex);
                    var cpnTagNrArr = cpnTagNr.split("_");
                    var tag = cpnTagNrArr[1];
                    var cpnInst = S + "_" + cpnTagNr;

                    // create component wiki pages
                    var pgContent = getPage(cpnTitle);
                    pgContent = pgContent.replace(SK, S);
                    editPage(cpnInst, pgContent);

                    // create component svg elements
                    var propList = getPropList(pgContent);
                    var cpnElem = makeSVG(tag, propList);
                    $(cpnElem).attr("title", cpnInst);
                    $(cpnElem).appendTo(mainElem);

                });


$(mainElem).children("text[text_id='ID']").text(textID);

$(mainElem).children("text[text_id='Name']").text(textName);
                $(mainElem).appendTo("g[zPos='"+zPos+"']");

                // select shape
                if (selected != 'null') deselectAll();
                selected =
$("g[zPos='"+zPos+"']").children("g:last");
                showSelHandles(selected);
                updateShapeData(selected);

                $(this).dialog("close");
            }

            },
            Cancel: function() { $(this).dialog("close"); }
        },
        close: function() { resetForm(this); }
    });
    // add new organization artifact
    $('#addOA').click(function() {
        $("#newOADialog").dialog("open");
    });

    $("#newOADialog").dialog({
        autoOpen: false,
        height: 350,
        width: 400,
        modal: true,
        position: { my: "center+10 center+20", of:
"#newShapeDialog" },
        buttons: {
            "Create OA": function() {

                if (valFields(this)) {

                    var OAK =
$(pltElem).attr("title").slice(0, -11);
                    var id = $("#noaID").val();
                    var name = $("#noaName").val();
                    var pgTitle = genTitle("OA");

                    editPropVal(pgTitle, "OA", [
                        ["Is a instance of organizational
artifact kind", OAK],
                        ["Is in a organization artifact
state", "Operationalized"],
                        ["Specific id", id],
                        ["Name", name],
                        ["Has scope of interest", curScope]
                    ]);

                    var OA;
                    if (id == "") OA = name;
                    else OA = id+'-'+name;

                    var elem = $('<option/>',
{'value':pgTitle, 'text':OA});
                    $("#nsOA").append(elem);

                    $(this).dialog("close");

                }

            },
            Cancel: function() { $(this).dialog("close"); }
        },
        close: function() { resetForm(this); }
    });

    // add new scope of interest
    $('#addSOI').click(function() {
        $("#newSOIDialog").dialog("open");
    });
    $("#newSOIDialog").dialog({
        autoOpen: false,
        height: 350,
        width: 400,
        modal: true,
        position: { my: "center+10 center+20", of:
"#newDiagramDialog" },
        open: function(event, ui) {

            loadSOI("#nsPartOf");

        },
        buttons: {
            "Create Scope of Interest": function() {

                if (valFields(this)) {

                    var title = genTitle("SCOPE_OF_INTEREST");
                    var name = $("#nsName").val();
                    name = capitalizeFirstLetter(name);
                    var partOf = $("#nsPartOf").val();

                    if (partOf == '') partOf = "N/A";

                    var elem = $('<option/>', {'value':title,
'text':name});
                    $("#ndSOI").append(elem);

                    // scope of interest properties and values
                    var content = "\n{{SCOPE OF INTEREST";
                    content += "\n|Scope of interest=" + name;
                    content += "\n|Is part of scope of
interest=" + partOf;
                    content += "\n}}<br />";

                    // diagrams of scope of interest
                    content += "\nDiagrams of Scope of
Interest:";
                    content += "\n{{#ask: [[Is a perspective
of scope of interest::"+title+"]]";
                    content += "\n| ?Is instance of diagram
kind";
                    content += "\n| format=table";
                    content += "\n}}";
                    content += "\n:+ [[Form:DIAGRAM|Create new
Diagram]]<br /><br />"

                    // OAs of scope of interest
                    content += "\nOrganization Artifacts (OAs)
of Scope of Interest:";
                    content += "\n{{#ask: [[Category:OA]]
[[Has scope of interest::"+title+"]]";
                    content += "\n| ?Is a instance of
organizational artifact kind";
                    content += "\n| ?Specific id";
                    content += "\n| ?Name";
                    content += "\n| format=table";
                    content += "\n}}";
                    content += "\n:+ [[Form:OA|Create new
Organization Artifact]]<br /><br />"

                    // OARs of scope of interest
                    content += "\nOrganization Artifacts
(OARs) of Scope of Interest:";
                    content += "\n{{#ask: [[Category:OAR]]
[[Has scope of interest::"+title+"]]";
                    content += "\n| ?Is a instance of
organizational artifact kind relation kind";
                    content += "\n| ?Is reference 1 of
organization artifact relation";
                    content += "\n| ?Is reference 2 of
organization artifact relation";
                    content += "\n| format=table";
                    content += "\n}}";
                    content += "\n:+ [[Form:OAR|Create new
Organization Artifact Relation]]<br /><br />"

                    // create diagram page
                    editPage(title, content);

                    $(this).dialog("close");

                }

            },
            Cancel: function() { $(this).dialog("close"); }
        },
        close: function() { resetForm(this); }
    });

    // delete current diagram
    $('#deleteDiagram').click(function() {
        $("#delDiagramDialog").dialog("open");
    });

