Using ontology mapping to automate derivation of transformations for data integration

D.G.A. Stolp

Thesis for a Master of Science degree in Telematics

University of Twente
Faculty of Electrical Engineering,
Mathematics and Computer Science

February 2009

Graduation Committee:

- Dr. ir. M.J. van Sinderen
- Dr. ir. M.W.A. Steen
- Dr. L. Ferreira Pires

Abstract

The Purchase Order Mediation scenario of the Semantic Web Service challenge describes an interoperability problem, in which two companies want to do business, and need to integrate their IT systems. Within the A-MUSE project, a solution to this problem is devised that uses ontologies to describe the two IT systems. This solution requires a tool that can create mappings between these two ontologies, and use these mappings to generate the necessary data transformations.

This master thesis describes the research and development process that resulted in the creation of that tool. A state-of-the-art survey was conducted which resulted in an overview of existing ontology matching algorithms and mapping tools. These findings confirmed that there is currently no tool that can readily be used, and provided the necessary knowledge to start the development of a mapping editor.

The resulting editor is capable of loading two ontologies that are expressed using the Web Ontology Language (OWL). The user of the editor can create mapping relations between classes and properties of the two ontologies, and provide additional information for each of these mappings. Based on the mappings, the tool can generate transformation code. This code is then used by other tools that are part of the solution. These tools perform the necessary translations of the messages that are exchanged between the two systems that have to be mediated, such that the systems can communicate with each other.

Acknowledgment

This work is part of the Freeband A-MUSE project. Freeband is sponsored by the Dutch government under contract BSIK 03025.

Preface

This thesis is the resulting product of my Master's assignment, which I have carried out at the Telematica Instituut. During the process of doing research, developing a software tool and ultimately writing this thesis, a lot of people have provided valuable assistance. I hereby would like to thank everyone for their contribution to this work.

A small number of people deserve a special acknowledgement. My direct supervisors and graduation committee: Maarten Steen at the Telematica Instituut, for providing the opportunity to do the assignment in the first place, and for his continuing efforts to steer my research in the right direction. Marten van Sinderen at the University of Twente, his suggestions and feedback have made this thesis more academically sound and better structured. And Luís Ferreira Pires, whose last-minute suggestions made this thesis all the more readable and correct.

I would like to thank Stanislav Pokraev at the Telematica Instituut for his help. He came up with more ideas and suggestions than five Master students could implement, but they always helped to improve the product. Thankfully, Jaap Reitsma was there to help me pick out the most relevant and doable ideas, and to assist in their implementation.

A big thank you goes out to my parents. Regrettably, my mother passed away just before I started this assignment, but I will always remember her for her interest and unconditional support. My father's curiosity and advice over the years made it possible for me to come to where I am now: writing a preface for my completed Master's thesis, with the prospect of a fun and challenging career.

Finally, I would like to thank my girlfriend Marlous very much for her continuing motivation, patience and advice. Our conversations helped me structure my own ideas, and her suggestions often made the text in this thesis more complete and understandable. Her support for me during this assignment has been invaluable.

Daan Stolp
Enschede, 19 February 2009

Contents

Chapter 1

Introduction

From the first moment that IT systems were being used in businesses, people were looking for ways to interconnect them. These interconnections allow for even more automation of business processes than individual IT systems can achieve. However, since most vendors of IT systems used their own format for data storage and communication, implementing such interconnections has never been an easy task. Over the course of time, numerous technologies have been developed that aim to make this process as easy as possible. In the last couple of years, research has focused mostly on the use of Web Services to achieve this task.

While Web Services facilitate the exchange of messages between two systems, this is only part of the interconnection problem. The other problem is that both systems may use different concepts, different ways to represent real-world entities. If you want to interconnect the two systems, you need to make sure that both systems understand each other's concepts, or at least that there is a mediator or 'interpreter' that can translate back and forth between the systems. This means that the systems should not only know *how* to represent messages exchanged with the other party (syntax), but also what the other party *means* with the messages it exchanges (semantics).

In order to make the semantics clear, Web Services can be 'enriched' with semantic information. Currently, research is conducted in order to find out how to best apply and make use of this semantic information, such that an automated interconnection of two systems becomes possible.

Non-automated solutions for connecting systems are already available and widely in use. Tools exist that make it very easy to specify how to convert messages from one system into messages that the other system understands. However, this is still a completely manual process and the only data that is taken into consideration are the data fields and message types of the two systems. The semantics behind this data, the meaning of the fields, is only known by the domain experts. So even with easy to use mapping tools available, this is still a very hard, time consuming and error prone process. It is also a costly process because of the time, energy, and manpower required to interconnect the systems. Studies indicate that even a small improvement in the ease with which

systems can be interconnected, could add a percentage point to global GDP (Bugajski, as cited in [8]).

From the above, it should be clear that automated, or partially automated solutions for interconnecting IT systems can play a big role in the current IT industry. One initiative that tries to stimulate research into this topic is the Semantic Web Services Challenge.

1.1 The Semantic Web Services challenge

The topic of this research is based on the work on the Semantic Web Service Challenge (SWS Challenge) [24]. This is an international challenge organized by DERI Stanford[1] and has participants from universities, research institutes, and commercial businesses. Its goal is "to develop a common understanding of various technologies intended to facilitate the automation of mediation, choreography and discovery for Web Services using semantic annotations". In order to achieve that goal, they created several problems that the participants need to solve.

This initiative is indeed set up as a challenge rather than a contest, meaning that workshop participants mutually evaluate and learn from each other's approaches.

The problem scenario that this thesis focuses on is the Purchase Order Mediation scenario. In this scenario, two fictitious companies want to do business and link their IT systems. However, both parties use different systems and communication protocols. Because of this they cannot directly communicate to each other. The challenge is therefore to design a mediator that allows the companies to link their IT systems and do business.

1.1.1 The Purchase Order Mediation scenario

In the SWS Challenge, two companies want to do business. These are Blue Company and Moon Company, who have the roles of customer and manufacturer, respectively, in the SWS Challenge scenario. Both companies have external interfaces to their IT systems in the form of web services. The WSDL specifications of these services, as well as a natural language description, are given by the SWS Challenge organizers.

Even though both parties use Web Services, this alone is not enough to ensure interoperability. The problem is that both parties use different choreographies and data representations. Blue company adheres to the RosettaNet specification, while Moon Company uses a proprietary legacy system. So even though it is technically possible for Blue Company to send a message to Moon Company, the latter will not understand the contents of the message. The mediator that is to be built, must enable Moon company to understand and exchange RosettaNet messages with outside parties (Blue Company). An overview of the scenario is given in figure 1.1. More details can be found on the SWS Challenge website [24].

[1] Digital Enterprise Research Institute Stanford. http://www.deri.us/

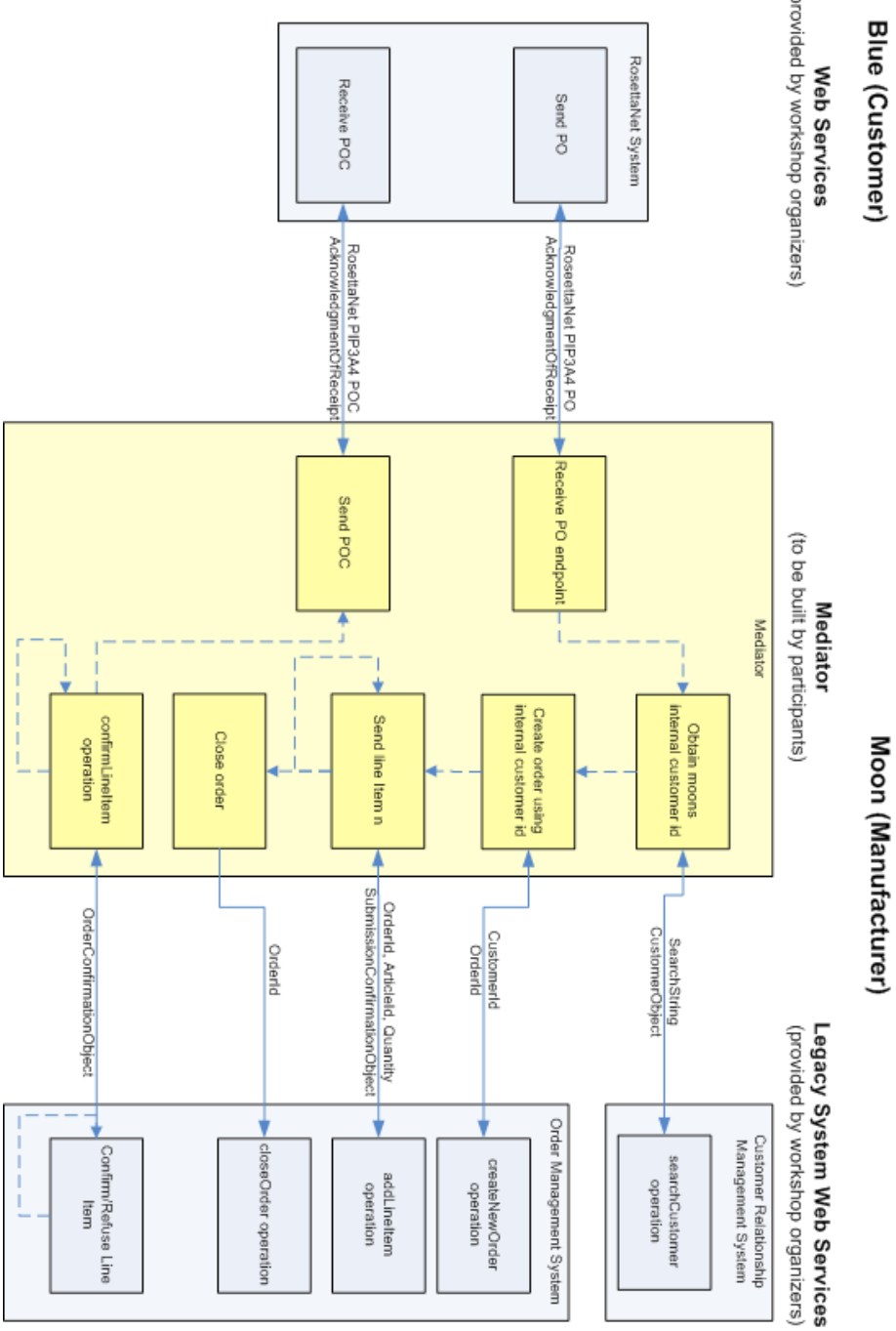


Figure 1.1: Overview of the mediation scenario of the SWS Challenge.

1.1.2 Solutions to the challenge

One way of looking at the problem that is proposed by the SWS Challenge is to divide it into two separate but related problems. Blue company uses different terms, names, and entities than Moon company, so there is a *data mismatch*. However, the order in which operations are called, the way responses are sent, errors are handled, etc. also differ between Blue and Moon company, so there is also a *behavior mismatch*. All solutions that are proposed need to solve both these sub-problems in order to solve the challenge.

The literature research on this topic [23] contains a more detailed overview of various solutions to the challenge. For clarity, the solution of the A-MUSE project is discussed below.

1.2 The solution of the A-MUSE project

As part of the Freeband A-MUSE project, the Telematica Instituut and the Center for Telematics and Information Technology (CTIT) of the University of Twente are working on a solution to solve the problem proposed by the challenge. The entire approach is described by Quartel et al. in [19] and [18]. The goal is to design and develop a mediator that conforms to Evaluation Success Level 2. This means that only the data used by this mediator (e.g. configuration files) needs to be changed when the problem level changes, not the application code. So if either of the parties decides to change their data or behavior models, only a reconfiguration of the mediator is required, not recompilation.

The Telematica Instituut and CTIT try to lift the solution to a higher level, away from the technicalities of the IT systems and more towards the business domain. In that case, the integration problem can be tackled directly by business domain experts. In contrast, nowadays the domain experts usually compile a requirements document that is sent to IT experts. The IT experts then build the integration solution, but this is often paired with miscommunication and misinterpretation between the business domain experts and the IT experts.

In order to accomplish this shift from the technical domain to the business domain, four steps are necessary (see figure 1.2). First, the service description of the IT systems needs to be transformed into a format that can be used by the domain experts. By adding semantic information, the technical descriptions get meaning. Then, the integration problem needs to be solved at the business layer. Finally, the solution needs to be transformed back to an IT solution. The following sections describe this process.

1.2.1 Transform the IT domain to the business domain

In this step, information models need to be derived from the IT systems. The IT systems are in fact web services, which are described using WSDL. The data models that are used are described using XML Schema. Using a defined set of rules, these descriptions can be transformed into an information model and a

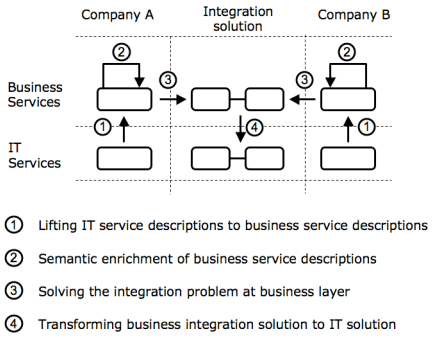


Figure 1.2: The necessary transformation in the solution.

behavior model. This process is described in more detail in [18]. The information model is called an *ontology,* which is represented using the Web Ontology Language (OWL). The behavior model can be represented using languages that support behavior representation, such as ISDL.

The concept *ontology* and the OWL language play an important part in this research. The following text box gives for a short introduction to ontologies and the Web Ontology Language (OWL).

Ontologies and OWL

The term *ontology* can have a different meaning, depending on the industry or context within which it is used. One definition that is very useful, and widely used, from the viewpoint of the IT industry, is the one by Gruber [10]. He defines an ontology as:

"a set of representational primitives with which to model a domain of knowledge or discourse"

These primitives are usually classes, attributes, and relationships. Using these primitives, it is possible to give meaning and constraints to the entities that describe the domain. These "semantic" properties of ontologies distinguish them from simple data models or database schemas. In addition, languages for specifying ontologies are usually completely independent from lower level data models. And this, according to Gruber, makes them a good tool for integrating different systems.

One such language for specifying ontologies is the Web Ontology Language (OWL) [3]. OWL is an RDF/XML based language and is endorsed by the World Wide Web Consortium, the main standards organisation for web technologies. Using OWL, one can describe the same representational primitives as mentioned by Gruber.

1.2.2 Semantic enrichment of business service descriptions

The next step is to add semantic information to the information models. This is information about relations or properties of entities that are present in the real world, but which are not encoded in the web services and WSDL documents of the IT systems. One way to achieve this is by using the Universal Data Element Framework (UDEF). UDEF provides globally standard identifiers with which one can tag elements in ontologies. One can now relate two different ontologies by looking at the associated UDEF tags.

This tagging is mostly a manual process. It requires knowledge from domain experts in order to make the correct semantic enrichment. The UDEF is structured as a hierarchical tree, so finding the most appropriate tag for a particular element means going through the tree top-down until the most appropriate tag for the element is found.

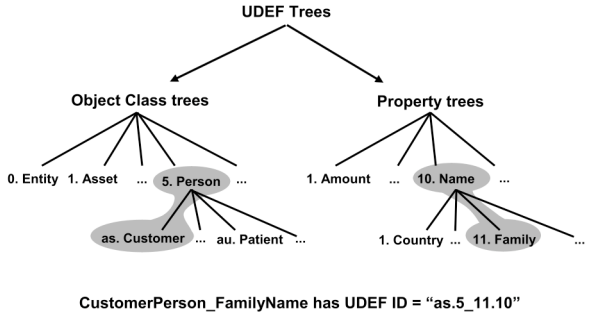


Figure 1.3: UDEF tree

1.2.3 Solving the integration problem at the business layer

When the problem is completely lifted from the IT domain to the business domain and the semantic information has been added, the integration problem can be tackled. This means that inconsistencies or mismatches in the data models and behavior models need to be solved. First, the relations between different elements in the two information models need to be defined. Second, these relations need to be formulated using some formal language. When this is done, it becomes possible to transform these mappings back to the IT domain.

Not only the information models need to be mapped, but also inconsistencies between the behavior models need to be solved. If, for example, one party expects an acknowledgement to a message, but the other party never sends such an acknowledgement, there is a mismatch. Such mismatches should also be solved by either automatically or manually identifying the mismatches and proposing solutions to solve them.

1.2.4 Transform the solution back to the IT domain

When all mappings have been defined, the solution should be transformed back to the IT domain. This is done by transforming the information mappings

to transformation specifications. These specifications 'translate' messages from one format to another, based on the relations that were specified in the previous step.

The behavior is translated into a BPEL specification. Using BPEL, the sequence, ordering and types of messages can be manipulated in order to facilitate interoperability between the two parties.

1.3 Aim of this research

This research focuses on the last two steps, namely the specification of the mappings between ontologies and the translation of those mappings to the IT domain by generating the required transformations. Furthermore, this research only deals with the mapping of information. The mapping of behavior to BPEL is dealt with o.a. in [5] and is not part of this research.

The specification of mappings or relations between concepts is always necessary in integration solutions. However, the problem with 'traditional' mapping approaches is that the domain experts, who need to map the concepts, and the technology experts, who have to implement the mappings on a technical level, have a hard time communicating with each other and staying on the same course. This results in a lot of communication overhead and at worst, even incomplete or incorrect mappings. The solution that is proposed in this thesis is to minimize the required communication between the domain and the technology experts by creating a tool that is able to create the technical transformations based on the input from the domain experts.

Mapping of ontologies is not new. Surveys such as [4] and [12] list various existing tools or algorithms that can aid in the mapping of ontologies. These algorithms do not, however, make it possible to derive transformations from the mappings. This research project aims to fill this gap. In short, the research objective can be formulated as follows:

The research objective of this project is to develop a tool for generating transformations from mappings between two ontologies by selecting appropriate mapping algorithms through a state-of-the-art survey and implementing the tool as a plug-in for the Eclipse platform.

1.4 Research approach

In order to achieve the formulated research objective, a number of activities had to be performed. This section lists these activities and describes how they relate to each other. A visual overview of these steps is given in Figure 1.4.

1. **Perform a state-of-the-art survey to determine what ontology matching algorithms are available for use in the solution to the Purchase Order Mediation scenario of the SWS Challenge.** This resulted in an overview of currently available ontology matching algorithms, describing their characteristics and capabilities. These algorithms

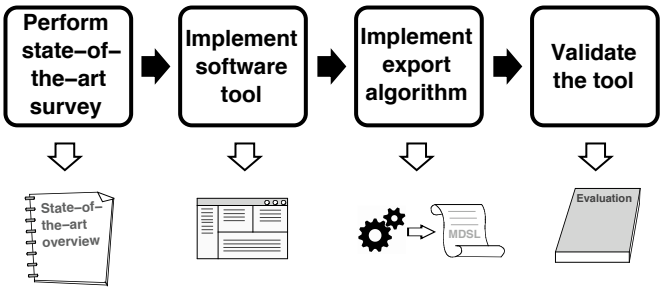


Figure 1.4: The research approach for this project.

can be used in a tool that supports the creation of mappings between two ontologies.

2. **Implement a software tool that supports the creation of mappings between two ontologies.** Such a tool enables the user to create meaningful mappings between entities in the ontologies. The exact requirements for the mapping tool have been derived from the work of Stanislav Pokraev [17].

3. **Implement an algorithm that creates transformations based on the resulting ontology mappings.** The mappings that were defined using the tool that was created in step 2 can now be used in order to derive transformations. In this step, an algorithm was developed that can create these transformations. These transformation describe how data from one ontology must be manipulated in order to map to the second ontology.

4. **Validate the tool by performing a case study, using the Purchase Order Mediation scenario as context for this study.** The case study served to demonstrate the use and evaluate the tool. This evaluation shows how the tool contributes to achieving the research objective, and to identify opportunities for further improvements.

1.5 Structure of this thesis

In order to report on the design and execution of this research, the following structure is used in the remainder of this report. Figure 1.5 shows how the structure of this thesis corresponds to the research steps that are outlined in the previous section.

Chapter 2 gives an overview of related work and the state-of-the-art in ontology matching algorithms and mapping tools. This provides an overview of existing solutions for automating the mapping between ontologies.

