ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Binder** Jméno: **Tadeáš** Osobní číslo: **469856**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Pokročilý webový editor OntoUML ontologií

Název bakalářské práce anglicky:

Advanced web editor of OntoUML

Pokyny pro vypracování:

Cílem práce je vytvořit pokročilou webovou komponentu pro tvorbu a úpravy OntoUML modelů vhodnou pro ontologické konceptuální modelování.
1. Proveďte rešerši
a) existujících webových knihoven vhodných pro tvorbu UML modelů
b) existujících komponent a systémů pro tvorbu OntoUML modelů
2. Na základě bodu 1 formulujte funkční a nefunkční požadavky na webovou komponentu pro grafickou tvorbu OntoUML modelů
3. Navrhněte webovou komponentu. Součástí návrhu bude i jazyk pro reprezentaci OntoUML konstruktů a jejich vzájemných závislostí. Svůj návrh formulujte v jazyce UML.
4. Implementujte komponentu ve vhodném JavaScript frameworku. Součástí implementace bude i validátor vytvářeného modelu využívající jazyk navržený v bodě 4.
5. Kvalitu implementace ověřte vhodnými automatickými testy. Webovou komponentu porovnejte se systémy v bodě 2b, a to na min. dvou OntoUML modelech různé složitosti a velikosti. Proveďte uživatelský test a zhodnoťte další využitelnost komponenty.

Seznam doporučené literatury:

[1] OntoUML Specification 1.0 - https://ontology.com.br/ontouml/spec/
[2] G.Guizzardi: Ontological Foundations of Structural Conceptual Models. 2005. Ph.D. Thesis - https://ris.utwente.nl/ws/portalfiles/portal/6042428
[3] Menthor editor - http://www.menthor.net

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Petr Křemen, Ph.D., skupina znalostních softwarových systémů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **24.01.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

Ing. Petr Křemen, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.	
Datum převzetí zadání	Podpis studenta

Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Advanced web editor of OntoUML ontologies

Tadeáš Binder

Supervisor: Ing. Petr Křemen, Ph.D.
May 2019

Acknowledgements

I would like to thank my supervisor, Dr. Petr Křemen, for offering to work with me and for having patience with me. I am also grateful for the people that contributed to testing of my project, as well as my friends for supporting me during tough times.

Declaration

I declare that I have made the submitted work independently and that I have listed all the sources used in line with the Methodological Guideline on Compliance with Ethical Principles of preparation of academic final theses.

Prague, May 23, 2019

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 23. května 2019

.....
Tadeáš Binder

Abstract

The aim of this thesis is to document the design, implementation, and testing of a web component for creating and editing OntoUML models appropriate for ontological conceptual modeling. There are solutions currently available that can model ontologies, however, they are not in the form of a web component, and therefore cannot be integrated into web systems. After an analysis of the requirements and available technologies, the component was implemented as a React component by heavily modifying and expanding the Storm React Diagrams library. Automated testing, user testing and comparisons to existing solutions revealed that while the component is lacking in features compared to the competition, its main advantage as a web component in terms of versatility suggests that the component has a lot of potential. We would, therefore, recommend that future work should be mainly directed at expanding the component's feature set.

Keywords: ontologies, OWL, React, web editor, web component, XMI, OntoUML

Supervisor: Ing. Petr Křemen, Ph.D. Karlovo náměstí 13, Praha 2

Abstrakt

Cílem této práce je zdokumentovat návrh, implementaci a testování webové komponenty pro tvorbu a editaci OntoUML modelů vhodné pro ontologické konceptuální modelování. V současné době jsou k dispozici řešení, která ontologie modelovat umí. Tato řešení však nejsou ve formě webové komponenty, a proto nemohou být integrována do webových systémů. Po analýze požadavků a dostupných technologií byla komponenta implementována jako komponenta React úpravou a rozšířením knihovny Storm React Diagrams. Automatické testy, uživatelské testy a porovnání s existujícími řešeními ukázaly, že zatímco komponenta ve srovnání s konkurencí postrádá určité funkce, její hlavní výhodou je její povaha webové komponenty, což jí nadzvihuje hlavně z hlediska univerzálnosti. Proto doporučujeme, aby budoucí práce směřovala především na rozšiřování funkcionality této komponenty.

Klíčová slova: ontologie, OWL, React, webový editor, webová komponenta, XMI, OntoUML

Překlad názvu: Pokročilý webový editor OntoUML ontologií

Contents

Figures

Tables

Chapter 1

Introduction

The goal of this work is to design and implement an advanced web component for the creation and editing of OntoUML models that is suitable for ontological conceptual modeling. First, we look at the reasons why such a component is necessary in the first place and what technologies can (and do) make it possible to realize. Second, we propose the functions and user interface that the component ought to have, based on currently available solutions and our discretion. We also design the component's architecture and choose its underlying structure. Third, we implement the component according to the design parameters and describe its function and basic usage. Lastly, we evaluate our work by using various testing methods and comparing it to other solutions. This evaluation will base the decision on where to take the component (or its underlying functions) next.

1.1 Motivation

As time goes on, more and more aspects of human activity and society is related to computers, increasing the amount (and types) of data that needs to be created and managed. Organizations from both the public and the private sector, therefore, need to design systems that allow for easy exchange of machine-readable data between other members of an organization or other institutions. To do so, however, it is first necessary to standardize the forms of data, their structure and the relationships between them. From there, various computer solutions can not only receive but also understand the nature of transferred information contained within.

A suitable method of achieving such functionality is through the use of *ontologies* – in the field of information technology, it is a term that represents explicit formalization of the conceptualization of knowledge of a given domain. Such knowledge can then be categorized, examined, represented in various ways and, above all, integrated into other ontological descriptions of knowledge.[1]

An important use of ontologies lies in the precise definition of terms or expressions in different contexts – for example, an *author* in the context of a book review may mean the author of the book or the author of the review. Therefore, if we want software systems to understand the difference

of such meaning, we need to divide these concepts and explicitly describe them along with the relationships with other concepts and, in some cases, its representation in different languages.

The aim of this work, therefore, is to design and implement a piece of software that can work with such ontologies. More specifically, it should be a component capable of creating and manipulating the diagram representation (in other words, model) of these ontologies in the OntoUML language. Since there are different categories of concepts in different types of usage contexts in each organization, the component must allow easy expansion of the pool of categories of concepts it is able to represent.

Furthermore, due to the nature of the assignment, the software should be developed as a web component in order to integrate it into existing software systems. The very idea of exchanging machine-readable data between data in a standardized manner is heavily inspired and in part realized through the Internet, nevertheless already existing solutions can store and make ontologies accessible on the Web. Thus, our solution should ideally be in the form of a web component as well, so that it is capable of integration into existing infrastructure with as little difficulty as possible.

1.2 Technologies

Several supporting technologies and protocols are necessary to implement the project. Previously defined and currently accepted standards for describing, graphically representing, and serializing ontologies on the Web allow integration of the component into existing solutions.

1.2.1 React

React is a JavaScript library first created by Jordan Walke and currently maintained by the Facebook team.[2] It is a library for easy and fast creation of user interfaces for the Web. These interfaces consist of individual components, which are structured together. A component can have attributes, or *states*, which, among other variables, can be passed to other components via component properties, or *props*.

One of the main advantages of React is the virtual DOM, which is a concept that, in effect, greatly reduces hardware requirements for rendering elements.[3] Another useful feature is JSX (JavaScript XML), which provides a way to represent user interface elements via a syntax very similar to that of HTML, for example.[4] React is one of the most used JavaScript libraries of all time in part because of these features.[5]

1.2.2 OWL

OWL (Web Ontology Language) is a descriptor for a family of languages defined by the World Wide Web Consortium (W3C). Their purpose is to

represent ontologies, the groupings those ontologies belong to, and the relationships between those ontologies. This representation is explicit and exact to allow a meaningful exchange of ontologies between different systems and solutions. As such, it is important for our work, since it provides an unambiguous description of knowledge that any compatible piece of software can understand.[6]

1.2.3 OntoUML

OntoUML is a language extension first envisioned in Giancarlo Guizzardi's Ph.D. thesis "Ontological foundations for structural conceptual models". In it, he designed the *UFO* (Unified Foundational Ontology), that he used to create an extension of *UML* (Unified Modeling Language). This extension enables the use of UML standards to create conceptual models of ontologies and, most importantly, visualize them and the relationships between them.[7] OntoUML has since its inception been adopted by many public and private organizations all around the world.[8]

1.2.4 XMI

XMI (XML Metadata Interchange) is a standard devised by the Object Management Group that uses XML syntax to represent metadata of UML data models.[9] This metadata is useful for our project since it gives us the ability to transform the metamodel (in other words, the set of available model elements) into a single serialization that is then easily distributed between organizations or individuals. It also allows for validation – a model can be checked for discrepancies between it and a given metamodel.

Chapter 2

Design

In this chapter, we will discuss the requirements of our potential solution. In order to do so, we need to first study other pieces of software that aim to fulfill similar objectives as the goals we set out to achieve, so that we may recognize which of their features and design elements we ought to implement as well. Next, we look at existing JavaScript libraries that support modeling and determine which can be used as a base which we will extend to accommodate our needs. From that, we construct requirements, both functional and non-functional, as well as produce use cases and activities that stem from those requirements. Lastly, we design the visual appearance of the component.

2.1 Existing solutions

There are solutions currently available claiming to fulfill some of our initially defined criteria. Through analysis of those solutions, we can determine what parts of our goals are they able to achieve, what parts they can't achieve, whether they are able to achieve those parts well, and if (or how) they can be extended to achieve other goals. We picked examples of those pieces of software according to our (the author's and the supervisor's) previous experiences and recommendations.

