**UNIVERSITY OF OSLO**
**Department of informatics**

# Semantic Model Annotation Tool

Master thesis

Odd Christer Brovig

**01.08.2008**

**Acknowledgements**

Thanks to my family and friends that have supported me through the process of getting this thesis done.

My gratitude goes toward Ismar and Trine who has had to suffer my rants, frustrations and outright nonsense on numerous occasions.

A special thanks to supervisor Arne-Jørgen Berre and Brian Elvesæter for fruitful discussions, tips and help during the process.

I would also like to thank to all the other people at SINTEF that have been helpful and accommodating.

Odd Christer Brovig
*Oslo, 1. august 2008.*

**Abstract**

Model driven interoperability is a research field that has gotten a lot of attention and money. Several research projects over the past years such as ATHENA, SWING and MODELWARE have sought to bring solutions to this field. The latest efforts involve the use of formal semantic descriptions to ease the interoperability task. Recent technologies on a lower level, such as SAWSDL has also presented new opportunities in the field. This thesis proposes to elevate the annotation to the platform independent level (PIM), complying with the Model Driven Architecture (MDA) vision of the MDE paradigm. Doing the annotation at this level will enable us to specify mappings to and from the ontology. Further this will enable us to generate specific mappings between PIMs that are annotated by, and mapped to, the ontology.

Benefits are many, especially in the fields of model integration, validation, constraint checking and the creation of a common vocabulary that the annotated models adhere to. The latter is especially an important property in a world that sees more distributed software development, where domain experts and developers often find themselves at the opposite sides of the world.

This thesis puts a focus on the *integration* aspect and shows how a annotated model can be transformed to another representation using the annotations and its definitions of the lifting and lowering operations. The thesis aim is to show that these annotations are feasible and represents added value in the context of MDA software development. We propose, and construct, a specific tool to handle annotation, mapping and validation of models and ontologies. They are annotated through a metamodel that support relations between both. This tool is based on the Eclipse platform, and is extensible with regard to future transformation technologies.

The proposal is evaluated in the context of a specific case that presents two domain models rooted in a buyer/seller context. These models are expressed as UML class diagrams. A reference ontology that is the equivalent of one of the UML models represents the common vocabulary that we want to connect the models to. We then evaluate the solution in the light of this and show that semantic annotation of models is possible and could provide assistance and concrete solutions to many problems that face developers.

# CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

CHAPTER

# ONE

# INTRODUCTION

This chapter provides the background for the thesis and introduces some of the concepts the thesis will be revolving around.

## 1.1 Thesis background

This thesis' theoretical foundation is in the domain of *Model Driven Engineering* and the *Semantic Web*. The subsequent sections will provide a brief introduction to each topic.

### 1.1.1 Model Driven Engineering

The Model Driven Engineering paradigm is not new, and has been around in various forms for the last 20+ years. The 80's offered CASE, Computer-aided Software Engineering, and over the last years, the hype has been so-called Domain Specific Languages. Although the means and challenges have changed, the goals are still the same:

- to provide an abstraction that allows computational independent design and architecture of software.
- to make software development more of an engineering discipline where finished components are assembled
- and to make the whole process of creating software more stable, precise and measurable.

### Model Driven Architecture

Model Driven Architecture(MDA) is the approach, and vision, from Object Management Group(OMG) on how model driven engineering should be executed. MDA consists of many different standards and meta models. A metamodel is, very simplified, a model that describes the available syntax for models instantiating it. The core functionality of MDA is provided by the Meta Object Facility(MOF) metamodel. This metamodel provides the structural features and vocabulary that most other metamodels in MDA are founded on.

The MDA approach identifies 4 model levels[23], shown in table 1.1. M0 is the *instance* level. The models here are instances of a model on level M1, e.g. generated Java code that reflects the expression in the M1 model . The M1 level contains a model, for instance a UML model that usually tries to capture the domain and describe a software system. Level M2 has the metamodel for the M1 level, i.e. it describes the vocabulary that can be used in M1. At the topmost level, M3 we find the meta-metamodel. This model describes what constructs that can be used to describe metamodels.

In MDA the MOF has been placed at the M3 level. Essentially the MOF is a self-describing model that in theory could be used to describe meta meta meta meta etc. models. Along with the model level separation comes the

| Level | Description |
|---|---|
| M0 | Instance level. E.g. Java code |
| M1 | Model level. E.g. UML diagrams |
| M2 | metamodel level E.g. UML metamodel |
| M3 | Meta metamodel level. E.g. MOF meta metamodel |

Table 1.1: MDA model levels

ideas of different viewpoints. The MDA approach identifies three viewpoints the models can take. The *Computation Independent Model*, CIM, prescribes that the model should not show a detailed description of structural features in the system. This level usually contains process descriptions and other abstract artifacts such as goal models and a preliminary domain model.

Further detailing of the system happens in the *Platform Independent Model*, PIM. This view describes the structural elements in greater detail, as for instance UML class diagrams do. Often it also provides mechanisms to describe behaviour in more detail (e.g. elaborating the processes from the CIM), for instance UML interaction diagrams or state chart diagrams.

At the lowest level we have the *Platform Specific Model*. At this level the system is described so that it can be transformed and deployed to a specific platform. An example could be the EJB stack from Java Enterprise Edition, where one has made e.g. a DSL to describe EJB based systems, or refined the model using UML stereotypes.

Essentially the three viewpoints gives us a crude methodology for how we should construct software — starting in the large with CIM and then refining the expressions at the level in lower levers such as the PIM, PSM and finally code.

To support automation one usually try to define clear cut mappings between the levels. This enables developers to build transformations that automatically transforms between the models. This can happened both vertically, as well as horizontally. Ideally the transformations should go both ways, i.e. allow round-trip engineering, but this is not always attainable, as the transformations loose information for each transformation step.

**Metamodels** As metamodels are the core of the MDA vision, it deserves some more attention. As mentioned earlier in this section, a metamodel is essentially a model that describes the available syntax for a instance of that model. Just like the Java language specification[1] that provides the syntax for the java language(albeit it is not like the models we deal with here).

Using UML to express a metamodel means that we also have the possibility to express constraints on it using Object Constraint Language(OCL). This, together with the model structure gives us the available syntax and some simple semantics for the model. We say that the metamodel provide the *abstract* syntax.

**UML Profiles** Having a abstract syntax in the shape of a metamodel is not that useful unless we create a *concrete* syntax. Today this is most often done either with UML Profiles or a Domain Specific Language.

A UML Profile is basically a extension of the UML metamodel. That means that we take the structural features present in that metamodel and extend them so that we inherit the same features, and gains the possibility to specialize these. This means that our metamodel will be tightly coupled to the UML metamodel, meaning that it has to be present when applying the metamodel in other environments.

[1] http://java.sun.com/docs/books/jls/

The extension makes the UML element extended upon a metaclass and the extender a stereotype. The latter can also have attributes in the form of tagged values. A tagged value is attribute with a name and a type, for instance we could add a tagged value to a stereotype called *logging* with boolean as a type. This would then indicate during code generation if the code from the stereotyped class should have logging enabled.

The main reason to develop a UML Profile is reuse and rapid tool development. Since one will extend upon the underlying UML model, there is already some syntax and semantics in place. This speeds up development at the cost of flexibility. It can also present a problem when the underlying semantics of UML are not properly understood or poorly defined. A second benefit is the ease of tool development. E.g. if one choose to build a profile upon UML Class diagrams, one will already have the tool support needed, with some adjustments to the visual representation. The other way of developing a concrete syntax for a metamodel is through a Domain Specific Language(DSL) which is described in the next paragraphs.

**Domain Specific Languages** A Domain Specific Language is, as the name implies, a language built for a specific domain. This domain could be anything, from the more general cases such as business process modeling to more specific cases where the language perhaps only will be used in one or few projects.

The DSL is constructed upon a metamodel that provides the abstract syntax. Using frameworks such as Eclipse GMF one can quickly build and generate visual editors that provides a concrete, visual, language to the users.

The creation of domain specific languages is not constrained to MDE, one can also use a language such as *Ruby* and do metaprogramming, essentially redefining parts of the language to suit ones needs. This is known as an *internal* DSL, where on adapts the host language to suit ones needs. The opposite is an external DSL, essentially the creation of a completely new language. Some of the most known projects employing (internal) DSLs in the Ruby world is Ruby on Rails[2] and RSpec[3]. The latter is a framework for creating tests for behaviour in a natural language (english), enabling greater readability and creating a environment for behaviour driven design(BDD)[4].

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

[3]http://rspec.info

[4]Behaviour Driven Design is a software construction method or technique where one focuses on the language and the interactions. It is usually used in combination with an ubiquitous language created with Domain Driven Design, see 2.1.4. More information on

Another approach taken in the Java world is to create DSL within the language with what is named as *fluent interfaces*.. This takes advantage of a feature that was added in Java 5: static imports. It works by always returning the modified object in a method call, effectively enabling a chaining of method call that allows the construction of sentences that are reminiscent of e.g. written english.

The creation of a DSL requires more work, since all the concrete syntax has to be constructed from the bottom up. But as said, in the previous paragraph, the use of modern framework and code generation helps reduce the time used on this task. It also gives the developer of the concrete syntax high flexibility on how the concrete syntax will look and behave.

### Architecture-Driven Modernization

While utilization of MDA is most often synonymous with developing new architectures and applications, Architecture-Driven Modernization takes on the opposite challenge of modernizing existing software.

In [17] the authors define three levels that are affected by modernization:

- Business Architecture: E.g. changing processes.
- Application & Data Architecture: E.g. changes in the architecture. Going from spaghetti code to a clean object-oriented design.
- Technical Architecture: E.g. changes in implementation language.

The former resides in the business domain, while the latter populates the IT domain. ADM is about doing a vertical transformation from A to B, crossing these domains. They explain the impact of this by a reverse horseshoe. the steeper the curve, the higher impact. The steepest curve will cross both domains and all three levels.

Modernization of existing software is often a challenging task, and ADM prescribes how to compose a clear vision of the existing system and transform it into a MDA compliant system that can be adjusted to new business requirements. This is described in [16] and involves:

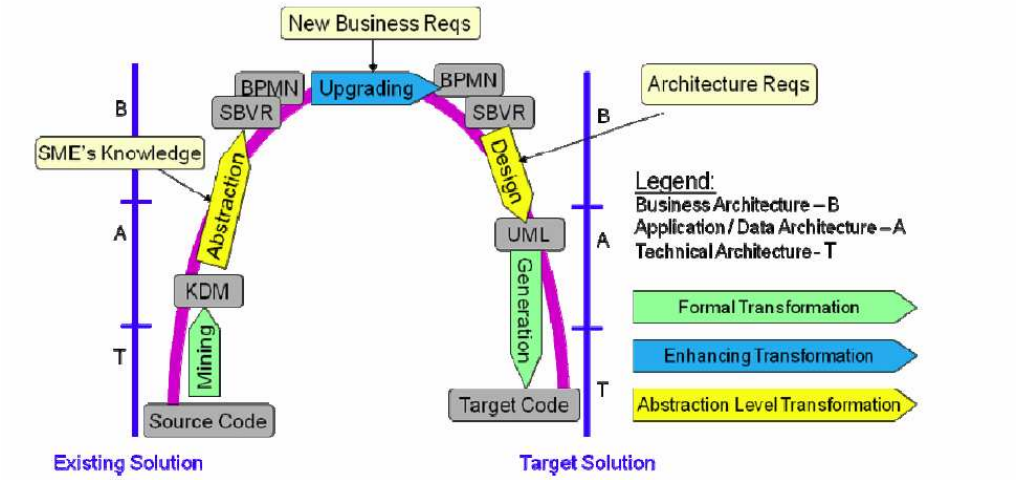
- Knowledge discovery through source code mining

BDD can be found at: http://behaviour-driven.org/

- Elevating knowledge and logic from knowledge discovery into abstract descriptions utilizing e.g. Business Process Modeling Notation(BPMN) and Semantic Business Vocabulary and Rules(SBVR)
- Specialization of the knowledge captured through UML
- Final transformation to new code

This forms the *ADM Transformation horseshoe*, which can be seen in figure 1.1.

**Figure 1.1** ADM Transformation horseshoe. From[16].




As we can see from this process, the discovery of knowledge in the existing software artifacts is vital to doing modernization of software. This implies also that the knowledge discovered should be expressed in a formal and comprehensible manner in the modernization effort of an ADM project, which again implies further that MDA systems would benefit formal semantics. This is further addressed in section 2.1.4.

### 1.1.2 The Semantic Web

The semantics "movement" has also been around for some time, but got a bootstrap when Tim Berners Lee published his now famous article *The Semantic Web*[5]. In this article he presented his vision and idea of providing

content and services on the *WWW* with meta-data; semantic descriptions. If implemented fully, it will allow us better and more advanced search capabilities as well as enhanced computability of the web.

At the same time as the semantic movement has gained traction, the system design paradigms also changed. It went from component-based architectures to Service Oriented Architectures, or SOA. A SOA can be realized in a number of ways, but the most common today is to realize it with web services. These services are usually described with a languages called Web Service Description Language(WSDL), expressed in XML. The WSDL contains a definition of the service. This definition covers the interfaces, data types and so-called grounding that says where the service is located, and what technology used to access it. The service messages are usually carried over HTTP, encapsulated in Simple Object Access Protocol(SOAP) envelopes.

Another web services approach that has gained traction over the past years is REpresentational State Transfer(REST). REST aims to utilize the power in HTTP, and use its operations GET, PUT, DELETE and POST to its fullest extent. This way of doing services is said to be resource-centered, i.e. the domain model is exposed over HTTP, and accessed through URLs that has namespaces that correspond with the model. An example url of a resource could be: *http://www.foo.bar/user/baz*. To manipulate the model the client would use the HTTP operations and e.g. use POST to persist changes about user information.

As can be seen with the above paragraphs there's a lot of syntax involved in utilizing the services, but not so much semantics. Of course, we have the semantics of the protocols involved in transport, data exchange and so-forth, but we lack the semantic of the service itself—we want to know what it does, and how it does it. Does it interact with other services? Does it guarantee a response time? What is the precision of the response? And so forth. We also want to know the semantics of the information it consume and/or produce.