Chapter 3 discusses the iterative approach with which the mapping tool has been developed. It outlines the basic requirements in the form of user stories.

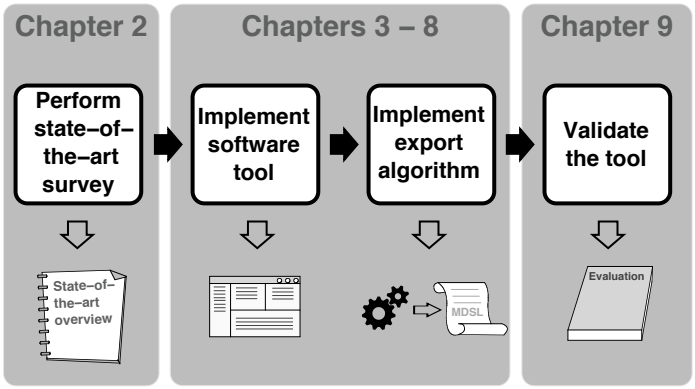


Figure 1.5: Structure of this thesis

Chapters 4–7 present the development iterations that led to the implementation of the mapping tool, including a description of the design and implementation details.

Chapter 8 presents the final resulting architecture and design of the tool. The chapter shows the result of the combined work described in the previous chapters.

Chapter 9 presents the results of a case study of the tool. This chapter shows how the tool is used in the process of solving the integration problem and presents an evaluation of the tool.

Chapter 10 presents a discussion on this research and the implemented mapping tool.

Chapter 11 presents the conclusions of this research and provides pointers for future research.

Chapter 2

State-of-the-art in ontology mapping

This chapter presents an overview of existing work on the topic of ontology matching. The types of work that are considered can roughly be divided into two separate categories: matching algorithms and mapping tools. It is convenient to make this distinction, because both types of work have a different influence on this research. Studying this work provides a good overview on the possibilities and challenges of ontology matching.

In the literature study [23], this state-of-the-art overview has been worked out. This chapter presents a summary and conclusion of the study, for the sake of clarity and completeness.

2.1 Ontology mapping algorithms

In the literature study, several ontology mapping algorithms have been reviewed. These are algorithms that have been engineered with the purpose of creating mappings between two ontologies. So when two ontologies are similar, but not identical (for example, they describe the same domain), a matching algorithm can try and generate mappings that allow instances of one ontology to be transformed into instances of the other ontology.

The algorithms that have been reviewed are S-Match [9], Anchor-PROMPT [2, 14], MAFRA [13], NOM [7], and QOM [6]. For each of these algorithms, the following characteristics have been determined and evaluated. First, the *kind of matching* that is performed by the algorithm is described. Basically, an algorithm can be classified as *heuristic* or *formal* and as *implicit* or *explicit* [21]. In most of the mapping approaches described here, the input models are seen as graph structures. Heuristic techniques try to guess relations which may hold between the graphs labels or graph structures. Formal techniques make use of semantics that are inherent to the model. Implicit techniques are syntax driven: the algorithm makes a comparison based on strings, data types, or soundex of

schema or ontology elements. Explicit techniques rely on tools such as thesauri or ontologies that explicitly codify semantic information.

Next, the *source formalism* for the algorithms is discussed. It is the 'input' for the algorithm. Some algorithms will accept any XML schema, others may only work with some proprietary description of the schema or ontology.

The third characteristic is the *mapping formalism*. This is the 'output' of the algorithm. The mapping formalism specifies what the result of the algorithm will be. Some may output their results as XSLT, XQuery, or other transformations. Others may use the input formalism as output formalism, or they may use some proprietary format or description for describing the mappings.

Another characteristic of interest is the *implementation*. This is a rather straightforward point: for each algorithm, the question is answered whether there is an implementation of this algorithm available.

Finally, it is determined whether a *specification* of the algorithm is available. Of course, some level of specification is available in the articles or other sources that describe the algorithm, but how deep does this go? Some articles may only describe the basic functioning of the algorithm, while others may go as far as listing the entire source code or pseudo code for the algorithm.

As mentioned before, the literature research report [23] provides a more detailed discussion on each of the evaluated algorithms. A summary of this research is given in table 2.2.

	S-Match	Anchor-PROMPT	MAFRA	NOM / QOM
Kind of matching	heuristic element-level explicit & implicit	heuristic structure-level	heuristic explicit	mostly heuristic
Source formalism	XML Schema	RDF	RDF(S)	RDF
Mapping formalism	?		DAML+OIL	?
Implementation	Private	PROMPT plug-in for Protégé	In KAON	Private
Specification	Pseudo-code	Description	Description	Description

Table 2.2: Comparison of ontology mapping algorithms

2.2 Ontology mapping tools

Although studying algorithms and papers describing possible solutions provides a good feel for what is current in ontology mapping research, studying the available tools gives a more concrete overview of the available methods and

techniques that can be used in practice. Tools that exist today usually do not include an intelligent matching algorithm. Rather, they provide a rich graphical user interface that makes it as easy as possible for a user to create mappings from one ontology to the other.

The tools that have been evaluated in the literature research report are the PROMPT plug-in for Protégé [2], TopBraid Composer[1], WSMO Studio[2], and the NeOn Toolkit[3] . In order to evaluate these applications, a list of characteristics has been created, just like with the evaluation of the algorithms. For the evaluation of the tools, the first characteristic was to see whether it is a *stand-alone* application, or a plug-in to an existing program. Knowing if a tool is stand-alone or not is an interesting characteristic, because it provides you with a better feel of what the tool looks like. This in turn makes it easier to decide whether the tool can be used as inspiration for developing the ontology mapping tool, with regard to the user interface, the benefits or drawbacks of either a stand-alone or plug-in based tool, etc. Of special interest are tools that are implemented as Eclipse plug-ins. The ontology mapping tool will eventually be integrated with other tools that have been developed by the Telematica Instituut. All these existing tools have been implemented as Eclipse plug-ins, and the ontology mapping tool will also be implemented as an Eclipse plug-in. Therefore, knowing whether the tool or algorithm that is being evaluated is an Eclipse plug-in might help in the decision of using it when implementing the mapping tool.

Next, the application is checked for *platform dependencies*. Some tools may run on Microsoft Windows only, others may need a Java Virtual Machine, etc. If the tool is implemented as a plug-in for another application, this is also a required dependency. Again, knowing this characteristic helps to get a better overall picture of the tool. If the tool provides code or functionality that could be copied one-on-one to the ontology mapping tool, knowing the dependencies is crucial, since these dependencies may then also be required for the ontology mapping tool.

Next, it is worth checking out if the tool operates in *interactive or batch* mode. Some tools may work in a batch mode: one needs to specify the inputs to map and the tool will generate mappings in a burst. Other tools may use a more interactive approach, getting intermediate feedback or additional input from the user. Depending on the mapping algorithm that the ontology mapping tool will perform, either of these modes of operation — and their implementation in various tools — may be useful. Having a number of implemented examples at hand while designing the tool may offer some inspiration.

The next characteristic discusses the *interface* of the tool. Some tools may have only a user interface, be this in the form of a command-line input / output system, a full-fledged GUI, or something similar. Others may offer an API, offer their services as a Web service, as some other TCP/IP-based service, or as something entirely different. If the tool is suitable for use in / with the ontology mapping tool, knowing the interface is a necessary requirement to incorporate

[1] http://www.topbraidcomposer.com
[2] http://www.wsmostudio.org
[3] http://www.neon-toolkit.org

it into the tool. If this is the case, the specification of the interface (such as API specification, WSDL description, etc.) has to be available.

Finally, the *availability* of the tool is determined. This answers some practical questions about the licenses for the tool, whether or not it is open source, the availability of binaries, installers, source code, etc. Knowing whether the tool is open source or not may be beneficial. If it is, parts of the source code may be re-used in the ontology mapping tool, if the tool provides some interesting functionality. Having the binary available makes it much easier to compare the tool to others, or, if so desired, include it in or use it in combination with the ontology mapping tool.

The literature research report gives a complete evaluation of the aforementioned tools, based on these characteristics. A summary of this evaluation is given in table 2.4.

	PROMPT plug-in	TopBraid Composer	WSMO Studio	NeOn Toolkit
Stand-alone	No	Yes, or Eclipse plug-in	Yes, or Eclipse plug-in	Yes
Platform dependencies	Protégé JVM	Windows Mac OS X	Windows Mac OS X Linux	Windows Mac OS X Linux
Interactive or batch	Interactive	N/A	N/A	Yes
Interface	GUI	GUI	GUI	GUI
Binary available	Yes	Yes	Yes	Yes
Open source	Yes	No	Yes	Yes
Licence		Commercial	GPL	GPL

Table 2.4: Comparison of ontology mapping tools

Chapter 3

Development Iterations

Now that the motivation and background for developing the ontology mapping tool is clear, it is time to formulate concrete requirements for the tool, and then to design and implement it. Numerous approaches and methodologies exist for this, ranging from very rigid, sequential processes such as the waterfall model or formal methods, to very flexible and iterative processes, such as the various agile methodologies.

Each of these approaches has its strengths and weaknesses, and the characteristics of the project determine which approach is best suitable. For the development of the ontology mapping tool within the scope of this Masters assignment, the following characteristics are considered:

1. **The exact requirements are subject to change.**
   There are several reasons for this. First, the implementation of certain requirements may lead to new insights and generate new ideas for the application, that were not considered before. Second, the exact capabilities and limitations of the underlying software platform (the Eclipse platform, the Eclipse Modeling Framework, and possibly other resources) with regard to handling ontologies and in particular the OWL language are not very well known in advance. This might cause the requirements to change a little for practical reasons. Third, others within the Telematica Instituut are working on different but related aspects of the SWS Challenge. Their research and findings may also lead to new insights, that have an impact on the requirements of this mapping tool.
2. **The development time required is hard to determine up front.**
   As stated before, it is not very clear in advance what the capabilities and limitations of Eclipse, EMF, and possibly other libraries are with regard to dealing with ontologies and OWL. Also, having to learn how to develop Eclipse plug-in applications and how to work with the EMF framework takes some time that is hard to estimate. All these factors make it very hard to come up with an accurate project planning.

These characteristics suggest that a rigid, sequential approach might not be very suitable, since a lot of factors are subject to change, which is hard to incorporate

in such methodologies. Therefore, an iterative approach is adopted. In each iteration, a selected subset of the requirements is chosen to be implemented, and a working, running implementation of this is delivered.

Starting the iterative development

Before the first iteration, a meeting was held to determine the initial set of *user stories*. This is a term that is used in agile practices such as eXtreme Programming (XP). It describes one particular thing that the application can do for a user. They are in the format of about three sentences of natural text. It is similar to a usage scenario, but without regard for user interface specific details. In some cases, there may be the need for more specifics than what can be captured by three sentences of natural text. If this is so, these details are provided through interviews with the 'customer' (in this case, the involved project members at the Telematica Instituut) and recorded as notes or annotations to the user story.

This first set of user stories is by no means definitive, but it serves as a starting point for the first iteration. From this initial set of user stories, a subset is selected for implementation in the first iteration. After this first iteration, another subset is chosen for the subsequent iteration, but in the mean time, stories might have been added, altered or removed altogether.

Because of this iterative approach, the concrete formulation of requirements and the prioritizing and planning of them, can be distributed over time. This makes it very much suitable for changing requirements and an unsure development speed.

The set of implemented user stories Although, in true XP projects, user stories are almost never put into formal documentation, they are nevertheless listed in the box below for the sake of illustrating the development process. This is the set of user stories that have eventually been implemented.

Implemented user stories

Visualizing an ontology

The tool should be able to visually display an existing ontology, that is stored as an EMF representation of an OWL ontology.

Use visual tree-representation of EMF editor

Test/research what OWL editor to use:

- EMF editor based on OWL Ecore model
- or EODM OWL editor

Load and display two ontologies

The tool should be able to load two ontologies from file (see story Visualizing an ontology) and display both of them visually in the same Eclipse perspective.

Indicate relations between ontology-entities

The user should be able to indicate that an entity from the first input ontology is related to an entity from the second input ontology. The user can specify as much of these 1-to-1 relations as he wishes.

Specify relations

The user should be able to give a more specific description of a previously indicated relation between two entities. (For example, specifying that a relation is an equivalence relation, or disjoint, etc.).

Edit relations

Once a user has created relations, he should be able to edit these. This means that the user must be able to change either the 'source' entity, the 'target' entity or the type of relation.

Save the mappings

The user must be able to save the mappings in a file, such that the relations that the user specified are stored an can be re-opened at a later time.

Opening a saved mappings-file

A user must be able to open a previously saved mappings-file. The appropriate ontologies and specified mappings are subsequently presented to the user.

Automatically inferring mappings

Based on the two input ontologies, the tool must be able to automatically infer mappings using some algorithm. These must then be visualized to and be editable by the user.

Ideal situation: based on one proposed mapping, which is either accepted or rejected by the user, other mapping proposals are calculated and suggested.

Constraints

There are a number of constraints and assumptions that have to be considered. These are the initial constraints that have been documented:

A reasoner for EMF representations of OWL models is available with the Eclipse platform or one of its plug-ins.

The application is to be build as a plug-in to the Eclipse platform.

The tool should deal with the input (and possibly the output) in form of EMF model repositories, in order to tie in with other tools that are developed in the context of developing a mediator for the SWS Challenge.

More of a 'hint' than a constraint: use as much existing plug-ins as possible to create the functionality for the tool

The input for the tool is available in the form of EMF representations of OWL models.

The following sections describe the iterations that make up the development process. Each section describes the following information about an iteration:

Goals Describes the specific goals of the iteration. What should have been accomplished when this iteration has been completed?

User stories This describes the user stories that are implemented in that specific iteration, which make sure that the specified goals are reached.

Design This section describes the design of the application for this specific iteration.

Solutions and decisions This describes the problems that were encountered while implementing the user stories of this iterations, and the relevant solutions and decisions that have been made in order to solve these problems.

Evaluation Finally, the iteration is evaluated. This section describes whether the stated goals have been reached and evaluates how this was (or was not) accomplished.

Chapter 4

Iteration 1
Starting the project

4.1 Goals

The main goal of this iteration is to provide a starting point for the rest of the development of the application. The iteration itself only delivers the results of one user story, but it provides the opportunity to setup a work environment and to get familiar with a number of tools and software frameworks that are needed throughout the implementation process.

After implementing the user story in this iteration, code is generated that can serve as a basis for further development of the ontology mapping tool. This code allows for working with OWL models in the context of Java code and the Eclipse environment.

4.2 User stories

Visualizing an ontology

The first user story that was chosen to be implemented is described as follows:

> The tool should be able to visually display an existing ontology, that is stored as an EMF representation of an OWL ontology.

When discussing this user story, a number of possible approaches and implementation details have been determined. It was quickly established that a visual tree-representation was to be used for displaying an ontology. The tree representation is an intuitive way to visualize a hierarchy of elements. In addition, the EMF (see section 4.3 for more information on EMF) can be used to automatically generate such a tree viewer.

Implementing this user story consists of a single step: using the capabilities of EMF to automatically generate the executable code for displaying an OWL ontology. The resulting product is a fully functional, runnable Eclipse plug-in that is capable of not only displaying, but also editing OWL files. This is all displayed as a tree view, with menu bar controls and context-menus that offer the means for editing the file. See also Figure 4.1.

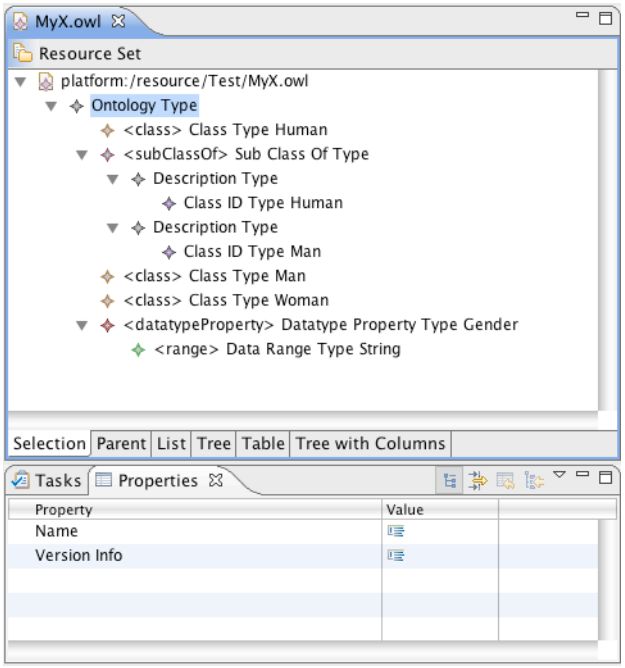


Figure 4.1: The OWL editor that is generated using the EMF framework.

There are several reasons for using EMF for this task. The framework offers all the facilities needed to manipulate models, such as OWL models, using Java code. Based on a single model specification, the framework can generate code to read, manipulate, create, and store instances of such a model. This means that is not necessary to deal with the syntax of OWL, or to handle and write text-based XML files. Rather, the only thing necessary is to provide EMF with an OWL model specification, and then only short Java code is required to manipulate the model.

Other frameworks and libraries exist that offer similar functionality. One such example is Jena[1], which is also used in later iterations. Even so, EMF is preferred over these alternatives. The reason for this is mostly to be compatible with other applications that have been (or will be) developed for the SWS Challenge. Eventually, all these tools may get integrated into one single toolkit. This integration is easiest when all individual tools are written for the same platform (Eclipse), and use the same libraries and frameworks (EMF).

[1] http://jena.sourceforge.net

4.3 Design

In subsequent iterations, it is very interesting to see how the design of the application evolves with the addition of more and more functionality. However, in this iteration a description of the design is not very interesting. There are two reasons for this. First, the design is an automatically generated one, so nothing can be said about design decisions, use of software patterns, etc. But second, most importantly, most of the code that was created will be discarded in later iterations.

To understand the design of the system, and see which generated code is useful and which can be discarded, let's first take a look at the EMF framework and its function.

Making use of the Eclipse Modeling Framework (EMF)

The EMF is a set of plug-ins for the Eclipse IDE. It provides a number of tools for modeling and automatic generation of code based on a structured data model. All that EMF needs is a specification of the model. The code that is generated are Java classes for the model, adapter classes that allow for easy editing and viewing of the model, and a basic editor for creating and manipulating documents that conform to this model.

The specification of the model that EMF requires, can be in various forms. It can be annotated Java interfaces, XML Schema, UML Class Diagrams (support for Rational Rose is built in), or XMI documents. Using this model specification as input, EMF generates two documents: an Ecore model and a generator model. Ecore is the meta-modeling language of EMF. It is a XMI document, that is used to describe application data models. The generator model is used to specify information that is needed for code generation, but that is not included in the Ecore model, such as: where should the code be generated, what is the plug-in ID for the generated application, etc.

Using the EMF generator and a sub-part of EMF, the EMF.Edit Framework, we can now generate the Java code for creating, manipulating, and viewing documents that conform to the specified model. The framework generates a number of items, among which:

Java interfaces For each class in the model, a Java interface is generated. This interface can be extended if needed, by manually changing or adding new methods.

Java classes For each class in the model, the EMF generator also creates a class that implements the corresponding interface. These implementation classes can also be changed after they have been generated.

A factory EMF generates a factory interface and implementation. This factory is used to create instances of model classes.

A package EMF generates package interface and implementation. This package provides some static constants and methods for accessing the model's metadata.

ItemProviderAdapterFactory This class is used to create the item providers that are used to work with the model code.

ItemProviders For each class in the model, an ItemProvider is generated. These ItemProviders provide all methods that are needed to support the standard viewers, commands, etc.

Editor An standard editor is created that can be run as an Eclipse plug-in.

ModelWizard Using this ModelWizard, new files of the model's type can be created. They can then also be edited using the generated Editor.

For a more in-depth overview of the framework, see the The Eclipse Modeling Framework (EMF) Overview [1] in the Eclipse documentation.

4.4 Solutions and decisions

EMF was used to implement (or actually, auto-generate) the code for this iteration. The generic editor that is generated by EMF, based on a model of OWL, is in fact an implementation that conforms to the user story. The following outlines the steps that were taken to obtain this editor.