    $("#delDiagramDialog").dialog({
        autoOpen: false,
        height: 250,
        width: 350,
        modal: true,
        open: function(event, ui) {

            var dgContent = getPage(curDiagram);
            var dgPropList = getPropList(dgContent);
            var dgName = dgPropList["diagram"];
            $(this).find("#confirmMsg").text("You are about to
delete " + dgName + " diagram.");
        },
        buttons: {
            "Delete": function() {

                resetEditor();
                // delete diagram page
                delPage(curDiagram);

                // delete all pages part of the diagram page
and its components
                var results = ask("[[Is part of diagram::" +
curDiagram + "]]");
                var pgTitle;
                $.each(results, function(index, symbol) {
                    pgTitle = $(symbol).prop("tagName");
                    delSymbol(pgTitle);
                });

                $(this).dialog("close");

            },
            Cancel: function() { $(this).dialog("close"); }
        }
    });

    // export diagram as SVG
    $('#exportImage').click(function(e) {

        deselectAll();

        var s = new XMLSerializer();
        var o = $("svg")[0];
        var svgcode = s.serializeToString(o);

        $.generateFile({
            filename    : 'diagram.svg',
            content     : svgcode,
            script      : 'php/download.php'
        });
    });

    // add connector move to selected connector
    $("#addCM").on("click", function() {
        if (($(selected) != "null") &&
($(selected).attr("id").substring(0, 4) == "con_")) {

            var length =
$(selected).children("polyline").attr("points").split("
").length;

            var p1 = getCM(selected, length-2);
            var p2 = getCM(selected, length-1);

            var point = [];

            point[0] = (p1[0] + p2[0]) / 2;
            point[1] = (p1[1] + p2[1]) / 2

            point[0] = parseFloat(point[0]).toFixed(2);
            point[1] = parseFloat(point[1]).toFixed(2);
            addCM(selected, point, length-1);

            updateConPage(selected);
        }
    });

    // remove connector move from selected connector
    $("#remCM").on("click", function() {
        if ((selected != "null") &&
(selected.attr("id").substring(0, 4) == "con_")) {

            var length =
$(selected).children("polyline").attr("points").split("
").length;

            remCM(selected, length-2);

            updateConPage(selected);
        }
    });

    // move symbol forward
    $("#moveForward").on("click", function() {
        if (selected != "null") {
            selected.appendTo(selected.parent());
        }
    });

    // move symbol backward
    $("#moveBackward").on("click", function() {
        if (selected != "null") {
            selected.prependTo(selected.parent());
        }
    });

    // deselect shape
    $("svg").on("click", function(e) {
        if ((e.target == this) && (selected != 'null')) {
            deselectAll();
            selected = 'null';
            $("#shapeData tbody").remove();
            $("#shapeData .title").text("Shape Data");
        }
    });

    $("#palette").on("mousedown", "img", function(e) {

        e.preventDefault();
        $("#thumbnails img").css("background-color", "");
        $(this).css("background-color", "#FFA500");
        $("svg").off("mouseover mouseleave", atdShapes);

        pltElem = $(this);
        var gType = $(pltElem).attr("type");
        var gSubType = $(pltElem).attr("subType");
        if ((gType == "con")) {
            // deselect shape (if selected)
            if (selected != 'null') {
                deselectAll();
                selected = 'null';
                $("#shapeData tbody").remove();
                $("#shapeData .title").text("Shape Data");
            }
            document.body.style.cursor = "pointer";
            showSCPs();

            var ckTitle = $(pltElem).attr("title");
            var ckContent = getPage(ckTitle);
            var ckPropList = getPropList(ckContent);
            var zPos = ckPropList["connector_z-position"];
            var OAKRK = ckPropList["oakrk"];

            // get connector SVG code
            $.ajax({
                url: $(pltElem).attr("src")
            }).done(function(data) {
                newCon = makeSVG('g', {id: gType + '_' +
gSubType});
                $(data).find("polyline").appendTo(newCon);
                newCon.zPos = zPos;
                newCon.OAKRK = OAKRK;
                var line = $(newCon).children("polyline");
                $(line).attr("points", "0,0 0,0");
                $(line).attr("fill", "none");

                // load markers
                var mStartID = $(line).attr("marker-start");
                mStartID =
mStartID.substring(mStartID.indexOf('#')+1, mStartID.length-
1);
                if ($("marker[id='"+mStartID+"']").length ==
0) $(data).find("marker[id='"+mStartID+"']").appendTo("defs");

                var mEndID = $(line).attr("marker-end");
                mEndID =
mEndID.substring(mEndID.indexOf('#')+1, mEndID.length-1);
                if ($("marker[id='"+mEndID+"']").length == 0)
$(data).find("marker[id='"+mEndID+"']").appendTo("defs");

            });

        }

        // try to add new shape
        else {
            document.body.style.cursor = "move";
            $(document).one("mouseup", function(e) {

                var el = document.elementFromPoint(e.pageX,
e.pageY);
```

```
                // if mouse up on SVG element
                if ( $(el).is("svg") ||
$(el).parents("svg").length == 1 ) {
                    $("#newShapeDialog").dialog("open");
                    pltMouseUp = [e.pageX, e.pageY];
                }

                document.body.style.cursor = "auto";
                $("svg").on("mouseover", atdShapes,
moveCursor).on("mouseleave", atdShapes, autoCursor);
                $("#thumbnails img").css("background-color",
"");
            });
        }
    });