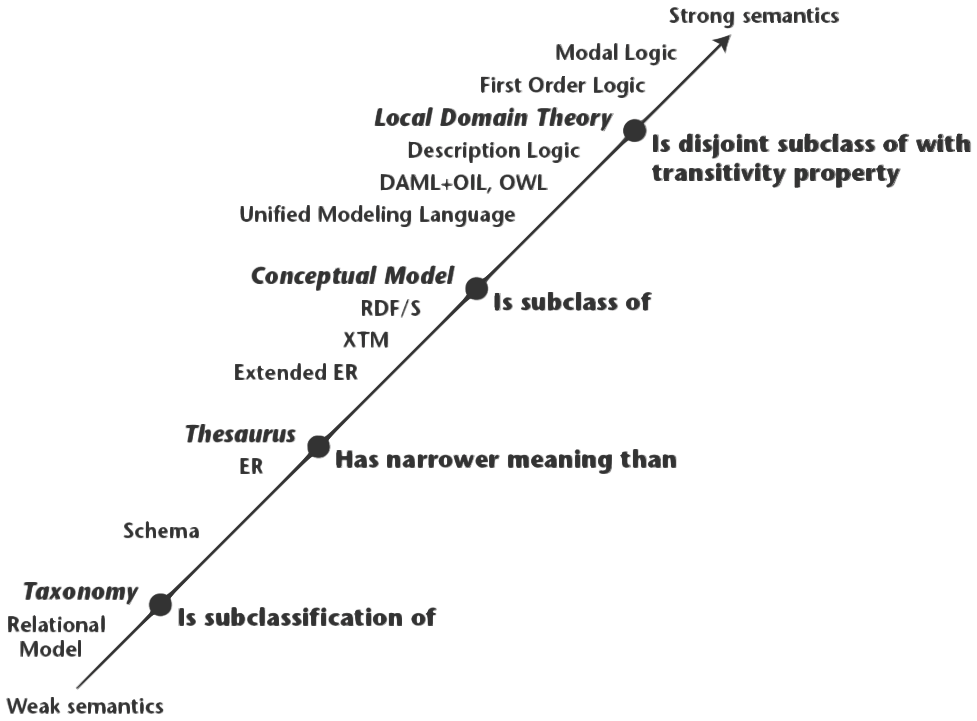
At the same as the semantic web vision came into existence, researchers saw the possibility to provide more metadata for services also. The idea was that this would allow for greater automation of the services, effectively allowing run-time mediation, composition and also greater findability.

To achieve this, the semantics has to be computational. As an effect of the Semantic Web Initiative at W3C, several technologies were developed. The core part of this is the Resource Description Framework[27]. This framework provides mechanisms to describe web resources. Extending upon RDF we have the Ontology Web Language(OWL), that provides the extra pieces needed to create ontologies.

### Ontologies

Ontologies is a knowledge representation system that allows us to express knowledge through logical constructs. The "weaker" variant of this is taxonomy, a classification scheme. For instance Carl Linnaeus created a method for classifying all living things, which is called a Linnaean taxonomy. The word taxonomy itself stems from the greek words *taxis*, meaning order, and *nomos* which means order and law, i.e. scientific law or order. The organized items in a taxonomy is has a parent-child relationships. This means that all constraints and properties of a parent is inherited in the child. Although there is a strong semantic binding in the relationships found in the taxonomy, it is not sufficient. We can refer to this a as *weak* semantic compared with ontologies, as we can see in 1.2. This is also evident in figure 1.3 that shows a spectrum of knowlegde management systems, where ontologies exist to the right of the red line, clearly distinguised from less formal systems. To gain the extra formality we turn to ontologies.

**Figure 1.2** Weak to strong semantics. From [8]




Merriam Webster states[22] that ontologies are:

> a branch of metaphysics concerned with the nature and relations of being

While Tom Gruber, in [14] provides a definition for domain of computer science:

> In the context of computer and information sciences, an ontology defines a set of representational primitives with which to model a domain of knowledge or discourse. The representational primitives are typically classes (or sets), attributes (or properties), and relationships (or relations among class members). The definitions of the representational primitives include information about their meaning and constraints on their logically consistent application.

From this definition we learn that ontologies are formal. This means that we will be able to carry out computations on them. For instance Concepts [5] can be related to each other. What separates relations in ontologies from other ways of capturing domain knowledge, e.g. UML Class diagrams, is that the relation also will have its own semantics. Whereas the semantic of e.g. a UML association is defined in the UML standard, ontological relations are defined by the ontology creator.

An example could be an relation, or association in UML class diagrams, named parentOf. With the UML association, the semantic of the name is implicit. Defined in a ontology the semantic would be unambiguous, the what the relation holds for would be defined in a computable axiom.

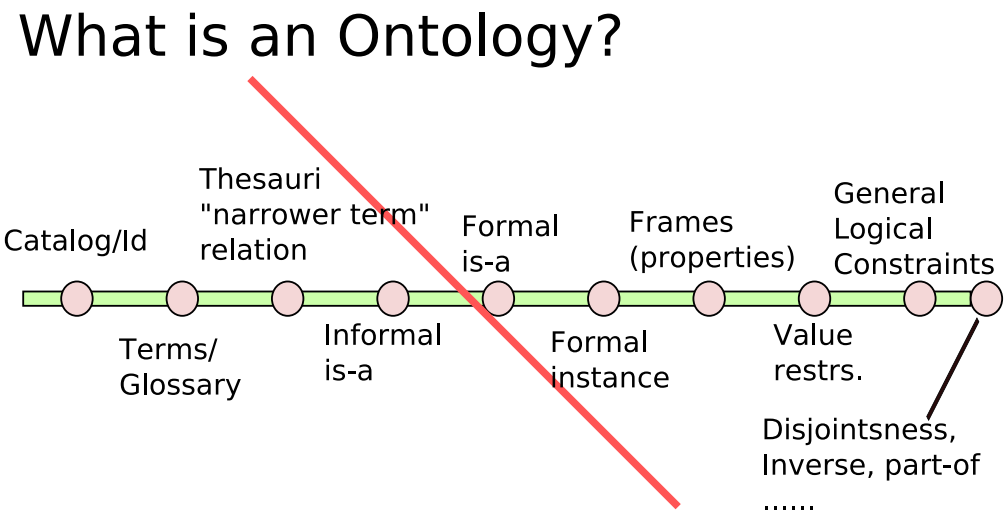
This means that the relation can be reused so that the knowledge captured in the relation can be used in other parts of the domain, or in other domains and still have the same meaning. It is also possible to impose rules, called axioms on the relations. These are truths that must be maintained for the relation to hold, somewhat similar to the use of OCL in UML. The result is a comprehensive representation of a domain, both its concepts, *Individuals*[6], relations and the rules that dictate these entities and relations.

Another distinction from UML class diagrams is that *Properties* are first class citizens in ontologies. They are concepts too, in the sense that they carry logical constructs. This makes them hard to map, since UML class attributes/properties are "dumb"; they belong to the class itself, and have no behaviour.

[5] *Concept* will be used instead of *Class* in this thesis.

[6] Instances of Concepts

**Figure 1.3** What is an Ontology. From [20]




Several languages exists to represents ontologies, one of the most popular is the Ontology Web Language, OWL[4], a recommendation from the W3C. OWL extends the Resource Description Framework - Schema, RDFS[27], and allows for greater expressivity.

### Resource Description Framework

Resource Descriptor Framework(RDF) is a framework built towards the idea of providing metadata for web resources. It does not only itself provide a way to build these relations, but also a foundation that can be extended for various needs.

The RDF relations are called *triples*. The triples makes statements about relationships resources have, and contains a subject, predicate and object. For instance, the statement: This document' author is John Doe, would have a subject *document*, predicate *author is* and object *John Doe*. Since RDF is a abstract model there exist several serialized formats. Of these, there a two formats in common use today: N3 and XML. An example of the same data using N3 and RDF below[7]:

For N3:

[7]Examples from: http://www.w3.org/2000/10/swap/Examples

```
@prefix p:  <http://www.example.org/personal_details#> .
@prefix m:  <http://www.example.org/meeting_organization#> .

<http://www.example.org/people#fred>
p:GivenName   "Fred";
p:hasEmail  <mailto:fred@example.com>;
m:attending  <http://meetings.example.com/cal#m1> .

<http://meetings.example.com/cal#m1>
m:homePage  <http://meetings.example.com/m1/hp> .
```

and for RDF:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:m="http://www.example.org/meeting_organization#"
    xmlns="http://www.example.org/people#"
    xmlns:p="http://www.example.org/personal_details#">

    <rdf:Description about="http://meetings.example.com/cal#m1">
        <m:homePage resource="http://meetings.example.com/m1/hp"/>
    </rdf:Description>