Find an OWL specification

EMF can generate model code and the editor, based on a specification of the model. This means that we need such a model specification of the OWL language. This specification is available from the W3C website [11]. It is an XML Schema definition (XSD) file containing the specification for OWL DL.

A note on the OWL specification

For this iteration, the OWL DL version of OWL 1 is used. All models, and of course the java code that is generated from that, are based on this specification. However, it might well be that in future iterations OWL 1.1 is used instead of OWL 1. One of the main reasons for this is that OWL 1.1 has better 'built-in' support for specifying mappings or relations between classes. That means that

the mappings that the ontology mapping tool produces may be stored in OWL itself, rather than in some other format.

However, since this iteration is only about displaying an OWL ontology and compatibility or persistency issues are not yet considered at this point, it suffices to simply choose OWL 1 DL. The reason for not immediately choosing OWL 1.1 anyway, is that at the time of implementing this iteration, it was still unclear whether there would be enough tool support for OWL 1.1. In particular, a reasoner needs to be available that supports it, since the ontology mapping tool will very likely make use of such a reasoner.

Create the models for the OWL specification

Using the New EMF Project Wizard, a new Eclipse project is created, based on the OWL DL specification. After finishing the wizard, the Ecore model and the generator model have automatically been created.

Generate the code

Using the generated models, EMF can now generate all the model and editor code. After this step, the editor is a separate project in Eclipse and can be run as an Eclipse plug-in. This is, in fact, the implementation for the first user story.

4.5 Evaluation

The main goal of this iteration was to provide a starting point for the development of the rest of the application. This has succeeded, in that the process of creating this iteration was a good way to get familiarized with the appropriate tools and technologies. The basis — in terms of gained knowledge and implemented code — is now there to continue development on the mapping tool.

The quality of the generated editor, however, is not as sophisticated as most OWL editors that are used by other tools. The three main different entities of an OWL model — classes, properties and individuals — are usually represented in three different views. See for example the screenshot of the Protégé tool in Figure 4.2. The generated editor does not do this, it lists them all in the same single tree view.

Figure 4.2: Three different views in Protégé, one for classes, one for properties, and one for individuals

The shortcomings of this generated editor will be solved in later iterations. The user interface of this editor will be discarded in iteration 2 in favor of a GUI that can display two ontologies at the same time. Dividing the tree viewer into three separate views, for the three different types of entities, is therefore not relevant for this iteration.

Another limitation is the use of a tree for representing ontologies. In an OWL ontology, the classes are ordered hierarchically, as are the properties. All classes are a subclass of the OWL class Thing, and classes can be arranged in a subclass / superclass hierarchical structure. The same is true for properties. But when displaying the relations between classes, as defined by the properties, a tree is not the best way to do this, since the resulting model is not necessarily a tree. This is illustrated by the Wine ontology[2] as explained in section 5.4.2. Nevertheless, a tree representation is chosen for the implementation of the mapping editor. A compelling reason for this is that the EMF framework uses trees to visually represent models. By using the EMF framework, you get the visual tree representation of the model 'for free'. The drawback is, as explained, that this tree does not have the expressive power to represent structures in an OWL ontology that are not hierarchical. However, since the mapping editor is concerned with creating mappings between elements in an ontology and not with visualizing an ontology as concisely and effectively as possible, this drawback does not outweigh the benefits of using the EMF framework.

[2] http://www.w3.org/TR/2004/REC-owl-guide-20040210/wine.rdf

Chapter 5

Iteration 2
Relate two ontologies

Now that iteration 1 has provided some insight into the technology, and some code to work with, it is now possible to start building more functionality into the application. This chapter describes the second iteration, which focuses on relating two ontologies using the mapping tool.

5.1 Goals

This iterations has two main goals. The first is to extend the results of the previous iteration by making it possible to display two ontologies instead of one. The obvious reason for this is that the tool should map between two ontologies, so having those two displayed on screen instead of just one at the time, makes it much easier to create mappings.

The second goal is to add mapping functionality. While the goal here is mostly to provide very basic mapping functionality, this iteration does provide a solid basis for further extending the mapping capabilities of the application.

5.2 User stories

In order to reach the goals of this iteration, the following two user stories will be implemented.

Load and display two ontologies

This user story builds on the work that has been performed for the first user story, visualizing an ontology. This user story allows the application to load and display *two* ontologies instead of visualizing just one. It is described as follows:

> The tool should be able to load two ontologies from file (see story Visualizing an ontology) and display both of them visually in the same Eclipse perspective.

The ontologies are again visualized as tree diagrams, in the same way as for the story in the first iteration. The tree controls should be positioned next to each other, so that controls for creating mappings can be placed in between or below them. The implementation of this user interface is displayed in figure 5.1.

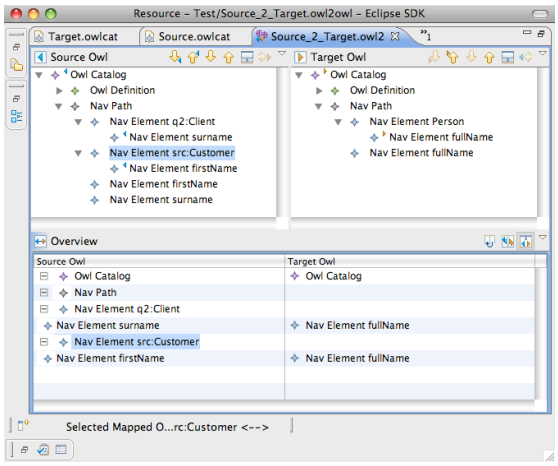


Figure 5.1: The mapping editor

Indicate relations between ontology-entities

This is the first in a series of user stories that add the actual mapping functionality to the application. This particular story starts the development of these functions by allowing the user to indicate that relations exist between entities of the two ontologies. The story is formulated as follows:

> The user should be able to indicate that an entity from the first input ontology is related to an entity from the second input ontology. The user can specify as much of these 1-to-1 relations as he wishes.

As is obvious from the story description, this functionality is minimal. Only indicating a relation is possible, but what kind of relation this is, cannot yet be specified. The next iteration will extend the capabilities of the application with more powerful mapping capabilities.

5.3 Design

The user stories that are described in the previous section dictate a number of demands on the application: there should be two tree viewers for the ontologies, an editor for creating mappings, code that can relate entities between the

ontologies and mappings, etc. Creating this from scratch as an editor for the Eclipse platform would be quite a large task. However, there is a component in the EMF framework that can provide most of this functionality. This component is in the package `org.eclipse.emf.mapping.ecore2ecore`. It is a basic mapping editor that allows for the creation of mappings between two ecore models. Basic as it may be, it has all the features required for this iteration, except that is uses ecore models instead of OWL files as its input.

In order to display OWL files using this editor, they first need to be converted to ecore models. The ecore model that was developed for this purpose is described in section 5.3.1.

5.3.1 The OwlCat file format

If the existing ecore2ecore mapping editor is to be used, the OWL files that are used as the input for the mapping editor, have to be converted to an ecore model. A metamodel for these ecore models has been created called OwlCat (originally short for OWL Catalog). Figure 5.2 gives an overview of this metamodel.

The top-most class in this model is the `OwlCatalog` class. From this point in the class hierarchy on, the OwlCat metamodel is basically split into two parts. One part deals with representing the original OWL file in terms of EMF classes, attributes, and relations. This means that OWL Classes as defined in one of the original OWL files are copied one-on-one to an OwlClass in the OwlCat file. Likewise, object and data properties in the original OWL file are also copied to `OwlObjectProperty` and `OwlDataProperty` classes, respectively.

The other part of the OwlCat metamodel hierarchy deals with visually representing the OWL file. The Solutions and Decisions section will go into more detail as to why this separation between the definition of the OWL model and its visual representation is necessary.

5.3.2 The conversion process

Using the existing ecore2ecore mapping editor and the OwlCat model that is defined in the previous section, the entire process for displaying the OWL files in the mapping editor can now be described. It is a four-step process. First, each OWL file needs to be read from the file system. These files are the original OWL files, serialized using XML as usual, describing the two input models for the mapping application. Second, the Jena[1] libraries are used to make Java representations of these OWL files. We now have the two complete ontologies in memory as Java objects. Third, two OwlCat models are created. These OwlCat models are ecore models, the format that is required by the editor to display the ontologies. It is easy to create these models in this step, because all information about the OWL models is represented using the Java objects that were generated by Jena. If this had been omitted, all information had to be read and interpreted line by line from the OWL files, which is quite a complicated job. All this is now handled by Jena. The fourth step is to load the newly

[1] http://jena.sourceforge.net/

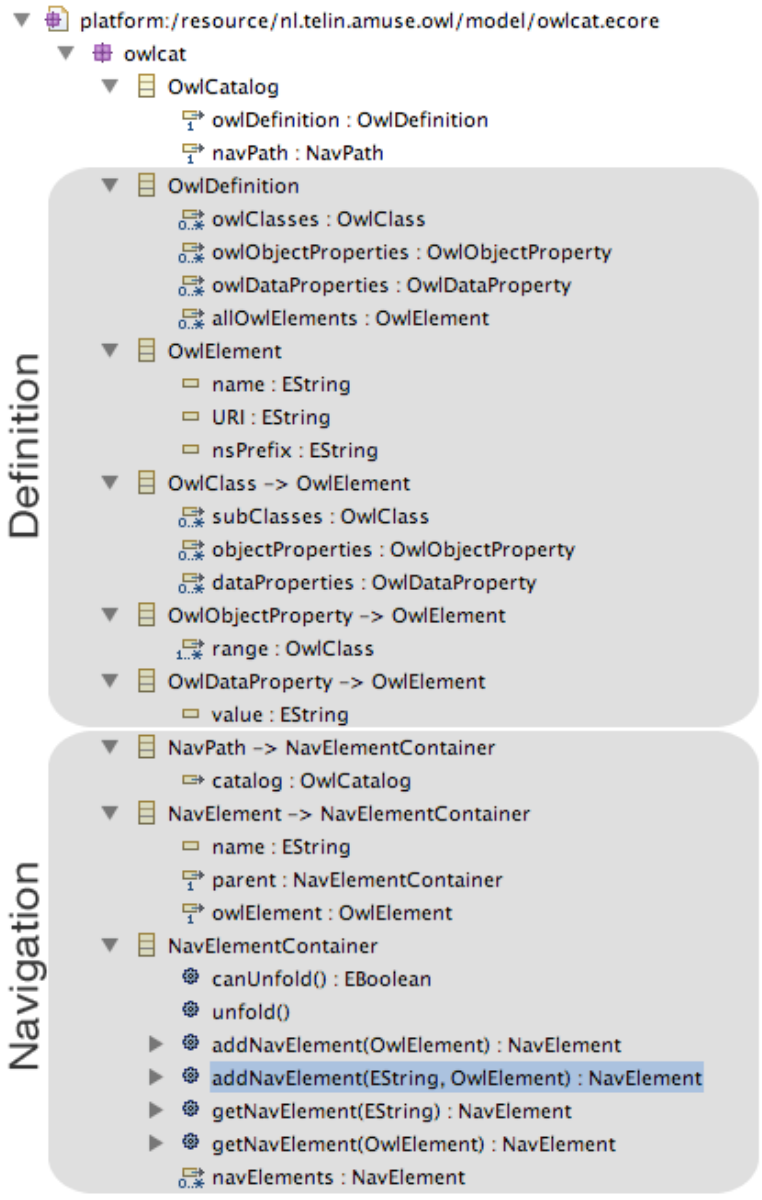


Figure 5.2: The OwlCat meta-model

created OwlCat models into the editor, where they are displayed in the two tree viewers. Figure 5.3 summarizes this process.

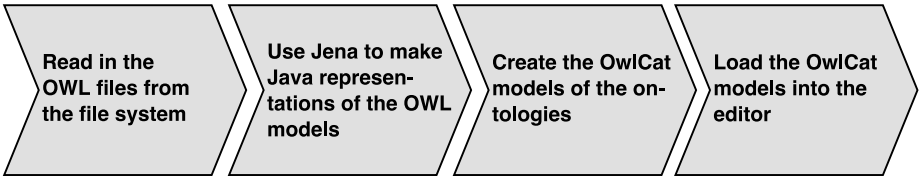


Figure 5.3: The conversion process

After changing the code of this editor so that it can handle OWL files, the code is now structured in several separate projects. These projects are:

`nl.telin.amuse.owl` This project contains all the code to represent OwlCat files. The code is automatically generated by the EMF framework, based on the OwlCat ecore model.

`nl.telin.amuse.owl.editor` This project contains the code for creating an OwlCat editor. The code is automatically generated by the EMF framework based on the OwlCat ecore model.

`nl.telin.amuse.owl.mapping.owl2owl` This is the same kind of project as nl.telin.amuse.owl, it is automatically generated code to deal with Owl2Owl models. It is based on the Owl2Owl ecore model. This Owl2Owl model is an adapted form of the ecore2ecore model of the EMF editor. It specifies how the mappings between the two models are structured. A more detailed description of this mapping model can be found in section 6.3.1.

`nl.telin.amuse.owl.mapping.owl2owl.editor` Similarly, this project contains an editor for Owl2Owl models. It is also based on the Owl2Owl ecore model.

`nl.telin.amuse.owl.owl2emf` This project contains the code that is necessary to convert the original OWL files to the OwlCat EMF models that are the required input for the mapping editor.

`nl.telin.amuse.owl.pelletjena` This project contains the libraries for the Jena and Pellet projects. The Jena libraries are used to convert the OWL files to Java objects. The Pellet project is a reasoner that is used to infer additional information from the ontologies.

Most of the code in the above project is library code, automatically generated code, or part of the existing ecore-to-ecore mapping editor. A number of items are worth mentioning separately though.

The `Owl2EmfConverter` class This Java class is located in the nl.telin.amuse.owl.owl2emf project and is contained in the package `nl.telin.amuse.owl.owl2emf`. This class contains the code that converts the OWL file,

using Jena, to an OwlCat EMF model. When the `convert` method of this class is invoked, the method will first load the OWL file and convert it to an `OntModel`, the root object that is used by Jena to represent ontology models using Java objects. Then, an empty OwlCat model is created using the code that was generated by the EMF framework. Finally, the method will copy all classes, all object properties, and all data properties from the Jena Java objects to the OwlCat model. The OwlCat model is written to the disk so it can later be loaded in to the mapping editor.

The `ConvertToOwlCatActionDelegate` class In this iteration, the conversion process that is described above is not executed automatically when the user tries to load the OWL files into the editor. The user first has to explicitly convert the input OWL files to OwlCat files, and can then load the latter into the editor. For this reason, a context menu option is added to the user interface. When the user invokes the context menu (by default by right-clicking) on an OWL file, a command is available to convert this file to OwlCat. The `ConvertToOwlCatActionDelegate` class contains the code that is executed when this command is invoked.

The `nl.telin.amuse.owl.mapping.owl2owl.test` package This is a package in the `nl.telin.amuse.owl.mapping.owl2owl.editor` project that contains a JUnit[2] test case for the `Owl2EmfConverter` class. The test case makes sure that the functionality of the converter class is as required, and also serves as a 'description in code' of how to use and invoke the converter.

5.4 Solutions and decisions

5.4.1 Alter the editor or convert the input

For the implementation of the user stories in this iteration, the existing editor code from the EMF framework was adopted instead of creating all necessary code from scratch. This makes a more or less complete user interface already available for use, as well as the underlying code to deal with mappings between two models. It does however introduce the need to alter the editor, so that it can deal with OWL files instead of ecore models. In order to accomplish this, two options are possible. The first is to completely rewrite the internal workings of the editor, such that it can work with OWL files directly. The second is to keep the internal workings largely intact, but convert the OWL files to ecore models, and feed those to the editor.

Initially, the first option might seem like the most direct approach. However, rewriting the editor so it can handle OWL files is a very complicated task and would require a large amount of time and coding effort. The second option introduces an extra step to the process, which unnecessarily complicates the process, especially when the user has to be aware of this step and take action (i.e. the user has to give the **convert** command before he can load the OWL files

[2] http://www.junit.org/

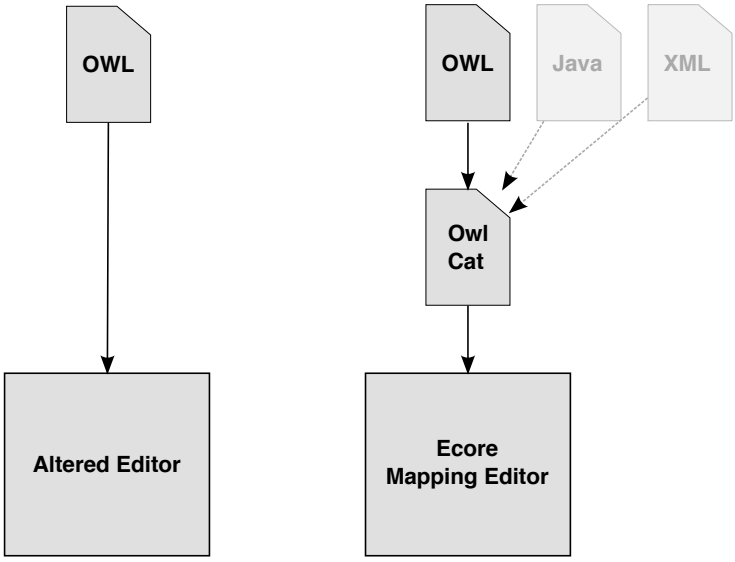


Figure 5.4: Two options for creating the editor

into the editor). Nevertheless, this options has been chosen since the amount of time and coding effort required to implement this is much less compared to the first option. This makes it possible to devote more time to other functionalities of the application.

Another advantage of this second method is that it separates the mapping functionality that is provided by the editor from the details of the input format. This separation is achieved by first converting the input format (OWL) to an intermediate format (OwlCat). The editor can now easily be adapted to map other formats than just OWL files. The only thing that needs to be written is an adapter from the other input format (such as Java classes or some XML schema) to the intermediate OwlCat format.

5.4.2 Avoid infinite loops caused by circular references

The ecore2ecore mapping editor that is used as the basis for this iteration comes with a number of default functions and behavior. One of these behaviors is that it traverses over the input OWL trees and the mappings, when loading a saved mapping file. When the input is a tree, this does not pose a problem. However, since an ontology is not really a tree, but the editor displays it as if it is one, this can cause some problems. Consider for example the Wine ontology (see footnote 2 on page 31), which contains a class **Region**. This class has an object property **adjacentRegion**, who's range in turn is a **Region** class. Such a circular reference cannot be expressed using a tree. The element **Region** would have a child **adjacentRegion**, which would have a child **Region**, and thus this goes on infinitely. Because of this, when the ecore2ecore mapping editor tries to traverse the entire tree, it gets locked in an infinite loop. Figure 5.5 illustrates this problem.

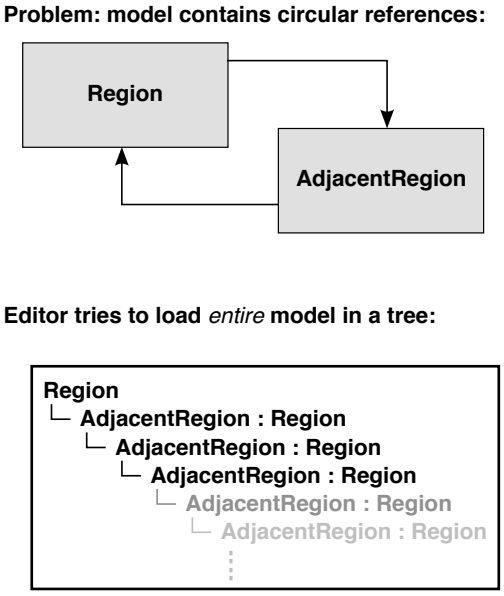


Figure 5.5: The infinite-loop problem of the ecore2ecore mapping editor

To solve this infinite loop problem, two solutions are possible. The first solution is to stop the editor from trying to traverse the entire tree, or to tell it to skip a branch as soon as it detects recursion. The other solution is to make sure that no circular references exist in the input models. Since the code of the ecore2ecore mapping editor is rather complex, the first solution is not an attractive one. Technically it is possible, but it requires a lot of effort to determine exactly how, where, and when the editor executes the tree traversal code. Therefore, the second solution is chosen. This solution is illustrated in Figure 5.6 and explained below.