2.1.1 Menthor Editor

Web address: `http://www.menthor.net/menthor-editor.html`

Author: Menthor

Analyzed version: 1.1.9

Menthor Editor is a Java-based application for creating ontologies with a large amount of functionality. It can model ontologies in the OntoUML language as well as export it to various serialization standards, validate ontologies, or generate documentation. In addition, it is available for download for free without restriction of use as an open-source application.

If a user wants to alter or expand the application's code, however, they will run into significant problems. For example, the definition of the metamodel

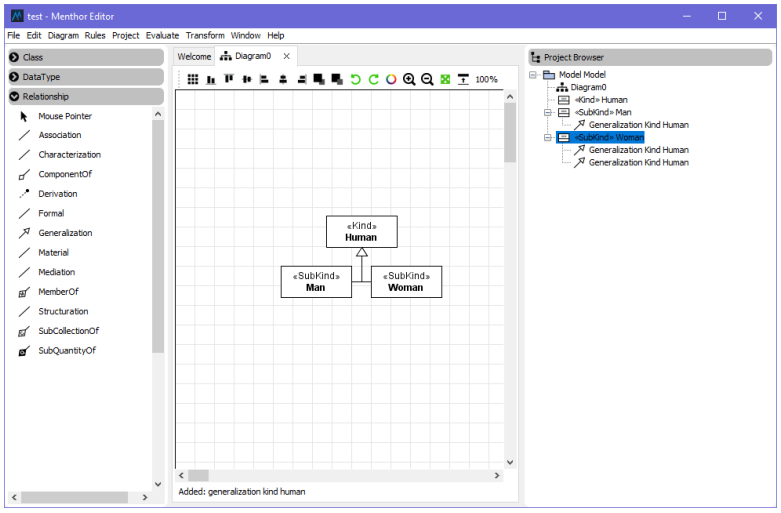

Figure 2.1: Menthor Editor

is located in a single file in the source code. Therefore, a person technically could change the pool of available diagram elements. The problem, though, is in how that definition is connected to the rest of the application. Since the other components heavily rely on the vanilla version of the metamodel, simply changing it would significantly break the application, so any seemingly simple change brings with it large consequences in terms of requiring a rewrite of major parts of the application. Not to mention, even if such a change were able to be easily done, *Menthor Editor* is currently unable to dynamically import other versions of the metamodel in a way that would facilitate simple integration into other systems, as our requirements explicitly require.

Another problem with *Menthor Editor* is its instability. Basic modeling is, from our experience, mostly uninterrupted by software issues, however, the reliability of more advanced functions, such as exporting and validation, is highly volatile. For example, simple changes of cardinalities of certain relationships can break simple exporting to OWL.

2.1.2 Enterprise Architect

Web address: `https://sparxsystems.com/products/ea/index.html`

Author: Sparx Systems

Analyzed version: 12

Enterprise Architect presents a way to create detailed diagrams of various types and specifications for purposes of modeling, analysis, testing and maintaining of systems, processes, and architectures. It supports many standards commonly used to assist software development (mainly UML) and for those use cases, it offers a very generous amount of functions. *Enterprise Architect* does not officially support OntoUML out of the box, nevertheless it

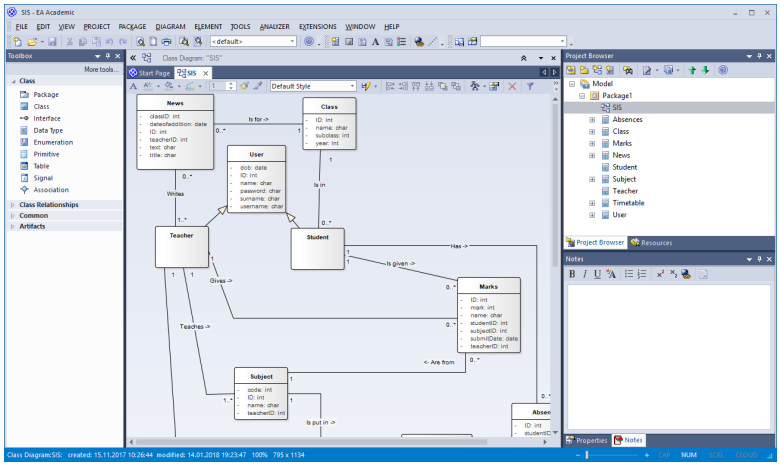

Figure 2.2: Enterprise Architect

supports extensions, and as such can be supplanted with OntoUML elements and standards.[1]

However, as the application's focus is mainly towards aiding software design and development, not ontology modeling, it is generally not suitable for use cases stemming from our defined goals. In addition, the license requirements pose a significant problem for the adoption of a potential extension that we would develop, as Sparx Systems ask 230 US dollars for an entry-level license.[10]

2.1.3 Summary

The solutions analyzed above do not present a valid solution or a base of a solution for our requirements, mainly because both do not fulfill the key requirement of being a web component that can be integrated into other web applications. Furthermore, although both applications can be improved with additional functionality, the way that the applications can be extended and the process of implementing those extensions leave much to be desired in both cases. However, there are design elements that we considered important enough to include them in our design, such as the ergonomics/user interface components of basic modeling processes.

2.2 Modeling libraries

Thanks to the architecture of the *React* library, we can aid the development of the solution with an adequate pick of an underlying library that is designed to allow graphical representation and modification of diagrams. By usefully evaluating and selecting such library, we can significantly ease development, since we will gain a base upon which we can build extensions that we won't

[1] For example, the extension developed by Menthor Editor authors, available at `https://github.com/MenthorTools/plugin-enterprise-architect`.

have to create from scratch. By choosing poorly, however, we invite the risk of using a base that cannot be improved with additional functionality, or can be, but with significant difficulties. Therefore, choosing the right criteria for determining which library is the best for our needs is crucial.

After analysis of our needs, the following criteria were chosen:

Extensibility. Whether it is possible to extend the library with additional functionality, and if so, how difficult is such an extension, and what parts of the library can be extended. This is important because of two reasons: firstly, because we need to include features that almost certainly will not be included in the base library in a form that is useful to us, such as exporting or validation, and secondly, it is necessary to develop the component in such a way that users down the line are able to extend the finished product themselves, since it should be able to potentially work with as many types of ontologies as possible.

License. Denotes the type of license agreement that is available to users of the library. An ideal pick would be one which has no restrictions to commercial and non-commercial use, both with standard use of the library and with modification of the library's source code.

Documentation. Here, we evaluate the completeness (what percentage of functionality does the documentation cover), quality (where it lies on a continuum between, for example, a mere mention of a function's existence and a detailed explanation including examples, etc.) and availability (whether it is available as a complete document, wiki, forum posts, or other forms).

Functionality. In other words, what percentage of the functionality we need is already implemented in the vanilla version of the library and what is the quality (in terms of reliability) of those functions.

Compatibility. Our component needs to be able to communicate with other components/software solutions. Therefore, this criterion describes the simplicity of transforming data of a given model, the component's settings, or the metamodel in such a way that it is possible to use that data in a meaningful exchange between other computer systems.

All of these criteria are subjective, and as such can only be evaluated ordinally; in other words, ranked from worst to best.

2.2.1 Detailed analysis

The five libraries subject to evaluation were found using the Google search engine based on "JavaScript", "UML", "diagram", and "React" keywords and chosen according to their popularity in the search results as well as their ability to fulfill our needs.

JointJS

Web address: `www.jointjs.com/opensource`

Author: client IO s.r.o.

In development: Yes

Licence: MPL

The modular library *JointJS* can model a large number of types of diagrams. The documentation is short but describes all categories of use of the library. Extension of the library is possible but very difficult. You can add other types of diagrams relatively easily, however, due to the nature of the internals of the library, any meaningful extension that would add significant functionality to the library would involve rewrites of major parts of the core source code.

Storm React Diagrams

Web address: `www.npmjs.com/package/storm-react-diagrams`

Author: Dylan Vorster

In development: No

Licence: MIT

Storm React Diagrams is the only library in this list that takes full advantage of the *React* library and it can be integrated with other *React* components, as well as exchange information with them. The documentation and the list of already implemented functions is small compared to other libraries on this list, but that is compensated with the fact that the library's customization and extension is simple and can be done at any level of the source code.

Rappid

Web address: `www.jointjs.com`

Author: client IO s.r.o.

In development: Yes

Licence: Commercial

Rappid is an extended, commercial version of *JointJS*. In addition to features contained within *JointJS*, this library is able to perform additional tasks, such as exporting and diagram canvas manipulation. Furthermore, the user can take advantage of a pool of already implemented tools and menu bars to use in their application. The authors also provide professional support and consultations, if needed.

GoJS

Web address: `www.gojs.net`

Author: Northwoods Software

In development: Yes

Licence: Commercial

The largest advantage of using *GoJS* is its versatility. Hundreds of examples of use cases and implementations of various types of diagrams, for example UML, spiderweb-like mind maps and graphs, are available at the library's web site. As a result, the documentation of this library is thorough and extensive.

jsUML 2

Web address: `www.uco.es/~in1rosaj/tool_jsUML2.html`

Author: José Raúl Romero, PhD. et al.

In development: No

Licence: GPL3

This library, designed by a professor from a Spanish university in Cordoba, has a large amount of functionality available out of the box, such as exporting, stereotype defining, or tabs (which allows editing multiple diagrams at the same time). On the other hand, extensibility of the library is remarkably difficult, in part due to bare documentation in Spanish and also due to source code that is not prepared for the addition of new functions.