    <rdf:Description about="http://www.example.org/people#fred">
        <m:attending resource="http://meetings.example.com/cal#m1"/>
        <p:GivenName>Fred</p:GivenName>
        <p:hasEmail resource="mailto:fred@example.com"/>
    </rdf:Description>
</rdf:RDF>
```

As we can see, the N3 format is easier to read, and more terse compared to the XML representation in RDF.

Today, the most prominent uses of RDF is RSS and in the Dublin Core Metadata Initiative[8]. The latter is a ISO-standard for interoperable metadata and specialized vocabularies for describing resources. Dublin Core uses RDF as one of its formats.

RSS on the other hand stands for RDF Resource Summary[9], and is used to summarize updated information from resources, often blogs or news head-

[8] http://dublincore.org/

[9] Applies for RSS versions 0.9 and 1.0. Actually the acronym RSS denotes three standards for publishing updated information.

lines. Usually it is read in a special software called RSS reader or aggregator, that pulls the RDF descriptions from a URI and presents the headlines and/or the content to the user.

#### Ontology Web Language

Ontology Web Language[4], OWL, is a language for knowledge representation in the form of ontologies. It is composed of three different "modules":

- OWL Lite
- OWL DL
- OWL Full

OWL Lite and DL share the same semantic basis, they are both based on description logics. Description logics(DL) are, as the name implies, based on descriptions, and handles the manipulation of complex predicates. Descriptions in DL are composed of *concepts* and *roles*, where concepts will be categorical nouns, such as Pilot, Adult and Teenager. These describe basic classes of objects. Roles on the other hand are relational nouns, such as Age, that describes objects that are parts of or properties of other objects.

The distinction between OWL Lite and DL is that Lite only uses some of the features of that exist in DL and Full, and that these features also have usage limitations, e.g. in the use of classes and a limited notion of cardinality. The differences are described further in [21]. Both OWL Lite and DL can be expressed in a restricted view of RDF.

OWL Full can, on the opposite, be expressed with all available features of RDF. OWL Full also features the most expressivity, but this comes at a cost. OWL Full doesn't guarantee computationality. It is not likely that reasoning software will support complete reasoning for all the features in OWL Full.

A more thorough look at OWL is presented in chapter 3.

### 1.1.3 MDE and ontologies

#### Ontology Driven Architecture

The W3C has looked at a combination of model driven engineering and the use of ontologies, in their note "Ontology Driven Architectures and Potential Uses of the Semantic Web in Systems and Software Engineering"[26](ODA).

They argue that

> The suite of standards supporting the MDA initiative streamline the mechanics of managing and integrating models of various aspects of a system's design, but say little about the underlying semantics of the domain being modeled.

and that:

> Formal representation can help to limit ambiguity and improve quality not only in automation of business semantics but in overall engineering of complex systems.

The main motivation of combining MDA and The Semantic Web is stated as that MDA provides the mechanics necessary to manage and integrate various modeled aspects of a systems, but that is says little on the semantics of the domain that is modeled.

They contend that formal definitions will limit ambiguity and improve the quality of the software, ranging from automation of business semantics to the construction of complex systems. The downside is that with higher formality, the tool support will have a higher abstraction too. This can lead to difficulty in constructing tools that can support the users in a good way, and limit the freedom of expression.

On the other end of the scale is of-course the less formally defined systems that can suffer under high ambiguity, even if the documentation is good. This is especially true if the person reading the documentation and models is not accustomed to the problem space.

A prime example of an area that can benefit from formal definitions if that of matching component interfaces. This involves defining pre, post and invariants to specify the behaviour of the component the interfaces exposes under various circumstances. This can be hard in the MDA vision alone, due to the scalability issues that arise as the rules increases, also the lack of unambiguous presentations present a problem. Integrating semantic web technologies in this problem space would impose unambiguous representation of the domain, automate consistency checking, validation and support mediation and transformation when composing components, based on the domain knowledge.

The authors identifies three main applications for semantics in model driven engineering:

- Ontologies as formal model specifications

- Software life cycle support
- Corpora[10] of reusable contents and the use of metadata as relational data

Some of the benefits of using ontologies as formal model specifications has been elaborated on earlier in this section, but the paper give a few more to ponder on. Within the area of quality, the use of formal model specifications can, amongst others, improve consistency, better typing and categorization, better communicate requirements between domain experts and developers and increase the potential for reuse, substitution and extension of software via "accurate content discovery on the Semantic Web".

In the field of software life cycle support the integration of semantics is expected to reduce the problems of inconsistency between different artifacts generated during the development life cycle. Inconsistencies here arise due to the fact of different perspectives on the domain between different stake holders in projects. Also semantics can enable better enforcement of traceability of the artifacts, augment sharing and interoperation of models made by different stake holders.

Last, the "corpora of reusable contents and the use of metadata as relational data" is about utilizing the semantic web to describe composites based on participating objects in runtime and artefact sharing systems. This differs from the normal way of identification in the semantic web, that is normally conducted through functional properties that are inverse, such as Friend of A Friend(FOAF). The aim is to provide better findability through these special relations. This would in turn provide for e.g.: formalization of associations between sub-components in a system, a framework for sharing of runtime data and

## 1.2 Methodology

The thesis will be carried out as a partly theoretical work and partly as a practical work. The process will be iterative and incremental, meaning that parts of the theoretical work and practical work will influence each other — from start to completion. The benefit of doing the work this way is that is allows us steer of dead ends in a better way than ditching large quantities of work. Also, since the practical work in this thesis is software development,

[10]From Merriam-Webster: a collection or body of knowledge or evidence

it pays of to be as agile as possible with regard to what can be accomplished within the time frame of the thesis. As Piet Hein put it:

> The road to wisdom? Well, it's plain and simple to express: Err and err again but less and less and less.

The theoretical work will be carried out in a qualitative fashion by literature reviews, discussions and evaluations of the literature against the arguments and thoughts presented in the thesis. The practical work will focus on implementing the ideas described in the thesis and providing a basis which upon the problem definition and solution requirements can be tested, evaluated and hopefully validated.

### 1.2.1 Literature reviews

The process of literature reviews has been iterative. Literature has been by and large retrieved by searches on either ACM, IEEE Xplore or Google Scholar and citations in previously found literature.

### 1.2.2 Implementation

The implementation work will as said earlier be carried out iterative and incremental, both influencing and being influenced by the theoretical work. Doing the development this way avoids a waterfall process where a large specification is hammered out in advance without enough knowledge of the problem domain. Prototyping is carried out, both on paper and with the use of computer. This serves to create a quick and tangible perception on how a idea might work out when it is (semi) formalized.

The process loosely followed in this thesis has been using the requirements set forth in chapter 2 as kind of *user stories*, that has been prioritized and edited as required to satisfy the new knowledge discover during the work.

## 1.3 Thesis overview

In chapter 2 we learn about the problems their associated domain that the thesis will try to provide a solution for. Prior associated solutions and standards are discussed in chapter 3. Chapter 4 introduces a proposal that will

try to solve these problems. Chapter 5 describes the application of the solution to the evaluation cases, evaluating it against the requirements presented in this chapter. These are then evaluated in chapter 6. At last a final conclusion and suggestions for further work is provided in chapter 7.

CHAPTER

# TWO

# PROBLEM DEFINITION

This chapter presents the problem definition and scope for this thesis. It also presents an evaluation/use case. The use case is used in the evaluation of the solution introduced in chapter 4.

## 2.1 Interoperability and its challenges

### 2.1.1 Introduction

What is interoperability? Tanenbaum defines[3] interoperability as:

> Interoperability characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard.

and IEEE states[2] that interoperability is:

> the ability of two or more systems or components to exchange information and to use the information that has been exchanged

These days integration of information systems is as big as ever. The spread of the internet, XML as an open and extensible information exchange format, and the introduction of web services has resulted in a ever-growing market

for services distributed over the internet. Companies such as Amazon.com[1] and Salesforce.com[2]has opened up and offers interfaces for integration of, through web services, their information and services.

This empowers developers all over the world to create new, or integrate existing solutions with these services. The openness of the technologies involved has allowed this. W3C[3] has published numerous recommendations, and OASIS[4] has provided us with a reference model for what SOA is and means.

As it stands, the purely technological challenge of integrating it-systems has advanced tremendously in recent years. But challenges remain, and one specific challenge is to increase the interoperability and integration of information, services and processes across domains and organizations.

### 2.1.2 Information interoperability

Knowledge of a certain domain, is often encapsulated in the organization that offer the services for utilizing it. If this knowledge isn't properly conveyed the users of the service, it can be hard to take advantage of the service. This, in turn, can lead to low adoption rate of the services and as a consequence developers will leave the service for better offerings from competitors.

Most information is relayed between different systems without any associated semantics other than what developers decide them to be—at the opposite ends of the "pipe". This goes for both the information being exchanged, and the semantics of the service itself. This creates a need for manual labour, resulting in slower development time and higher software cost. In many cases this very problem could have been solved (partially) automatically through the means of annotations to or embedment of the formal *semantics* that define the service and information it provides/consumes.

In [12] Goh identifies three main causes leading to semantic heterogeneity:

- Context
- Scale
- Naming

[1]Amazon Web Services - `http://aws.amazon.com`
[2]Salesforce.com - `http://www.salesforce.com/developer/`
[3]World Wide Web Consortium
[4]Organization for the Advancement of Structured Information Standards.

These three causes leads to problems during interchange of information. The first, *context*, tells us that information can be interpreted in various (temporal) contexts that the originator of information doesn't have control over, e.g. due to different cultures, laws and norms. For instance, the swastika is a sign for good luck in asian culture. In western culture it is highly stigmatized due to the use of the symbol in nazi-Germany. This is an example of a differing cultural context.

*Scale* is the second cause of heterogeneity. For instance information does often not contain any information about what scale a number is expressed in relation to, or what currency it has. Different countries use different standards on how to express different numeric values such as time and date and currency. Exchanging information crossing these boundaries goes wrong without the proper semantics defined and understood.