    $("#shapeData").on("click focusout keydown", "td:nth-
child(2)", function(e) {
        var keycode = (e.keyCode ? e.keyCode : e.which);
        var i = $(this).parent().index();
        var txtElem = $(selected).children("text:eq("+i+")");
        var txt = $(this).text();
        if (e.type == "click") {
            $(this).selectText();
        }
        else if (e.type == "focusout") {
            updateLabels(txtElem, txt);
        }
        // enter key pressed
        else if (keycode == '13') {
            e.preventDefault();
            move(i, "down");
            updateLabels(txtElem, txt);
        }
        // down arrow key pressed
        else if (keycode == '40'){
            e.preventDefault();
            move(i, "down");
        }
        // up arrow key pressed
        else if (keycode == '38'){
            e.preventDefault();
            move(i, "up");
        }
    });

    $(document).on("keyup", function(e) {

        if ((e.keyCode == 46) &&
!($(document.activeElement).is("tr > td:nth-child(2)"))) {

            var title = $(selected).attr("title");
            delSymbol(title);
            $(selected).remove();
            selected = "null";
            $("#shapeData tbody").remove();
            $("#shapeData .title").text("Shape Data");
        }
    });

});

function updateLabels(txtElem, txt) {

    $(txtElem).text(txt);

    var selected = $(txtElem).parent(),
        spTitle = $(selected).attr("title"),
        spContent = getPage(spTitle),
        spPropList = getPropList(spContent),
        oaTitle =
spPropList["represents_organization_artifact"],
        txtID = $(txtElem).attr("text_id"),
        txt = $(txtElem).text();

    if (txtID == "ID") txtID = "Specific id";

    editPropVal(oaTitle, "OA", [[txtID, txt]]);

}

function loadSOI(field) {

    var elem = $('<option/>');
    $(field).append(elem);
    var scopes = ask("[[Category:SCOPE_OF_INTEREST]]");
    $.each(scopes, function(index, value) {

        var pgTitle = $(value).prop("tagName");
        var pgContent = getPage(pgTitle);
        var propList = getPropList(pgContent);
        var name = propList["scope_of_interest"];
        elem = $('<option/>', {'value':pgTitle, 'text':name});
        $(field).append(elem);

    });
}

function valFields(dialog) {

    var isValid = true;
    var tips = $(dialog).children("p:first");
    var inputs =
$(dialog).find(":input:not(:button):not(.optional)");
    $(inputs).removeClass("ui-state-error");

    $.each(inputs, function(index, elem) {
        if ($(elem).val() == "" || $(elem).is('select:empty'))
{
            $(elem).addClass("ui-state-error");
            isValid = false;
        }
    });

    if (!isValid) {
        tips
            .text("Highlighted fields are empty.")
            .addClass( "ui-state-highlight" );
        setTimeout(function() {
            tips.removeClass( "ui-state-highlight", 1500);
        }, 500);
    }

    return isValid;
}

function resetForm(dialog) {

    $(dialog).children("form")[0].reset();
    $(dialog).find("select").empty();
    $(dialog).children("p:first").text("Enhanced form fields
are required.");
    $(dialog).find(":input").removeClass( "ui-state-error" );
}

function resetEditor() {

    $("#mainTitle").text("Universal Diagram Editor");
    $("#thumbnails").empty();
    $("svg").children("g").empty();
}

function moveCursor() {
    document.body.style.cursor = "move";
}

function autoCursor() {
    document.body.style.cursor = "auto";
}

function move(i, direction) {
    length = $("#shapeData tr").length;
    if (direction == "down") {
        i++;
        if (i >= length) i=0;
    }
    else if (direction == "up") {
        i--;
        if (i < 0) i=length-1;
    }
    var nextCell =
$("#shapeData").find("tr:eq("+i+")").children("td:nth-
child(2)");
    nextCell.selectText();
}

// creates a new SVG element with custom attributes
```

```
function makeSVG(tag, attrs) {
    var el =
document.createElementNS('http://www.w3.org/2000/svg', tag);
    for (var k in attrs)
        el.setAttribute(k, attrs[k]);
    return el;
}

// deselects a shape
function deselectAll() {
    $("[id^='sh_']").remove();
    $("#conPoints").remove();
}

function showSCPs() {

    $("[id^='cp_']").remove();
    var shapes = $("g[zPos='2']").children();
    var cpRad = 6;

    $.each(shapes, function(index, shape) {

        var elem = $(shape).children().first();

        if (elem.is("rect")) {

            var spWidth = parseFloat(elem.attr("width"));
            var spHeigth = parseFloat(elem.attr("height"));

            var x = elem.attr("x"); if (typeof x ==
'undefined') x = 0; x = parseFloat(x);
            var y = elem.attr("y"); if (typeof y ==
'undefined') y = 0; y = parseFloat(y);

            // top connector points
            var tl = makeSVG('circle', {id: 'cp_tl', cx:
x+spWidth*0.35, cy: y, r: cpRad});
            var tc = makeSVG('circle', {id: 'cp_tc', cx:
x+spWidth*0.50, cy: y, r: cpRad});
            var tr = makeSVG('circle', {id: 'cp_tr', cx:
x+spWidth*0.65, cy: y, r: cpRad});