As mentioned in section 5.3.1, the OwlCat meta-model that is used to describe the input models for the mapping editor contains two main parts: the "definition" part for describing the original OWL input file and the "navigation" part for displaying the ontology. The definition part may contain circular references. Since this part is not used to display the ontology, this poses no problem. The key to the solution is to use elements from the navigation part to describe elements from the definition part, in such a way that this navigation part will *not* contain circular references. This navigation part is then used to generate the tree representation from. The following steps illustrate this procedure:

1. When first loading an OwlCat file, a `NavElement` class is created for each top-level class and property in the "definition" part of the OwlCat model. This `NavElement` class only contains a reference to the `OwlElement` it represents, not to any children that this `OwlElement` might have.

2. When the mapping editor then tries to load the entire model, it sees only the `NavElement` objects. It displays these, resulting in one flat list.

3. When the user wants to go one level deeper, he can expand individual

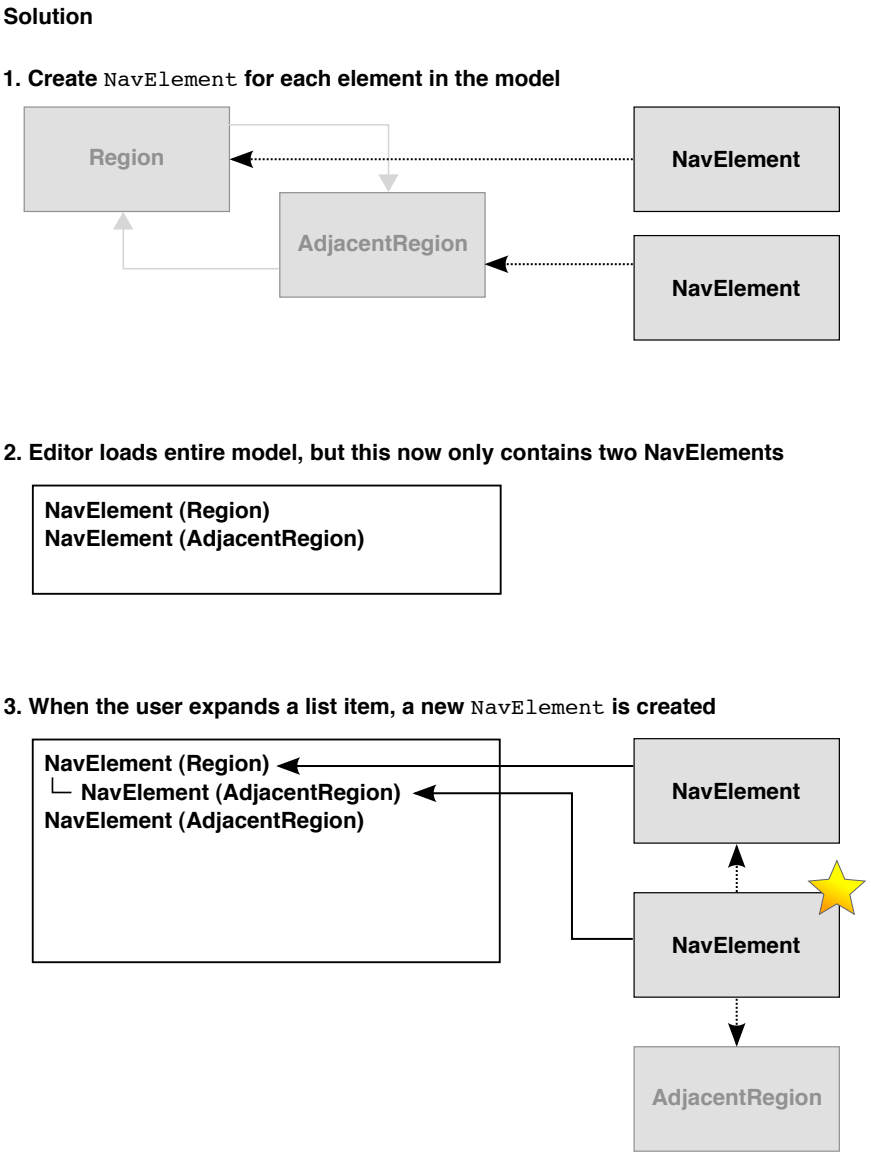


Figure 5.6: The solution to the infinite-loop problem

elements in the tree. For example, he can select a class and expand only that class. At this point, new `NavElements` are created for the subclasses and properties of the selected class. These `NavElement` classes contain references to the `OwlElement` classes they represent, and to their parent `NavElement`.

When the user re-opens the mapping file, the editor wants to traverse the entire tree. However, the tree only goes as deep as the user expanded the elements in the first place. A circular reference is never encountered, and the editor displays the root elements of the ontology, as well as all sub-elements that the user had previously expanded.

5.5 Evaluation

The first of the two goals for this iteration was to make an editor that can display two ontologies. This is accomplished nicely by using the existing EMF editor code and adapting it to the needs for the mapping application. As seen in the screen shot in figure 5.1, the interface supports exactly what the user story describes.

The second goal was to add mapping support to the editor. This has also been accomplished by using the EMF editor, and creating the conversion code so that it can handle OWL files.

Two remarks are worth mentioning at this point. First, in future development or extensions of the mapping application, it is desirable to make the conversion process seamless. It makes for a much more intuitive and friendly user experience if the user can simply select two OWL files and start creating mappings, instead of having to convert these OWL files first. Second, the OwlCat model that is used in the conversion process may need extending. At this point, the model that is used is a very basic one, defining OWL Classes, Object Properties, and Data Properties. However, the OWL language is much richer than this. In a real life situation, this simple model may not be sufficient to accurately model the ontologies and create a mapping between the two systems.

Chapter 6

Iteration 3
Create meaningful mappings

At this point, the mapping editor can load OWL files, convert them to the intermediate OwlCat format, display two ontologies side-by-side and support the creation of relations between two ontology entities. This iteration will refine the mapping capabilities of the editor, such that the relations between two ontology entities can be expressed with much greater and accurate detail.

6.1 Goals

The previous two iterations laid the foundation for the ontology mapping application. But as the name suggests, the main purpose of the application is to enable the creation of meaningful mappings between two ontologies. Based on the now created application foundations, the actual mapping functionality is implemented in this iteration.

The goal is to implement all required features for creating mappings between ontologies. These features include both the user interface and the underlying code. The user interface should allow the user to create mappings and describe these mappings in a meaningful way. The underlying code should be able to represent these mappings in the internal model of the application, so that later iterations can use this for further extension of the functionality, such as persisting the mappings or adding more automation to the mapping process.

6.2 User stories

Specify relations

With the implementation of this story, it is now possible to give a more detailed specification of a mapping relation, rather than just indicating that two items are

related. This is an important step towards creating more meaningful mappings. The description of this user story is the following:

> The user should be able to give a more specific description of a previously indicated relation between two entities.

This 'specific description' consists of a number of items. The user must be able to enter all of these items into specific fields on the user interface. For each relation, the following information must be recorded:

Mapping Name The user must be able to specify a name for this mapping.

Source Element The path that identifies the source element in this mapping relation, which is an OWL Class or Property from one of the two input ontologies. The path is automatically determined by the mapping editor, based on the element that the user has selected in the tree representation of the ontology.

Target Element The path that identifies the target element in this mapping relation, which is an OWL Class or Property from the other input ontology. The path is automatically determined by the mapping editor, based on the element that the user has selected in the tree representation of the ontology.

Relation Function In some situations, some operations needs to be performed in order to map classes or properties. For example, in order to map the property **first name** and **surname** to the property **full name**, one can imagine that a function such as `concatenate` is to be executed for this relation. Using this field, the user can specify the name of the function that is to be executed. This functionality is explained in more detail in section 6.3.1 on the next page.

Function Argument Index In the previous example, the order of the properties is important. The property **surname** is to be append to the property **first name**, not the other way around. The Function Argument Index is used to specify the order in which the properties are processed by the specified Relation Function.

Edit relations

This story makes the application more user friendly. It allows the user to edit the relations that have been specified. This means that when a mapping needs to be altered, the user does not have to delete it and create a new one, but can instead edit the existing one to make it correct. The user story is described as follows:

> Once a user has created relations, he should be able to edit these. This means that the user must be able to change either the 'source' entity, the 'target' entity or the detailed specification of the relation.

6.3 Design

6.3.1 The owl2owl mapping meta-model

The ecore2ecore mapping editor that forms the basis of the owl2owl mapping editor makes use of the EMF framework for both its input and its output. The input is defined by the OwlCat meta-model, and describes the two OWL ontologies that need to be mapped. Similarly, the output is defined by the owl2owl meta-model. This model extends the Mapping meta-model that is used by the ecore2ecore mapping editor. It defines the classes and attributes that are necessary to describe all required mapping information that is described in the user story Specify Relations.

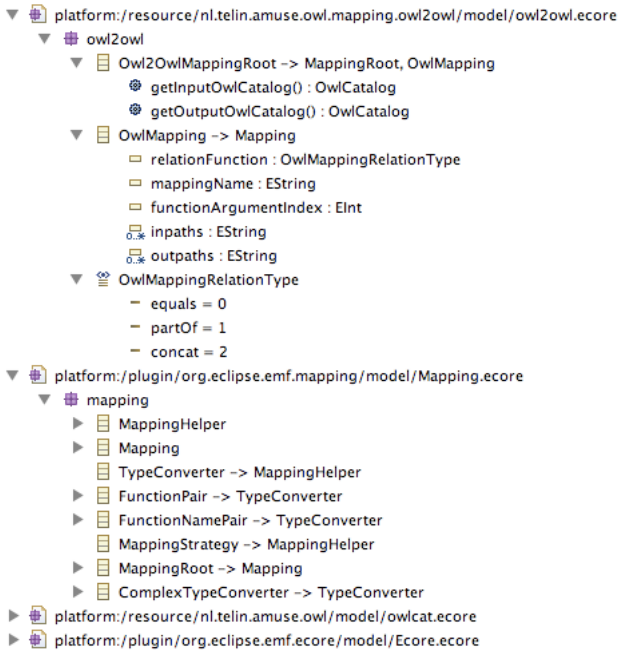


Figure 6.1: The owl2owl mapping model

The owl2owl meta-model is presented in Figure 6.1. The class `OwlMapping` contains all the attributes that are required to store the information that is described in the user story. As described in the user story, the Relation Function field must indicate the function that is to be called when the mappings are processed. This information is indicated by the `relationFunction` attribute. Its value is of type `OwlMappingRelationType`, which is an enumeration that describes which functions are available. At this iteration, it is not yet clear which exact functions will be available. Three functions are likely to be required anyway, so these have been added by default. These are the `equals` function (the two mapped entities are completely equal to each other), the `partOf` function (one entity is a part of the other entity, one can e.g. imagine an entity **street** that is **partOf** an entity **shippingAddress**), and the `concat` function, which concatenates the values of the two mapped entities.

6.3.2 Editing the mapping information

Now that the mapping model is defined, the editor needs an interface that provides the user with the means to supply and edit this information. The EMF framework and the existing ecore2ecore mapping editor provide most of this functionality. The user sees this functionality in the form of the Properties view (Figure 6.2) in Eclipse. This is a view that displays the mapping information of the selected mapping. Within this view, the user can also directly edit the mapping information.

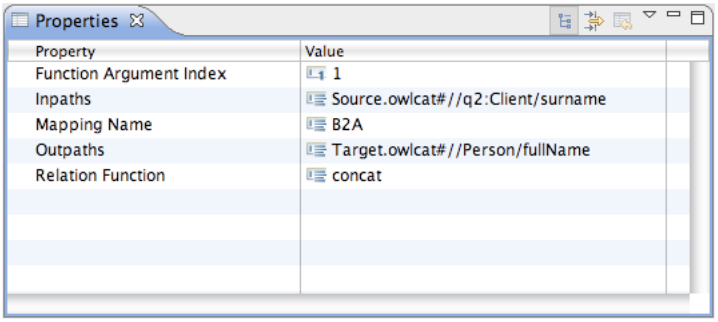


Figure 6.2: The Properties view of the owl2owl mapping editor

A convenient feature of the mapping editor and the EMF framework is that the contents of the Properties view is built directly from the owl2owl mapping model and the user-defined mappings. When the underlying mapping model changes, the EMF framework can update the supporting Java code and the Properties view automatically reflects these changes. During development, this occurred frequently, so not having to manually update the editor each time the model changed saved time and effort. After development, it can be expected that the set of Relation Functions will have to be changed or expanded. This is now easily achieved by changing the `OwlMappingRelationType` enumeration, and having EMF update the Java files, and it will be automatically reflected by the editor.

6.4 Solutions and decisions

6.4.1 Storing the information

In the first user story of this iteration, the user must be able to add information to previously defined mappings. Therefore, the editor needs some mechanism for holding this information. In addition, a user story in the next iteration also requires that this information can be saved to a file. In order to implement this, the following two alternatives were considered. The first requires that a custom mapping file format is created. The second, which is the chosen method, builds on the already existing ecore2ecore mapping meta-model.

Implementing a custom mapping format

In order to store the required mapping information, it is possible to devise a custom mapping file format. An XML Schema that was considered to store the mappings is listed below. The information that is represented by this schema does not exactly match that which is expressed in the user story and represented in the final owl2owl mapping model, because this schema was devised early on in the development process, when the exact details and requirements were still subject to change.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.telin.nl/Mapping"
        elementFormDefault="qualified"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:map="http://www.telin.nl/Mapping">

    <element name="Mapping" type="map:MappingType"></element>

    <complexType name="MappingType">
        <sequence>
            <element name="relation" type="map:RelationType"
                maxOccurs="unbounded" minOccurs="1"></element>
        </sequence>
        <attribute name="name" type="string"></attribute>
    </complexType>

    <complexType name="RelationType">
        <sequence>
            <element name="source" type="map:PathType"
                maxOccurs="unbounded" minOccurs="1"></element>
            <element name="target" type="map:PathType"
                maxOccurs="unbounded" minOccurs="1"></element>
            <element name="expression" type="string" maxOccurs=
                "1" minOccurs="0"></element>
            <element name="condition" type="string" maxOccurs="
                1" minOccurs="0"></element>
        </sequence>
        <attribute name="name" type="string"></attribute>
    </complexType>

    <complexType name="PathType">
        <sequence>
            <element name="instanceType" type="string"
                maxOccurs="unbounded" minOccurs="1"></element>
            <element name="path" type="string" maxOccurs="1"
                minOccurs="1"></element>
        </sequence>
        <attribute name="name" type="string"></attribute>
    </complexType>
</schema>
```

The advantage of using a custom mapping format is that it can be modeled to exactly meet the requirements. If this solution had been chosen, the user-defined mappings would be stored in an XML document that followed this schema.

The downside of this method is that it abandons the functionality and benefits that the ecore2ecore mapping editor and the EMF framework provide. Every basic operation such as saving and loading the mapping file would have to be implemented manually, and then integrated with the editor.

Extending the ecore2ecore mapping meta-model

As explained in the Design section (6.3), the solution that has been implemented uses a newly defined mapping meta-model, the owl2owl model, that is based on the existing ecore2ecore mapping model. Since this is the model that is used by the ecore2ecore mapping editor, using and extending this model has the benefits that loading and saving operations are automatically supported. In addition, this approach has the benefit that updating the editor when the underlying mapping model changes is relatively easy, as previously explained (see section 6.3.2).

The limitation of this approach is that the format in which the mapping information is saved, is determined by the ecore2ecore mapping model. Therefore, it is not easily adjustable to some custom format. This is of no consequence for the stories in this iteration, since the user stories in this iteration are not concerned with saving the information to disk. However, it will have consequences for the implementation of user stories in the next iteration. That is because the goal of the owl2owl mapping editor is to provide information to some other application that will execute the mappings. Not being able to control the exact format in which the mapping information is stored, may cause problems when trying to communicate this information to the other application. This issue and the solution is further addressed in the next chapter, and more specifically discussed in section 7.4.1.

6.4.2 The `OwlMappingRelationType` class

In the defined owl2owl mapping meta-model, the enumeration `OwlMappingRelationType` is used to store the functions that are available for processing the mappings. An alternative implementation would have been to leave the field 'open', i.e. it is simply a String value that the user can supply. This allows for more freedom, since the set of functions can be expanded without having to modify the mapping editor in any way. However, it also means that the user becomes responsible for knowing what functions are available in the first place, and entering the function names manually and without spelling errors.

Because of these drawbacks, the functions names are now explicitly listed in the `OwlMappingRelationType` enumeration. Because of the way the mapping editor and EMF framework work, this automatically means that the user is presented with a selection field, where he only has to select the appropriate function instead of typing it in manually. When a function is added to the set, EMF can make sure that the selection field on the GUI is automatically updated. The only cost of this method is that the model and the derived Java code need to be updated whenever the function set changes.

6.5 Evaluation

With the implementation of these user stories, the owl2owl mapping editor is now able to support the creation of meaningful, fully defined mappings between the elements of the input ontologies. Most of the benefits and drawbacks of several design decisions have already been explained in the previous section. One implementation decision deserves some extra discussion: the way the function names are encoded in the model.

6.5.1 An alternative way to encode the function names

The solution to encode the function names into the mapping model through the `OwlMappingRelationType` enumeration offers a number of benefits, as explained in the previous section. However, it may prove that the inflexibility of this solution is too great a cost. On the other hand, the complete flexibility that is achieved when the user can enter any String value as function name may cause too many problems when the user specifies non-existing functions. A solution might be to populate the function selection field on the GUI automatically, based on the actually available functions, rather than on some values in an enumeration in the model. This requires code that can determine what functions are available, update the model accordingly, and use EMF to also update the supporting Java code of the editor. While this may not be as trivial to implement as a simple model change, which is the current implementation, the improved convenience for the user may warrant the extra development effort needed.

Chapter 7

Iteration 4
Save and export the mappings

The mapping functionality of the owl2owl mapping editor is now in place. The remaining user stories, that are implemented in this iteration, deal with the routine but important functions of saving, opening, and exporting the mappings.

7.1 Goals

This is the final iteration in the development process of the owl2owl mapping editor. Its goals are to persist the information that the user creates using the editor, and to make this information available to other applications. This means that the user must be able to save the mapping information to disk, to open a previously saved file containing mapping information, and to export this information to a format that can be used by other applications. The final three user stories that implement this functionality are explained in the next section.

7.2 User stories

This iteration consists of the following three user stories. Two of the stories deal with saving and re-opening the mapping information that the user creates using the owl2owl mapping editor. The final user stories concern an export function, such that the mapping information can be used by other applications.

Saving the mappings

This user story is formulated in the following way:

> The user must be able to save the mappings in a file, such that the relations that the user specified are stored and can be re-opened at a later time.

There are no additional requirements for the file format that is used. In the early phases of the requirements gathering and implementation, it was considered whether to use a certain, custom defined file format, which would facilitate integration with other applications. This requirements was dropped in favor of an export functionality, as described in the final user story of this iteration. More on this decision is described in section 7.4.1 on page 53.

Opening a saved mappings-file

The corollary to the function to save mapping information is the functionality to open a file containing mapping information. This user story implements this functionality. It is formulated as follows:

> A user must be able to open a previously saved mappings-file. The appropriate ontologies and specified mappings are subsequently presented to the user.

This user story, combined with the previous one, provides the save and open functionality that is common in almost all document-based applications in use today.

Exporting the mappings

The owl2owl mapping editor will be used to solve a particular problem within the SWS Challenge. In order to develop a complete solution to this challenge, a suite of applications will be used, of which the mapping editor is only one part. This final user story makes the information created using the mapping editor available to the other tools in this suite. The description of the user story is:

> The user must be able to export the mapping information, such that it can be used by other applications that are being developed by the A-MUSE project in order to solve the SWS Challenge.

With the implementation of this user story, the mapping editor has all the basic functionality needed in order to be used as part of the suite of applications that try and solve the SWS Challenge.

7.3 Design

7.3.1 File format

In order to save the mapping information to a file, a formalism needs to be created. In the case of the owl2owl mapping editor, this formalism is already determined by the ecore2ecore mapping editor and the EMF framework. The use of these two technologies predetermine the file format that is used to save the mapping information. This file format has the extension `owl2owl`. It is

a serialization of an ecore model. This serialization is in the form of an XMI document, which is the default way for the EMF framework to serialize ecore models. A sample code of such a file is listed here.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <owl2owl:Owl2OwlMappingRoot xmi:version="2.0"
3        xmlns:xmi="http://www.omg.org/XMI"
4        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5        xmlns:owl2owl="http://www.telin.nl/OwlCat/Owl2Owl"
6        topToBottom="true">
7
8     <nested xsi:type="owl2owl:OwlMapping"
9           relationFunction="concat"
10          mappingName="B2A"
11          functionArgumentIndex="1">
12       <inpaths>Source.owlcat#//q2:Client/surname</inpaths>
13       <outpaths>Target.owlcat#//Person/fullName</outpaths>
14    </nested>
15
16    <nested xsi:type="owl2owl:OwlMapping" mappingName="B2A">
17       <inpaths>Source.owlcat#//src:Customer/firstName</inpaths>
18       <outpaths>Target.owlcat#//Person/fullName</outpaths>
19    </nested>
20
21    <inpaths>Source.owlcat#/</inpaths>
22    <outpaths>Target.owlcat#/</outpaths>
23
24 </owl2owl:Owl2OwlMappingRoot>
```

This file format contains all the information that is necessary to store and re-load the mapping information that the user specified. The `inpaths` and `outpaths` element on lines 21 and 22 specify which two ontologies are being mapped. Each mapping is described using a `nested` element, where the enclosed `inpaths` and `outpaths` elements contain an XPath query describing the individual elements in the ontologies that are being mapped. The attributes of the `nested` elements describe the remaining properties of the mappings, such as the mapping name, the relation function, etc. The mapping that is described in the code above would be represented on screen in the editor as in Figure 7.1.