Third, *naming* conflicts can lead to ambiguity when interpreted by a third party, because they will not be accustomed to or know about the meaning and/or differences of the names. How is a third party to implicitly know what the word *accommodation* means housing in british-english and way of public transportation in american-english? Or that a *brew* means tea in England and beer in the US? These are examples of naming conflicts, where the same word means different things for different interpreters. It is also an example of a contextual conflict.

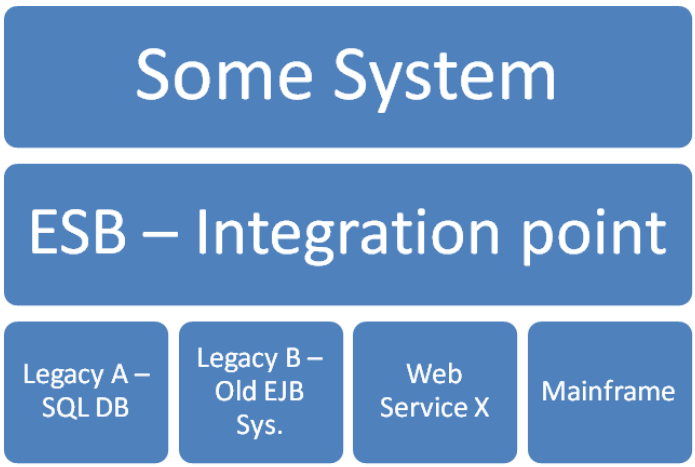
A compound example of this can be the statement that[5] "Jones' salary is 2000". This statement is ambiguous and has several problems. First, we don't know what currency, or what scale the amount is in. It could be in yen, dollars or norwegian kroners, and the scale could possibly be 1 or a 100. Second, does it include any bonuses or benefits? We can't possibly know, and this is important in many countries due to taxation. And what periodicity has the statement? Is it yearly, monthly, weekly or something else? As we can see, this statement has a lot of room for different interpretations, especially if we cross borders and languages, which is quite common in corporations today. These problems present themselves clearly when we are tasked with the integration of system that transcend borders, governments and corporations. Especially today when there is a heavy emphasis on reporting, e.g. utilizing so-called Business Intelligence.

[5] from [12]

### 2.1.3 Interoperability between systems

Much of the integration work done today is done using a technology/pattern called Enterprise Service Bus(ESB). ESBs are essentially *the* integration point. E.g. if one has a silo application[6] environment, the ESB enables interoperability and integration by being the coordinator and transformer of messages. These messages can be sent to and from systems using widely different technologies, such as FTP, Message Queue, Web Services and RPC. For all intents and purposes the ESB acts as a abstraction layer. In theory (if the ESB is modular enough) one could completely replace a (say technological old) system with a new one on one end and the system at the other end wouldn't notice anything except perhaps some downtime. See figure 2.1.

**Figure 2.1** Enterprise Service Bus Integration Architecture




The ESB is absolutely not a silver-bullet. Scaling will become a problem as the number of transformations grow as we add systems to the bus.

[6] A silo application refers to a system doing one specific task. Since the systems often lack horizontal integration, silo application often end up duplicating functionality.

Another area that has a need for interoperability is sensor networks. This is an area that has seen a lot of attention, especially as the military is very interested in this technology. Sensor networks are networks composed of sensors that are, most often, geographically distributed. They need to interoperate to create networks between them, coordinate and reduce load on the network while running applications and so forth.

At the moment, the sensors deployed are usually very specialized, and interoperate in a prescribed manner. They have their protocols and message exchange formats pre-programmed. If one takes the idea of creating ad-hoc networks with different devices a step further one realizes that interoperability is not just "nice to have", it is a necessity.

For instance, most mobile phones today are sold with Bluetooth in addition to the regular networking capabilities (GSM/EDGE/3G). These devices are usually tightly bound to the software supplied from the vendor. One could imagine that a common standard could be created that allowed these devices to interoperate seamlessly. The network standards deployed is one step on the way, but building services on top of these is a completely different matter.

With the introduction of "smarter" phones such as the Apple iPhone and the phones, likely to be released in 2008, deploying Google Android the opportunities for building e.g. some sort of mesh/P2P network is growing.

Another technological paradigm that would benefit from higher interoperability is *event driven* systems. These systems consists of event producers and consumers, where a event is a significant change in the state of an object. A example of an event can be a change in a stock price. The event would basically compromise the new price and ticker of the stock. It could be published in a message queuing system such as JMS[7] or in a space based architecture. The benefits of adding some form of semantics to the events, is that is would enable the consumers of events to reason about them before acting. Of course there is the implicit semantics already present in the event, in our example it is that the stock price in the event is a change from an earlier price. This is a pretty banal example, but since the event lacks any formal definition of what a price is, it could be misinterpreted. A slightly more advanced example is that of for instance a scenario where nodes in a distributed systems announce capabilities. E.g. they announce that they have 1 hour of spare cpu-time to offer, and the consumers of those events would then be able to schedule work for that time. Semantics here would allow a formal definition of what work that could be performed, any constraints and so forth. This could of course, perhaps more easily, be presented in a

[7]Java Messaging System

proprietary metadata format, but it would require interested consumers of the service to implement parsing of that special metadata. Variances in the metadata would also require re-engineering of the consumers to adapt, contrary to a formal definition with logics that merely should require a dynamic re-adaption[8].

From both these examples we see that there is still a need for clearly defined semantics for both information and processes to allow for automation of both finding services, agreeing on interaction and transforming information. With (de-facto) standardized technologies such as RDFS/OWL from W3C[9] this is also starting to be a reality. At the same time, software engineering technologies such as MDA has made headway. The question then looms—is there any overlap or connection between the two areas?

The answers is yes. Through the projects, standards and technologies discussed in chapter 3 we are seeing integration of semantics and model driven engineering technologies influenced by the MDA vision.

### 2.1.4 MDA and semantics

This thesis will focus on a specific aspect of this interplay that has seen little work. The use of ontologies as a *shared vocabulary* between several Platform Independent Models. The importance of a shared vocabulary is discussed at length by Eric Evans in [11], and semantic technologies (such as ontologies) is just one, but powerful, way of capturing this vocabulary. Evans argument is that shared vocabularies allows the domains of discourse to be properly communicated between the involved parties, usually comprised of non-technical domain experts and technical experts (developers). Without a shared vocabulary the definition of the domain will be biased, creating ambiguity. This ambiguity can both present itself at the same level, e.g. using different terms to express the same idea and therefore failing to communicate, or in different abstraction levels. Developers can for instance be at a conceptual level that is more entangled with the actual implementation, while the domain expert is at the "business" level, worrying about processes and abstract relationships in the domain.

So, in an effort to minimize the gap between the experts in both camps, Evans suggests the development of a *ubiquitous* language, i.e. a shared vocabulary. This is essentially a representation of the domain that both parties agree on, and must corporate on to change. This vocabulary is not rooted in

[8]Provided that all providers and consumers share the same semantics

[9]Recommendations.

a certain language or visual representation, but Evans uses simplified UML class diagrams in his book as an example. Together the two camps explore the domain and creates diagrams and textual descriptions that capture the concepts, behaviour and relations found in the domain. Further, the shared vocabulary is should be rooted to the actual implementation, as early as possible. This ensures that the two representation of the domain don't diverge from each other.

Doing development this way, goes a long way to ensure that we end up in the favorable situation where the development team and domain experts speak the same ubiquitous language without ambiguities. In relation to MDA we can see that a shared vocabulary can be very beneficial for the development with models, some of which can be very complex. The shared vocabulary can of course be manifested by the models themselves, but for the domain experts this is something that also can be a bit complex. A more lightweight, less complicated approach, would suit the communication better—with the appropriate bindings to the implementation models.

As described in section 1.1.1 the PIM is an abstraction layer, positioned between the CIM and PSM layer. This enables MDA practitioners to vary the concrete implementation of a software system independently from the CIM, and allows them to modify their concrete implementation of the software without (in most cases) being dependent on changes in the higher (PIM and CIM) levels. To allow for easier, and automated, interoperability between the concrete systems at the PSM level we need to ensure that they share the same vocabulary. This can either be done by retrofitting semantic annotations to either the PSM model level or the generated code. Another, perhaps better, method is to do it at the PIM level.

The inclusion of semantics at the PIM makes it easier to generate interoperable systems, but it also, more importantly, allows for interoperability between the models themselves.

A model practitioner could be faced with a integration task involving his own model and a "foreign" model[10] faces the same problems as a systems developer trying to integrate to legacy code based systems with poor documentation. The exact semantics in the models is not defined, and as a consequence both the practitioner and developer will use time to map between the systems/models and resolve conflicts. This hassle could have been removed, or made less burdensome, if the semantics were defined at either levels.

[10]possibly only just a skeleton with defined interfaces, generated from e.g. WSDL

For instance, a semantic match-maker could search for PIMs with matching interfaces, or suggest mappings/transformations automatically when needed. This in turn would allow for transformation to target PSMs, such as one for an ESB that included the relevant mappings and transformations for run-time interoperability.

The problem statements this thesis seek to answer is:

*Will integration of semantics, through the means of ontologies, allow greater interoperability?*

and:

*Will the integration of semantics allow easier transformation of models?*

and:

*Will the integration of semantics provide better validation of models?*

To thoroughly answer this question this thesis evaluates different approaches, part and whole solution in chapter 3. To vet the idea properly, a proposal for a solution is made in chapter 4. To help verify that the proposal addresses the said problems, and that it is done in a proper way the next sections define:

- evaluation cases - provides scenarios where the solution will be applied for evaluation and tested against
- solution requirements - statements about what the proposal will address and how it should be addressed

## 2.2 Evaluation cases

### 2.2.1 Domain model with UML class diagrams

This use case is simple and serves to test the basic functionality a solution need to deliver. The use case will test the ability of a given solution to annotate classes with semantics and allowing the classes, and their features, to be re-conciliated.

#### Evaluation case design

The evaluation case will be based upon a classic buyer/seller example case. It has, as earlier mentioned, two different domain models created with class

diagrams, as well as a model expressed as an ontology. The challenge is then to be able to transform from one of the UML models to the other, using the ontology as a common vocabulary.

The domain model for UML model A looks like figure 2.2:

The domain model for UML model B looks like figure 2.3.

The domain model for the Ontology is presented in figure 2.4. The ontology is very basic and just resembles the structural features present in Model A. The different classes are related to each other using simple *ObjectProperties* that have no more than the *domain* (source) and *range* (destination) in their expression.

**Figure 2.2** Buyer/Seller UML class, diagram example A

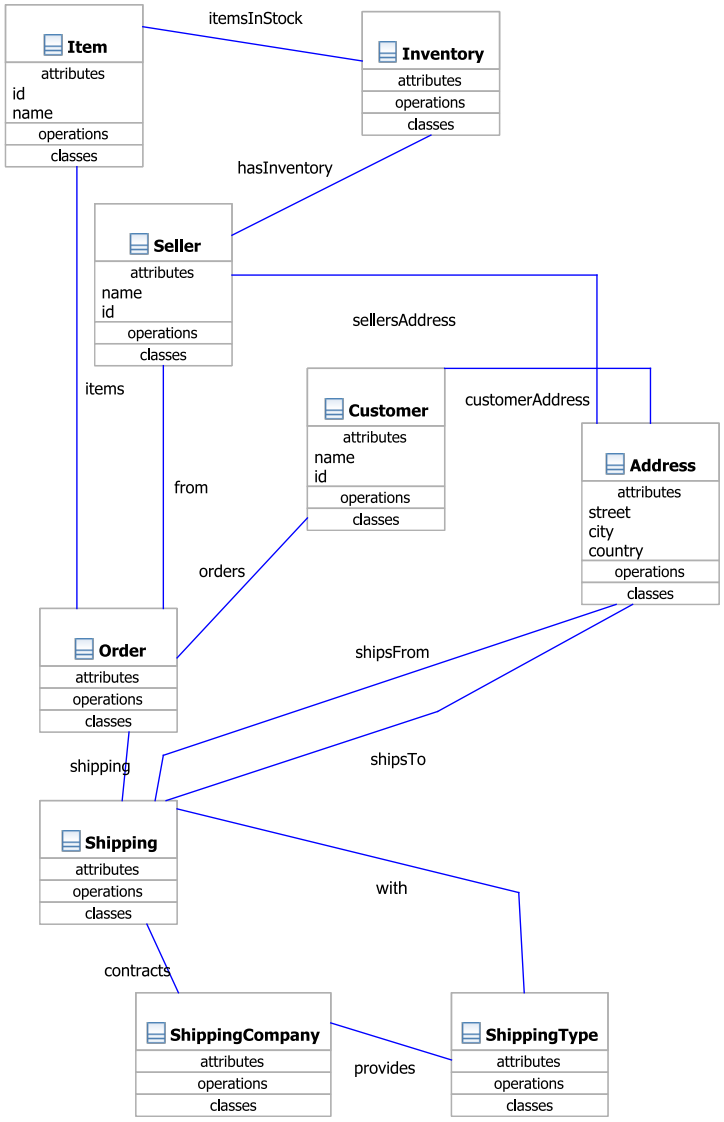


**Figure 2.3** Buyer/Seller UML class diagram, example B

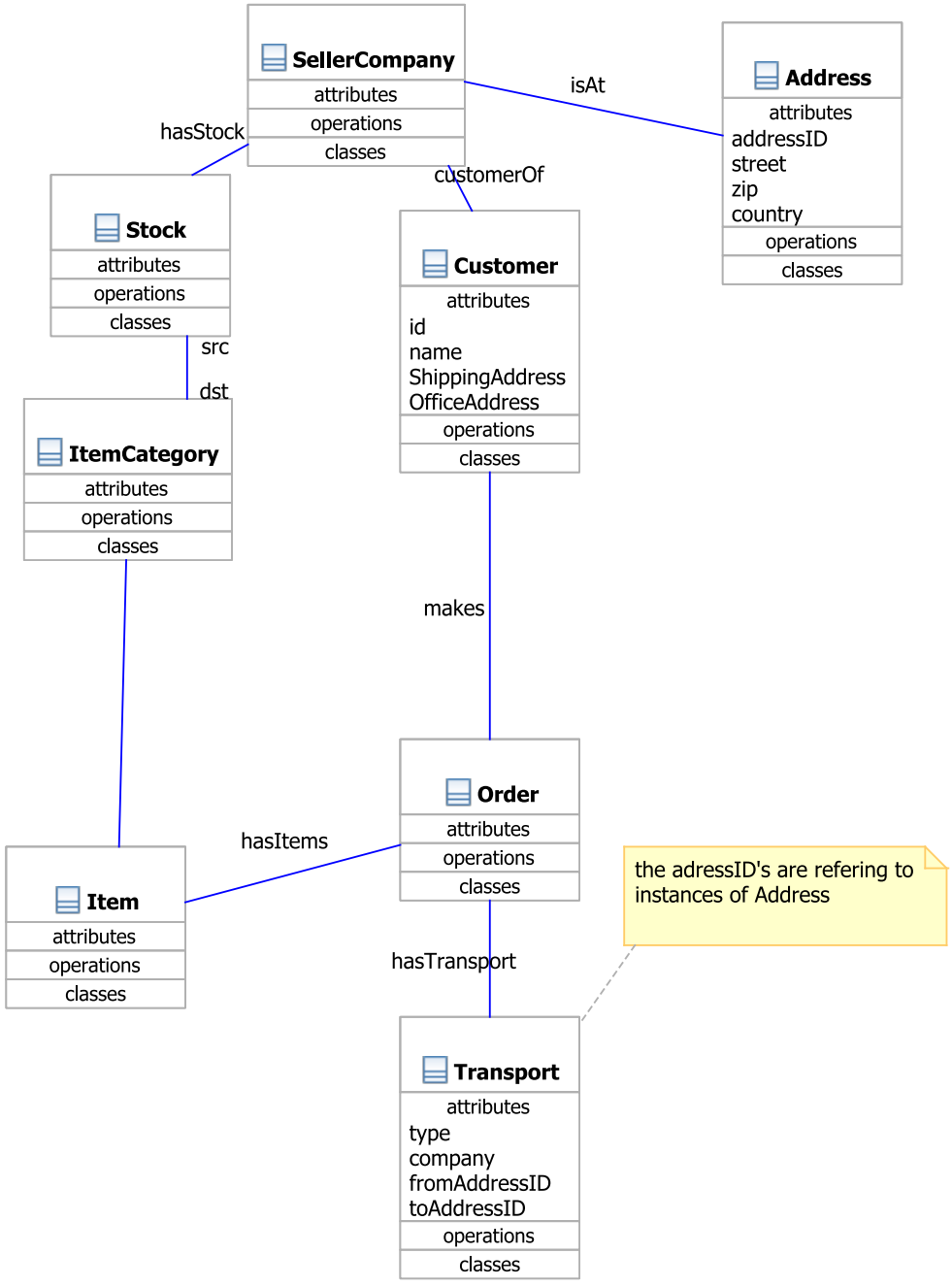
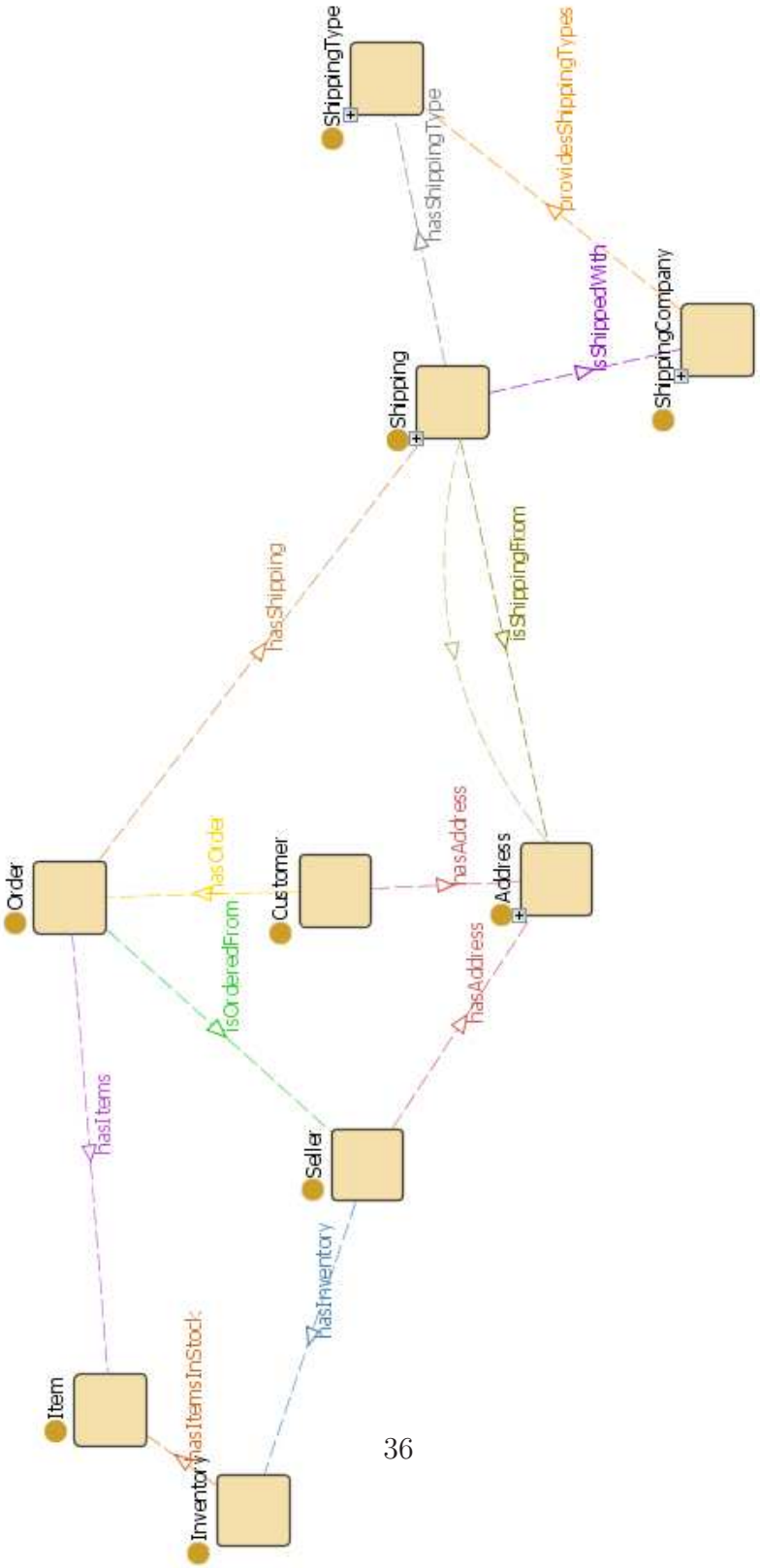


**Figure 2.4** Buyer/Seller ontology example. Created in Protégé



A snippet from the OWL code showing an object property:

```
<owl:ObjectProperty rdf:ID="hasAddress">
    <rdfs:domain>
        <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#Customer"/>
                <owl:Class rdf:about="#Seller"/>
            </owl:unionOf>
        </owl:Class>
    </rdfs:domain>
    <rdfs:range rdf:resource="#Address"/>
</owl:ObjectProperty>
```

and a class/concept:

```
<owl:Class rdf:ID="Seller"/>
```

As can be seen for both models, they are not complete, and perhaps not entirely realistic. They have been derived from real examples in the ATHENA project and the UBL standard. To test the basic scenarios, cardinality and types has been omitted— both in the UML models, as well as in the ontology. The most important thing is that they in fact could be encountered in similar forms in the wild. In our case, we do not need complete or completely realistic models. What we need is the required situations that we need to evaluate the solutions to later in the thesis.

Model A is a somewhat more object-oriented, or normalized in relational sense. It has somewhat separation of concerns. In model B on the other hand, we have several cases where attributes and what would be independent attributes or classes are mingled. To be able to do integration of information exchange between the two models the following cases would need to be solved:

1. Concatenated string will need to be split according to a separator or by index. Examples are: *ShippingAddress* and *OfficeAddress* in *Customer* from *model b*

2. Classes that are in-cohesive, e.g. they model more than one concern will need to be split up. An example is the class *Transport* from *model b*. It contains both type information, company and address. To conform with *model a* it would need to transformation to that structure.

3. Naming differences need to be resolved according to the references ontology. Example is *Stock* in *model b* and *Inventory* in *model a*. The reference uses the word *Inventory*.

## 2.3 Solution requirements

The previous section established several concrete mapping cases that the tool need to support. The requirements that stem from these are expanded upon in this section. In addition some general requirements are set forth. The requirements are both of non-functional and function character. The requirements are summarized in table 2.1 in section 2.3.7.

### 2.3.1 Shared model driven vocabulary

The tool need to support a shared vocabulary. Since the tool will need to support several different types of input models, this vocabulary needs to be implemented in a language that supports all the formalisms in these models.

The tool shall implement an ontology infrastructure, that can support solutions such as OWL for ontology development.

The tool will support this infrastructure through the use of either one or more properly documented metamodels.

### 2.3.2 Visual tooling for vocabulary development

The tooling will support an easy formalism for developing and/or editing the shared vocabulary. The tool will have emphasis on a quick process to develop the domain model.

Some possible formalisms here are simplified class diagrams, topic maps and concept maps.

### 2.3.3 Horizontal mappings between models

The tool need to support horizontal mappings between different models related to the shared vocabulary.

This involves supporting different operations so that the mapping will "raise" and "lower" the structural features present in the model so that they fit with the shared vocabulary concepts that they are connected to.

Implicit through the mapping up/down to/from the shared vocabulary one shall be able to generate a mapping model that exist on the model level.

In this regard the shared vocabulary becomes a “hub” model.

### 2.3.4 Vertical mappings to PSM/Code level

The tool need to have support for transformation technologies that enable the tool to transform the aforementioned implicit model to PSM or Code. The technologies here should be interchangeable as long as it adheres to a specific interface.

### 2.3.5 Open implementation environment

The environment for implementation shall be open, in the sense that the platform is under a open source license that allow both for commercial as well as private use and further development.

### 2.3.6 Extensible implementation

The implementation shall be extensible to cater for future work. This involves the implementation to be extensible in a way that allows separate modules to be added at defined interaction points.

### 2.3.7 Requirements summarization

The last previous sections have laid out 6 requirements for a solution to be able to solve the problems stated in this chapter. These requirements are summarized in table 2.1.

| No. | Requirement | Described in |
|---|---|---|
| 1. | Support a shared, model driven, vocabulary for multiple models | 2.3.1 |
| 2. | Supply visual tooling to develop/edit the vocabulary | 2.3.2 |
| 3. | Support horizontal mapping operations between the models | 2.3.3 |
| 4. | Support code generation from the vertical mappings | 2.3.4 |
| 5. | Use an open implementation environment | 2.3.5 |
| 6. | The implementation has to be extensible wrt. future additions | 2.3.6 |

Table 2.1: Main requirements

## 2.4 Execution of evaluation

To do the evaluation of the solution according to the requirements properly, a description of how the evaluation is to be executed is required. This description is given in this section.

The evaluation will be done using a reference ontology for a domain, and two class diagrams both depicting the same domain in a different way[11].

It shall be shown[12] that the solution can fulfill the following requirements:

1. Relate the elements in the class diagram to the concepts in the ontology.
2. Identify semantic mismatches.
3. Resolve semantic mismatches through the use of mapping operations.
4. Generate a reconciled class diagram from the mappings.

### 2.4.1 Annotate the class diagrams with concepts

The solution shall relate the elements in the class diagram and the concepts in the ontology. This means that the solution will need to:

1. Allow creation, editing and storage of the relations
2. Validate that the relations are live
3. Depict the relations

### 2.4.2 Identify semantic mismatches

Identifying semantic mismatches is vital to the effectiveness of the tool. This requirement is not intended to measure the effectiveness of the mismatch detection, but rather show that the tool can be enhanced to support various schemes.

Specifically, the solution will support these requirements:

1. Automatic semantic mismatch detection.

[11] one of the diagrams can depict it in the same way as the ontology, as this is a base case

[12] either by an implementation or a detailed description

2. (Semi)automatic semantic mismatch correction proposal (mapping).
3. Validate that the mappings hold.

### 2.4.3 Resolve mismatches with mappings

The solution will show that it can handle the following scenarios that require mappings to. . . :

1. Split strings. E.g. addresses that are concatenated.
2. Split classes. E.g. a class that holds two attributes will be split up to two classes containing one attribute each
3. Merge strings. Concatenate strings according to the reference.
4. Merge classes. Create a merged class containing the attributes from the merged classes.

### 2.4.4 Generate a reconciled class diagram from mappings

The solution will need to show that it is, from the relations and mappings, possible to generate a new UML class diagram that incorporates these—*a unified representation*.

CHAPTER

# THREE

# EVALUATION OF EXISTING SOLUTIONS

The purpose of this chapter is to provide an overview of existing frameworks, projects, solutions and standards that has relations to this thesis.

## 3.1 Existing platforms and projects

This sections details two research projects that encompasses some or all of the problem defined earlier in the thesis.

A walkthrough of objectives and accomplishments is provided before the impact and relations to the problem(s) is discussed.

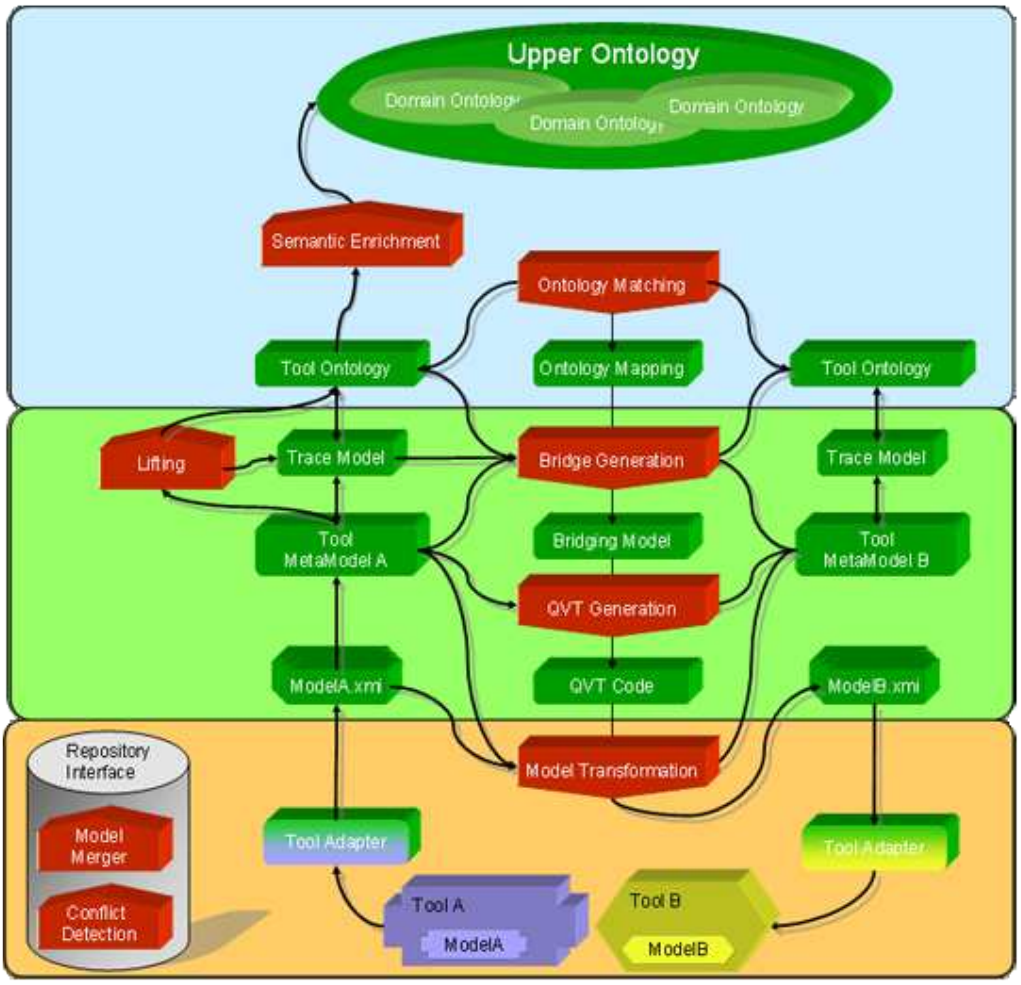
### 3.1.1 ModelCVS

ModelCVS presents a approach where metamodels are *lifted* and *lowered* from and to ontologies. Motivated by the need to do integrations between the models in two other projects, Modelware and Ontoware, they have produced a tool which translates metamodels to ontologies, creates a mapping model an then transforms this to a integration model at the metamodel level.

The reason to lift and lower the models is stated as: [15]:

> The integration can be made easier, when concentrating on the concepts described by a language, only, without needing to worry how the language implements these concepts

**Figure 3.1** ModelCVS Overview



The concept of *lifting* refers to bringing concept at one level up to one or several levels above the current one. For ModelCVS it means bringing a metamodel up to the level of ontologies. Since metamodels don't necessarily contain all the needed information to be presented as full ontologies, this is done in three steps, as can be seen in the corresponding layers of figure 3.1.1[1]. The first step entails conversion, where the metamodel is transformed into an intermediate ontology. This involves a mapping model from the meta-metamodel (M3) to an ontology metamodel, holding the more or less the same information as in the original model. This transformations is considered a pseudo-ontology as it is not perfect with regard to how formal concepts should be defined in ontologies. The next step is to refactor this pseudo-ontology to make the concepts more explicit, and also to find hidden concepts in the metamodel that should be made explicit. This is a partly manual and partly automatic step. It entails the assistance of a ontology developer that knows the domain(s) and its corresponding ontologies.

The third step is to enrich the newly refactored ontology with axioms, and also relate it to other ontologies that share the same vocabulary about the domain. This is also a manual step, requiring expert knowledge of the domain, and the ability to work with logic constructs (to express the axioms). This is referred to as *semantic enrichment*. To be able to relate the ontology back to the metamodel it was lifted from, the tool holds traces from the ontology concepts to metamodel constructs. This is an important property of the tool, as it allows the users of it to trace back to the roots of the model on later occasions if there is seemingly anything wrong with the resulting ontology.

When more than one metamodel is lifted to the same level as the common ontology, i.e. the shared vocabulary, it means that the metamodels at the ontology level, has a set of mappings to and from the common ontology. This in turn means that executing these mappings in turn will create a *bridge* from metamodel A to metamodel B through the common ontology. This achieves a star mapping, requiring far less mappings than when mapping directly between all the involved metamodels.

As of such ModelCVS represents a solution where the emphasis is on semantic interoperability between models through the use of ontologies. The process of transforming models to ontologies is a process that consists of several steps that require manual intervention by the user.

[1]From: http://www.modelcvs.org/thirtyminute.html

### 3.1.2 ATHENA Project

The ATHENA, short for Advanced Technology, Interoperability, Heterogeneous Enterprises, project ran from 2004 to 2007. The project goals were to enhance interoperability, through use of models and semantics.

A tool called Semaphore[19] was developed as part of ATHENA. It is described as a:

> ... a syntactic and semantic mapping tool.

This means that is will support the mapping of differing syntax, the representation of the data and then the semantic meaning of the data. This could be as easy as mapping synonyms or words in different languages or harder; e.g. mapping between concepts that share some of the same semantics. The latter require that the tool will be able to represent operations such as split, join, merge etc.

To achieve the mapping between different information representation and semantics, Semaphore requires that the information format is represented in a model at the PIM level. When the source and target models are properly represented at this level the developer can start mapping between them.

The mappings are preserved in a mapping model. Semaphore supports a range of different mapping and associated operations:

- Root mapping
- Simple mapping
- Concatenate mapping
- Split mapping
- Substring

Since the mapping process can be very tedious, Semaphore also includes pluggable mapping helpers, i.e. these are plug-ins that analyze the source and target models and suggests different mappings that can be made. Semaphore itself only presents a simple sample implementation of this.

After the creation of the mapping model, the user is left with a populated mapping definition model. This model can in turn be used as an input to a transformation, either a model to model transformation or model to text transformations. For instance it could be possible to generate e.g. code that

utilize a web service and saves the responses to a CSV file, or code that reads from a SQL database and outputs xml. Another prospective use of the transformation code is to generate mappings in ESBs.

The focus of Semaphore is to provide the mappings needed to produce code for transformation at lower levels. The tool is agnostic regarding what it maps, as long it is represented as a UML compliant model at the PIM level. This allows for mapping of ontology to models and ontologies to ontologies, but the tool isn't really geared towards providing mappings to/from models and ontologies and preserving the mappings up to the concept for later reuse and in combination with other models.

## 3.2 Standards and solutions

This section provides an evaluation of standards that may apply fully or partly to the problems stated earlier.

### 3.2.1 UML

The Unified Modeling Language, UML for short, is part of the MDA direction in Model Driven Engineering(MDE). UML does not just provide a standard for information modeling, but also modeling of behaviour. The most interesting aspect for us though is information modeling through the use of class diagrams.

#### UML Class diagrams

Most software developers have a relationship to class diagrams. The class diagram is in many respects the de-facto way of modelling a domain, and the most widely used diagram of the 13 available in UML 2. A class diagram expresses classifications and relations between them, for instance inheritance and association. There is also a possibility to express some constraints on the model using the Object Constraint Language, or OCL for short.

Class diagrams is at the object orientation abstraction level. That means that they will map easily to object oriented languages such as Java, Python or Simula. Indeed there are presently many tools, for instance AndroMDA[2],

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

available that allows you to build a large part of your application using class diagrams.

Often these systems at the base level will only off CRUD functionality, Create/Read/Update/Delete and needs more refinement to offer more complex functionality. Nevertheless they give developers a head start in projects where this functionality will be the base of the application.

Although UML class diagrams can be sufficient to capture the structure of a domain, the semantics of the domain can still be ambiguous. The reason for this is that the class diagram can be read differently by different persons in different contexts, since the vocabulary used normally is not connected to a more formal defined shared vocabulary. UML also comes with its own set of predefined semantics for e.g. the relations. The semantics of some of these, such as aggregation, is debated.

This prescription of how exactly one are supposed to model a domain is in some ways both a necessary evil (generating code would be hard without) and a good feature (fewer options can enable more consistency).

### 3.2.2 ODM

Ontology Definition Metamodel[13] is the semantic initiative from the OMG group Ontology Working Group and provides for the semantic in MDA. The metamodel consists of several packages:

- Common logic
- Topic maps
- RDF
- OWL
- Description logic

The main items of interest are the RDFS and OWL packages. These are a reflection of the W3C recommendations, and provide a way to build a model driven environment to model RDFS and OWL. A popular tool for modeling OWL based ontologies is Protégé[24], which is a Stanford research project. It consists of a platform supporting to ways of modeling ontologies: either by frames or by OWL.

As stated ODM is modular in the respect that it is split into packages based on the area the models in each package addresses. I.e. a separation of concerns. This means that ODM is open to extension, for instance reusing concepts in the RDFS model to support a ontology language other than OWL,

The Eclipse project provides an implementation of ODM, named EODM[25], that provides a EMF-based RDF/OWL Model[3].In addition the project also provides reasoners, transformers, parsers and simple editors for RDF/OWL. Currently[4] EODM is labeled as in incubation and the project activity is low[5].

### 3.2.3 SAWSDL

XML is one of, if not the leading data representation format in the world today. The format lends itself to creation of self-describing documents. But as in the world of models, often this is not enough. The description is merely a syntax, and can cause ambiguities and conflicts as it is interpreted by different parsers, people and in different contexts.

One prominent use of XML is to describe web services through the use of WSDL. As was described in section 1.1.2, a WSDL document contains a description of the associated interfaces, data types, exchanged messages and the so called grounding that makes up the service. An example[6] WSDL document is shown next:

```
<?xml version="1.0"?>
<definitions name="StockQuote"

targetNamespace="http://example.com/stockquote.wsdl"
          xmlns:tns="http://example.com/stockquote.wsdl"
          xmlns:xsd1="http://example.com/stockquote.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
          xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
       <schema targetNamespace="http://example.com/stockquote.xsd"
              xmlns="http://www.w3.org/2000/10/XMLSchema">
```

[3]Only in code form, no Ecore is provided
[4]Summer 2008
[5]Last release as of 9. October 2007
[6]From: http://www.w3.org/TR/wsdl.