            // right connector points
            var rt = makeSVG('circle', {id: 'cp_rt', cx:
x+spWidth, cy: y+spHeigth*0.35, r: cpRad});
            var rc = makeSVG('circle', {id: 'cp_rc', cx:
x+spWidth, cy: y+spHeigth*0.50, r: cpRad});
            var rb = makeSVG('circle', {id: 'cp_rb', cx:
x+spWidth, cy: y+spHeigth*0.65, r: cpRad});

            // bottom connector points
            var bl = makeSVG('circle', {id: 'cp_bl', cx:
x+spWidth*0.35, cy: y+spHeigth, r: cpRad});
            var bc = makeSVG('circle', {id: 'cp_bc', cx:
x+spWidth*0.50, cy: y+spHeigth, r: cpRad});
```

```
            var br = makeSVG('circle', {id: 'cp_br', cx:
x+spWidth*0.65, cy: y+spHeigth, r: cpRad});

            // left connector points
            var lt = makeSVG('circle', {id: 'cp_lt', cx: x,
cy: y+spHeigth*0.35, r: cpRad});
            var lc = makeSVG('circle', {id: 'cp_lc', cx: x,
cy: y+spHeigth*0.50, r: cpRad});
            var lb = makeSVG('circle', {id: 'cp_lb', cx: x,
cy: y+spHeigth*0.65, r: cpRad});

        }

        else if (elem.is("circle")) {

            var spRad = parseFloat(elem.attr("r"));

            var x = elem.attr("cx"); if (typeof x ==
'undefined') x = 0; x = parseFloat(x);
            var y = elem.attr("cy"); if (typeof y ==
'undefined') y = 0; y = parseFloat(y);

            var tc = makeSVG('circle', {id: 'cp_tc', cx: x,
cy: y-spRad, r: cpRad});
            var rc = makeSVG('circle', {id: 'cp_rc', cx:
x+spRad, cy: y, r: cpRad});
            var bc = makeSVG('circle', {id: 'cp_bc', cx: x,
cy: y+spRad, r: cpRad});
            var lc = makeSVG('circle', {id: 'cp_lc', cx: x-
spRad, cy: y, r: cpRad});

        }

        $(shape).append(tl, tc, tr, rt, rc, rb, bl, bc, br,
lt, lc, lb);
    });
}

function getPointValues(shape, curPointID) {

    var translation = $(shape).attr("transform").slice(10,-
1).split(' ');
    var absX = parseFloat(translation[0]);
    var absY = parseFloat(translation[1]);
    var cx, cy;

    var elem = $(shape).children().first();

    if (elem.is("rect")) {

        var spWidth = parseFloat(elem.attr("width"));
        var spHeigth = parseFloat(elem.attr("height"));

        var x = elem.attr("x"); if (typeof x == 'undefined') x
= 0; x = parseFloat(x);
```

```
            var y = elem.attr("y"); if (typeof y == 'undefined') y
= 0; y = parseFloat(y);

            switch(curPointID) {

                // top connector points
                case "cp_tl": cx = x+spWidth*0.35; cy = y; break;
                case "cp_tc": cx = x+spWidth*0.50; cy = y; break;
                case "cp_tr": cx = x+spWidth*0.65; cy = y; break;
                // right connector points
                case "cp_rt": cx = x+spWidth; cy =
y+spHeigth*0.35; break;
                case "cp_rc": cx = x+spWidth; cy =
y+spHeigth*0.50; break;
                case "cp_rb": cx = x+spWidth; cy =
y+spHeigth*0.65; break;
                // bottom connector points
                case "cp_bl": cx = x+spWidth*0.35; cy =
y+spHeigth; break;
                case "cp_bc": cx = x+spWidth*0.50; cy =
y+spHeigth; break;
                case "cp_br": cx = x+spWidth*0.65; cy =
y+spHeigth; break;
                // left connector points
                case "cp_lt": cx = x; cy = y+spHeigth*0.35; break;
                case "cp_lc": cx = x; cy = y+spHeigth*0.50; break;
                case "cp_lb": cx = x; cy = y+spHeigth*0.65; break;
            }
        }

        else if (elem.is("circle")) {

            var spRad = parseFloat(elem.attr("r"));

            var x = elem.attr("cx"); if (typeof x == 'undefined')
x = 0; x = parseFloat(x);
            var y = elem.attr("cy"); if (typeof y == 'undefined')
y = 0; y = parseFloat(y);

            switch(curPointID) {

                case "cp_tc": cx = x; cy = y-spRad; break;
                case "cp_rc": cx = x+spRad; cy = y; break;
                case "cp_bc": cx = x; cy = y+spRad; break;
                case "cp_lc": cx = x-spRad; cy = y; break;
            }
        }

        var x = absX + cx;
        var y = absY + cy;
        var point = [];
        point[0] = x;
        point[1] = y;

        return point;
}
```

```
function updateLinkedCon(selected) {

    for (i=0; i<begConList.length; i++) {

        var curPointID = $(begConList[i]).attr("begPoint");
        var bPoint = getPointValues(selected, curPointID);
        var pointLoc = curPointID.substring(0, 4);

        var length =
$(begConList[i]).children("polyline").attr("points").split("
").length;

        if (length > 2) {

            var p1 = getCM(begConList[i], 0);
            var p2 = getCM(begConList[i], 1);