7.3.2 The export function

A number of user stories from this and previous iterations have built on the functionality provided by the ecore2ecore mapping editor and the EMF framework. The last user story however, *Exporting the mappings*, needs to be implemented manually, with little or no support from existing code. Fortunately, the specific requirements for the export functionality are well known. In order to understand these requirements and the resulting design of the export function, it is helpful to know a bit more detail about the export target. The next section will explain how the exported mapping information fits in the entire suite of applications that solve the problem of the Purchase Order Mediation scenario of the SWS Challenge.

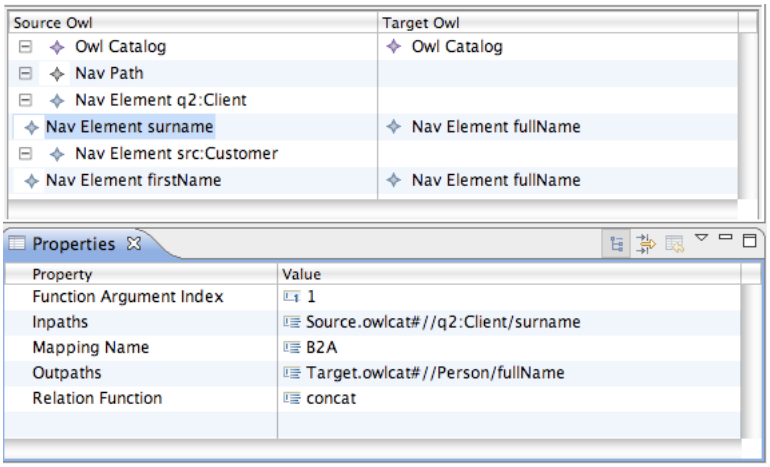


Figure 7.1: The mapping as it shows in the editor

How the exported information is used

In order to solve the integration problem of the Purchase Order Mediation scenario of the SWS Challenge, a mediator has been developed. An overview of the general solution has already been given in section 1.2 and a concise overview of the entire solution is given in [20]. One of the tasks of the mediator is to solve the information mismatch between the two IT systems that need to be connected. The owl2owl mapping editor is used to specify the relations between entities in the two systems, so that data transformation can be derived from these relations. The mediator makes use of a custom notation for describing these data transformations. This notation is a domain specific language (DSL) for specifying data mappings, called the Mapping DSL (MDSL). All data transformations that need to be performed by the mediator have to be specified in this notation. Since the transformations are initially specified using the owl2owl mapping editor, the export function should be able to convert these mappings to an MDSL representation.

In order to work with transformations that are described using MDSL, a number of library classes have been developed that can manipulate the MDSL files. A new package containing these classes has been added to the project. In the design of the mapping editor, the following code has been added to support the export functionality:

Class `ExportToMDSLAction` This newly created class contains the code to execute the export function. It is responsible for reading the mapping information, constructing the MDSL Transformation, and exporting the result to an MDSL file.

Package `nl.telin.swsc.parsers.mdsl` This package contains the code to read and create MDSL files.

7.4 Solutions and decisions

7.4.1 Save vs. Export

As stated before, the owl2owl mapping editor is part of a suite of applications that aims to solve the integration problem in the Purchase Order Mediation scenario of the SWS Challenge. This means that the mapping information that is specified using the editor needs to be available to the other tools in this suite. There are two main ways to accomplish this: one is to devise a file format in which the mapping information is saved, which can be read by the other applications. The other is to leave the file format another issue entirely, and create a separate export functionality to interact with the other applications. This is the approach that is taken with the owl2owl mapping editor.

There are two arguments in favor of an export function as opposed to a custom save file format. The first argument is that *a fully functional save (and re-open) function is already provided by the ecore2ecore mapping editor and the EMF framework*. Creating a custom save file format would mean that all of this functionality would have to be rewritten. Since this functionality is deeply embedded within the ecore2ecore mapping editor and relies heavily on the EMF framework, this would be a difficult and time-consuming task. Implementing an export function in addition to the existing open and save functionality would be just as effective (from the user's point of view) and cost less development time. The second argument is that *the export function can now be exactly tailored to the specific needs of the application that consumes the mapping information*, without being limited by the fact that the exact same file also needs to be used for saving and re-opening the mappings file. The export functionality is realized by making use of existing code that handles MDSL files. This direct integration would not have been possible if a single file-format for saving and exporting the information would have been chosen.

7.5 Evaluation

The goal of this iteration was to add save and export functionality to the owl2owl editor. This will allow it to be effectively used as part of the solution to the SWS Challenge. By implementing the specified user stories, these goals have been reached. Now that this final iteration has been completed, the end result is a fully functional owl2owl mapping editor.

A note about the export function

One decision in this iteration is worth discussing though: the decision to implement an export function that is separate from the save function. This decision leads to both flexibility in one way, and a restriction in another. The flexibility lies in the fact that the export function can be implemented any way that is necessary. This freedom of implementation is due to the fact that it has no impact on the save function at all. The export function can be completely changed

if necessary, without affecting the capability of opening and saving previously defined mapping definitions.

The restriction that was mentioned lies in the fact that the export functionality is very tightly integrated with the MSDL library classes. Instead of simply writing the mapping information to a file using some predefined format (e.g. an XML file, defined by an XML Schema that can be understood by the mediator), the mappings are now used to update or create an MDSL file.

Chapter 8

Resulting architecture

This chapter presents the resulting architecture and design of the mapping editor. While the previous chapters discussed the specific outcome and challenges per iteration, this chapter looks at the resulting product as a whole. The chapter is structured as follows. The first section discusses the architecture of the mapping editor. The second second section goes into more detail, and describes the design of the various components that make up the editor.

8.1 Architecture of the mapping editor

The owl2owl mapping editor is implemented as a plug-in for the Eclipse platform. It is basically an extension of the ecore2ecore mapping editor, a tool that is shipped with the Eclipse Modeling Framework (EMF). In order to provide all required functionality, the editor is extended with modules to convert OWL files to Ecore models, to export created mappings to an MDSL file and to deal with the OwlCat format, the intermediate format that is used by the editor. An overview of this architecture is presented in Figure 8.1.

The following sections describe each of these architectural elements in more detail.

8.1.1 Eclipse

The basis of all development on the owl2owl mapping editor is the Eclipse platform. Eclipse is:[1]

> "... an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle."

[1] http://www.eclipse.org/org/

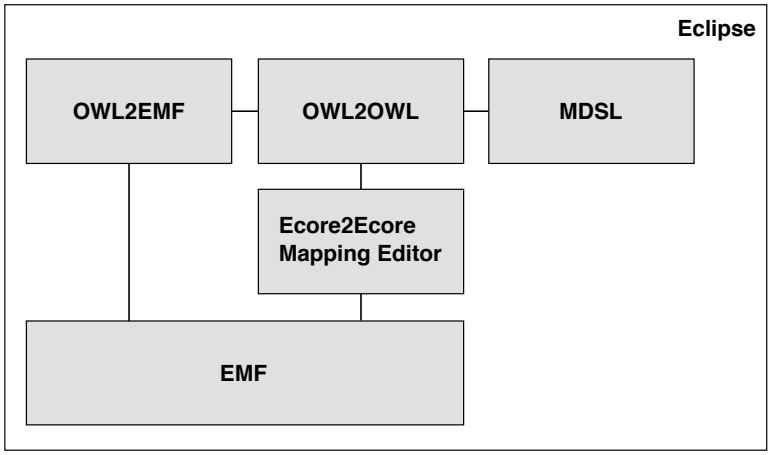


Figure 8.1: Architecture of the mapping editor

The main product, the Eclipse Platform, is a software platform for the development of integrated tools and rich client applications. It is used as the main platform for all tools that are developed for the solution to the Purchase Order Mediation scenario. As such, it is also the platform on which the owl2owl mapping editor is developed. In addition to the main Eclipse Platform, another Eclipse-based project is used for the development of the editor. This is the Eclipse Modeling Framework (EMF).

8.1.2 The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is "a framework and code generation facility for building tools and other applications based on a structured data model"[2]. It is used as the underlying framework for the development of the owl2owl mapping editor. More precisely, both the OwlCat format and the format in which the mappings are expressed are defined as EMF models. The ecore2ecore editor is also heavily based on EMF: the information that can be stored for each mapping is entirely determined by the underlying mapping model. Changing this model automatically changes the information that is stored for each mapping, as well as the user interface for manipulating this information.

8.1.3 Ecore2Ecore mapping editor

The mapping editor that is created as part of this research is based on the ecore2ecore mapping editor. This existing editor is at the heart of the owl2owl mapping editor. The ecore2ecore mapping editor ships with EMF. It is capable of creating mappings between ecore models. The resulting file, which describes these mappings, is itself an ecore model. In order to provide the necessary functionality for the owl2owl mapping editor, the ecore2ecore mapping editor

[2] http://www.eclipse.org/projects/project_summary.php?projectid=modeling.emf

has been extended with several modules that provide additional functionality, or alter the functionality of the editor itself. These modules are described in the following sections.

8.1.4 OWL2OWL

This module extends the ecore2ecore editor, such that is becomes capable of mapping two OwlCat files, rather than ecore files. In addition, it changes the underlying mapping model. The model now contains all information that is required to describe every important aspect of a mapping. The ecore2ecore mapping editor, combined with the extensions in this module, are responsible for the mapping capabilities of the owl2owl editor. All that is required now is a means to provide the editor with the appropriate input, and to generate the correct output. This is accomplished by the next two modules.

8.1.5 OWL2EMF

The previously described modules provide the basic functionality for the mapping editor. This module is responsible for generating the correct input for the editor. Is takes an ontology, described using OWL, and transforms this to the intermediate OwlCat format (which is an ecore model, generated and manipulated by EMF, hence the name "OWL2EMF"). The editor can now load the OwlCat file and display the ontology.

8.1.6 MDSL

This module is responsible for generating the output of the mapping editor. All previously described models can provide the functionality up to the point where all required mappings between the ontology elements are described in the exact detail that is required. All that is required now is to export these mappings to the MDSL format. This is the responsibility of this module. The output it generates is an MDSL script that describes the mappings that have been defined using the owl2owl editor.

8.2 Design of the editor's modules

The previous section identified the different modules that make up the owl2owl mapping editor. This section goes into more detail for most of these modules. It describes the high level design of the modules that have been implemented to create the owl2owl mapping editor. Some modules, such as the Eclipse Platform itself or EMF, will not be described in great detail. For the design of these components, see the official documentation of these projects.

In order to realise the design that is described in this section, the Java code that implements the editor is divided into eight separate projects. Each project contains one or more Java packages, which in turn contain one or more classes.

Sections 8.2.1 through 8.2.3 describe how these projects contribute to the design of the editor's modules, and thus to the entire architecture of the editor. This is summarized in Figure 8.2.

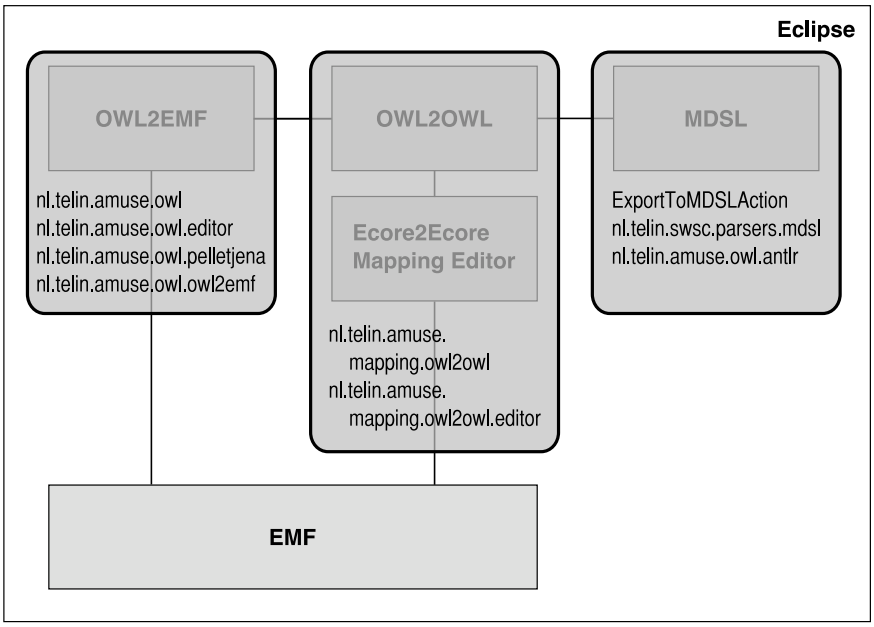


Figure 8.2: Design of the owl2owl mapping editor

8.2.1 OWL2OWL and the ecore2ecore module

As described in section 8.1.4, this module is responsible for giving the editor its mapping capabilities. It is closely tied in with the ecore2ecore module. In fact, the architectural distinction between the two modules is more of a conceptual nature than that it is clearly reflected in the code.

There are several projects that concern the implementation of this module. These project are:

`nl.telin.amuse.mapping.owl2owl`

This project contains the owl2owl mapping meta-model. This is the meta-model that completely describes what a mapping between two ontology entities looks like. A more complete description of this meta-model can be found in 6.3.1 on page 44. Based on this meta-model, EMF can generate the supporting Java code to build and manipulate instances of such models. This code is also contained within this project.

nl.telin.amuse.mapping.owl2owl.editor

This project contains the code to override and extend the functionality of the existing ecore2ecore mapping editor. It also provides the hooks to extend the editor with additional menu commands, operations, etc. A number of classes in the `action` package are worth mentioning separately. The class `ConvertToOwlCatActionDelegate` provides the hook for the action to convert the selected OWL file to an OwlCat file. The actual execution of this conversion is handed of to the OWL2EMF module, which is described in section 8.2.2. The `ExportToMDSLAction` class contains the logic of the MDSL module, which is discussed in section 8.2.3. The class `MapToOwlActionDelegate` provides the necessary logic for loading two OwlCat models and creating a mapping model for these two ontologies. Finally, the `UnfoldNavElement-ActionDelegate` class implements the custom unfold command. This command unfolds the next level in the tree that visualizes an ontology. The reasoning and functionality behind this custom unfold command is described in section 5.4.2.

8.2.2 OWL2EMF

The responsibility of this module is to convert OWL ontologies to the intermediate OwlCat format. The following projects implement this module.

nl.telin.amuse.owl

At the basis of the conversion process to the OwlCat format lies the meta-model that describes the OwlCat format. The meta-model is contained within this project, along with the code that EMF generated to support the creation and editing of OwlCat model instances. An elaborate description of the meta-model itself can be found in section 5.3.1.

nl.telin.amuse.owl.editor

EMF not only generates code to create and manipulate OwlCat models, it also generates an OwlCat model editor. Using this editor, it is possible to create and edit OwlCat models via a Graphical User Interface. While this editor is not used directly by the owl2owl editor, it is a valuable tool during development in order to create custom OwlCat models or evaluate generated models.

nl.telin.amuse.owl.owl2emf

This project contains the algorithm for the actual conversion of an OWL file to an OwlCat model. It consists of one class, `Owl2EmfConverter`. The `convert` method of this class takes as input the path to an OWL file, and returns an `OwlCatalog` class, the root class for an OwlCat model. A high level outline of the algorithm that is used by this class is given in Figure 8.3.

Figure 8.3: The conversion algorithm of the `Owl2EmfConverter` class

If one wants to add support for other input formats besides OWL, this project would be a good candidate for refactoring. By giving it a more appropriate name, the project can become the container for various conversion algorithms. Each specific conversion is handled by a class that follows the same interface: it contains a public `convert` method which takes a file path as input and returns an OwlCat model (in the form of an `OwlCatalog` class). The OWL to OwlCat convertor is currently the only class that has been implemented.

`nl.telin.amuse.owl.pelletjena`

The aforementioned `Owl2EmfConverter` class makes extensive use of two existing Java applications: the Pellet OWL reasoner[3] and the Jena framework[4]. The Jena framework is used to convert the OWL file to a Java model. Using Jena, one can use simple Java method calls to query the model and obtain information about it. Jena in turn uses the Pellet reasoner in order to infer additional information about the model which is not explicitly expressed in the OWL file.

8.2.3 MDSL

This module is used to generate the output of the mapping editor. It converts the user-defined mappings to an MDSL script. Its functionality is spread over three parts, the `ExportToMDSLAction` class that was already briefly mentioned in section 8.2.1, the `nl.telin.swsc.parsers.mdsl` project, which contains the libraries for generating MDSL files, and the `nl.telin.amuse.owl.antlr` project, which contains the ANTLR tool.

The `ExportToMDSLAction` class

This class contains the algorithm to convert the user-defined mappings to MDSL code. It is contained within the `action` package of the nl.telin.amuse.owl.mapping.owl2owl.editor project. Its `run` method is bound to the **Export to MDSL** menu command of the owl2owl mapping editor. When this command is invoked, the algorithm will convert all user-defined mappings to an MDSL script.

[3] http://clarkparsia.com/pellet/
[4] http://jena.sourceforge.net/

nl.telin.swsc.parsers.mdsl

The MDSL script files do not have to be written manually. A collection of library classes has been developed specifically for the case of the Purchase Order Mediation scenario of the Semantic Web Service Challenge. With the aid of these library classes, creating an MDSL script is accomplished by building the appropriate structure using the API of these classes. The library classes are contained within the `nl.telin.swsc.parsers.mdsl` project.

nl.telin.amuse.owl.antlr

The library classes that were mentioned in the previous paragraph, make use of the ANTLR tool [5]. This is a "tool that provides a framework for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in a variety of target languages". The `nl.telin.amuse.owl.antlr` project contains this tool, such that it can be used by the MDSL library files.

[5] http://www.antlr.org

Chapter 9

Case study: Purchase Order Mediation scenario

Now that the tool has been developed, it needs to be put to the test. The tool is part of the suite of applications that try to solve the integration problem in the Purchase Order Mediation scenario of the SWS Challenge. That is exactly what this chapter will show: how this tool can be used in the process of solving the integration problem.

In section 9.1, the specific goals for this case study are discussed first. Then, section 9.2 discusses the method for performing the case study. The results of the case study are presented in section 9.3, and finally, section 9.4 gives an evaluation.