```
        <element name="TradePriceRequest">
           <complexType>
               <all>
                   <element name="tickerSymbol" type="string"/>
               </all>
           </complexType>
        </element>
        <element name="TradePrice">
           <complexType>
               <all>
                   <element name="price" type="float"/>
               </all>
           </complexType>
        </element>
    </schema>
</types>

<message name="GetLastTradePriceInput">
    <part name="body" element="xsd1:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
    <part name="body" element="xsd1:TradePrice"/>
</message>

<portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
       <input message="tns:GetLastTradePriceInput"/>
       <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>

<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http'
    <operation name="GetLastTradePrice">
       <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
       <input>
           <soap:body use="literal"/>
       </input>
       <output>
           <soap:body use="literal"/>
```

```
            </output>
        </operation>
    </binding>

    <service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteBinding">
            <soap:address location="http://example.com/stockquote"/>
        </port>
    </service>

</definitions>
```

As can be seen this document brings with it a lot of implicit semantics. First of there is the data types. Except for the structure of the type, we don't really know what a TradePrice or TradePriceRequest is, and the computer certainly don't. The meaning of TracePrice could be one in one company, and another in a different company. This pertains to what was discussed in section 2.1.2 regarding information heterogeneity, and is clearly a case of a contextual mismatch, and perhaps also naming.

Further, the document has elements called portType. These define the operations the service has and their ins and outputs, but nothing about the behaviour of the service or pre or post conditions that has to be satisfied for that matter. There are more implicit semantics in this document, but with examples provided here, the reader should have at least an idea of what challenges there are to automating consumption of such services without formal semantics.

To alleviate this issue, a working group within W3C started working on a set of extensions to WSDL know known as SAWSDL. SAWSDL[28] is now a recommendation from the W3C for expressing relations to ontologies in XML documents adhering to a schema. The original intent for the standard was to provide a way to annotate WSDL with semantics, but it was soon realized that this could be done as a general all purpose solution for annotations on the XML level.

The standard is simple and does not say anything what-so-ever of which semantic technology one is supposed to use. It adds the ability to specify a:

- Reference to a Concept through the use of a *modelReference*.
- A mapping from a Concept to XML through the use of a *loweringSchemaMapping*.

- A mapping from XML to a Concept through the use of a *liftingSchemaMapping*.

The reference, specifically a *modelReference*, can be applied for the following elements in WSDL:

- interface
- operation
- faults

The elements for XML schema documents are:

- simpleType
- complexType
- element
- attribute

The recommendation also supports embedment of semantic models in the WSDL documents themselves[7].

An example of applying SAWSDL is in the tool *hyperModel*. This tool is made to support the modeling of XML Schema Documents through UML. The application of SAWSDL is described in [6]. The reason to integrate the semantic annotations is to realize the following benefits:

- Semantic classification
- Message or component discovery and reuse
- Unify XML naming and design rules
- Semantic validation of messages
- Mapping between message vocabularies

[7]These are contained within the *<WSDL:description>* element.

To do this a prototype was built using Eclipse technologies as EMF and EODM[25]. The implementation of the annotation mechanism was done through a small UML profile that had one stereotype that allowed the annotation to be captured. This profile was based on the SAWSDL specification. Further work on this implementation was planned to feature realization of some of the benefits mentioned above.

Another recent[8] effort to use semantic annotations through SAWSDL was described by Derouiche and Nicole in [9]. This effort was aimed at producing a prototype for using semantic web services in scientific work flows. They regard scientific work flows as data-driven work flows where the compatible parameters of the activities they structure are connected. The reasons to introduce semantics in these work flows is stated to be that:

> ...scientific work flows operate on large, complex and heterogenous data.

This leads to resource mismatches that Derouiche et al states requires intervention by the designer and is time consuming for scientists. The solution they envisage is the incorporation of semantics through annotations.

Their prototype incorporated SAWSDL as annotations for web service operation parameters. This enabled them to develop a Windows Workflow Foundation[9] custom activity for semantic web services. This activity enables the prototype to compose semantic web services and the ability to bind the operation parameters at execution time. Given to services, A and B, the binding works by first translating service A's XML data to the corresponding semantic (ontological) data (lifting) and after the successful execution this is then mapped(lowered) to XML data conforming to the XSD of service B's input parameter.

This prototype shows that using SAWSDL in scientific work flows is possible.

## 3.3 Discussion and Evaluation

This chapter has provided an overview of possible solutions for the problems outlined in chapter 2. The solutions reviewed in this chapter addresses different aspects of the problems.

[8] 2008

[9] WWF is a Microsoft Technology, part of .NET 3.X, that supports definition, execution and management of long-running work flows in a Microsoft Windows environment.

### 3.3.1 Discussion

To represent the domain of discourse the chapter has review UML class diagrams and a model driven approach for ontologies; ODM. These two technologies has common roots, as they both are represented using a metamodel based upon MOF, but ODM represents a much more advanced, powerful and versatile system for representing knowledge—ontologies.

The biggest "problem" with ODM usage would be that there are few constraints on how to represent the information. Luckily there are best practices to do this in the field of ontology engineering. This theme is not further discussed here, as it out of the scope of the thesis.

The Semaphore tool and ModelCVS project both contend in the same space, which is concerned with mapping models between each other. The approach they take are radically different. Semaphore works more as a procedural tool, with a input, mapping functions and output. ModelCVS has more focus on extracting the inherit knowledge in the input models, and works by transforming them to ontologies in two steps before the mappings are done.

Given this, it is fair to say that Semaphore represents a simple and fast, but probably less accurate, tool for mapping between models. ModelCVS is more complex, slower to work with and very accurate (if enough care is taken in the mapping process). ModelCVS also adds some more value in its process, since the input models in fact are transformed upwards to ontologies.

Another technology that is related to the work done in Semaphore and ModelCVS is SAWSDL. This technology is somewhat narrow (works for XML), but this doesn't exclude it from model driven development done MDA style. This is because the models are exchanged as XMI, which is XML with a OMG provided namespace. SAWSDL is very *light*, since it doesn't embed more information than strictly needed:

- a reference to a ontology concept
- a reference to a lowering operation
- a reference to a lifting operation

This means that is would probably be simple to incorporate some sort of use of SAWSDL in model driven tools. For instance, if tool A allows model elements to annotated with references to ontology concepts, these can be embedded in the XMI documents with SAWSDL. Tool B, also supporting references via SAWSDL, can then read the XMI file with the annotations

and allow the developer and or tool. to utilize the same vocabulary as was used when creating the model in tool A.

The "problem" with SAWSDL is that it is bound to a specific technology (XML) for its annotations. A higher abstraction level would ensure that this specificity is a matter of translation between the higher and lower levels. The general idea of embedding references to external resources nonetheless is a simple and feasible idea that could be implemented at a higher level.

### 3.3.2 Evaluation according to solution requirements

To evaluate the two primary solutions according to the solution requirements outlined in chapter 2 is not easy, as the solutions differ in terms of what they want to solve and how they do it. Based on this fact, this evaluation is in no way a score card, merely a way of summarizing the capabilities the solutions provide in light of the solution proposed in this thesis.

The evaluation is based on the different requirements outlined in table 2.1.

| #. | Description | ModelCVS | Semaphore |
|---|---|---|---|
| 1. | Shared, model driven, vocabulary for multiple models | 2 | 0 |
| 2. | Visual tooling to develop/edit the vocabulary | 2 | 0 |
| 3. | Horizontal mapping operations between the models | 2 | 2 |
| 4. | Code generation from the vertical mappings | 2 | 0 |
| 5. | Open implementation environment | 1 | 2 |
| 6. | Extensible implementation wrt. to future additions | 1 | 1 |
| | Final score | 10/12 | 5/12 |

Table 3.1: Evaluation of solutions. Range from 0 to 2.
Note: Evaluation based on available information and not through actual use.

As seen in table 3.1 the evaluated tools only satisfy parts of the requirements of the solution this thesis proposes in chapter 4. This is due to the fact that the area is under active research, where different research projects naturally will have different goals to accomplish. There is also a lack of commercial solutions in this field.

CHAPTER

# FOUR

# SMAT PROPOSAL

This chapter outlines the proposal for the Semantic Model Annotation Tool(SMAT), which will provide a proof of concept for this thesis.

## 4.1 Vision

The vision for SMAT is to provide a tool-set for semantic annotation of models on the platform independent level. This will be accomplished through a tools et that provides:

- A annotation editor
- A ODM implementation
- A component providing infrastructure for transformations, based on the annotations

The tool will be open and flexible with regard to future improvements in the technology stack. To achieve this openness and flexibility, the tools et will developed on the Eclipse platform. This platform provides both the infrastructure, tools and standard implementations that are needed for the development of the tools et. The platform also encourages development of plug-ins with high coherence and low coupling through the use of standard called OSGi, discussed further in the section 4.2.1. The Eclipse platform also has a vibrant and living ecosystem that gives a certain guarantee that the platform will undergo further improvement for the years to come and allow for further improvement of SMAT as new features become available.

## 4.2 Architecture

### 4.2.1 Eclipse Platform

The architecture will be realized with technologies that are part of the Eclipse project. At the core of the Eclipse project is the Eclipse platform. This core part provides the infrastructure that other projects within the Eclipse project builds on. The project itself started out as an attempt from IBM to provide a more open platform for development of tools and end-user features[7]. It became open source in 2001 together with the creation of the Eclipse consortium that besides IBM consisted of eight other organizations such as Borland and Rational. Even with the consortium in place, many in the industry perceived the project as a IBM controlled initiative. This resulted in that many third party vendors, and potential contribute rs, were reluctant to devote resources to the project. In a move to increase credibility of the project IBM and its partners in the consortium formed the Eclipse Foundation in 2004, which is a independent non-profit organization that employs its own staff, supported by the organization members.

Today, the Eclipse project spans multiple subprojects, that provides functionality ranging from JPA implementations (Eclipselink), Java development to development productivity (Mylyn). The projects of interest for this thesis are the Eclipse Modeling Framework(EMF) and Graphical Modeling Framework(GMF). They provide frameworks for both metamodeling and construction of domain specific languages. These frameworks provide the needed modeling facilities, code generation and a methodology for development of metamodels and associated domain specific languages. This enables developers to quickly either have all their needs fulfilled with the code generated from the frameworks, or use the code as the foundation for further development. EMF and GMF are further discussed in sections 4.3.1 and 4.3.1.

#### Eclipse and OSGi

The Eclipse platform also features an Open Services Gateway Interface(OSGi) framework and container called Equinox. OSGi is not a new technology, but recently it has gained more and more attraction from the Java developers community. In a nutshell, an OSGi runtime provides a Service Oriented Architecture for Java (on the JVM level) where services can be dynamically published, started, stopped, monitored and more at run-time. This encourages, and enables, developers to create modular systems with high cohesion and low coupling—a well known best practice from object oriented system

development. It also shifts the programming paradigm towards contract-first, where the one program against the provided service interface which in effect serves as a contract to what the service should deliver and consume, avoiding to programming against the concrete implementation.

The use of OSGi requires that application developers program against a OSGi API and deploy their application to a container that implements OSGi, Eclipse Equinox[1]is as said one, others include Knopflerfish and Apache Felix. To enable the services of the different modules in the application to be dynamically loaded, instantiated and consumed on has to adhere to the OSGi standard format of packaging the modules. This means that they will be packaged and organized into so-called *Bundles*. These bundles are nothing more than Java Archive(JAR) files that follows a standard structure and provides a file (*manifest.mf*) with metainformation . This file provides information about the bundle, such as its name, required bundles, exposed service interfaces and how to activate the bundle.

A manifest file could for instance look like this[2]:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: HelloWorld Plug-in
Bundle-SymbolicName: foo.bar.baz.HelloWorld
Bundle-Version: 1.0.0
Bundle-Activator: foo.bar.baz.helloworld.Activator
Bundle-Vendor: Foobar
Bundle-Localization: plug-in
Import-Package: org.osgi.framework;version="1.3.0"
```

As can be seen the manifest defines a *Bundle-Activator*, this is a special purpose class that implements a *BundleActivator*-interface. This allows the container to issue start and stop commands to the bundle itself and providing the bundle with a context An example of an implementation can be seen here:[3]:

```
package foo.bar.baz.helloworld;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
public class Activator implements BundleActivator {
```

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

[2]Adapted from: http://www.javaworld.com/javaworld/jw-03-2008/jw-03-osgi1.html

[3]Adapted from the same article as the manifest

```
    public void start(BundleContext context) throws Exception {
        System.out.println("Hello world");
    }
    public void stop(BundleContext context) throws Exception {
        System.out.println("Goodbye World");
    }
}
```

The purpose of this is to allow the bundle to e.g. do resource initialization. This could range from eager loading of data from a file or creating a pool of database-connections to be used by the bundle. If the bundle throws a exception during the initialization the container will not put it into service.

OSGi was adopted for the Eclipse infrastructure as of version 3.0[4] and the plug-ins that are made for use on the Eclipse platform are essentially OSGi bundles. OSGi is important in the implementation context of this thesis, since it allow great modularity and enables dynamic behaviour. This is done by allowing interchange of, starting of and stopping of modules at run-time. This is an important property with regard to e.g. interchanging and/or transformation-engines, validation frameworks and so on. It gives the user a real possibility to extend the tool. As of such it adheres to another important principle in software development, the *Open/Closed principle*: Open for extension, closed for modification. This is an important principle, as it ensures that no modifications should be needed in the entity that is extensible. This avoids propagating changes through several interdependent modules.

As of such the proposal contained in this chapter outlines only plug-ins that implements the OSGi standard—in accordance with how the Eclipse platforms works.

### 4.2.2 Architecture description

To fulfill the vision previously stated, a core architecture need to be established. SMAT will consist of five main components:

- ODM Implementation
- ODM Editor
- Semantic Annotation Model (SAM) Implementation

[4]http://www.research.ibm.com/journal/sj/442/gruber.pdf

- SAM Editor
- SAM Mapping Service

Adding to that, SMAT will in addition take advantage of an existing, third party, UML2 implementation. This implementation is part of the Eclipse project, and is called UML 2 Tools. It will provide the necessary parts to access UML 2 PIMs, validation of them and, if desired, editing. It is implemented in the usual Eclipse APIs (EMF, GEF, GMF).