Licences

GoJS and *Rappid* both use commercial licences. Therefore, they can't be used for the development of internal (for use within the organization that ordered it) nor external (for further distribution to other organizations) software solutions without a certain fee. In the case of *GoJS*, this fee is differentiated between external and internal licenses.[11] Commercial licenses in general are the ones that are favored the least in our solution, since high prices limit the ability to distribute the component between as many organizations as possible, the modification of source code is accessible for a higher price or prohibited altogether, and the use of the component/library may be subject to other laws. This relationship between the developer, distributor, and user is further complicated if any of the parties reside in different countries.

The *MPL* (Mozilla Public Licence) license employed by *JointJS* is much more flexible. In this case, any unchanged code of the library must be distributed under the *MPL* license, whereas the license of any modified code or code added on top is up to the developer's discretion.[12] *GPL3* (General

Public License) chosen by *jsUML2*'s developers differs from *MPL* in that the modified or additional code must also be licensed under *GPL3*.[13]

The *MIT* license is by far the most lax. If we were to use *Storm React diagrams* for any use, commercial or non-commercial, all we have to do is to include a copy of the MIT license into the source code.[14]

Functionality

All of the libraries mentioned above support important functions such as serialization and deserialization, definition of custom diagram elements (relationships and classes), interactive diagramming (in other words, the ability to edit diagrams in a visual interface, without having to define diagrams in code) and keyboard shortcuts. *GoJS* as a solution is specifically focused towards providing the user with the largest possible pool of available tools for creating diagrams in general – for example, transactions, connected buttons, diagram trees (and graphs overall), or model debugging. On the other hand, *Rappid*'s main objective is to offer a mostly complete (but still expandable) solution with already implemented panels, menu bars, drop-downs, help functions, diagram maps, etc.

Results

The chosen libraries were evaluated according to the criteria defined above on a range from 1-5 (1 being the most favorable, 5 being the least favorable). These evaluations were done at our discretion with regards to the solution's requirements and the detailed analysis of the chosen libraries.

	Extensibility	Licence	Documentation	Functionality	Compatibility	Mean
JointJS	3	2	2	3	2	2,4
Storm React Diagrams	2	1	3	3	1	2
Rappid	3	5	1	2	2	2,6
GoJS	3	5	2	2	3	3
jsUML 2	4	2	4	3	3	3,2

Table 2.1: Evaluation of modeling libraries

After careful consideration and analysis, we selected the *Storm React Diagrams* library for our component. Compared to the other freely available libraries, it if the most easily modifiable and extensible. It already takes advantage of the *React* library and does not have to be edited to fit into the existing solution. Its design makes it easy to implement the features necessary to meet our requirements.

2.3 Software design

In this section, we seek to define and describe the exact requirements of the project, its possible use cases and base architecture.

2.3.1 Functional requirements

The functional requirements defined for the projects are as follows:

Ontological modeling. The component is, at its core, an ontology modeling tool. As such, it is crucial that it is able to perform all the basic tasks that are required for that use, such as providing a canvas for manipulating diagram elements, having the ability to name elements and add attributes of various types to them in multiple languages, enabling the user to view models in read-only mode, naming models, and saving/loading them.

Customization. As mentioned previously, ontologies can take many forms. Because of that, necessary steps need to be made to ensure that a user can modify the available pool of elements and their features, such as graphical representation, names, and underlying information, as well as attribute types, cardinalities, and languages.

Conversion. One of the main purposes of ontologies is that they can be shared and parsed by multiple software solutions in a meaningful way. In order to facilitate that, our project has to provide a way to convert models created within the component into a serialization that is useful to other users. The standard we have chosen for that function is *OWL*. Next, the component must be able to interact with serialized definitions of stereotypes and include them into the available stereotypes the user can place in a model, as well as transfer those that the user created in the component to a form that can be shared with others.

Validation. To determine whether a model conforms to certain standards in terms of what diagram elements were used and in what way, the user has to have the option to validate that model against either the settings the component is operating under at that time or a settings file that was created from a certain configuration that the user created/received previously. To implement this, a definition of a language based on UML standards is required.

Constraints. During modeling, the user needs to be able to evaluate the model created using certain constraints that the user can define. They can then see what constraints were violated and where the violation occurred. As the model and its elements are represented via objects in *Storm React Diagrams*, the choice of language is relatively straightforward: *OCL* (Object Constraint Language).

2.3.2 Non-functional requirements

The non-functional requirements defined for the projects are as follows:

Reliability. The reliability, or vulnerability towards bugs and unintended behavior, will be measured in automated testing for the base algorithms and with user testing for the user experience.

Documentation. As the component is directed towards use by other users, general documentation for the use of the component ought to be provided.

Source code availability. The source code of the application must be available so that other users can expand the functionality of the component if they so desire.

Extensibility. The component must be designed and implemented in such a way as to open the avenues through which to expand the functionality as widely as possible.

2.3.3 Use cases

We can deduce the possible use cases and available activities of the component through our requirements. These activities and the actors that can perform those activities are shown in figure A.3. An activity diagram showing an example of possible activities is in figure A.1.

Actors

In a given use case, actors can have one of three possible roles:

- Viewer – a user who can only view an embedded, read-only version of the model, for example on a web page.
- Modeler – someone who uses the component for the purposes of model creation and manipulation but does not have access to the source code.
- Developer – a person who integrates the component into a web page/-software system and has access to the source code.

2.3.4 User interface

The design of the user interface was influenced by the analysis of other solutions (see section Existing solutions) and was created with consideration to *React* and *Storm React Diagrams*' limitations.

The list of stereotypes and the connection between them is present on the left panel; when you select a diagram element on the canvas, a panel appears on the right with a detailed description of the element and its attributes, complete with forms to change those attributes.

A menu bar on top shows the model's name and a series of drop-downs containing most of the component's tools and features. Selecting an option requiring further input will open a modal dialog. If the user decides to evaluate the diagram's constraints, a panel on the bottom will appear detailing the failed constraints.

A wireframe detailing the component's user interface is available in appendix A.2.

Chapter 3

Implementation

The component is based on the *React* library and uses *Storm React Diagrams* as the underlying base package that enables it to provide a diagram canvas on which models can be created and modified. Due to the fact that *Storm React Diagrams*' available documentation does not present a detailed explanation of how the library work at its core, the process of understanding the library's architecture and taking advantage of it was quite challenging at first. *Storm React Diagrams*' code hides within it several peculiarities that are not described thoroughly, if at all, in the documentation; however, after taking the time to master those oddities, the main challenges of implementing the project's requirements were mainly the question of designing algorithms that correctly fulfill the desired tasks or taking advantage of the right JavaScript packages that provide the required functionality, for example, user interface elements, serialization, or document fetching.

The implementation is available at the web address `https://gitlab.fel.cvut.cz/bindetad/oUML-diagram-app/tree/bachelorProject` and the latest version (as of May 24th, 2019) is included in the attachments.

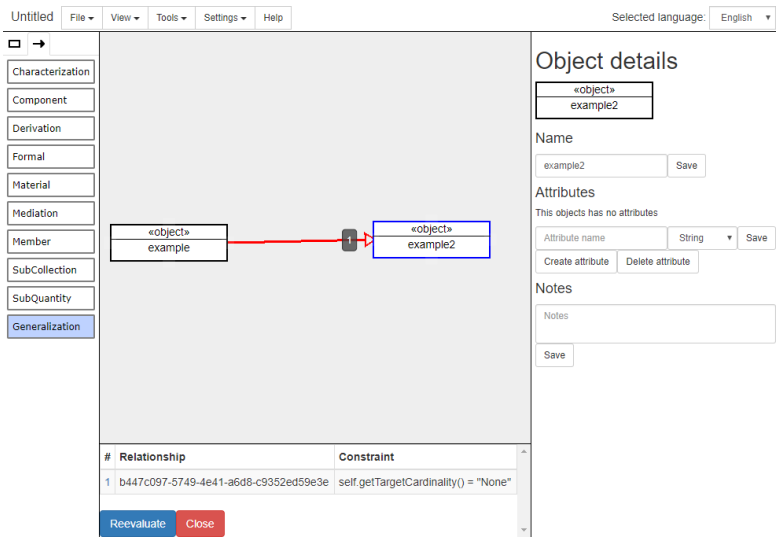

Figure 3.1: The component

3.1 Architecture

The *Storm React Diagrams* library is based on the Model-View-Controller architecture, where each diagram element (stereotype or relation) exists in a data form (or *model*) and in a visual form (or *view*) that is bound to that data form. All of these elements are categorized by the diagram controller that keeps a list of each diagram element in the form of an object. As a result, models can be easily expanded with new features/attributes, and theoretically allow serialization of the chart with a variety of standards, such as JSON or RDF/XML. It also simplifies communication between components wherever needed. Element visuals are based on SVG graphics.

In addition, the project uses additional support libraries, especially for constructing the user interface - react-bootstrap[1] for general appearance, such as tabs, buttons, menus and modal dialogues; rdf-ext[2] to load and parse RDF serializations for the purpose of loading new diagram elements; ECore.js[3] to load, parse and create serializations of component settings; OCL.js to build and evaluate contraints[4]; sax.js to construct XMI-compliant serializations[5].

The component offers an interface for various properties:

- loadSettings: accepts an XMI serialization to fetch settings.
- loadDiagram: loads a diagram from a serialization.
- readOnly: sets the component mode to read-only mode. Any action that would alter the diagram is prohibited. Accepts boolean values.
- loadOntology: loads stereotypes from a .ttl external source.

3.2 Components

The structure of the component is divided into 6 main React components unified under the main DiagramApp class - the menu bar, the left panel for selecting diagram elements, the right panel for detailed description and modification of diagram elements' properties, the bottom panel for evaluating constraints, the context menu, and the canvas itself. Panels and the context menu are ordinary React components that occasionally refer back to the diagram controller - but the canvas is a component created from a modified *Storm React Diagrams* core that is extended with the ability to allow communication between other components.