            if (pointLoc == "cp_t" || pointLoc == "cp_b")
p2[0] = bPoint[0];
            else p2[1] = bPoint[1];

            updateCP(begConList[i], p2, 1);
        }
        updateCP(begConList[i], bPoint, 0);
    }

    for (i=0; i<endConList.length; i++) {

        var curPointID = $(endConList[i]).attr("endPoint");
        var ePoint = getPointValues(selected, curPointID);
        var pointLoc = curPointID.substring(0, 4);

        length =
$(endConList[i]).children("polyline").attr("points").split("
").length;

        if (length > 2) {

            var p1 = getCM(endConList[i], length-1);
            var p2 = getCM(endConList[i], length-2);

            if (pointLoc == "cp_t" || pointLoc == "cp_b")
p2[0] = ePoint[0];
            else p2[1] = ePoint[1];

            updateCP(endConList[i], p2, length-2);
        }
        updateCP(endConList[i], ePoint, "last");
    }
}

function updateCP(connector, point, index) {

    var points =
$(connector).children("polyline").attr("points");
```

```
    points = points.split(" ");
    length = points.length;
    if (index == "last") index = length-1;
    points[index] = point[0] + "," + point[1];

    var pointsStr = "";
    for (var i=0; i<length; i++) {
        pointsStr += points[i];
        if (i != length-1) pointsStr += " ";
    }

    $(connector).children("polyline").attr("points",
pointsStr);

$(connector).children("#conPoints").find("circle:eq("+index+")
").attr("cx", point[0]).attr("cy", point[1]);
}

function addCM(connector, point, index) {

    var points =
$(connector).children("polyline").attr("points");
    points = points.split(" ");
    var length = points.length;
    point = point[0] + "," + point[1];

    var pointsStr = "";
    for (var i=0; i<length; i++) {
        if (i == index) pointsStr += point + " ";
        pointsStr += points[i];
        if (i != length-1) pointsStr += " ";
    }

    $(connector).children("polyline").attr("points",
pointsStr);
    showCCPs(connector);
}

function remCM(connector, index) {

    var points =
$(connector).children("polyline").attr("points");
    points = points.split(" ");
    var length = points.length;

    if (length > 2) {

        var pointsStr = "";
        for (var i=0; i<length; i++) {
            if (i == index) { /* do not add the point */ }
            else {
                pointsStr += points[i];
                if (i != length-1) pointsStr += " ";
            }
        }
        $(connector).children("polyline").attr("points",
pointsStr);
        showCCPs(connector);
    }
}

function showCCPs(connector) {
    $(connector).children("#conPoints").remove();

    var points =
$(connector).children("polyline").attr("points");
    points = points.split(" ");
    var length = points.length;
    var cps = makeSVG('g', {id:'conPoints'});

    for (i=0; i<length; i++) {
        var point = points[i].split(',');
        var x = point[0];
        var y = point[1];
        var cp = makeSVG('circle', {id:'conPoint', cx:x, cy:y,
r:6});
        $(cp).appendTo(cps);
    }
    $(cps).appendTo(connector);
}

function showSelHandles(shape) {

    var elem = shape.children().first();
    var topLeft, topCenter, topRight, middleLeft, middleRight,
bottomLeft, bottomCenter, bottomRight;
    var side = 7;

    if (elem.is("rect")) {

        var spWidth = parseFloat(elem.attr("width"));
        var spHeight = parseFloat(elem.attr("height"));

        var x = elem.attr("x"); if (typeof x == 'undefined') x
= 0; x = parseFloat(x);
        var y = elem.attr("y"); if (typeof y == "undefined") y
= 0; y = parseFloat(y);

        var devX = x - (side/2);
        var devY = y - (side/2);

        topLeft = makeSVG('rect', {id: 'sh_topLeft', x:
(devX), y: (devY), width: side, height: side, cursor: 'nw-
resize'});
        topCenter = makeSVG('rect', {id: 'sh_topCenter', x:
(spWidth/2+devX), y: (devY), width: side, height: side,
cursor: 'n-resize'});
        topRight = makeSVG('rect', {id: 'sh_topRight', x:
(spWidth+devX), y: (devY), width: side, height: side, cursor:
'ne-resize'});
```

```
        middleLeft = makeSVG('rect', {id: 'sh_middleLeft', x:
(devX), y: (spHeight/2+devY), width: side, height: side,
cursor: 'w-resize'});
        middleRight = makeSVG('rect', {id: 'sh_middleRight',
x: (spWidth+devX), y: (spHeight/2+devY), width: side, height:
side, cursor: 'e-resize'});
        bottomLeft = makeSVG('rect', {id: 'sh_bottomLeft', x:
(devX), y: (spHeight+devY), width: side, height: side, cursor:
'sw-resize'});
        bottomCenter = makeSVG('rect', {id: 'sh_bottomCenter',
x: (spWidth/2+devX), y: (spHeight+devY), width: side, height:
side, cursor: 's-resize'});
        bottomRight = makeSVG('rect', {id: 'sh_bottomRight',
x: (spWidth+devX), y: (spHeight+devY), width: side, height:
side, cursor: 'se-resize'});
    }

    else if (elem.is("circle")) {

        var r = parseFloat(elem.attr("r"));

        var x = parseFloat(elem.attr("cx")); if (typeof x ==
'undefined') x = 0; x = parseFloat(x);
        var y = parseFloat(elem.attr("cy")); if (typeof y ==
"undefined") y = 0; y = parseFloat(y);