As for the five main components, the *ODM Implementation* component will be a implementation of the OMG ODM metamodel for ontologies. It will need to feature the metamodel itself and facilities for manipulation, saving and loading of files. This component will implemented using EMF.

For the ODM implementation to be useful there will also be a need for a simple editor. The is provided by the *ODM Editor* component. This editor will be a simple editor based on the graphical formalism of Concept Maps, described later in the design section(4.3). The implementation will be layered on top of the ODM metamodel implementation and realized through the means of GMF.

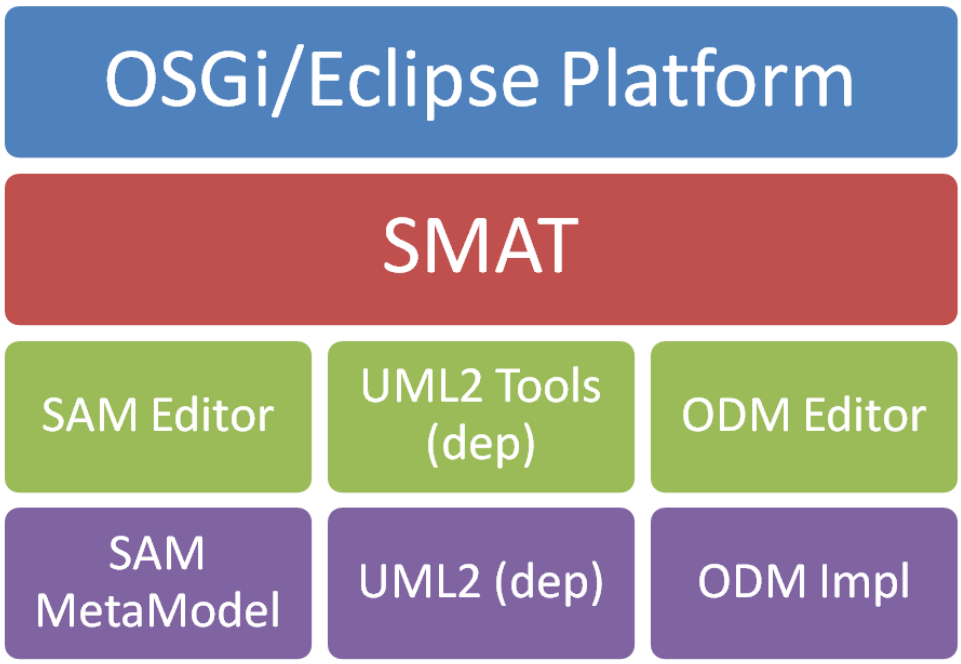
To relate the different PIM models and the common ontology, the *Semantic Annotation Model(SAM) component* will be developed. This model will also be implemented using EMF. The sole purpose of this component is to represent the binding between ontologies and PIMs and provide facilities for import and export of these. This is realized through EMF.

Utilizing the Semantic Annotation Model, and providing the actual annotation functionality will be the responsibility of the *SAM Editor* component. This editor is also developed using GMF.

At last the *SAM Mapping Service* component will provide the services necessary to facilitate (automatic) mapping of models and ontologies. This service is responsible for providing services for automatic mapping and export of these mappings to target platform specific models. This component will need to cater for different types of transformation languages and different transformations. This component will be implemented as an OSGi module.

The previous architectural description enables us to draw figure 4.1, shown below:

**Figure 4.1** Overall SMAT Annotation Architecture



### 4.2.3 Ontology integration strategies

Since the tool potentially will need to handle multiple ontologies, either interconnected or stand-alone there is a need for a strategy or at least view, on how they should be integrated with both each is needed. Three strategies for using ontologies in information integration is described in [29]. These three are:

- Global ontology
- Multiple ontologies
- Hybrid ontology

The global ontology suggests using a lone ontology that encompassed all the knowledge. This creates a shared vocabulary that all information is described with. This is unfeasible since it can lead to ambiguities if several information sources have different views of the domain, since it will lead to several levels of granularity and added complexity.

Multiple ontologies on the other hand entails the use of several ontologies, that are related horizontally. This allow the heterogeneous information sources to be described by a ontology that matches the domain. The problem is that the horizontal relation between source ontologies makes it difficult to compare them due to the differences. The ontologies would probably also be expressed with varying degrees of granularity and aggregations. Essentially the use of the strategy would only push the problem of heterogeneity to a more abstract level.

The last strategy is the hybrid ontology. In this approach, one defines a common vocabulary. This vocabulary expresses the knowledge all information sources have in common—common constraints, concept definitions and so forth. This becomes the base ontology that ontologies beneath specializes on. This approach assures that the specialized ontologies will have a common ground, making it easier to map between them, but at the same time allowing heterogeneous view on the domain as the lower ontologies specializes it. This approaches is also known as upper/lower ontology.

Essentially SMAT will resemble the last approach since the PIM models will be, amongst other, the specialization of the domain knowledge expressed in the core ontology. Further, one can also build layers of ontologies in the core ontology part, but they shall also be "governed" by an upper ontology, essentially a shared vocabulary to avoid inconsistencies and ambiguities to propagate to lower levels.

### 4.2.4 Annotation process

#### Usage patterns

The tool will not feature a specific process with steps that has to be followed to use it, but the architecture and design of the application will dictate some usage patterns. These are briefly described in this section.

The main usage patterns of the application are summarized in table 4.1.

| # | Name |
|---|---|
| 1. | Annotation |
| 2. | Mapping |
| 3. | Transformation |
| 4. | Validation |

Table 4.1: Usage patterns

The patterns revolve around the use of an instance of the SAM metamodel and the tool surrounding it. The metamodel instance is the key artifact.

The *Annotation* pattern concerns the primary use of the tool where the models will be related with annotations. The primary operation is of course *annotate*. This operation allows the user to select a model element and relate it to an element in the ontology. This relationship is persisted in a instance of the SAM metamodel (see section 4.3.4). To be able to annotate anything, the user will have to load the model(s) and ontolog(y|ies) that he or she wants to use in a annotation. These can either be loaded into an existing annotation model, or in a new one. The current relations, if there are any, are then displayed in the SAM Editor, and the user can choose to either edit, delete or create new mappings. The editor will feature feedback for the user with regard to validation of the relation.

The editor will also feature a possibility to explore possible annotations that can be applied through different ways of examining the related models and ontologies using e.g. matching words, synonyms etc.

In the *Mapping* and *Transformation* usage patterns the primary concern is both the construction and execution of the mappings made. Hence, two primary operations exist here: *construction* and *execution*. Construction is about constructing mappings that can both lower and lift the model from and to the ontology level, using the available constructs in the SAM metamodel. The execution is concerned with making the execution options, i.e. both the transformation of mappings and the transformation itself, available to

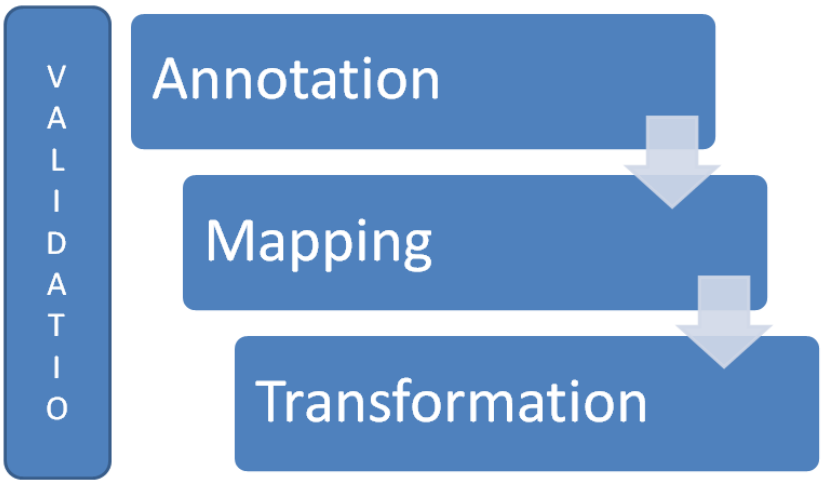
the user. This implies that the user will be able to select from different registered mapping engines and outputs.

Last, the *Validation* pattern will allow the user to determine if the relation and mappings validate—both in relation to the ontology, as well as a target model.

### Overall process

The overall usage of the process will consist of the 4 steps outlined in the previous section. The user will develop a annotation model, mappings and execute transformations. The validations will be done throughout the process to ensure that the relations and mappings are correct. A depiction of the process is seen in figure 4.2.

**Figure 4.2** General SMAT usage process



## 4.3 Design

This section describes the five core components in greater detail and provides an introduction to the frameworks used.

### 4.3.1 Eclipse frameworks

#### EMF

Eclipse Modeling Framework is an umbrella project offering several different APIs/frameworks for use in metamodeling. The projects range from distributed models to validation. The main project is the EMF Core, which offer us the possibility to define and use metamodels. This can be done programmatically by annotation of Java files, by XML (using a XMI/Ecore namespace), by XML Schema, or by using one of the provided (GUI) editors, or a third party editor that support EMF.

XML Metadata Interchange(XMI) is a OMG technology that supports exchange of models in a standardized way. This enables model sharing between different implementations of the same standard. E.g. if a UML Profile[5] is defined and provided a visual notation, serializing it as XMI will enable other tools supporting XMI to present the profile exactly as it was conceived in the first tool.

EMF provides its own metamodel API called Ecore. EMF started as a implementation of MOF, but evolved further based on the experiences made. This resulted in Ecore[10], which is a core subset of MOF, and said to be MOF compliant. In MOF 2.0, a similar subset was made, called EMOF. The differences is mainly naming, and EMF can read and write EMOF serializations transparently.

When a Ecore file has been produced, one has the possibility to generate code from it. This process creates a so called *genmodel* file that can be customized with regard to how the code will be generated. Instantiating the code generation on the genmodel creates three Java projects that contain:

- Model code
- Edit code
- Editor code

[5]UML Profiles are user-adaptions of the UML metamodel.

The *model* code is a reflection of the domain that was modeled in the Ecore step. The code is standard Java, with generated interfaces implemented by generated classes. The *edit* code represents the API for serializing and manipulating the model. Other facilities include code for building editors. The *editor* code provides editor code that conforms to the proper style for a EMF model editor and serves as a starting point for further customization.

### GMF

GMF extends upon Graphical Editor Framework(GEF), and provides a model driven design of these, allowing generation of code for diagramming tools.

The architecture of GMF is compromised of several metamodels that provides the necessary infrastructure and information needed to define the components in a Domain Specific Language. The instances of the metamodels convey the graphical definition, tooling definition and the mapping between the source metamodel, graphical definition and tooling to create the diagramming tool. The metamodels are:

- Notation model
- Graphical definition model
- Tooling definition model
- Mapping model
- Generator model

Of these metamodels one will be instantiated at runtime while other will be used in the work of creating the diagramming tool. The *notation* model is instantiated at runtime and stores the visual information of the diagram—where the nodes are positioned, their size, where links bend and so forth. Each instance of a diagram tool developed with GMF has a notation model instance attached. To actually generate the code necessary to get to the stage with the notation model in use, we need to execute three main activities[6]:

1. Defining a graphical definition
2. Defining a tooling

[6]The definition of the abstract syntax with EMF is a dependency before carrying out further work with GMF

3. Defining the mapping between the metamodel, graphical definition and tooling.

The *graphical* definition is the activity that determines how the visual language of the diagramming tool will look like. For instance one can define how the canvas(drawing area), nodes and associations will look. The definition is expressed in a model, using a tree view editor. This definition is not tied up against the metamodel. To separate common graphical definitions from the more specific parts of the language, it is possible to make several graphical definition files to enable easier reuse.

To instantiate nodes, associations and in general do anything worthwhile with the tool we need *tooling*. The tooling model will specifies what tools we'll have, where they will be placed (palette) and how they will look (colors, icons etc.). The tooling definition model cannot be used directly to generate any code, and is utilized in the mapping model.

After these to preliminary activities it is time to sew together these models. To do this we create a *mapping* model. In this model we define relations between the metamodel elements, graphical definition and tooling definition. To allow a tooling element to create a node with a certain look, we need to create a definition in the mapping model that states that a node in the diagram can be created with so-and-so tool, and that the node will look like the definition in the graphical definition model.

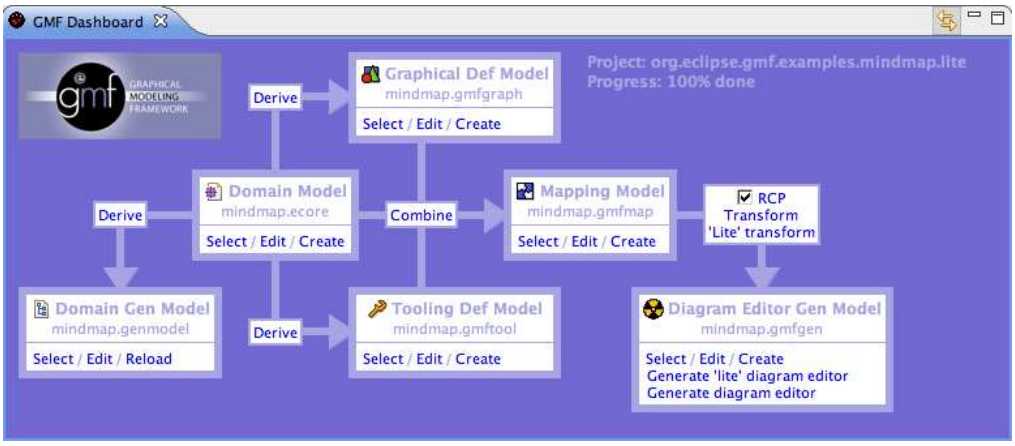
With these three activities done we can move on to the code generation. On the way to this there is a intermediate step that generates a so-called *generative model*, as in EMF. This model is in some respects the PSM of GMF, and allows us to do further configuration with regard to the diagramming application itself. Customizations are not necessary, and often not needed. Some examples are the possibility to change the plug-in name, ID and package statements.

The process is summoned in figure 4.3[7]. Some

When the generative model is ready, we can proceed to generate java code for the diagram tool. This code is produced, as in the case of EMF, as a separate plug-in project in Eclipse. After generation, it can be launched as an Eclipse application or optionally packaged as a standalone rich client application(RCP).