[1] Available at `https://react-bootstrap.github.io/`.
[2] Available at `https://github.com/rdf-ext/rdf-ext/`.
[3] Available at `http://emfjson.org/projects/ecorejs/latest/`.
[4] Available at `https://ocl.stekoe.de/`.
[5] Available at `https://github.com/isaacs/sax-js/`.

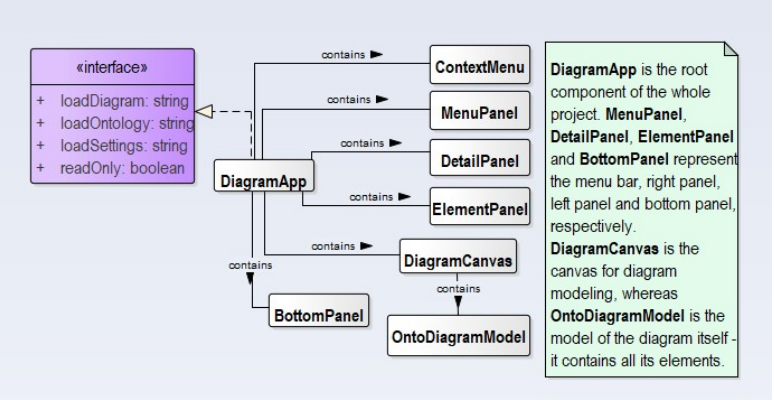

Figure 3.2: Conceptual diagram of project components

3.2.1 Model elements

Every stereotype is an instance of the same class. Information about its name and definition is taken from the left panel when the object is created. The same is true for relationships, but with one difference: since each relationship can have a different look (in terms of lines and ends – for example arrows or points), a specific view must be manually assigned to each defined session. Stereotypes can be loaded automatically from a predefined .ttl file when the component is started - the component takes the appropriate classes from the file, which it then offers for use in the left panel.

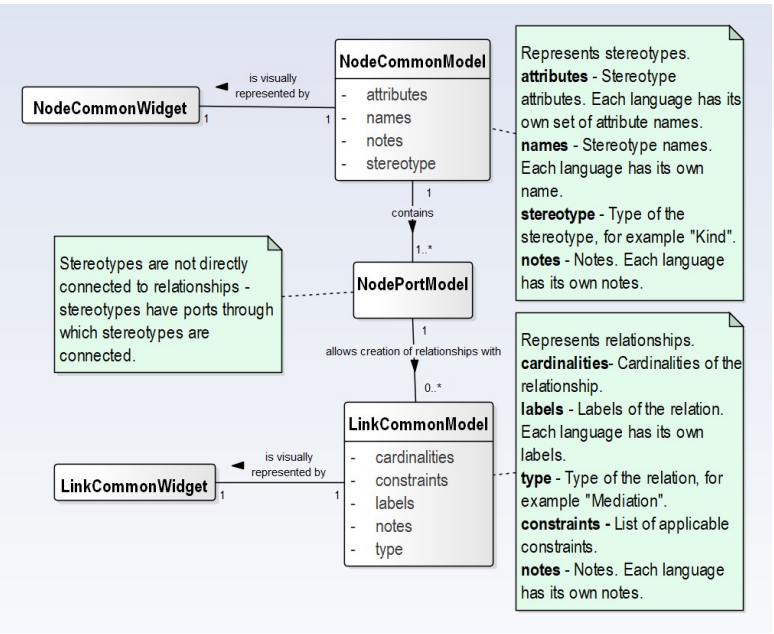

Figure 3.3: Conceptual diagram of diagram element classes and interface

3.2.2 Example

Consider an example ontology with a Person, who has an ID and a name whose role is a Patient that is characterized by one or more Symptoms. There

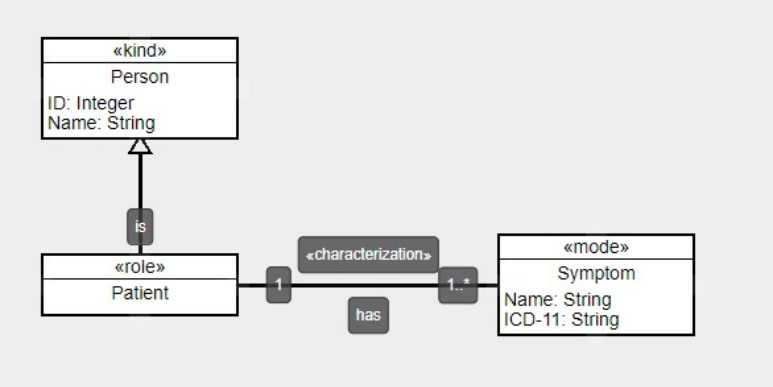

Figure 3.4: Example ontology visualized in the component

are five elements in this diagram - 3 stereotypes and 2 relationships, which are described in Czech and English. What the end user sees on the screen are the visual elements - they take information about the stereotype, attributes, cardinalities, etc. from the element model. The object model of the patient, the symptom, and their relation from the implementation point of view is shown in figure 3.5.

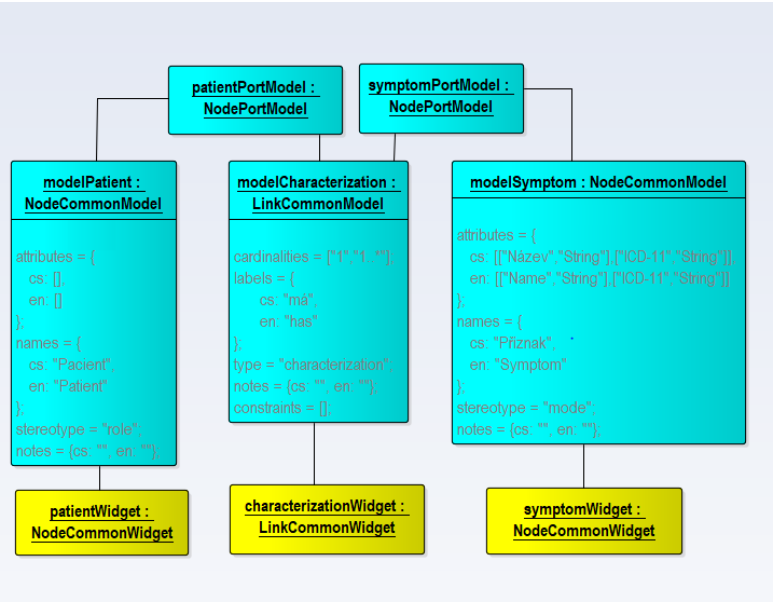

Figure 3.5: Object diagram of diagram element objects

3.3 Advanced features

This section contains the specifics of more advanced features, including their implementation and basic usage.

3.3.1 Constraints

Constraints of the model are implemented through collections of OCL statements pertaining to given relationship types. For instance, if a user wanted to include a constraint of "a `Kind` cannot be a subclass of a `Phase`", that constraint would be realized through the `Generalization` relationship; therefore, that statement would be in effect re-framed as "if a `Generalization` relationship has a `Kind` at its source, then a `Phase` cannot be its target", where the source and target refer to the element from which the relationship is created and the element to which the element is connected, respectively.

OCL (Object Constraint Language) is the language used to produce these constraints. Functions enabling access to diagram elements' attributes were created in order to make the construction of OCL statements as simple as possible. These include:

- for relationships:
  - getSourceNode(): returns the stereotype of the source end of the relationship
  - getTargetNode(): returns the stereotype of the target end of the relationship.
  - getLinktype(): the type of the relationship, for example "Generalization".
  - getName(language): the name of the relationship in a given language.
  - getSourceCardinality(): the cardinality at the source end of the relationship.
  - getTargetCardinality(): the cardinality at the target end of the relationship.
- for stereotypes:
  - getName(language): the name of the stereotype in a given language.
  - getRDF(): the RDF source of the stereotype.
  - getStereotype(): the type of the stereotype, for example "Kind" or "Relator".
  - getAttributes(language): a list of attributes in a given language, where attribute.getName() returns the name of the attribute and attribute.getType() returns the type of the attribute.

Using these functions and OCL syntax, we can construct the statement mentioned above as:

```
self.getSourceNode().getStereotype() = "Kind"
implies
self.getTargetNode().getStereotype() <> "Phase"
```

This statement would be included in `Generalization` relationships.

Evaluation

After creating constraints and building a model, the user can evaluate the constraints using the "Evaluate" tool available on the menu bar. Upon activating the tool, the bottom panel appears with the list of relationships whose constraint checks failed. The user can then investigate and remedy any issues.

3.3.2 Validation

The act of "validating" in the context of the component compares the settings (that is, the list of available stereotypes, relationships, OCL constraints, cardinalities, languages, and attribute types) to either different settings or a particular model. There are three modes that the component supports:

Comparing the current model to the current settings. This option adds up diagram elements and their properties and tests if the model can be recreated with the component's current settings.

Comparing the current model to an external file. In this case, the component takes an XMI input (see below) and validates the current model with it.

Comparing the current settings to an external file. Here, the XMI input is analyzed to find out if the current settings of the component and the input are one and the same.

A potential use case of validation is comparing a model or settings to a certain standard. For example, an organization could have its own set of defined stereotypes, relationships, constraints, etc. with which to create models. Therefore, they might want to find out whether an imported model or settings are in conformance with their standard.

As mentioned above, in certain cases, the validation tool accepts an external input as a source for comparison. This input is based on XMI and ECore standards. More specifically: XMI syntax is used to convey ECore elements. These elements represent the elements available in the component or the elements present in a model.