9.1 Goals for the case study

As stated in this chapter's introduction, the aim of the case study is to put the mapping editor through its paces. The motivation for conducting this case study is twofold. First, it will *demonstrate* how the tool can be applied as part of the solution to solve the integration problem of the Purchase Order Mediation scenario. Second, it is desirable to have an evaluation of the tool. The case study provides the means to perform such an evaluation. The goals for the case study follow naturally from this motivation. They are:

1. **Demonstrate how the editor works as part of the entire solution**
   The case study will apply the tool in order to solve an actual mapping problem of the Purchase Order Mediation scenario. This section will document this process, in order to demonstrate how the tool fits in the entire process of solving the integration problem and how the tool itself can be used.

2. **Perform an evaluation of the mapping editor**
   The second goal for conducting the case study is to make an evaluation of

the mapping editor. While using the tool in order to create the demonstration for the previous goal, its strong points, weak points, and opportunities for improvement will be recorded. This forms the basis for an evaluation of the mapping editor. This evaluation can be used later to improve the tool, and with it the entire solution approach to the integration problem.

9.2 Method

Recall from Chapter 1 that the aim of the mapping editor is to help solve the integration problem of the Purchase Order Mediation scenario of the SWS Challenge. In this scenario, the IT systems of two different companies have to be connected. The organizers of the SWS Challenge have provided the description of these two IT systems, along with the WSDL descriptions of the Web Services that expose their functionality.

9.2.1 The process

As described in [20], the process of developing the mediator consists of five steps. The first step is to derive Platform Independent Models (PIMs) from the service descriptions of the two IT systems. These PIMs consist of two parts: an ISDL description of the behavior of the system, and a UML class diagram, based on Java classes, describing the information model of the system. The second step is the semantic enrichment of the PIMs. In this step, information is added to the behavior and data models that is not encoded in the WSDL descriptions, but is important for integration. Such information can e.g. be the order in which operations are to be called. Another possibility in this second step, is to describe the elements of the IT systems using an ontology language such as OWL, and mapping them onto a domain specific ontology such as Universal Data Element Framework[1].

The third step consists of mapping the PIMs. The information models of both systems need to be merged in an intelligent way in order to construct the information model of the mediator. This is the step where the owl2owl mapping editor is used. The first two steps produced OWL files describing the information models of the two systems. Figure 9.1 shows a portion of these models, displaying a number of entities of Blue company on the left hand side, and a number of entities of Moon company on the right hand side. Using the full data models of both the IT systems, a business domain expert can identify the relations between the two models. Within this set of entities in the figure, three mappings have been highlighted. These mappings have been identified by the domain expert (in this case, by reviewing the scenario description given by the SWS Challenge). They will be defined using the owl2owl mapping editor.

In the fourth and fifth steps, the PIM of the mediator is validated and a PSM of the mediator is created. This makes it possible to implement the mediator, such that the necessary transformations of the data that is sent between the two mapped systems can be executed.

[1] http://www.opengroup.org/udefinfo/

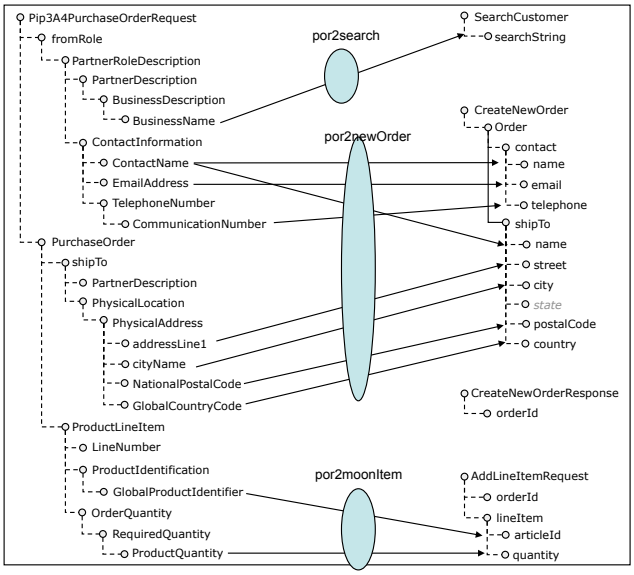


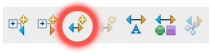
Figure 9.1: Example Mappings

9.2.2 The owl2owl mapping editor

The first step in using the mapping editor, is to load the ontologies. These ontologies should be generated in the semantic enrichment step. However, at the moment the semantic enrichment step generates Java files as output, rather than OWL files, as described in [20]. Therefore, the ontologies describing the two IT systems have been created manually for the purpose of this case study. Recall from the design of the editor, that it is not possible to load OWL files directly, but that they have to be converted to an intermediate file format first (see chapter 5). This is accomplished by right-clicking an OWL file in the editor, and selecting the **Convert to OwlCat** option (Figure 9.2).

When the two ontologies have been converted to OwlCat files, these OwlCat files can be selected for creating mappings. To do this, right-click one of the two files (which will be designated the 'source' ontology), and select the **Map to Owl** function. In the dialog box that has appeared, select the OwlCat file that should be the 'target' ontology (Figure 9.3).

Now that the ontologies are loaded, everything is in place to start defining the mappings between ontologies. To demonstrate this, the `por2newOrder` mapping that is indicated in Figure 9.1 will be defined step by step.



When both ontologies have been loaded, the mappings can be defined by selecting the appropriate elements in the 'source' and 'target' ontology and clicking the Create Mapping button. This will create a new mapping between the selected elements. The user can now enter the details of this mapping

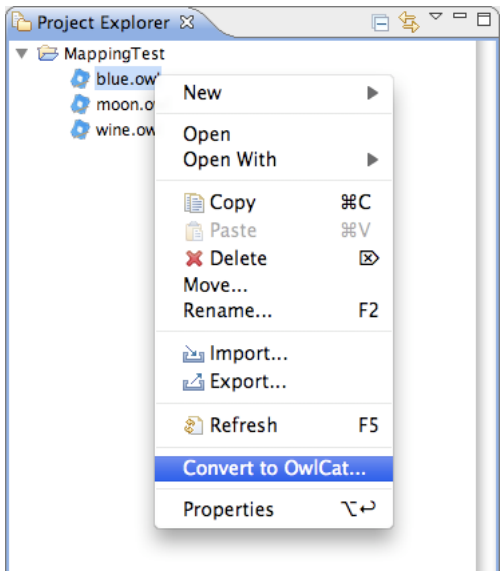
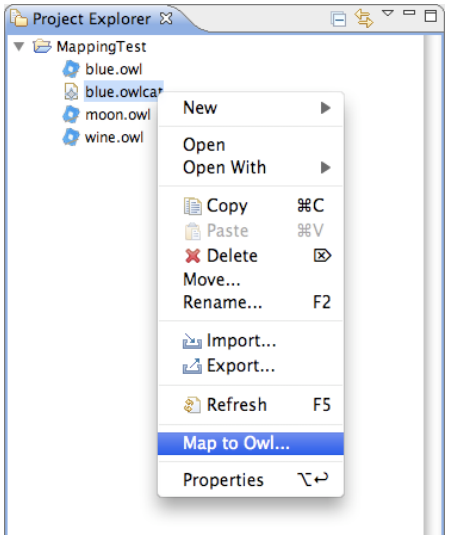
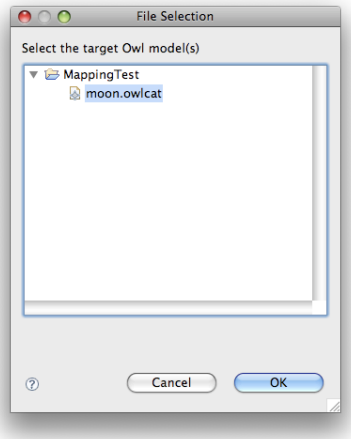


Figure 9.2: Convert the OWL files to OwlCat



(a) The Map to Owl... command



(b) Select the 'target' file

Figure 9.3: Select two OwlCat files to create mappings

in the Properties view, associated with the mapping. Figure 9.4 displays the first relation in the `por2newOrder` mapping. When the user has completely specified this mapping, the Properties view looks like this: the Mapping Name is 'por2newOrder', the relation function is 'equals', the function argument index is not required for the 'equals' function, but it defaults to 0, and the paths of the source and target elements are displayed.

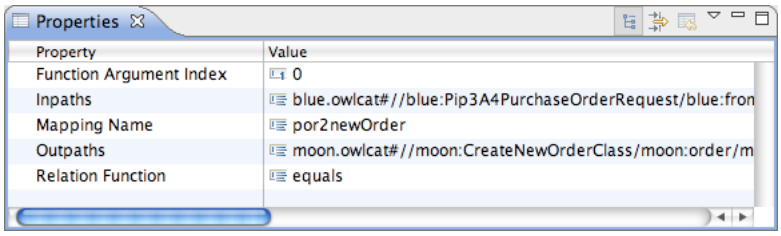


Figure 9.4: The Properties of the first relation in the `por2newOrder` mapping

In the same way, the user specifies all other relations of the `por2newOrder` mapping. When all mappings have been specified, the mapping editor shows this as displayed in Figure 9.5.

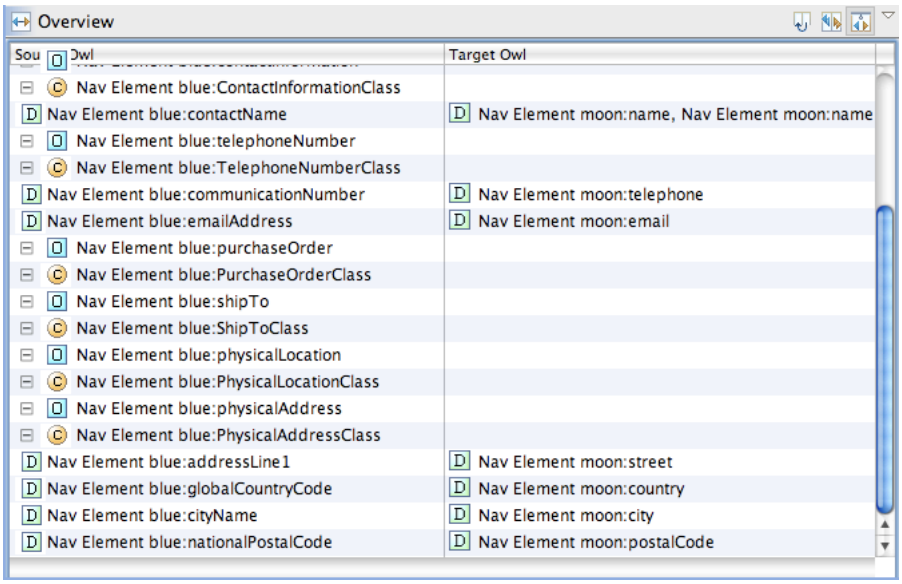


Figure 9.5: All relations in the `por2newOrder` mapping

When all mappings have been specified, there is one final task to be done in the owl2owl mapping editor. This is the task of exporting the mappings to an MDSL file. Recall that this file is used by other applications in order to execute the transformation of messages between the two IT systems. To invoke the export function, the user needs to select the Export to MDSL command from the Owl to Owl Mapping Editor menu (Figure 9.6). This will generate the MDSL code for the defined mappings and output this on the command line. The mapping code for the `por2newOrder` mappings looks like this:

```
1  transformation GeneratedFromOwlCat {
2      namespaces {
3          blue = "http://blue.com/blue.owl";
4          moon = "http://moon.com/moon.owl";
5      }
6      mapping por2newOrder {
7          target moon:CreateNewOrderClass owlcat_z3 {
8              x0 = "CreateNewOrderClass/order/OrderClass/contact/
                   ContactClass/name";
9              x1 = "CreateNewOrderClass/order/OrderClass/shipTo/
                   ShipToClass/name";
10             x2 = "CreateNewOrderClass/order/OrderClass/contact/
                   ContactClass/email";
11             x3 = "CreateNewOrderClass/order/OrderClass/contact/
                   ContactClass/telephone";
12             x4 = "CreateNewOrderClass/order/OrderClass/shipTo/
                   ShipToClass/street";
13             x5 = "CreateNewOrderClass/order/OrderClass/shipTo/
                   ShipToClass/city";
14             x6 = "CreateNewOrderClass/order/OrderClass/shipTo/
                   ShipToClass/postalCode";
15             x7 = "CreateNewOrderClass/order/OrderClass/shipTo/
                   ShipToClass/country";
16         }
17         source blue:Pip3A4PurchaseOrderRequest owlcat_z2 {
18             x0 = "Pip3A4PurchaseOrderRequest/fromRole/
                   FromRoleClass/partnerRoleDescription/
                   PartnerRoleDescriptionClass/contactInformation/
                   ContactInformationClass/contactName";
19             x1 = "Pip3A4PurchaseOrderRequest/fromRole/
                   FromRoleClass/partnerRoleDescription/
                   PartnerRoleDescriptionClass/contactInformation/
                   ContactInformationClass/contactName";
20             x2 = "Pip3A4PurchaseOrderRequest/fromRole/
                   FromRoleClass/partnerRoleDescription/
                   PartnerRoleDescriptionClass/contactInformation/
                   ContactInformationClass/emailAddress";
21             x3 = "Pip3A4PurchaseOrderRequest/fromRole/
                   FromRoleClass/partnerRoleDescription/
                   PartnerRoleDescriptionClass/contactInformation/
                   ContactInformationClass/telephoneNumber/
                   TelephoneNumberClass/communicationNumber";
22             x4 = "Pip3A4PurchaseOrderRequest/purchaseOrder/
                   PurchaseOrderClass/shipTo/ShipToClass/
                   physicalLocation/PhysicalLocationClass/
                   physicalAddress/PhysicalAddressClass/
                   addressLine1";
23             x5 = "Pip3A4PurchaseOrderRequest/purchaseOrder/
                   PurchaseOrderClass/shipTo/ShipToClass/
                   physicalLocation/PhysicalLocationClass/
                   physicalAddress/PhysicalAddressClass/cityName";
24             x6 = "Pip3A4PurchaseOrderRequest/purchaseOrder/
                   PurchaseOrderClass/shipTo/ShipToClass/
```

```
                physicalLocation/PhysicalLocationClass/
                physicalAddress/PhysicalAddressClass/
                nationalPostalCode";
25          x7 = "Pip3A4PurchaseOrderRequest/purchaseOrder/
                PurchaseOrderClass/shipTo/ShipToClass/
                physicalLocation/PhysicalLocationClass/
                physicalAddress/PhysicalAddressClass/
                globalCountryCode";
26      }
27  }
28  }
```

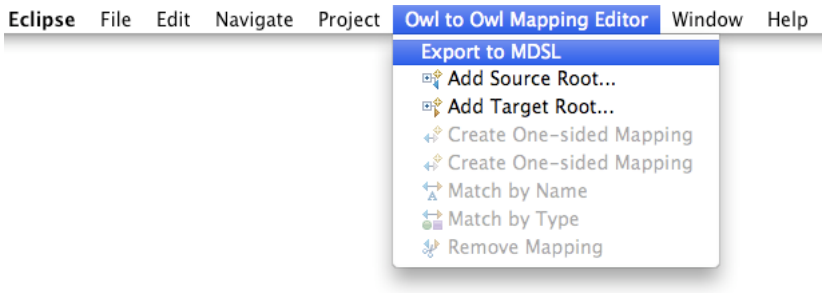


Figure 9.6: The Export to MDSL command in the Eclipse menu bar

With this final export action, the job of the owl2owl mapping editor is done. The PIMs have been mapped and the relations between the information models of the two IT systems have been defined in the MDSL format.

9.2.3 Configuring the mediator

Now that all mapping information is defined, the Mediator can be configured. Using the mapping information from both the information and the behavior models, a Mediator PIM can be constructed. This PIM is validated, and when is passes validation, is ready to be converted to a PSM, i.e. a concrete implementation of the Mediator. Since the functionality of the Mediator is completely specified in a Platform *Independent* Model, the execution platform for actually creating the Mediator can be chosen freely. In [20], a WS-BPEL application was chosen, but EJB or .NET applications are also a possibility.

9.3 Results

Performing the case study with the implemented mapping editor yielded a number of results. The results can be roughly divided into two categories. First, there are the results regarding the prerequisites for using the editor, such as the input files, or the way the editor is used in the entire process. Second, there are results that directly concern the use or the implementation of the editor, such as improvements in functionality, or user interface issues. This next sections will discuss these results.

9.3.1 Issues regarding the process

The input ontologies

The editor assumes that the input is in the form of OWL ontologies. These ontologies have to be generated, based on the WSDL description of the Web Services that are used in the systems that have to be integrated, and the natural language description of the behavior of both systems. Currently, this is mostly a manual process. While tools exist that can generate ontologies from WSDL descriptions, this does not easily yield the correct OWL ontology files. Therefore, for the purpose of this case study, the two OWL ontologies have been created manually. These ontologies do not describe the entire IT systems, but only the parts that are used in this case study. A graphical representation of the two ontologies would look similar to the trees in the example mapping, in Figure 9.1.

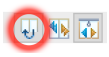
The export process

In the current implementation, the owl2owl mapping editor supports two export 'modes'. It can either read an existing MDSL file and add the mapping information to it, or it can create an new MDSL file and populate it with the mapping information. Which mode to use is up to the user do decide. In the case study, no preexisting MDSL file was available, so it was generated by the editor. However, if the user wants to edit existing mapping information using the editor, it is useful to export to an existing MDSL file. This is a decision that the user needs to make. In the current implementation, the editor does not offer a choice between the two modes. It automatically chooses to edit an existing MDSL file if a file with a specified URI exists. If not, it will create a new MDSL file with this same URI. The user should be aware of this, so he can make a conscious choice between the two export modes.

9.3.2 Issues regarding the editor

A number of issues are related to the editor itself. These are problems with the user interface, small implementation quirks, etc. The following issues were discovered in the case study.

Source and target columns reversed in mappings list



The list in the editor that displays the mappings, has a unwanted default setting: the column containing the elements from the source ontology is displayed on the right and the column containing the elements from the target ontology is displayed on the left. This is precisely opposite from the tree views of the ontologies, where the source is left and the target is right. With the press of a button, the columns can be switched, but this should not be necessary. An additional confusion occurs when many-to-one mappings are defined. Mapping two elements from the source ontology to one element on the target ontology,

results in two mapping. However, when the columns are switched, only one line is displayed, containing the target element in the first column and an enumeration of the source elements in the second column. This is confusing, because the convention is that each line contains exactly one mapping.

Paths of mapped elements not immediately displayed

When the user creates a mapping between two elements in the ontologies, the Properties view shows the details of this mapping (see Figure 7.1). Among these details are the paths of the selected source and target elements. However, these paths are not displayed in the view until the entire mapping file is saved. This behavior does not cause any data integrity problems, but it might confuse the user and cause him to think that information might not get saved.

Additional Relation Function types are required

In one of the example mappings, the `por2moonItem` mapping that is displayed in Figure 9.1, a Relation Function is required with the name `convertToInt`. This Relation Function is not supported by the current implementation of the mapping editor, but can easily be added by adding it to the owl2owl ecore model. When more mappings than just these sample of three are defined using the tool, this will most likely reveal the need for even more Relation Function types. If the `convertToInt` function is implemented, it is possible to generate the following MDSL output for the `por2moonItem` mapping. Notice that now an `expression` block is added that makes use of the new function.

```
1  mapping por2moonItem {
2      target moon:AddlineItemRequestClass owlcat_z11 {
3          x0 = "AddlineItemRequestClass/lineItem/LineItemClass/
               articleId";
4          y1 = "AddlineItemRequestClass/lineItem/LineItemClass/
               quantity";
5      }
6      source blue:Pip3A4PurchaseOrderRequest owlcat_z10 {
7          x0 = "Pip3A4PurchaseOrderRequest/purchaseOrder/
               PurchaseOrderClass/productLineItem/
               ProductLineItemClass/productIdentification/
               ProductIdentificationClass/globalProductIdentifier";
8          x1 = "Pip3A4PurchaseOrderRequest/purchaseOrder/
               PurchaseOrderClass/productLineItem/
               ProductLineItemClass/orderQuantity/
               OrderQuantityClass/requiredQuantity/
               RequiredQuantityClass/productQuantity";
9      }
10     expressions {
11         y1 = convertToInt(x1);
12     }
13 }
```

Minor user interface improvements

There have never been strict requirements regarding the user interface. While it was determined that the editor should be relatively easy to use, very strict rules or guidelines have never been laid out. That being said, there are few comments to be made regarding the graphical user interface of the owl2owl mapping editor.