        var devX = x - (side/2);
        var devY = y - (side/2);

        topLeft = makeSVG('rect', {id: 'sh_topLeft', x: (devX-
r), y: (devY-r), width: side, height: side, cursor: 'nw-
resize'});
        topCenter = makeSVG('rect', {id: 'sh_topCenter', x:
(devX), y: (devY-r), width: side, height: side, cursor: 'n-
resize'});
        topRight = makeSVG('rect', {id: 'sh_topRight', x:
(devX+r), y: (devY-r), width: side, height: side, cursor: 'ne-
resize'});
        middleLeft = makeSVG('rect', {id: 'sh_middleLeft', x:
(devX-r), y: (devY), width: side, height: side, cursor: 'w-
resize'});
        middleRight = makeSVG('rect', {id: 'sh_middleRight',
x: (devX+r), y: (devY), width: side, height: side, cursor: 'e-
resize'});
        bottomLeft = makeSVG('rect', {id: 'sh_bottomLeft', x:
(devX-r), y: (devY+r), width: side, height: side, cursor: 'sw-
resize'});
        bottomCenter = makeSVG('rect', {id: 'sh_bottomCenter',
x: (devX), y: (devY+r), width: side, height: side, cursor: 's-
resize'});
        bottomRight = makeSVG('rect', {id: 'sh_bottomRight',
x: (devX+r), y: (devY+r), width: side, height: side, cursor:
'se-resize'});
    }
    shape.append(topLeft, topCenter, topRight, middleLeft,
middleRight, bottomLeft, bottomCenter, bottomRight);
}

function resizeShape(shape, selHandle) {

    // get absolute coordinates of shape
    var translation = $(shape).attr("transform").slice(10,-
1).split(' ');
    var xpos = parseFloat(translation[0]);
    var ypos = parseFloat(translation[1]);

    if ($(selHandle).is("#sh_topLeft"))
document.body.style.cursor = "nw-resize";
    else if ($(selHandle).is("#sh_topCenter"))
document.body.style.cursor = "n-resize";
    else if ($(selHandle).is("#sh_topRight"))
document.body.style.cursor = "ne-resize";
    else if ($(selHandle).is("#sh_middleLeft"))
document.body.style.cursor = "w-resize";
    else if ($(selHandle).is("#sh_middleRight"))
document.body.style.cursor = "e-resize";
    else if ($(selHandle).is("#sh_bottomLeft"))
document.body.style.cursor = "sw-resize";
    else if ($(selHandle).is("#sh_bottomCenter"))
document.body.style.cursor = "s-resize";
    else if ($(selHandle).is("#sh_bottomRight"))
document.body.style.cursor = "se-resize";

    var elem = shape.children().first();

    var rescale = elem.attr("rescale");
    if (rescale == "none") return;
    else if (rescale == "Horizontal") dy = 0;
    else if (rescale == "Vertical") dx = 0;

    if (elem.is("rect")) {

        var spWidth = +$(elem).attr("width");
        var spHeigth = +$(elem).attr("height");

        if ($(selHandle).is("#sh_topLeft")) {
            $(elem).attr("width", spWidth - dx);
            $(elem).attr("height", spHeigth - dy);
            $(shape).attr("transform", "translate(" + (xpos +
dx) + " " + (ypos + dy) + ")");
        }
        else if ($(selHandle).is("#sh_topCenter")) {
            $(elem).attr("height", spHeigth - dy);
            $(shape).attr("transform", "translate(" + xpos + "
" + (ypos + dy) + ")");
        }
        else if ($(selHandle).is("#sh_topRight")) {
            $(elem).attr("width", spWidth + dx);
            $(elem).attr("height", spHeigth - dy);
            $(shape).attr("transform", "translate(" + xpos + "
" + (ypos + dy) + ")");
        }
        else if ($(selHandle).is("#sh_middleLeft")) {
            $(elem).attr("width", spWidth - dx);
            $(shape).attr("transform", "translate(" + (xpos +
dx) + " " + ypos + ")");
        }
        else if ($(selHandle).is("#sh_middleRight")) {
            $(elem).attr("width", spWidth + dx);
        }
        else if ($(selHandle).is("#sh_bottomLeft")) {
            $(elem).attr("width", spWidth - dx);
            $(elem).attr("height", spHeigth + dy);
            $(shape).attr("transform", "translate(" + (xpos +
dx) + " " + ypos + ")");
        }
        else if ($(selHandle).is("#sh_bottomCenter")) {
            $(elem).attr("height", spHeigth + dy);
        }
        else if ($(selHandle).is("#sh_bottomRight")) {
            $(elem).attr("width", spWidth + dx);
            $(elem).attr("height", spHeigth + dy);
        }
    }

    if (elem.is("circle")) {

        var r = +$(elem).attr("r");

        if ($(selHandle).is("#sh_topLeft")) $(elem).attr("r",
r - (dx + dy)/2);
        else if ($(selHandle).is("#sh_topCenter"))
$(elem).attr("r", r - dy);
        else if ($(selHandle).is("#sh_topRight"))
$(elem).attr("r", r + (dx - dy)/2);
        else if ($(selHandle).is("#sh_middleLeft"))
$(elem).attr("r", r - dx);
        else if ($(selHandle).is("#sh_middleRight"))
$(elem).attr("r", r + dx);
        else if ($(selHandle).is("#sh_bottomLeft"))
$(elem).attr("r", r + (-dx + dy)/2);
        else if ($(selHandle).is("#sh_bottomCenter"))
$(elem).attr("r", r + dy);
        else if ($(selHandle).is("#sh_bottomRight"))
$(elem).attr("r", r + (dx + dy)/2);
    }