ECore structure

ECore is a standard within the *Eclipse modeling framework* which allows for definition of metamodels.[6] A definition of a single metamodel, or a *package*,

[6]For more information, visit `https://wiki.eclipse.org/Eclipse_Modeling_Framework`

contains within it definitions of all diagram elements and attribute types, or *classes*. A package has a name, a URI source, and a prefix, where prefix refers to a shorthand for the source. For instance, if we were to serialize a metamodel with a `www.example.com` source and an `ex` prefix, a `Kind` class within the metamodel could be referred to as `ex:Kind`, which would translate to `www.example.com/Kind`.

In the context of the component, there are five classes that act as categories (or *supertypes* in ECore terminology) that are analogous to abstract classes from which other classes are derived. These five supertypes are:

- Stereotype,
- Relationship,
- Attribute Type,
- Cardinality,
- Language.

Every class is individually encapsulated in an `eClassifiers` tag and it also has several attributes: a name, which is mandatory (for example `Kind`, if we're considering a stereotype), a supertype, and any additional attributes that are defined according to the abstract class from which the class was derived from. These attributes are defined as `eAnnotations` within the class definition, and each `eAnnotations` contains a *source*, meaning a URI source for the attribute type, and a key-value pair that describes the attribute's value.

Consider a relationship of the `Mediation` type with the following attributes:

- `linkEnd = "Empty"`,
- `labeled = true`,
- `dashed = false`.

Here, the relationship would be transformed into this XMI notation:

```
<eClassifiers
            xsi:type="ecore:EClass"
            name="Material"
            abstract="false"
            interface="false"
            eSuperTypes="//Relationship">
            <eAnnotations source="ex:linkEnd">
                    <details key="linkEnd" value="Empty"/>
            </eAnnotations>
            <eAnnotations source="ex:labeled">
                    <details key="labeled" value="true"/>
            </eAnnotations>
            <eAnnotations source="ex:dashed">
                    <details key="dashed" value="false"/>
            </eAnnotations>
</eClassifiers>
```

Now, continuing with our `www.example.com` example, after compiling all classes into the package, we get:

```
<?xml version="1.0" encoding="UTF-8"?>
<ecore:EPackage
    xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
    name="Example"
    nsURI="www.example.com/"
    nsPrefix="ex">

...eClassifiers...

</ecore:EPackage>
```

This XMI serialization can then be used to store the component's settings in a single file for purposes of validation and importing settings.

3.3.3 Configuration

There are four main ways through which the user can customize the component.

Configuration dialogues

The Settings drop-down on the menu bar contains options to create and delete stereotypes, constraints, attribute types, languages, and cardinalities via modal dialogues. They can, therefore, be modified right through the component itself.

Turtle importing

The component supports importing stereotypes with a .ttl (*Turtle*) file. When importing, the component is specifically looking for `owl:Class` definitions

with `rdf:label@language` descriptions, where `language` is the specified language code you are looking for. For example, these triples

```
<http://onto.fel.cvut.cz/ontologies/ufo/kind>
rdf:type owl:Class;
rdfs:subClassOf
    <http://onto.fel.cvut.cz/ontologies/ufo/non-rigid-type>;
rdfs:label
    "Kind"@en ,
    "Kind"@cs .
```

with the `en` language parameter return a
`http://onto.fel.cvut.cz/ontologies/ufo/kind: "Kind"` stereotype.

The file can be imported via a hyperlink in either the local configuration or the `Stereotypes` option in the Settings on the menu bar.

XMI importing

The XMI serializations mentioned in section 3.3.2 can be used to import settings that have been previously created.

Local configuration

If the user does not want to use XMI files to configure the component nor the settings modal dialogues within the component itself, or perhaps they wish to translate the component, they can customize the settings located in the `src/config` folder. In these files, the user can change the default settings that are present without any external sources. The instructions for customizing these are within the files themselves.

3.3.4 OWL exporting

Since no JavaScript library that supports OWL serialization and is compatible with the component is available on the Internet, a custom function that serializes the current model into OWL language had to be implemented. This function was designed by studying *Menthor Editor*'s OWL "simple" output with various models.

Chapter 4

Evaluation

In this chapter, we will evaluate the work we have done over the previous chapters, how it stands up to existing solutions, what testing strategies we have employed to assess the component's viability and what were the results.

4.1 Comparison to existing solutions

To make the comparison as formal as it can be, let us consider two examples of ontologies that we will use to judge the solutions' comparative strengths and weaknesses.

First, consider an ontology in which a Person has a name and a date of birth that can be categorized into either a Man or a Woman, where a Man and a Woman can be related to one another via Marriage as Husband and Wife. Figure B.1 portrays this ontology in the component.

The second example looks at a similar concept, but through a different lens: in it, we have a Person that is a member of a Family with a Head of family. The Family is located at a Home, which is a role of a Property. A Property can be either a House or an Apartment. The Property is located in a City. A Person can also enroll in a School, which functions in a Property. Figure B.4 portrays this ontology in the component.

4.1.1 Menthor Editor

The user experience of modeling with *Menthor Editor* is on a subjective level slightly more bothersome than our component, because sometimes certain actions cause unintended behavior - for example, confirming a dialogue sometimes does not save the data inputted within or making lines of relationships look the way the user wants them to requires precision that other solutions do not require. Also, implementing a `2..*` cardinality necessary for one relationship in the second example is not possible. However, there is one significant advantage – in OntoUML, the `Derivation` relationship is special in that is not drawn between two stereotypes, but between a stereotype and a relationship, which is currently not supported in our component. It is, therefore, able to correctly portray the Enrollment part of the second example. Exporting the models to OWL in "simple" mode produces very similar results

to our component – this is to be expected since our exporting algorithm was based on the results of *Menthor Editor*'s algorithm. Figures B.2 and B.5 show the representation of the example models in *Menthor Editor*.

4.1.2 Enterprise Architect

As *Enterprise Architect* is a professional commercial application, its modeling experience and tool set is excellent. The extension used also provided prefab patterns, allowing for instance a Kind with two Sub-kinds connected via Generalization to be added via a single drag of the mouse. Nonetheless, OWL exporting is nonexistent; the option is there, however selecting it produces an empty XML document. Default OntoUML constraints were also added in out of the box, and *Enterprise Architect* automatically suggests constraint-conforming relationships when connecting stereotypes. However, the extension's constraints contain a bug affecting this feature, which forbade the modeling of the Enrollment section of the second example, which is why that part is not present in the figure. Figures B.3 and B.6 show the representation of the example models in *Enterprise Architect*.

4.1.3 Summary

The component's main advantage that makes it stand out from the rest is its nature as a web component; neither of the two other solutions allow integration into web systems. This, combined with its comparably simple import/export capability of not only the model, but settings as well put the component into its own category.

This is not to say that the other solutions do not have their advantages – as mentioned above, *Enterprise Architect*'s design makes modeling a breeze, while *Menthor Editor*'s feature set relevant to OntoUML modeling is vast compared to our component. The most glaring omission is the lack of support for `Derivation`'s connectivity, another is the inflexibility of stereotypes – they currently cannot be resized, for example. These facts are not unchangeable, however – there are not, as far as we can tell, insurmountable issues that would prevent our component to catch up, feature-wise, to the other solutions.

4.2 Automated testing

Automated testing was implemented via *Jest*, a library designed for testing React components.[1] Twenty tests were designed to check the internals of the component, more specifically the functions that are used in modeling and interfacing with OWL, XMI and Turtle standards. Specific examples include:

- Manual modeling - because model elements are JavaScript objects with attributes, they can be tested using conventional object testing without using the user interface at all.

[1] Available at `https://jestjs.io`.

- Constraints - whether OCL statements of various complexity can be evaluated.
- Interfaces - checks fetching, importing and exporting functionality.
- Configuration testing - assesses the reliability of modeling with various configurations - different element pools, unusual setups, etc.

After automatic testing, conventional modeling was evaluated without problems – this result is expected since almost all functionality tested is either getters/setters or implemented through the *Storm React Diagrams* base. However, some bugs were found during the testing of importing, exporting and validation, which were then promptly fixed. Interfaces and configuration implementation did not import non-compliant inputs and recognized compliant ones.

4.3 User testing

User testing was devised in order to test the user experience of the component. Three scenarios analogous to common usage were created and handed to test subjects along with a user's manual and forms that they filled out describing their experiences. Unfortunately, due to time constraints, only one person out of six was able to participate in testing. The scenarios, user's manual, forms, and results are available in the attachments.

Chapter 5
Conclusion

The goal of our work was to design, implement and evaluate an advanced web component for the creation and editing of OntoUML models that is suitable for ontological conceptual modeling. We started with defining our requirements and use cases, designing the interface and selecting the adequate JavaScript package serving as the base for our modeling feature. Implementation of the component had its challenges, but we were able to meet all of our functional requirements, the most important of which are graphical modeling, OWL exporting, constraint evaluation, settings importing/exporting, and validation. Comparing the component to other solutions and testing revealed that while the component has a few bugs and the amount of implemented functionality is relatively smaller, its nature as a web component gives it a significant advantage in terms of expandability and interoperability.

5.1 Future work

We recommend that future work is directed at bug fixing and expanding functionality, especially in terms of exporting – the component could support more types of OWL languages and various UML exporting protocols. The experience of modeling itself could also be improved, for example with allowing realignment of elements, resizing stereotypes, and so on. Also, significantly more time needs to be dedicated towards testing, especially user testing, and integration testing with other web components and web systems.

Bibliography

[1] "What Is An Ontology?". 2019. www-ksl.stanford.edu. Accessed May 22 2019. http://www-ksl.stanford.edu/kst/what-is-an-ontology.html.