In the first place, the GUI of the editor seems to be optimized for use on Windows platforms. The mappings list of the editor uses custom ⊞ and ⊟ symbols for the unfold and fold commands of the list. These are indeed the default symbols in Windows for folding and unfolding items in a list. However, on other platforms, these symbols might be different. Mac OS X uses triangles (▶ and ▼), for example. Also, the indentation of the mappings, similar to the indentation of sub-elements in the tree views of the ontologies, works perfectly in an Eclipse instance under Windows, but is absent under OS X.

One interface improvement that had already been implemented at the time of the case study, is the use of custom icons. While the EMF framework provides default icons to represent entities in the model (e.g. the different OWL elements), it is much more convenient for the user to make use of custom icons that clearly identify an element. However, the editor sees all elements in the tree as instances of NavElement, so all would get the same icon, regardless of whether the element represents a class, object property or data property. For this reason, the NavElement class has been modified, such that is serves up a different icon, depending on what kind of element it represents. The custom icons that have been used are the same as those used in the Swoop[2] tool.

9.4 Evaluation

The goal for this case study was twofold. First, the study should demonstrate how the editor works as part of the entire suite of applications that solve the Purchase Order Mediation scenario of the SWS Challenge. Second, the study should perform an evaluation of the mapping editor, in order to find strong points, weak point, and opportunities for improvement.

In order to reach these goals, the study has been conducted as described in section 9.2. The description given in that section demonstrates the functionality of the owl2owl mapping editor, along with its input and output requirements. Hence, the first goals for the case study is reached.

More precise results regarding the use of the owl2owl mapping editor have been described in section 9.3. This section also identifies a number of aspects of the editor that can be improved upon. With this section, the second goal for conducting the case study has also been reached. But before concluding this chapter, it is worth discussing one of the results in more detail.

[2] http://code.google.com/p/swoop/

Improvements on the export functionality

The results section already commented on the fact that the export functionality is effectively composed of two different modes. One mode creates a new MDSL file, the other mode adds the mapping information to an existing MDSL file. Currently, this choice is invisible to the user. If an MDSL file exists with a predetermined URI, the mapping information is added to this file. If the file does not exist, a new one is created. If the user is not aware of this, the results may be unexpected.

A number of solutions are possible in order to solve this hidden behavior. One possibility is to give the user an explicit choice. When exporting the mapping information, the user should explicitly state whether to add the information to an existing MDSL file, or to create a new one. In the case the user wants to add the information to an existing file, the application can prompt the user to supply the URI of this file. Another possibility is to abandon one of the choices altogether, and require that there should always be an MDSL file to write to. This could be an empty file, thereby effectively having the editor create a new file anyway, but it does make the workflow more clear and consistent from the user's point of view. Whatever possibility is chosen, it should improve the current situation by making clear to the user what effect the export action will have.

Chapter 10

Discussion and reflection

In the course of conducting this research and implementing the tool, a number of questions and issues arose. Some of these are directly related to the implemented editor. Others concern the use of the tool as part of the entire solution of the A-MUSE project. Even others are of a more general nature, regarding the research as a whole. In addition, a state-of-the-art survey has been conducted that resulted in an overview of existing solutions for ontology mapping. This chapter presents a discussion on these topics, based on the questions and issues that came up. In addition, it discusses where the implemented tool differs from existing solutions. The discussion is roughly organized based on the following three areas. The first section discusses questions and issues related to the input that the tool requires, and the output that it generates. The second section then deals with issues concerning the role of the tool in the entire solution process of the Purchase Order Mediation scenario. The third section discusses a number of possible changes or improvement opportunities for the tool. Finally, the fourth section discusses the differences between the implemented owl2owl mapping editor and existing solutions.

10.1 The input and output of the mapping editor

This first section discusses several issues related to the input and output of the mapping editor. To start the discussion, the input of the editor is considered first. In the current implementation of the tool, the input consists of two ontologies, specified using the ontology language OWL. Stating this as a fact raises several questions, for example: why does the input consist of ontologies, instead of other types of schemas, java classes, or some other input? And why is OWL chosen as the ontology language of choice, and not some other ontology representation? Using OWL as input format, can the tool be used to map all possible ontologies, or are there still some restrictions? How well does the application scale up, when the tool has to handle quite large ontologies? And can it perhaps be used to map more than just *two* ontologies, or even map other things besides ontologies? Section 10.1.1 addresses all of these issues. Section 10.1.2 then goes on to discuss several questions related to the output of the mapping editor.

10.1.1 Input

The first question concerns the use of ontologies as input. As described in the previous chapters, the input for the editor should be in the form of ontologies. But Quartel et al. [20] describe that they use Java classes and UML class diagrams at the moment, rather than OWL files. So why not use those as input, instead of ontologies? The answer lies in the differences between simple Java classes and ontologies. Using an ontology-based approach makes it possible to map classes and properties onto some domain specific ontology such as UDEF (see footnote 1 on page 63). In addition, ontologies allow one to reason about the semantics of the elements in the ontology. In practice, this means that unspecified relations may be inferred, and specified relations can be validated for mathematical correctness. Consider for example an ontology that describes a class **Man**, a class **Child**, an object property **hasChild** and a class **Parent**. The ontology specifies that any **Man** is also a **Parent** if he has a **Child**. Now if we have some instance of **Man** in our ontology and we know that this **Man** has a **Child**, we can infer that the man must also be a **Parent**, even though this was never explicitly specified anywhere. Without using ontologies, such an inferences could not have been computed as easily. Quartel et al. acknowledge this, and indicate that they plan on using ontologies rather than Java classes in future versions of the solution. However, the current version still uses Java classes, for the sake of ease of implementation. A relatively quick solution to adapt the mapping editor to the current situation would be to write an adapter that can convert Java classes to OwlCat files. This is already explained in section 5.4.1.

From the above, it is clear that is has benefits to use ontologies as the input format for the mapping editor. In the previous chapters, it is made clear that OWL is the ontology language of choice. But OWL is not the only ontology language that is used. A number of other languages exist, that are also capable of describing ontologies. Examples of this are the Resource Description Framework (RDF)[1], DAML+OIL[2], or F-logic[3]. OWL has several benefits over other ontology languages, which is why it was the language of choice for this project. First, it can be seen as the successor of DAML+OIL. Based on experiences with DAML+OIL, the language was revised and the result would become the OWL language. Second, OWL has a number of characteristics that are very convenient for describing ontologies and relations between entities. From the OWL language overview of the W3C[4]:

> "OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes."

However, while OWL may be the language of choice, the editor is not restricted to it. In order to use the editor with some other ontology language, one only

[1] http://www.w3.org/RDF/
[2] http://www.daml.org/2001/03/daml+oil-index
[3] http://www.cs.umbc.edu/771/papers/flogic.pdf
[4] http://www.w3.org/TR/owl-features/

needs to create an adapter from the required ontology language to the OwlCat format. In fact, any type of input format that can be converted to the OwlCat format, can in principle be mapped using the editor. Because even though it is called the owl2owl mapping editor throughout this work, it is in fact a meta-model mapper, and thus it can map anything that conforms to the OwlCat meta-model format. That being said, if the editor will be improved / expanded and more of the advanced features of OWL will be used, this has to be reflected in the OwlCat format. And the more complex this OwlCat format becomes, the less trivial it will be to write converters for other input formats.

Another issue concerns the scalability of the application. There is only one relevant variable with regard to the scalability, and that is the size of the input ontologies. Without having conducted specific scalability experiments, the operation that seems to be most sensitive to scalability issues is the conversion process from OWL to OwlCat. During development, the difference between converting an ontology that contains tens of items and one that contains hundreds of items is in the order of several seconds. A more detailed study would be necessary in order to determine whether ontology size and subsequent memory use and computing time might become a problem with very large ontologies. For the scenario of the Semantic Web Service challenge and the ontologies that are being used within this context, these limits are not yet reached, so scalability is not a problem within this limited scope

Apart from the type of ontologies and their size, another factor is worth considering. That is the number of ontologies that are used as input for the mapping editor. The current implementation of the tool requires that exactly two ontologies are provided as input: one ontology can be mapped onto another one. But there might be situations where it is desirable to map more than two ontologies. For example, one might want to map two ontologies onto a third (global) ontology. Consider the case of the Purchase Order Mediation scenario: one might want to map the ontologies that describe the Blue and Moon company systems onto a third, global ontology, which describes the entire business domain. This can aid in the mapping between the Blue and Moon systems, because whenever two elements in the two ontologies map onto the same element in the global ontology, this suggests that these elements are related.

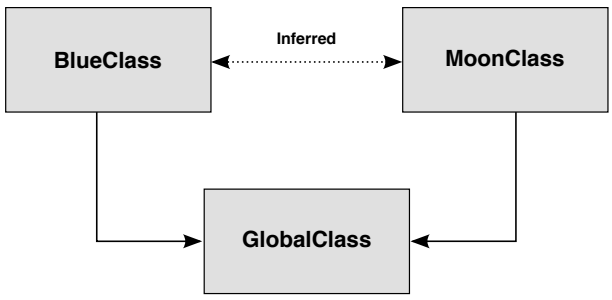


Figure 10.1: Mapping two ontologies onto a third, global ontology

The current implementation of the editor does not support this directly. It is only possible to load two ontologies and create mappings between these two. It would be very difficult to change the editor so that it directly supports the

creation of mappings between three or any other arbitrary number of ontologies. The entire structure and user interface of the editor is based on the ecore2ecore mapping editor that can be found in the EMF framework, and this editor only supports mapping between two models. Mapping two or more ontologies onto a global ontology is still possible, but this should then be done one ontology at a time, i.e. first map one ontology onto the global ontology, then map the second one to the global. However, since this does not create one integrated ontology, but only a set of mappings, it is not yet possible to reason and infer using all these mappings.

10.1.2 Output

The previous section discussed the input of the mapping editor. This is very important, because the constraints on the input determine whether the tool can be used at all in the given scenario. If the tool requires input that cannot be provided by the environment, the tool is of no use. The output of the tool, on the other hand, determines whether it is useful to apply the tool. If the output of the tool does not add value to the solution, why use the tool at all? So when evaluating the tool, it is important to look at the output and determine whether the output is actually useful as part of the solution to the Purchase Order Mediation scenario.

So what is 'useful' in this case? The most obvious criterium might be that the output describes what it should, and is ready to be used by the other applications that try to solve the Purchase Order Mediation scenario of the SWS Challenge. In this sense, the output is indeed useful. Since the output format is MDSL, which is exactly the format that is used by the other tools in the suite of applications, it is immediately applicable.

Another criterium would be whether the output can be used *immediately*, without modification. In theory, this is possible with the current implementation of the tool, but only when the input consists of exactly two complete ontologies. In practice, as was described in the case study, the input might consist of multiple ontologies, which have to be mapped in turns. This results in multiple output files, which have to be merged manually, before they can be used. [check with revised case study]

Yet another criterium to define 'useful' is to see if the output is complete. Complete means that all necessary transformations are defined in the resulting MDSL code and no additional coding is necessary to describe additional data mismatches. This depends on two things. First, the input ontologies should be complete. If the input fails to describe crucial data elements, the output can never cover all required transformations. Second, the tool must output the complete set of transformation. Currently, the completeness in this sense is controlled by the user of the mapping editor. The user is the one who determines and enters all mappings between the ontologies, thereby controlling which transformations are outputted by the tool. The current implementation of the tool offers only a rudimentary check for completeness, by highlighting items in the input ontologies which have not yet been mapped to anything.

Finally, the usefulness of the output can depend on level at which the problem is viewed. Looking at it from a very practical, hands-on point of view, the output

is useful if it helps solve the Purchase Order Mediation scenario. As long as the output is of immediate use to the other tools that provide parts of the solution, it is useful. In this sense, the current implementation has succeeded, as concluded several paragraphs earlier. From a somewhat higher point of view, one can look at the editor as a more general meta-model mapper. The MDSL output that can be generated is now quite useless, since it is much too specific for this scenario and only applies to the specific case of OWL files and the SWS Challenge. The other output of the tool, the owl2owl meta-model which can be saved to disk, is of a somewhat higher order. This model simply stores the information about mappings between the two input models that the user has entered. But even now, most of this information about the mappings is information that is specifically needed for generating the required MDSL, not information that describes the relation on a meta-level. So while it is possible to indicate relations between two meta-models on a high level, the editor is clearly designed for other purposes.

10.2 The tool as part of the entire process

This second section of the Discussion chapter deals with the role that the tool plays as part of the entire process for solving the Purchase Order Mediation scenario. Several questions are related to this issue, such as: does the tool fit in nicely and correctly with other tools in the suite of applications? And are there perhaps similar tools available that can solve the data integration problem of the Purchase Order Mediation scenario of the SWS Challenge? And what practical problem did the mapping editor actually solve, such that it is a valuable part of the solution to the challenge? And finally, can the tool be used by domain experts alone, as is the vision of the entire suite of solutions as described in [20]? The following paragraphs try to answer these questions.

The first question to answer is whether the tool fits in nicely and correctly with other tools in the suite of applications. The previous section already showed that the output of the tool is exactly in the format that is used by other tools, and that only a few minor adjustments may be necessary to make the integration complete. It also showed that the input does not quite match the current approach, since the mapping editor uses OWL as input, while the other tools use Java classes and UML. However, the section also described that adapting to the current situation should not be difficult, and that the OWL-based approach is in fact the proposed, preferable solution. So if we disregard the input and output, what else can improve or worsen the integration with other tools? One way to improve the integration is to *actually integrate* all the different tools. So instead of having a suite of tools that form the entire solution, there is one integrated tool that offers all the functionality that is now provided by several individual tools. There is no concrete implementation of any such tool, but there are some constraints in place throughout the entire project that ensure that integration should in principle be possible. These constraints also apply to the mapping editor (see also the box Implemented User Stories in chapter 3): use Java as the main programming language, implement the tool as an Eclipse plug-in, and use EMF / Ecore as the underlying framework and representation for the input

models. The more each individual tool adheres to these constraints, the easier it will be to implement one integrated tool.

Another issue concerns the intended users of the mapping editor. As noted earlier, the reason for lifting the problem and solution to a higher level, is to allow for easier reasoning about the problem, and to allow business domain experts to use their knowledge of the problem domain without being hindered by technical difficulties. This implies that those same business domain experts should be able to use the tool, without any help from IT professionals. To answer this questions decisively requires an elaborate user study, but some properties of the tool already hint at a possible answer. The first phase, loading the ontologies, should not be too difficult. With the proper instruction, this is not a very difficult task. The next step, indicating mapping relations, is also quite easy using the tool, and should not be hard to learn. Where it becomes more involved, is when the user has to provide detailed information for each mapping. The specific details that the user needs to provide, such as the Relation Function, Function Argument Index, etc. (see section 6.2), require that the user has a more technical mindset and considerable knowledge of the 'next' tool in the chain. While this is of course speculative, it is not likely that such intimate knowledge can be expected from the domain experts. The mapping editor, and perhaps also other tools with which it interacts, would have to be altered to provide an even higher level of abstraction, and it should provide explicit instructions to the user, in order to be usable for domain experts alone.

10.3 Improvement opportunities

The previous sections discussed various issues concerning the tool as it is now. This section on the other hand discusses a number of opportunities for changing the tool. This ranges from functional changes to usability changes to entirely different ways of applying the tool.

One of the first changes that this tool *will* see when it is deployed in a live environment, is an increase in the number of relation functions that is supports. Recall from section 6.3.1 that only three relation functions are initially specified (`equals`, `partOf`, and `concat`), but in practice there will probably be a demand for quite a larger set of functions. When the IT systems describe e.g. prices and quantities, one can imagine that this might require relation functions that can perform arithmetic operations. Adding relation functions is relatively simple, thanks to the design of the mapping editor. The only change required is to add the new operation to the owl2owl mapping model and recompile the tool. Thanks to the EMF framework, this will automatically ensure that all model code and the user interface also support the newly created relation function.

These changes concerning the relation functions are necessary in order to make the tool useful in a live environment. There are also a number of improvement opportunities that are not necessary for effective use of the tool, but that do enhance its usability and functionality. One such opportunity concerns the graphical user interface of the tool. The current implementation of the mapping editor is based on the ecore2ecore mapping editor of the EMF framework, and inherits its GUI. While this interface offers the user the required functionality,

it has a lot of opportunity for improvement. Among others, it can be changed to guide the user through the required workflow, to offer clearer graphical and textual clues of the functionality, to be more consistent in layout on different platforms, and to better represent the ontologies and the associated mappings.

Another area for improvement concerns the use of an OWL reasoner. A reasoner is a software application that can infer additional information about the ontologies by evaluating their classes, properties, and additional constraints. Currently, a reasoner is only used when the original OWL file is converted to OwlCat. The Jena framework uses a reasoner to make implicit information in the model available as Java objects. When the OWL file has been converted to an OwlCat file, the work of the reasoner is done. However, if the reasoner can also be used with the specified mappings, and evaluate relations between elements of the two ontologies, it is possible to check for validity and possibly to aid in the creation of additional relations.

To add even more intelligence to the tool, it can be modified to include an automatic matching algorithm. Chapter 2 presents a number of such algorithms. An automatic matching algorithm can make the detection and creation of mapping relations easier by suggesting candidate classes and properties that should be mapped. Chapter 2 and [23] describe how these algorithms determine their suggestions. Even though the algorithms are never 100% accurate or complete, using an algorithm like this saves the user from having to manually identify and specify all mappings and can save considerable time and effort.

This paragraph concerns a change on a somewhat higher level. Currently, the focus of the mapping editor is clearly on creating mappings between ontology elements. And naturally so, because identifying such mappings and deriving the appropriate transformations from them was the entire purpose of creating the editor in the first place. But when mapping two ontologies, it is possible to follow another approach. Instead of creating mappings, a new, integrated ontology is created, based on the two (or more) input ontologies. The benefit of this approach is that the functionality of reasoners can now be immediately applied to not only the input ontologies, but also the combined ontology. In addition, the mapping solution is of a more general nature, since one only specifies how the ontologies are to be combined, without worrying about mapping implementation details. The drawback of this approach is that is harder to implement, and harder to derive the required mappings from the general solution. In the end, concrete transformations are to be generated, and this is much easier to do from explicitly defined mappings than from a general combined ontology.

10.4 Comparison with existing solutions

The owl2owl mapping editor that has been implemented as part of this research, solves a quite specific problem. However, the editor is basically an ontology mapping tool. It is therefore interesting to compare it to other ontology mapping solutions, in order to see where the implemented tool differs from the existing solutions, where it improves upon them, and where other solutions may offer a better solution than the implemented owl2owl editor. Chapter 2, as well as [23], already discuss a number of alternative ontology mapping solutions.

This sections discusses the most striking differences or similarities between these solutions and the implemented editor.

10.4.1 The editor's output

Let's first consider the output. The owl2owl mapping editor produces two outputs: (1) an ecore model that contains references to the input ontologies and stores the specified mappings, and (2) a very specific output in the form of the MDSL script, targeted solely at the Purchase Order Mediation scenario of the Semantic Web Services challenge. It is not easy to compare this output to that of other tools. As Noy and Musen [15] note, tools for ontology mapping differ so much, that making a meaningful comparison is difficult. However, despite their differences, all tools have in common that they produce *some* output that describes relations between the input ontologies. This can be in very different ways — such as a merged ontology (PROMPT [16]), a list of relations between entities (QOM [6]), or queries — but all tools produce some output nonetheless.

The contribution of the owl2owl mapping editor is that it adds yet another output: the MDSL script. This output is generated specifically for solving the Purchase Order Mediation scenario, and therefore no other tool can be a better substitute for the owl2owl mapping editor in this case. In addition, it is relatively easy to expand the owl2owl mapping editor with additional export algorithms. This means that if another application requires a different kind of ontology mapping output, it is relatively easy to adapt the owl2owl mapping editor for that situation. The converse is also true: if the highly specific output of the owl2owl mapping editor is not quite the required output for any particular situation (which it very probably is not), one first needs to write a custom export module for it.