    updateLblPos(shape, elem);
    updateLinkedCon(shape);

    deselectAll();
    showSelHandles(shape);
}

function updateLblPos(shape, elem) {
    if ($(elem).is("rect")) {

        var spWidth = $(elem).attr("width");
        var spHeigth = $(elem).attr("height");

        shape.children("text:eq(0)").attr("x", spWidth*0.5);

        shape.children("text:eq(1)").attr("x", spWidth*0.5);
        shape.children("text:eq(1)").attr("y", spHeigth*0.65);

    }
}

function updateShapeData(selected) {

    $("#shapeData tbody").remove();
    var shapeTitle = $(selected).attr("title");
    $("#shapeData .title").text(shapeTitle);

    // if shape selected
    if ($(selected).attr("id").substring(0, 3) == "sp_") {

        var textElems = $(selected).children("text");
        $.each(textElems, function(index, txtElem) {
            var propName = $(txtElem).attr("text_id");
            var propValue = $(txtElem).text();
            var row = $("<tr><td>"+propName+"</td><td
contenteditable=true>"+propValue+"</td></tr>");
            $("#shapeData table").append(row);
        });
    }

    // if connector selected
    else if ($(selected).attr("id").substring(0, 4) == "con_")
{

        var begshape = $(selected).attr("begshape");
        var endshape = $(selected).attr("endshape");
        var row1 = $("<tr><td>Beg.
Shape</td><td>"+begshape+"</td></tr>");
        var row2 = $("<tr><td>End.
Shape</td><td>"+endshape+"</td></tr>");
        $("#shapeData table").append(row1, row2);
    }
}

// returns an array with the coordinates of the shape
connector point that was clicked
function getCP(sel) {

    var translation =
$(sel).parent().attr("transform").slice(10,-1).split(' ');
    var absX = parseFloat(translation[0]);
    var absY = parseFloat(translation[1]);
    var relX = +$(sel).attr("cx");
    var relY = +$(sel).attr("cy");
    var x = absX + relX;
    var y = absY + relY;
    var point = [];
    point[0] = x;
    point[1] = y;

    return point;

}

function startCon(newCon, sel, begPoint) {

    var begshape = $(sel).parent().attr("title");
    var pointID = $(sel).attr("id");

    $(newCon).attr("begshape", begshape);
    $(newCon).attr("begpoint", pointID);

    showCCPs(newCon);

    updateCP(newCon, [begPoint[0], begPoint[1]], 0)
    updateCP(newCon, [begPoint[0], begPoint[1]], 1)
    $(newCon).appendTo("g[zPos='"+newCon.zPos+"']");
}

function alignCP(selected, index) {

    var prevPoint = getCM(selected, index-1);
    var curPoint = getCM(selected, index);
    var nextPoint = getCM(selected, index+1);

    var dev = 30;

    // check if current CM is close to previous CM
    if ( (prevPoint[0]-dev) < curPoint[0] &&
(prevPoint[0]+dev) > curPoint[0] ) {
        curPoint[0] = prevPoint[0];
    }
    if ( (prevPoint[1]-dev) < curPoint[1] &&
(prevPoint[1]+dev) > curPoint[1] ) {
        curPoint[1] = prevPoint[1];
    }
    // check if current CM is close to next CM
    if ( (nextPoint[0]-dev) < curPoint[0] &&
(nextPoint[0]+dev) > curPoint[0] ) {
        curPoint[0] = nextPoint[0];
    }
    if ( (nextPoint[1]-dev) < curPoint[1] &&
(nextPoint[1]+dev) > curPoint[1] ) {
        curPoint[1] = nextPoint[1];
    }

    updateCP(selected, curPoint, index);
}
```

```
// get point coordinates of a connector at a specified index
function getCM(selected, index) {

    var points =
$(selected).children("polyline").attr("points");
    points = points.split(" ");

    var point = points[index].split(',');

    point[0] = parseFloat(point[0]);
    point[1] = parseFloat(point[1]);

    return point;
}

function updateConPage(connector) {

    var conTitle = $(connector).attr("title");
    var conMoves =
$(connector).children("polyline").attr("points");
    var begshape = $(connector).attr("begshape");
    var begpoint = $(connector).attr("begpoint");
    var endshape = $(connector).attr("endshape");
    var endpoint = $(connector).attr("endpoint");

    var conContent = getPage(conTitle);
    var conPropList = getPropList(conContent);
    var oarTitle =
conPropList["represents_organizational_artifact_relation"];

    var begShapeContent = getPage(begshape);
    var begShapePropList = getPropList(begShapeContent);
    var ref2 =
begShapePropList["represents_organization_artifact"];

    var endShapeContent = getPage(endshape);
    var endShapePropList = getPropList(endShapeContent);
    var ref1 =
endShapePropList["represents_organization_artifact"];

    // edit organization artifact page
    editPropVal(oarTitle, "OAR", [
        ["Is reference 1 of organization artifact relation",
ref1],
        ["Is reference 2 of organization artifact relation",
ref2]
    ]);