[2] "The History Of React.Js On A Timeline | @Risingstack". 2018. Risingstack Engineering - Node.Js Tutorials & Resources. Accessed May 23 2019. https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/.

[3] "Virtual DOM And Internals – React". 2019. reactjs.org. Accessed May 22 2019. https://reactjs.org/docs/faq-internals.html.

[4] "Introducing JSX – React". 2019. reactjs.org. Accessed May 22 2019. https://reactjs.org/docs/introducing-jsx.html.

[5] "npm". 2019. npmjs.com. Accessed May 22 2019. https://www.npmjs.com/browse/depended.

[6] "OWL 2 Web Ontology Language Primer (Second Edition)". 2019. W3.org. Accessed May 22 2019. https://www.w3.org/TR/2012/REC-owl2-primer-20121211/.

[7] Guizzardi, Giancarlo, Gerd Wagner, João Paulo Andrade Almeida, and Renata S.S. Guizzardi. 2015. "Towards Ontological Foundations For Conceptual Modeling: The Unified Foundational Ontology (UFO) Story". Applied Ontology 10 (3-4): 259-271. IOS Press. doi:10.3233/ao-150157.

[8] "OntoUML". 2016. OntoUML Community Portal. Accessed May 22 2019. https://ontouml.org/ontouml/.

[9] "What Is XMI (XML Metadata Interchange)? - Definition From Whatis.Com". 2019. searchmicroservices.techtarget.com. Accessed May 22 2019. https://searchmicroservices.techtarget.com/definition/XMI-XML-Metadata-Interchange.

[10] "Pricing - Enterprise Architect | Sparx Systems". 2019. sparxsystems.com. Accessed May 22 2019. https://sparxsystems.com/products/ea/shop/index.html.

[11] "Pricing And Ordering". 2019. nwoods.com. Accessed May 22 2019. https://www.nwoods.com/sales/index.html.

[12] "MPL 2.0 FAQ". 2019. Mozilla. Accessed May 22 2019. https://www.mozilla.org/en-US/MPL/2.0/FAQ/.

[13] "A Quick Guide To Gplv3 - GNU Project - Free Software Foundation". 2019. Gnu.Org. Accessed May 22 2019. https://www.gnu.org/licenses/quick-guide-gplv3.html.

[14] "The MIT License | Open Source Initiative". 2019. Opensource.Org. Accessed May 22 2019. https://opensource.org/licenses/MIT.