10.4.2 Degree of automation

Another aspect by which to compare the various solutions is the degree of automation that it offers. Some algorithms are specifically designed to automate (part of) the ontology mapping process (e.g. NOM [7], QOM [6], and S-Match[9]). Other solution are merely mapping tools that provide a user interface, such that all mappings can be defined manually by a user. The owl2owl mapping editor falls in the latter category, along with tools such as the NeOn Toolkit[5] and WSMO Studio[6].

The owl2owl mapping editor offers only one small automation feature. It can automatically create mappings between entities with exactly identical names. This is by no means a practical or sophisticated mapping algorithm, but it is automation nonetheless. But other than that, the editor only provides the means to create mappings by hand. Other solutions on the other hand, such as the aforementioned NOM, QOM, and S-Match, among others, focus specifically on automating the mapping process. They use various techniques in order to find similarities between ontologies. This is discussed in more detail in the research

[5] http://www.neon-toolkit.org
[6] http://www.wsmostudio.org

paper [23] or articles such as [22]. In future efforts, the owl2owl mapping editor may be expanded to include such a mapping algorithm.

10.4.3 Ontology visualization

It should be clear by now that there are various different ways by which the solutions try to solve the ontology matching problem. One other aspect in which they differ is the way they visualize ontologies. Some solutions, such as NOM and QOM, are only algorithms, which offer no visual representation of ontologies. Others, such as the NeOn Toolkit, WSMO Studio or Protégé, do offer a GUI and subsequently implement some method for visualizing an ontology. They use different approaches. Some solutions, such as Protégé, use a combination of Tree views. They create separate Trees for classes, object properties, data properties, and individuals within an ontology. The trees show the subclass–superclass (or subproperty–superproperty, etc.) relation of the entities. Other solutions, such as the NeOn Toolkit, use a graph view to visualize an ontology. The implemented owl2owl mapping editor uses an integrated Tree view: an ontology is shown as one Tree in which the classes hierarchy is used as basis. However, the 'children' of a class node are not only that class' subclasses, but also the properties for which that class is defined as the domain.

This method of visualization has a clear advantage, but also a clear disadvantage. The advantage is that it allows one to visualize an entire ontology in one view. All of the entities that make up the ontology can be displayed in one single Tree view. The disadvantage is that it is impossible to describe the ontology completely and correctly, since an ontology is in essence more similar to a graph than to a Tree. Forcing it to display in a Tree means that at some point, information is lost. The 'infinite loop' problem that is discussed in section 5.4.2 is a nice illustration of this problem.

When observing these different visualization methods, one might ask which is the 'best' method. The answer depends on the intended use of the visualization. For the purpose of understanding the ontology and creating a visual representation of the interrelations between ontology elements, a graph view is the most suitable method. For the purpose of quickly listing and navigating the entities that make up the ontology, a Tree view is more suitable. This is why the owl2owl mapping editor employs this integrated Tree view. It should visualize the ontologies so that the user can quickly find the elements that need to be mapped, and support the mapping process. The intended user is a domain expert, so he should already be familiar with the interrelations of the various element in the ontology.

Chapter 11

Conclusions and future work

This chapter presents the main conclusions of this research. Section 11.1 presents the general conclusions of the research and outlines its main contributions. Section 11.2 identifies opportunities for future research and development.

11.1 Conclusions

The Purchase Order Mediation scenario of the Semantic Web Service challenge describes an interoperability problem. Two companies want to do business, and need to integrate their IT systems. Within the A-MUSE project, a solution to this problem is devised that uses ontologies to describe the two IT systems. This solution requires a tool that can create mappings between these two ontologies, and use these to generate the necessary data transformations. The objective of this research was to create such a tool, and in the Introduction chapter, it was formulated as follows:

The research objective of this project is to develop a tool for generating transformations from mappings between two ontologies by selecting appropriate mapping algorithms through a state-of-the-art survey and implementing the tool as a plug-in for the Eclipse platform.

Four discrete steps were derived from this objective. These steps have been performed and the resulting research and development effort has been described in this thesis. The following sections aim to give a succinct conclusion for each of these steps.

Perform a state-of-the-art survey to determine what ontology matching algorithms are available for use in the solution to the Purchase Order Mediation scenario of the SWS Challenge

The state-of-the-art survey resulted in a comparison between different ontology mapping algorithms and various mapping tools. Each of these tools and algorithms have been compared based on a number of characteristics. This yielded

an overview of currently available tool and algorithms. In addition, the process of creating the survey proved to be a good introduction in the field of ontology research.

The research into ontology mappings tools showed that there are currently no tools that exactly match the requirements for the SWS Challenge. Mapping is supported by a number of tools, but no tool offers a way to generate the required transformations based on these mappings. The tools *did* offer inspiration on how to create a user interface for a mapping tool. In particular, most tools use some sort of tree representation to display the input schemas or ontologies, and most tools offer a way to view and edit very specific details for each individual mapping.

The research into ontology matching algorithms resulted in an overview that compares a number of these algorithms on various characteristics. This comparison showed that there are a lot of different approaches to create an ontology matching algorithm. While one approach focuses mostly on lexicological methods, others focus primarily on ontology structures and patterns. The more effective algorithms combine several of these approaches in an attempt to obtain the best results. From a practical point of view, the study showed that many of these algorithms are not implemented in any readily available form or product, but only exist as proposals or proof-of-concept implementations.

Implement a software tool that supports the creation of mappings between two ontologies

The chapters 3 through 7 describe the entire process for creating this software tool, the owl2owl mapping editor. Using the described iterative approach, the requirements and functionality of the tool have been gradually developed into the resulting end product, the owl2owl mapping editor plug-in for the Eclipse platform.

This product is capable of loading two ontologies that are described using the OWL language. The ontologies are converted to an intermediate format, in order to allow for ease of implementation and to make the implementation more generic. Other input format besides OWL can be supported by simply creating a conversion module that can translate the input format into the intermediate format. When the ontologies are loaded, the user can create mappings between elements in the two ontologies by using the graphical user interface of the tool. Additionally, the user can specify detailed information for each mapping, which will later be used to generate the correct transformation based on these mappings.

The implementation is subject to a number of constraints, such as the requirement to use Java, implement the tool as an Eclipse plug-in and make use of the EMF framework. These constraints ensure that the tool can interact with other tools that have been developed as part of the A-MUSE project, which solve other parts of the interoperability problem in the Purchase Order Mediation scenario.

Implement an algorithm that creates transformations based on the resulting ontology mappings

In the last phases of implementing the mapping editor, an algorithm has been implemented that can create transformations, based on the specified ontology mappings. These transformation are expressed using a Domain Specific Language (DSL) called the Mapping DSL or MDSL. They are used to transform the data from the source system to match the information structure of the target system. Other tools, which are part of the solution to the Purchase Order Mediation scenario, will execute the transformations, such that the actual messages that are exchanged between the two coupled IT systems are translated.

Validate the tool by performing a case study, using the Purchase Order Mediation scenario as context for this study

With the completed owl2owl mapping editor available, a case study was conducted within the context of the Purchase Order Mediation scenario of the Semantic Web Services challenge. This case study served two goals: to demonstrate how the editor can be used as part of the solution for the given scenario, and to evaluate the editor. This resulted in a step-by-step description of the working of the editor, thereby meeting the first goal for the case study. It also resulted in a list of observations, which can be used to improve the editor. This meets the second goal for the case study.

11.2 Future work

This section presents a number of opportunities for further research or development. It can roughly be divided into two parts. The first section lists a number of ways in which the mapping editor can be improved. This is mostly a summary of some of the findings in the Discussion chapter. The second section takes a more general view of the integration problem and the proposed solution, and identifies a number of research opportunities in this area.

11.2.1 Mapping editor improvements

The mapping editor can be improved or expanded in the following ways:

- Create a module that converts Java classes to the OwlCat intermediate format. This way, the editor can be used with the current implementation of the Mediator, which uses Java classes and UML instead of OWL to represent the two IT systems of Blue and Moon company.
- Automate the conversion process. In the current implementation, the user has to manually convert the input ontologies to OwlCat files, and then select the two OwlCat files for mapping. The tool can be improved by automating this conversion. The user can then select the two input ontologies for mapping, which are automatically converted to OwlCat files by the tool, and then loaded.

- Expand the set of supported RelationFunctions. The current implementation contains only three functions, `equals`, `partOf`, and `concat`, but it is very likely that more functions are required.
- Improve the graphical user interface and general usability of the tool. If the tool is really intended to be used by business domain experts with limited technical knowledge, the user interface should be improved such that it provides more guidance and hints. In addition, the general usability can be improved by reducing the number of necessary mouse-clicks, providing keyboard shortcuts, improving the graphical design of the editor, correcting minor layout errors, etc.
- Integrate an OWL reasoner and matching algorithm with the tool. This would allow one to make better use of the full potential of the OWL ontology language, because the reasoner and the algorithm can infer additional information about the input ontologies, provide validation, and assist in the finding and creation of new mappings.

11.2.2 Research and development opportunities

Apart from the mapping editor itself, a number of other research or development opportunities can be identified. These are the following:

- The Mediator that is developed as the solution to the integration problem in the Purchase Order Mediation scenario, currently consists of multiple different applications. These applications work together to achieve the desired integration solution. An improvement opportunity here it to merge these applications and create one single application that can perform all necessary steps. This reduces the complexity of the solution and provides the opportunity to create a much more polished solution for the user.
- The automatic derivation of ontologies based on the WSDL descriptions of the two IT systems proved to be difficult. It is possible, but did not result in two properly built ontologies. Therefore, the case study described in chapter 9 used two manually built ontologies. Further research and development is required in order to obtain a software tool that can automatically generate the required ontologies.
- In the course of developing the editor, a careful balance was maintained between supporting OWL on one hand and on the other hand keeping the mapping editor as generic as possible. When expanding the tool with other functionality, such as reasoners and ontology matching algorithms, it might be difficult to maintain this balance. At that point, the benefits of keeping the editor generic have to be carefully considered. It may be possible to maintain the balance, but it may also be the case that the OWL specific functionality makes it next to impossible to keep the editor generic. At this point, further research into the requirements and implementation possibilities is required to make the appropriate decision.

References

[1] The Eclipse Modeling Framework (EMF) Overview [online]. June 2005 [cited December 2007]. Available from World Wide Web: http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.emf.doc//references/overview/EMF.html.

[2] ProtegeWiki: Prompt [online]. November 2006 [cited January 2008]. Available from World Wide Web: http://protege.cim3.net/cgi-bin/wiki.pl?Prompt.

[3] Web Ontology Language OWL / W3C Semantic Web Activity [online]. October 2007 [cited February 2009]. Available from World Wide Web: http://www.w3.org/2004/OWL/.

[4] Namyoun Choi, Il-Yeol Song, and Hyoil Han. A survey on ontology mapping. *SIGMOD Record*, 35(3), September 2006.

[5] Teduh Dirgahayu. *Model-Driven Engineering of Web Service Compositions: A Transformation from ISDL to BPEL.* Master's thesis, University of Twente, 2005.

[6] Marc Ehrig and Steffen Staab. QOM - Quick Ontology Mapping. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 683–697. Springer, 2004.

[7] Marc Ehrig and York Sure. Ontology Mapping – An Integrated Approach. In Christoph Bussler, John Davis, Dieter Fensel, and Rudi Studer, editors, *Proceedings of the First European Semantic Web Symposium*, volume 3053 of *Lecture Notes in Computer Science*, pages 76–91, Heraklion, Greece, May 2004. Springer Verlag. Available from World Wide Web: http://www.aifb.uni-karlsruhe.de/WBS/meh/publications/ehrig04ontology_ESWS04.pdf.

[8] David Frankel. Business network transformation and semantic interoperability [online]. December 2007 [cited February 2009]. Available from World Wide Web: https://www.sdn.sap.com/irj/sdn/go/portal/prtroot/docs/library/uuid/b00cb8c5-6788-2a10-efa2-d7a7c7c74aeb.

[9] Fausto Giunchiglia, Mikalai Yatskevich, and Pavel Shvaiko. Semantic matching: Algorithms and implementation. *Journal on Data Semantics IX*, 4601:1–38, 2007.

[10] Tom Gruber. Ontology (computer science) - definition in encyclopedia of database systems [online]. September 2007 [cited February 2008]. Available from World Wide Web: `http://tomgruber.org/writing/ontology-definition-2007.htm`.

[11] Masahiro Hori, Jérôme Euzenat, and Peter F. Patel-Schneider. OWL XML Syntax: XML Schemas [online]. June 2003 [cited February 2009]. Available from World Wide Web: `http://www.w3.org/TR/owl-xmlsyntax/apd-schema.html`.

[12] Yannis Kalfoglou and Marco Schorlemmer. Ontology mapping: the state of the art. *The Knowledge Engineering Review*, 18(1):1–31, 2003.

[13] Alexander Maedche, Boris Motik, Nuno Silva, and Raphael Volz. Mafra - a mapping framework for distributed ontologies. In *EKAW '02: Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, pages 235–250, London, UK, 2002. Springer-Verlag.

[14] Natalya F. Noy and Mark A. Musen. Anchor-PROMPT: Using non-local context for semantic matching. In *Proceedings of the Workshop on Ontologies and Information Sharing at the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-2001)*, August 2001.

[15] Natalya F. Noy and Mark A. Musen. Evaluating ontology-mapping tools: Requirements and experience. In *In Proceedings of OntoWeb-SIG3 Workshop at the 13th International Conference on Knowledge Engineering and Knowledge Management*, pages 1–14, 2002.

[16] Natalya Fridman Noy and Mark A. Musen. Prompt: Algorithm and tool for automated ontology merging and alignment. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 450–455. AAAI Press / The MIT Press, 2000.

[17] Stanislav Pokraev. *Model-Driven Semantic Integration of Service-Oriented Applications*. PhD thesis, Telematica Instituut Fundamental Research Series, 2009. Forthcoming.

[18] Dick Quartel, Stanislav Pokraev, Rodrigo Mantovaneli Pessoa, and Marten van Sinderen. Model-driven development of a mediation service. In *EDOC '08: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 117–126, Washington, DC, USA, 2008. IEEE Computer Society.

[19] Dick A. C. Quartel, Stanislav Pokraev, Teduh Dirgahayu, Rodrigo Mantovaneli Pessoa, and Marten van Sinderen. Model-driven, Semantic Service Integration using the COSMO framework. Technical Report TI/RS/2008/031, Telematica Instituut, 2008.

[20] Dick A.C. Quartel, Stanislav Pokraev, Teduh Dirgahayu, Rodrigo Mantovaneli Pessoa, and Marten van Sinderen. Model-driven Service Integration using the COSMO framework. In *Proceedings of 7th*

International Semantic Web Services Challenge Workshop, in conjunction with the 7th International Semantic Web Conference (ISWC 2008), Karlsruhe, Germany, October 2008.

[21] Pavel Shvaiko. A classification of schema-based matching approaches. *Proceedings of Meaning Coordination and Negotiation Workshop at ISWC'04*, August 2004.

[22] Pavel Shvaiko and Jérôme Euzenat. A survey of schema-based matching approaches. *Lecture notes in computer science*, (3730):146–171, 2005.

[23] D.G.A. Stolp. Ontology Mapping – An overview of existing algorithms and tools, 2008.

[24] SWS Challenge. Main Page - SWS Challenge Wiki [online]. 2007 [cited February 2009]. Available from World Wide Web: `http://sws-challenge.org`.

Appendix A

Installation and operation instructions

This appendix describes how to install the owl2owl mapping editor, how to run it, and how to use the functions of the editor. It is structured as follows. Section A.1 presents the requirements for the editor. Section A.2 then describes how to obtain, install and launch the editor. Finally, section A.3 shows a basic, step-by-step instruction on how to operate the editor. These instructions assume a basic familiarity with Eclipse.

A.1 Requirements

- Eclipse version 3.4 needs to be installed. The Downloads page[1] of the Eclipse website contains a package titled **Eclipse Modeling Tools (includes Incubating components) (297 MB)**, downloading this version ensures that all required plug-ins are installed.
- Java version 1.5.0 (more recent versions of Java may work, but may also cause problems on some platforms).
- An SVN Client and access to the Telematica Instituut's SVN repository is required in order to obtain the latest version of the editor. A convenient method is to install the Subversive SVN plug-in for Eclipse. This page[2] describes how to install and configure Subversive.

A.2 Installation

1. Connect to the SVN repository and download all projects that are required to run the editor:

[1] http://www.eclipse.org/downloads/
[2] http://osgi.mjahn.net/2008/08/27/getting-svn-running-under-eclipse-34-ganymede/

- **nl.telin.amuse.owl**
- **nl.telin.amuse.owl.antlr**
- **nl.telin.amuse.owl.editor**
- **nl.telin.amuse.owl.mapping.owl2owl**
- **nl.telin.amuse.owl.mapping.owl2owl.editor**
- **nl.telin.amuse.owl.owl2emf**
- **nl.telin.amuse.owl.pelletjena**
- **nl.telin.swsc.parsers.mdsl**

2. Ensure that the run configuration is downloaded and installed. This is a file called **Owl2Owl.launch**, which is contained in the **nl.telin.amuse.owl.mapping.owl2owl.editor** project.

3. Note: the run configuration *should* be complete. However, minor differences between platforms, Java versions or individual Eclipse configurations may require that additional plug-ins are included in the run configuration. If problems occur when launching the editor, check if all required plug-ins are included in the run configuration. To do this, take the following steps:

   (a) Open the **Run** menu and select the **Run Configurations** option
   (b) Select the **Owl2Owl** configuration in the left menu, under **Eclipse Application**
   (c) Under the **Plug-ins** tab, click the button titled **Add Required Plug-ins**
   (d) Click **Apply** to save these changes, or **Run** to immediately run the editor

4. The owl2owl mapping editor can be launched by running the project using the Owl2Owl launch configuration.

A.3 Operation instructions

The following steps describe how to load two OWL files into the editor, how to create and edit mappings, and how to export these mappings to MDSL.

Load the ontologies

1. Open the editor and create a new empty Eclipse project
   This creates a new empty project, which is listed with a folder icon in the Project Explorer view

2. Import the OWL files into this project (on most systems, this can be easily done by dragging and dropping the files onto the project folder icon in the Package Explorer view)
   The two OWL files are now displayed under the project listing in the Project Explorer view

3. Convert the OWL files to the intermediate OwlCat format, by right-clicking the OWL files in the Project Explorer view and selecting the **Convert to OwlCat** command from the context menu
   The Project Explorer view now contains two additional files, one for each OwlCat file that was generated

4. Create a new mapping file for the two OwlCat models, by right-clicking one of the OwlCat files in the Project Explorer view and selecting the **Map to Owl** command from the context menu. In the File Selection dialog that pops up, select the other OwlCat file and click **Ok**
   The new mapping file now shows up as a new file with the owl2owl extension in the Project Explorer view. In addition, an instance of the mapping editor is opened, showing the two selected OwlCat models in the two tree views of the editor

Define the mappings

1. Navigate the ontologies to find two elements that need to be mapped. Start by right-clicking the **NavPath** element and clicking the **Unfold** command
   This makes it possible to unfold this node and show its sub-nodes, usually by providing a ⊞ icon or triangle that can be clicked to shows the sub-nodes

2. Continue unfolding elements in this way until you reach the elements that needs to be mapped
   This creates a tree structure of the two ontologies in the two ontology views

3. Create a mapping by selecting the two elements that need to be mapped, and clicking the **Create a new mapping** button in the editor's toolbar
   The Overview pane of the editor now contains a new entry, showing the selected elements in the Source Owl and Target Owl columns

4. Fill in the details of this mapping, by selecting the mapping in the Overview pane and editing the fields in the Property view. If the Properties view is not visible, then open it first by selecting the **Properties** entry in the **Windows** → **Show View** menu
   The Properties view now shows all the details of the specified mapping

5. Repeat steps 2–4 until all required mappings have been specified

Export to MDSL

1. To export the mappings to MDSL code, select the **Export to MDSL** command from the **Owl to Owl Mapping Editor** menu
   This shows a new blank window containing the MDSL code for the specified mappings. This code can be copied and used in other applications