    // edit connector page
    editPropVal(conTitle, "CONNECTOR", [
        ["Connector moves", conMoves],
        ["Connector beggining shape", begshape],
        ["Connector beggining shape point", begpoint],
        ["Connector ending shape", endshape],
        ["Connector ending shape point", endpoint]
    ]);
```

```
}

function capitalizeFirstLetter(string) {
    return string.charAt(0).toUpperCase() + string.slice(1);
}

jQuery.fn.selectText = function(){
    var element = this[0];
    if (document.body.createTextRange) {
        var range = document.body.createTextRange();
        range.moveToElementText(element);
        range.select();
    } else if (window.getSelection) {
        var selection = window.getSelection();
        var range = document.createRange();
        range.selectNodeContents(element);
        selection.removeAllRanges();
        selection.addRange(range);
    }
}

function getPropList(pgContent) {

    // remove prefix
    var remIndex = pgContent.indexOf('|');
    pgContent = pgContent.substring(remIndex+1);

    // remove postfix
    remIndex = pgContent.indexOf('}}');
    pgContent = pgContent.substring(0, remIndex);

    var props = pgContent.split('|');

    var list = {};
    $.each(props, function(index, value) {
        var el = value.replace(/(\r\n|\n|\r)/gm,
"").split('=');
        var key = el[0].toString().toLowerCase().split('
').join('_');
        var val = el[1];
        if (val == "undefined") val = 0;
        list[key] = val;
    });
    return list;
}

function loadPalette(diagramType, version) {

    var elem, subType;

    // load shapes
    var shapes = ask("[[Category:SHAPE_KIND]] [[Is allowed in
diagram kind::" + diagramType + "]] [[Version::" + version +
"]]");
    $.each(shapes, function(index, value) {
        subType = value.tagName.substring(0,
value.tagName.indexOf("_V" + version));
        elem = $('<img/>', {'class':'logo',
'title':value.tagName, 'type':'sp', 'subType':subType,
'src':'../mediawiki/images/' + value.tagName + '.svg'});
        $("#thumbnails").append(elem);
    });

    // load connectors
    var connectors = ask("[[Category:CONNECTOR_KIND]] [[Is
allowed in diagram kind::" + diagramType + "]] [[Version::" +
version + "]]");
    $.each(connectors, function(index, value) {
        subType = value.tagName.substring(0,
value.tagName.indexOf("_V" + version));
        elem = $('<img/>', {'class':'logo',
'title':value.tagName, 'type':'con', 'subType':subType,
'src':'../mediawiki/images/' + value.tagName + '.svg'});
        $("#thumbnails").append(elem);
    });

}

function genTitle(category) {

    var pages = ask("[[Category:" + category + "]]");
    var pagesArray = [];

    $.each(pages, function(index, value) {
        pagesArray[index] = $(value).prop("tagName");
    });
    var i = 1;
    var name = capitalizeFirstLetter(category.toLowerCase() +
'_');
    var title = name + i;

    while ($.inArray(title, pagesArray) != -1) {
        i++;
        title = name + i;
    }
    return title;
}

function delSymbol(title) {

    delPage(title);

    var components = ask("[[Is part of::" + title + "]]");

    $.each(components, function(index, component) {

        var cpnTitle = $(component).prop("tagName");
        delPage(cpnTitle);
    });
}
```

```
function getToken() {
    var token;
    $.ajax({
        url: 'http://localhost/mediawiki/api.php',
        async: false,
        data: {
            action: 'query',
            prop: 'info',
            intoken: 'edit',
            titles: 'default',
            format: 'xml'
        }
    }).done(function(data) {
        token = $(data).find("page").attr("edittoken");
    });
    return token;
}

function getPage(title) {
    var text;
    $.ajax({
        url: 'http://localhost/mediawiki/api.php',
        async: false,
        data: {
            action: 'query',
            prop: 'revisions',
            rvprop: 'content',
            titles: title,
            format: 'xml'
        }
    }).done(function(data) {
        text = $(data).find("rev").text();
    });
    return text;
}

function delPage(title) {
    $.ajax({
        url: 'http://localhost/mediawiki/api.php',
        type: 'POST',
        data: {
            action: 'delete',
            title: title,
            token: getToken()
        }
    });
}

function ask(myQuery) {
    var xmlPage;
    $.ajax({
        url: 'http://localhost/mediawiki/api.php',
        async: false,
        cache: false,
        data: {
            action: 'ask',
            query: myQuery,
            format: 'xml'
        }
    }).done(function(data) {
        xmlPage = $(data).find("results").children();
    });
    return xmlPage;
}

function editPage(title, text) {
    $.ajax({
        url: 'http://localhost/mediawiki/api.php',
        type: 'POST',
        data: {
            action: 'edit',
            title: title,
            text: text,
            token: getToken()
        }
    });
}

function editPropVal(title, form, array) {
    var query = new String();
    $.each(array, function(index, value) {
        query += form + '[' + value[0] + ']=' + value[1];
        if (index < array.length-1) query += '&';
    });
    $.ajax({
        url:
'http://localhost/mediawiki/api.php?action=sfautoedit&form=' +
form + '&target=' + title + '&' + query
    });
}

//console.log($('<div>').append($(element).clone()).html());
```