Appendix A

Design diagrams

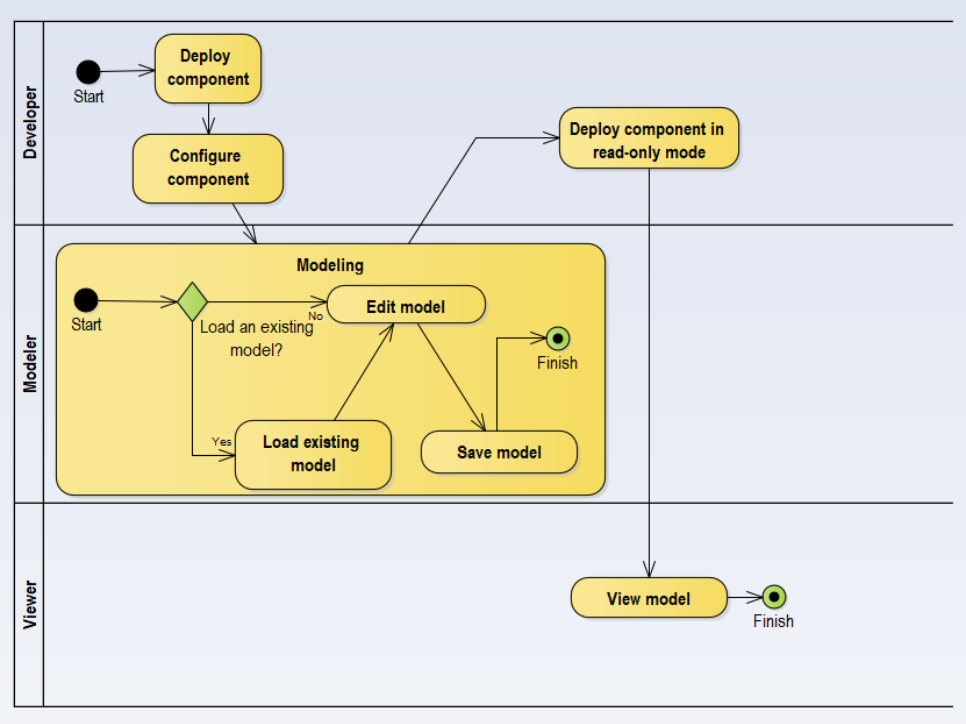

Figure A.1: Activity diagram of deployment, modeling and viewing

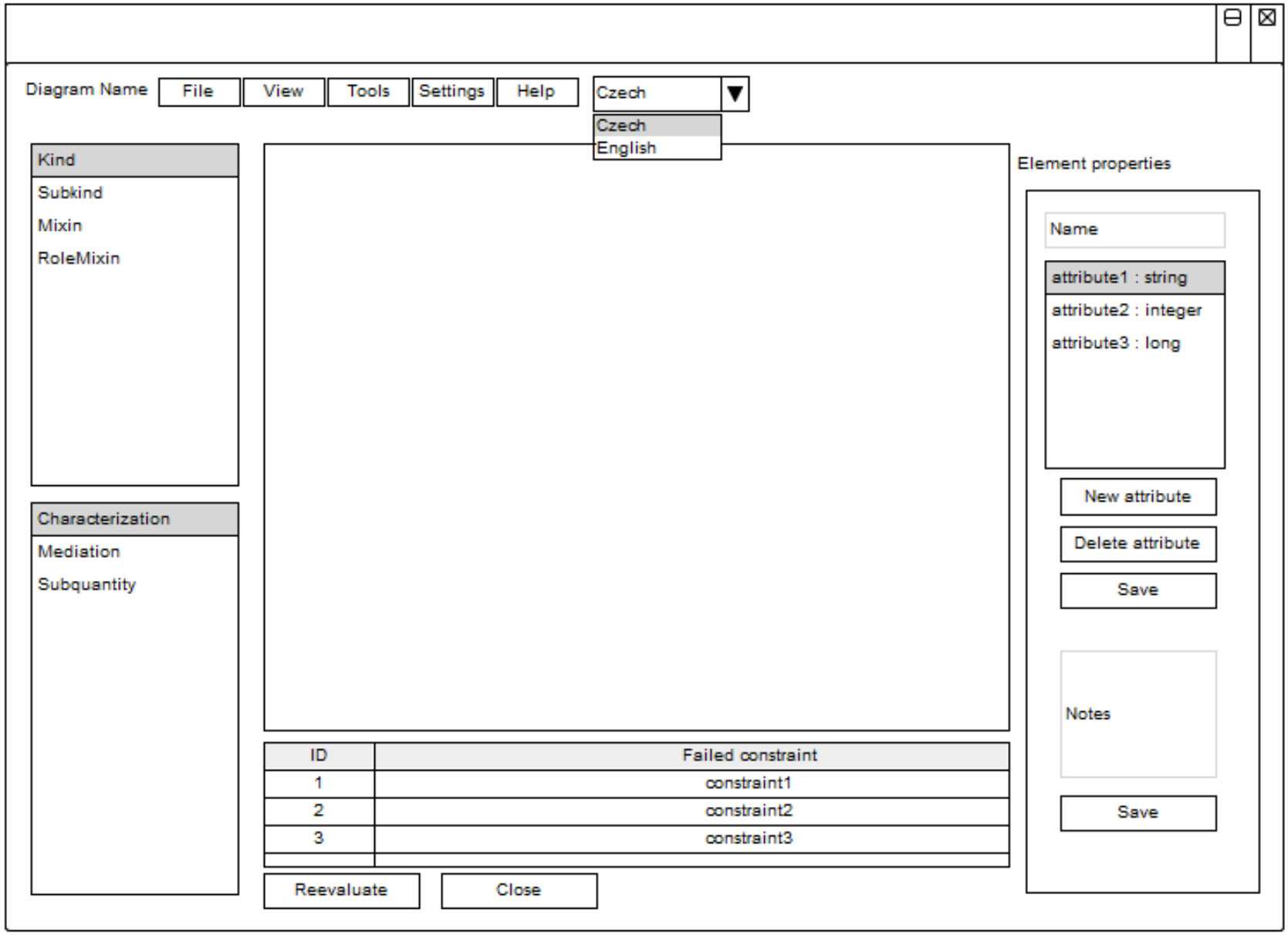

Figure A.2: Wireframe of the component

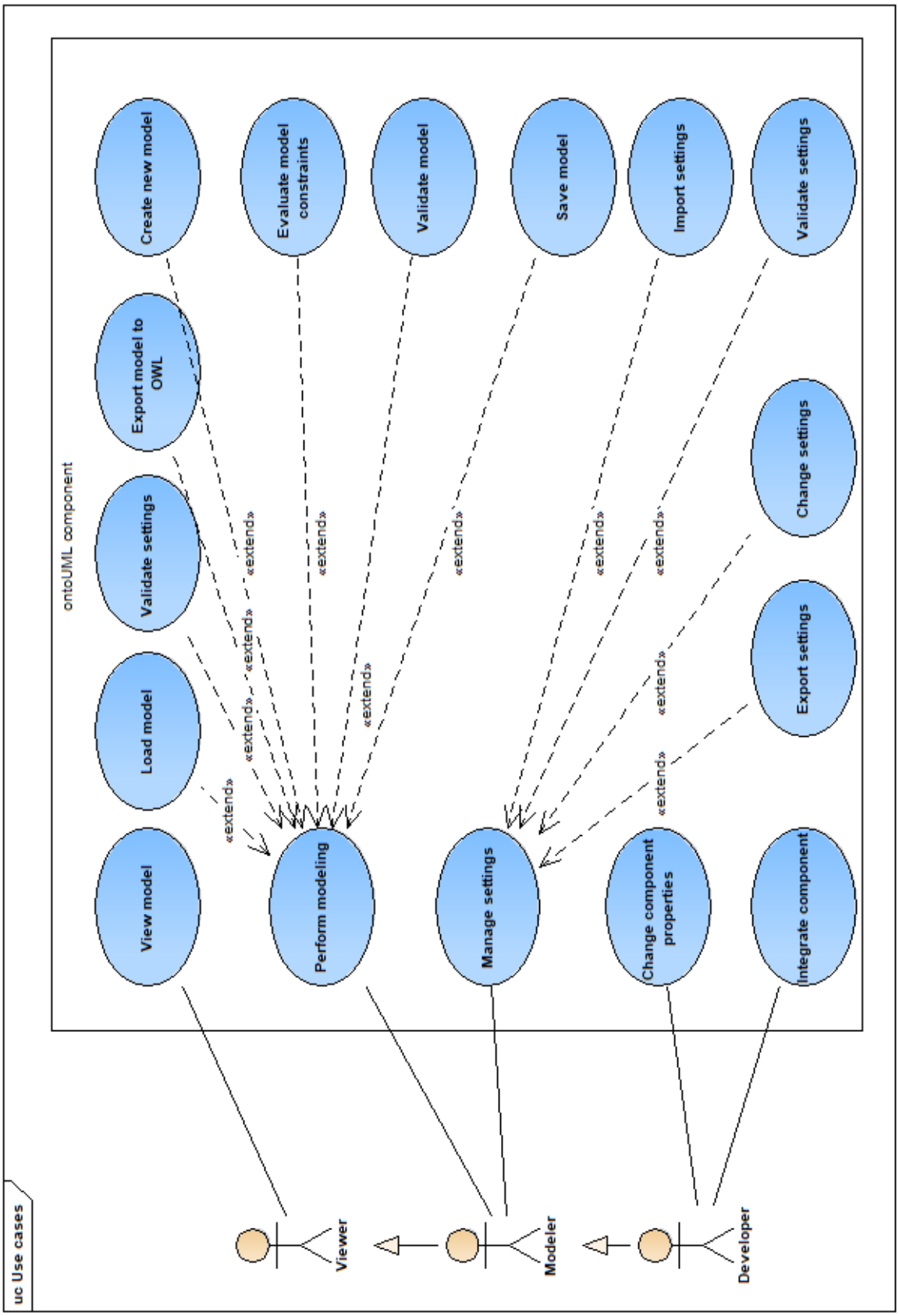

Figure A.3: Use cases

Appendix B

Example models

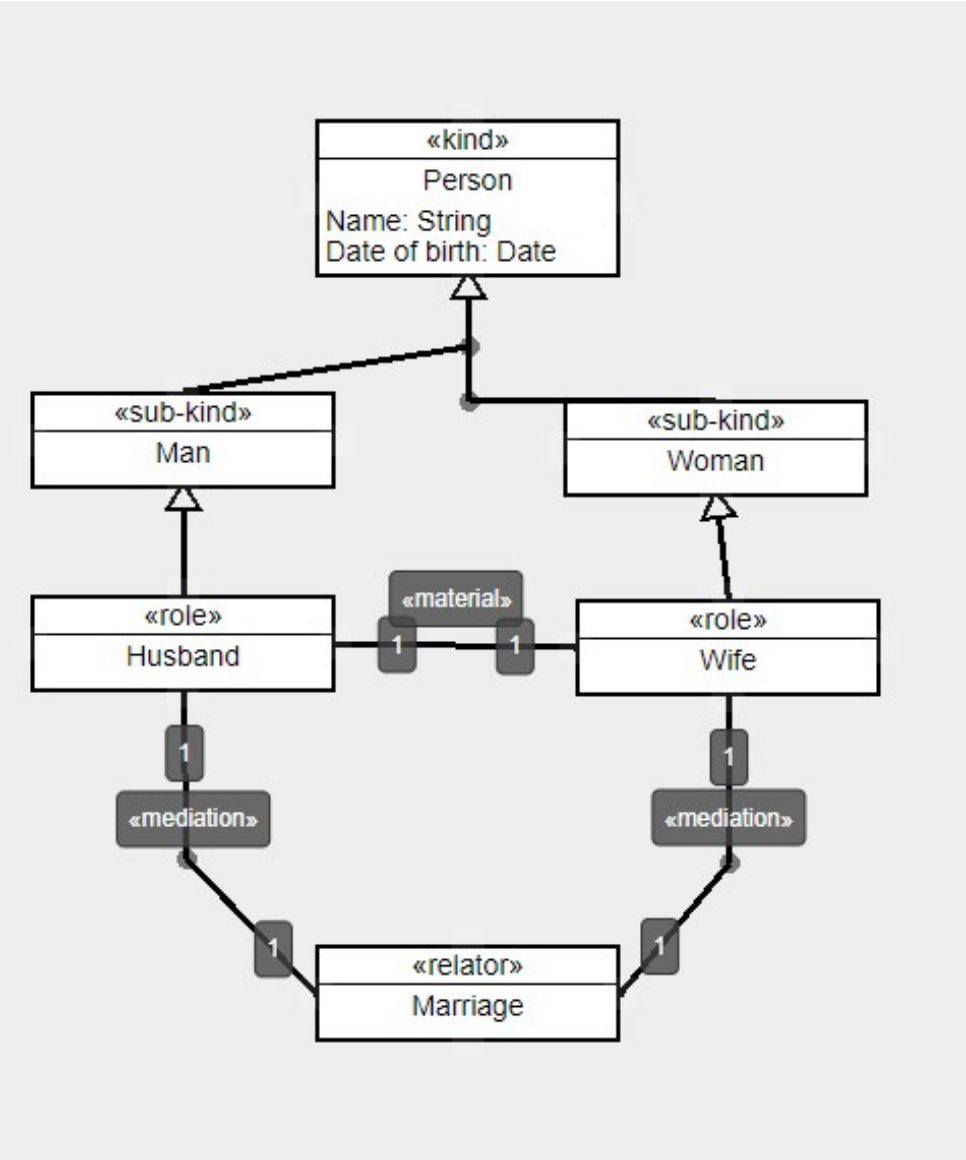

Figure B.1: The first example ontology modeled in the component

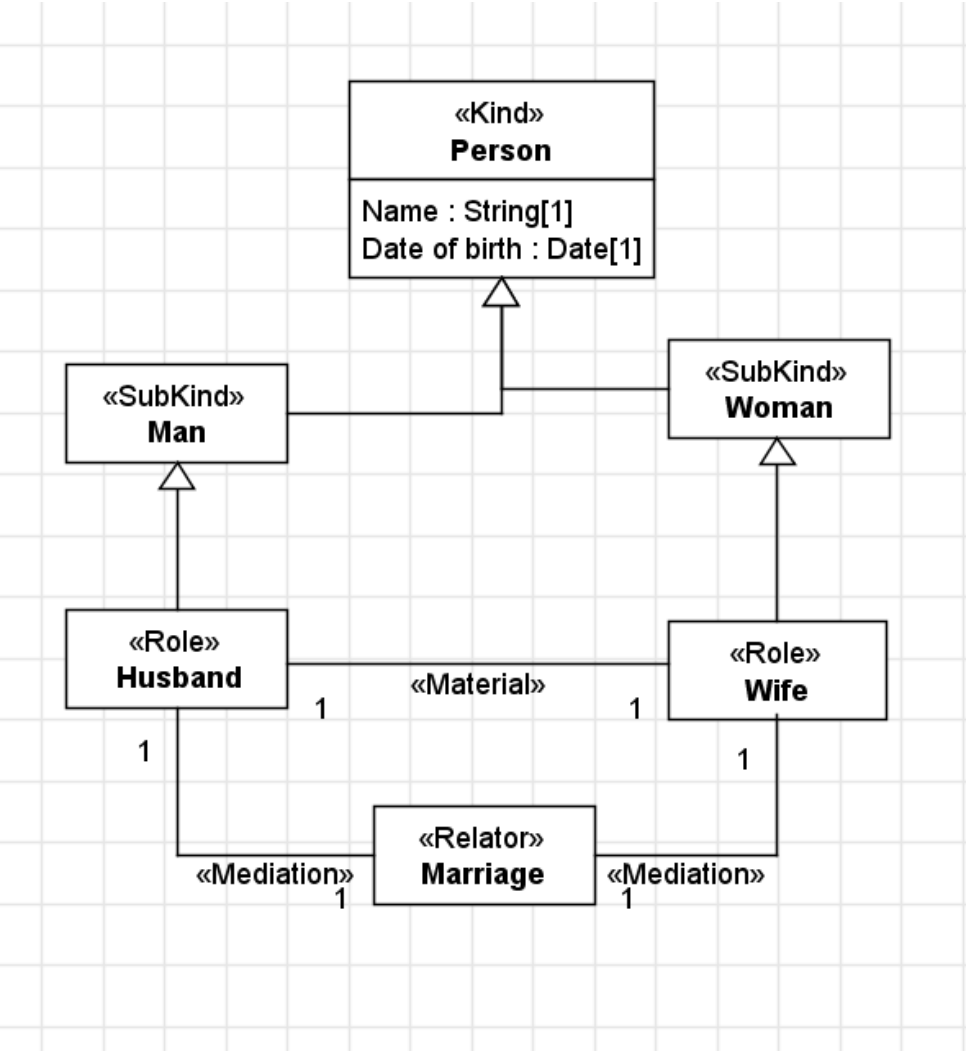

Figure B.2: The first example ontology modeled in Menthor Editor

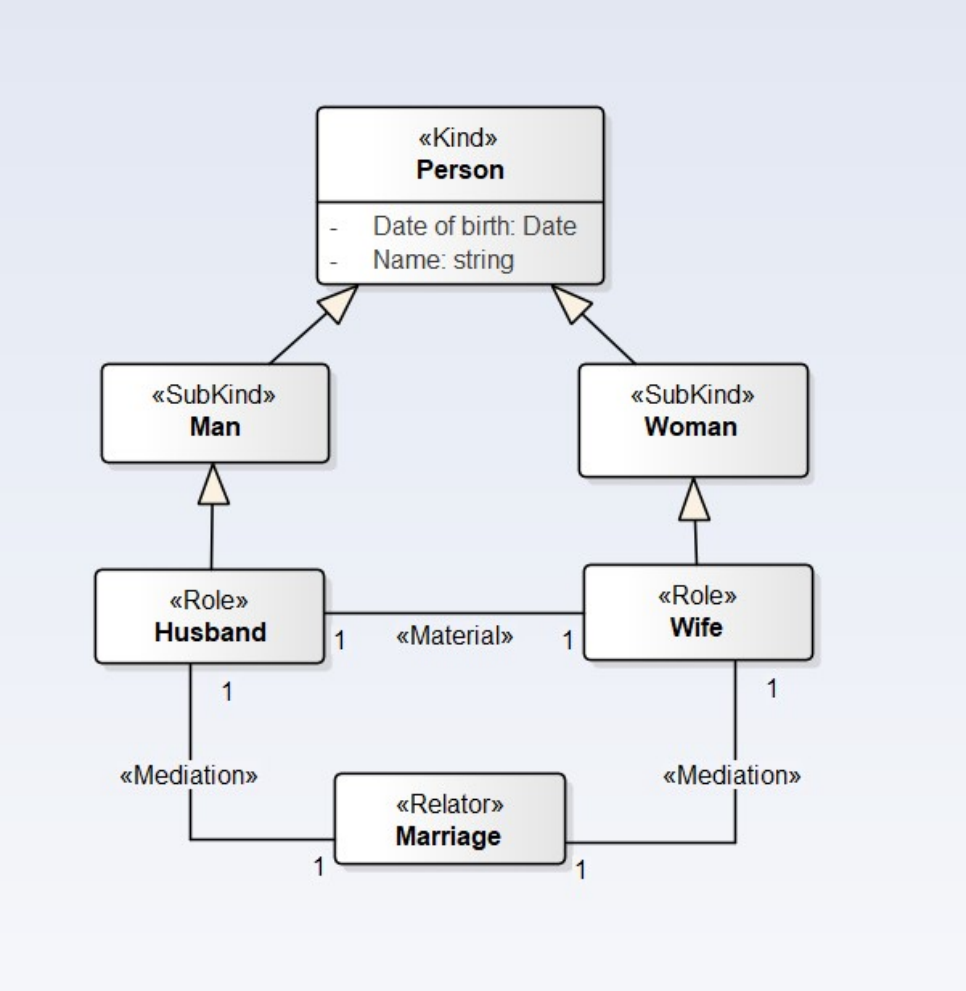

Figure B.3: The first example ontology modeled in Enterprise Architect

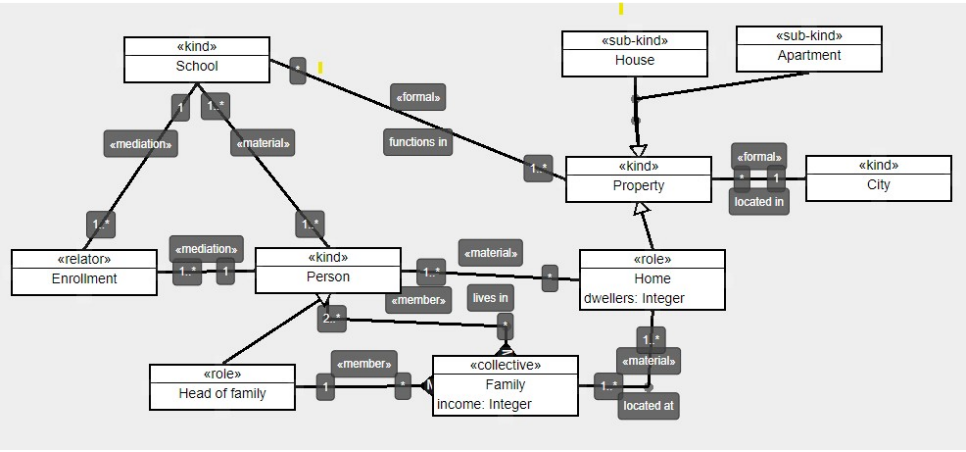

Figure B.4: The second example ontology modeled in the component

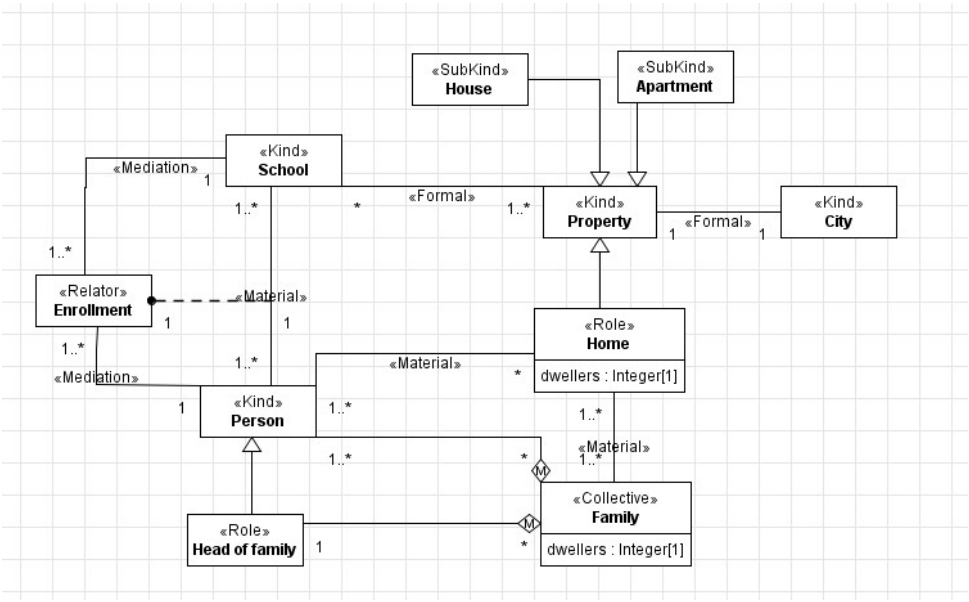

Figure B.5: The second example ontology modeled in Menthor Editor

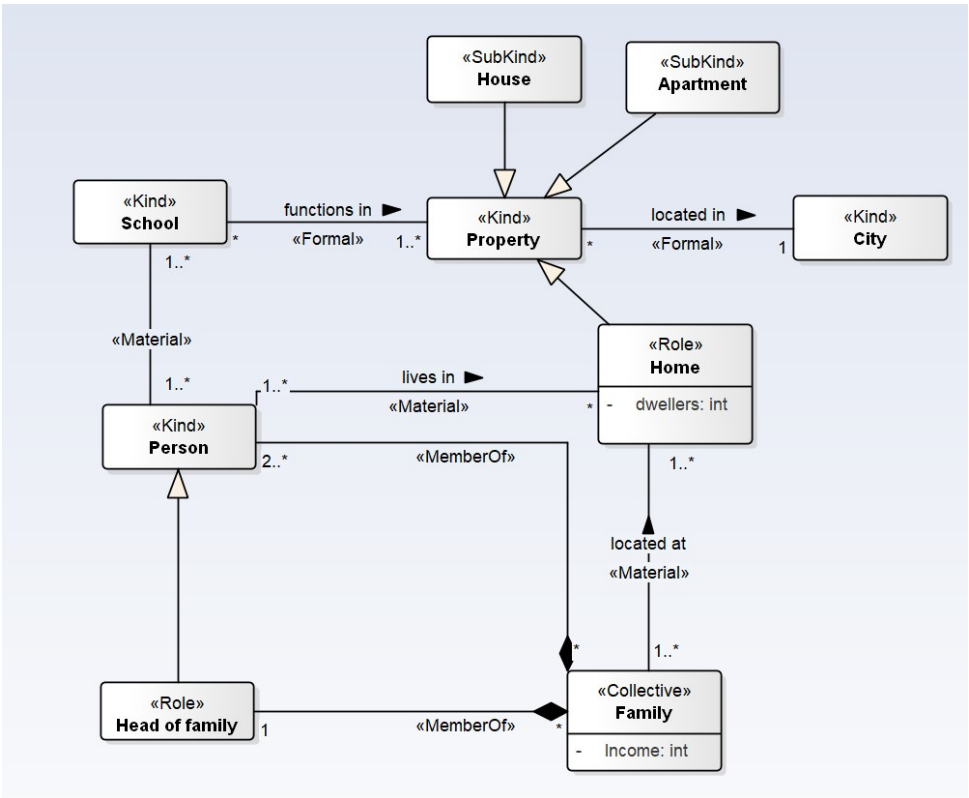

Figure B.6: The incomplete second example ontology modeled in Enterprise Architect

Appendix C

Guide to the attachments

- `readme.txt` - short description of attachment contents
- `src/` - the component's source code
- `user-testing-manual` - the user's manual, testing scenarios, and feedback forms
- `user-testing-results` - the results of user testing