[7]From GMF itself

**Figure 4.3** GMF Dashboard. Actual process outline in GMF




### UML2 and UML2 Tools

UML2 and UML2 Tools is Eclipse subprojects that provides plug-ins for working with UML 2. This includes both editors, metamodels, persistence and other necessities for working with UML 2. Since SMAT have to be able to read UML 2 models, from XMI, it makes the project ideal for inclusion in SMAT. UML2 and UML2 Tools is plug-in based, and SMAT will have a dependency to these projects.

Depending on the distribution of Eclipse that the user has acquired, the UML2 and UML2 Tools might either be downloaded via an update site in the Eclipse update manager by the user or be required as an dependency when installing SMAT via the Eclipse update manager.

The update manager is component in Eclipse that allows seamless installations via so-called update sites. These sites are published web resources with special index files that tell which plug-ins are available on the server.

## 4.3.2 ODM implementation

ODM consist of 5 packages. The implementation used in this thesis will only feature the two most important packages: RDFS and OWL. The implementation is done a straightforward Ecore model using EMF. Constraints are expressed using OCL, and utilizing the OCL plug-in[8] for Eclipse the result-

[8]http://www.eclipse.org/modeling/mdt/?project=ocl

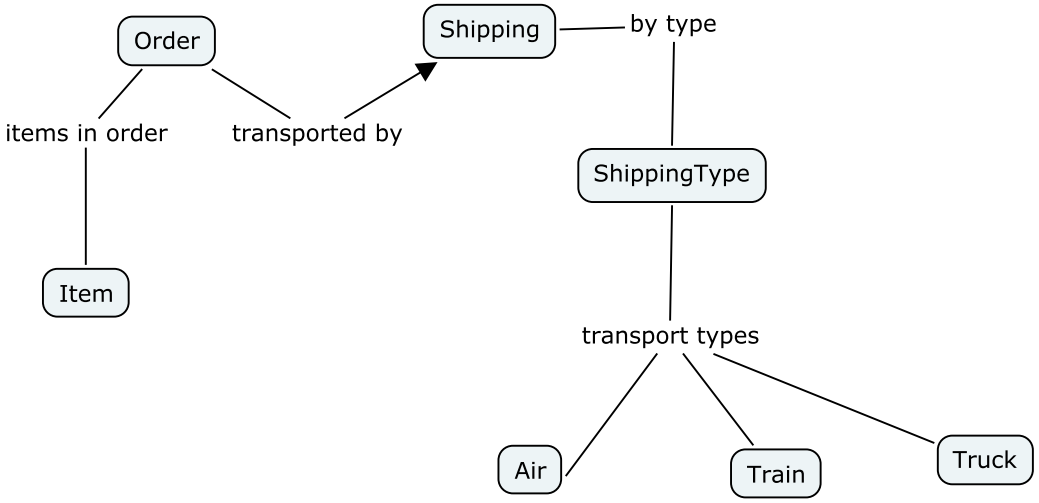
ing EMF code will adhere to these. Code generation is performed to produce both the model domain code and the model manipulation and I/O packages.

### 4.3.3 ODM editor

The ODM editor is built using GMF and feature the simple graphical notation Concept Maps. This notation is used for instance in a tool called IHMC CmapTools[9]. The notation itself was developed in the 70's at Cornell University, and has been in use for capturing the relationships between concepts since then.

A concept maps is simply a diagram with concepts that are related through *linking phrases* and *connections*, resembling named associations in UML. This makes the diagrams easy to grasp for non-technical people and fast to produce for everyone. An example diagram made with the tool IHMC CmapTools is shown in figure 4.4.

**Figure 4.4** Example concept map




The mentioned tool is a good example on how quick the maps can be developed, since it requires so little interaction, other than a some quick clicks with a mouse and some typing to name the relationships and concepts. Other tools that e.g. use UML as a formalism are often harder to work with, they also has a greater level of detail such as explicit inheritance, composition and

[9]http://cmap.ihmc.us

so on. On the other hand one can argue that this level of detail is required to make a decent and useful, especially in a MDE setting. Evans[11] argues that:

> A project faces serious problems when its language is fractured. Domain experts use their jargon while technical teams members have their own language tuned for discussing the domain in terms of design

The language Evans talks about here doesn't necessarily have anything to do with what graphical notation one uses to express ideas, but that domain experts and technical experts speak about the domain in different terms. He encourages a *binding* between the domain model and implementation. In a MDE setting, this binding could be between a ontology and a PIM. The Semantic Annotation Model described in the next section would then provide the necessary glue to relate these models. Since both the domain experts and implementers need to speak about the same domain it makes sense to choose visual notation that is as simple as can be with regard to the constructs used. Concept Maps is a good fit in this sense, since it essentially just features concepts and their relations. Adding to that, it could also be possible, and easy, to make a gradual expansion of this notation to support some more detailed statements, such as e.g. cardinality, which is also available in OWL and implicitly in ODM.

Another question with regard to the editor is to show the relations between the ontology and target models. It is of course a possibility, but could very well create a lot of visual "pollution". The realization could e.g. be done on the concepts in the map itself, or on a separate pane that listed the relations in for instance a table. In SMAT the ODM Editor will *not* show any relations, to keep things as simple as possible. Further extensions of the editor is discussed in chapter 7 under "further work".

Since the editor and metamodel infrastructure are separate projects, the editor can later be expanded or replaced with relative ease if deemed necessary. It also allows us for instance to replicate the current editor, enhance and expand it while still supporting the original editor without worrying about legacy issues that can arise in projects where modules are tightly coupled.

### 4.3.4 Semantic Annotation Model

The Semantic Annotation Model is also built using EMF.

### SAM metamodel

The Semantic Annotation Model, SAM for short, is a lightweight metamodel that stems from the arguments in 4.3.4.

### Annotation Scheme

The architecture provides annotation of target information models through a simple and flexible mechanism: a third model. This model will allow us to relate concept from an ontology model based off ODM to the information model (usually UML Class Diagrams). This is also one of the approaches in the field of *traceability*, and the main feature of this thesis, annotations is a form of traceability

Basically there are two approaches to relate models to each other. Either one can extend the target models themselves and annotate them directly. This can either be done in a light way with only references to the related model, or more heavily with full embedment. The other option is to use a third part model that contains the relations between the source and target model, as described in the paragraph above. Kolov et al. discusses both approaches in [18] and their pros and cons.

In favor of an embedded annotation technique is that it allows for tooling that easily and instantly can show the annotations on the model, this is of value to the user. The con is that it will also contribute to the pollution of the model. For instance the environment can easily be so "crowded" by both annotations and their targets so that the user eventually will have a hard time separating the two and/or figuring out the model.

Therefore Kolov et al[18] states that an external model is often preferred(in the traceability context). It allows the target and source models to be decoupled and it doesn't have the problem with pollution of the target model. Though, this approach comes with it own set of problems too. The models, since they are decoupled by the external, will be allowed to vary independently, reminiscent of the structural bridge pattern where an abstraction (ontology) is decoupled from the implementation (e.g. UML Class diagram). This gives flexibility at the expense of less control, e.g. changes in either models will have to be carefully monitored to detect inconsistencies and broken constraints in either models.

Implementing a tool that can support this is not necessarily straight forward, but can be accomplished in a least two ways:

- Monitoring the model for changes and checking for consistency after updates. Post validation scheme.
- Monitoring the model for changes as they happened, and check for consistency before the change is allowed to happen. Pre validation scheme.

The first approach is the most lightweight, as it can be implemented independently of the information model tool, it only needs to monitor for changes in the model file(s). The second approach is harder, as it needs to be integrated directly into the information modeling tool, and more resource intensive as it has to determine if the changes conflict while the user is interacting with the model(s).

Another advantage of the external model is that it will be simpler to introduce new functionality without the need to do another modification in the target model. An example could be to define mappings, e.g. how a concept in an ontology maps to entities in a class diagram.

Other more easily achieved functionality could be the introduction of versioning and other meta data. Round-trip engineering would probably be easier to. One special case also presents itself: for instance if one already has a large information model contained in one or more class diagrams, it/they could be converted to an ontology and in this same process the annotations could be made with the annotation model to support the relation between the ontology model which would now probably be the authoritative model and the class model that would be dictated by the conditions imposed by the ontology model. Also, another problem presents itself during the annotation itself—in what way shall we represent this to the user? There's no given answer here, but the most likely answer is to show the relations themselves in a simplified view that contains only the annotated elements and their relationship.

On the other hand it gives us the possibility to annotate models without the need to modify their metamodels to accommodate the annotation. This is also a challenge. When we can annotate "anything", then what will we annotate? This is a valid question, but outside the scope of the thesis, as it relates to which methodology one uses when working with the technologies and tools in the this domain.

At the other end of the spectrum is a model that is not annotated "enough". E.g. it does not provide any substantial value, and incentive, to annotate a information model when you don't add more information through the annotation, other than the possibility of transformation to semantic technologies

later on(although this can be argued to be a substantial contribution to a model, this could also be partly achieved through suitable transformations from the information model).

### Mapping operations

The metamodel must support different mapping schemes for the description of mappings to and from the ontology. These mappings will also need to be described in an abstract enough manner to allow for different technologies to execute the mappings. The basic operations that will need support are:

- Split (String)
- Join (String)
- Substring (String)
- Replace (String)

The string operations work on character streams, such as splitting a field based on a delimiter or joining one to adhere to the ontology definition. The substring operation will allow the transformation language to extract the portions of knowledge, and the replace operation will allow complete replacement of values.

These operations will contained within the SAM metamodel, and be constructed in a way that allow the specific units of mappings to be reused in other model-ontology relations. Since the operations in some cases need to be applied procedurally to reach the required result, the metamodel will support a sequenced composition of the operations.

The service providing the execution of the transformation operations is discussed in 4.3.6.

### Transformation scheme

A central aspect of SMAT is the transformation scheme—how do we transform from one model to the other? The transformation operations were outlined in the previous section, this section will take a look at how we can execute these mappings in a sensible manner.

As we saw in chapter 3, there was two tools that supported the transformation between models utilizing ontologies: Semaphore and ModelCVS. These two

tools represent two approaches to the transformation problem. ModelCVS takes the "hard" approach by lifting the input model to the ontology level in three consecutive steps. All of these steps will require human intervention to achieve the intended result. In Semaphore, the focus is a bit different—ontologies or UML models are used as a means to describe the reference model, and with that reduce the number of mappings needed to go from one model to another.

This presents to possibilities for the transformation scheme in SMAT:

- Lifting the source model up to the ontology and lowering it on the target side. E.g. ModelCSV.
- Generating transformation rules that can be used at the PIM or PSM level. E.g. Semaphore.

SMAT will use a combination of both approaches. The mapping operations will lift the source model to an intermediate format and then lower the model to the target using the latter' lowering operations. The transformation will preserve the annotations to the ontology elements for the whole duration of the process, enabling us to do a simple traceability on the mediated models.

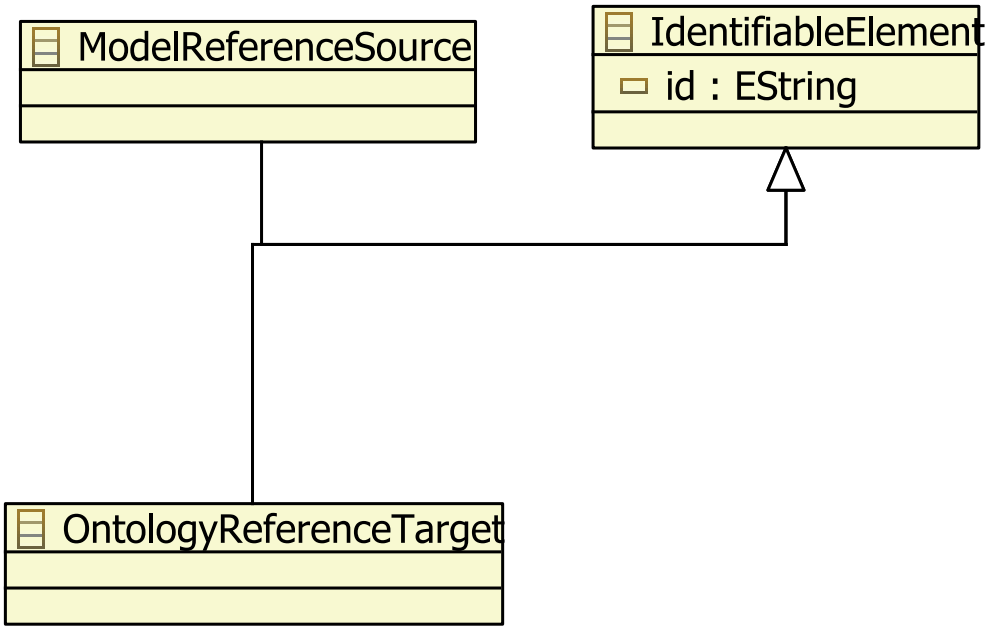
This mapping approach ensures that we avoid the $n^2 - 1$ situation where all models must features mappings between each other, lending us instead to the scenario of describing $2n - 1$ mappings. This is much more feasible when working with a large number of models, and reduces redundancy and lowers coupling between the models.

### Metamodel structure

The metamodel will use the structure of SAWSDL as its starting point. SAWSDL provides a modelReference, lifting and lowering references that point towards some resource that can do the mappings. The core concern of the functionality that needs to be supported is obviously the annotations themselves. Since SAWSDL has the annotation embedded in the source itself and SMAT will have it externally we need to add another reference. To achieve this, and avoid confusion, the SAWSDL modelReference is named *ModelReferenceSource*—meaning it is referring an element of a model instance. The opposite, the target of the annotation, is named *OntologyReferenceTarget*. This reference points to the ontology concept that the annotation targets.

**Metamodel core annotation structure** This core is depicted in figure 4.5. The source and target are two distinct classes that both inherits from *IdentifiableElement*. It is a abstract class that makes subclasses identifiable via a *id* property. The relationship between the source and target that make up one annotation is that of a normal association. One could make the case that the proper relationship here would be a composition, but that would imply that the source and target only could be part of one annotation. It is not entirely obvious that they should be allowed to be part of more annotations than one, but it could be the case that a domain model contained several model elements that mapped to the same concept in the target ontology.

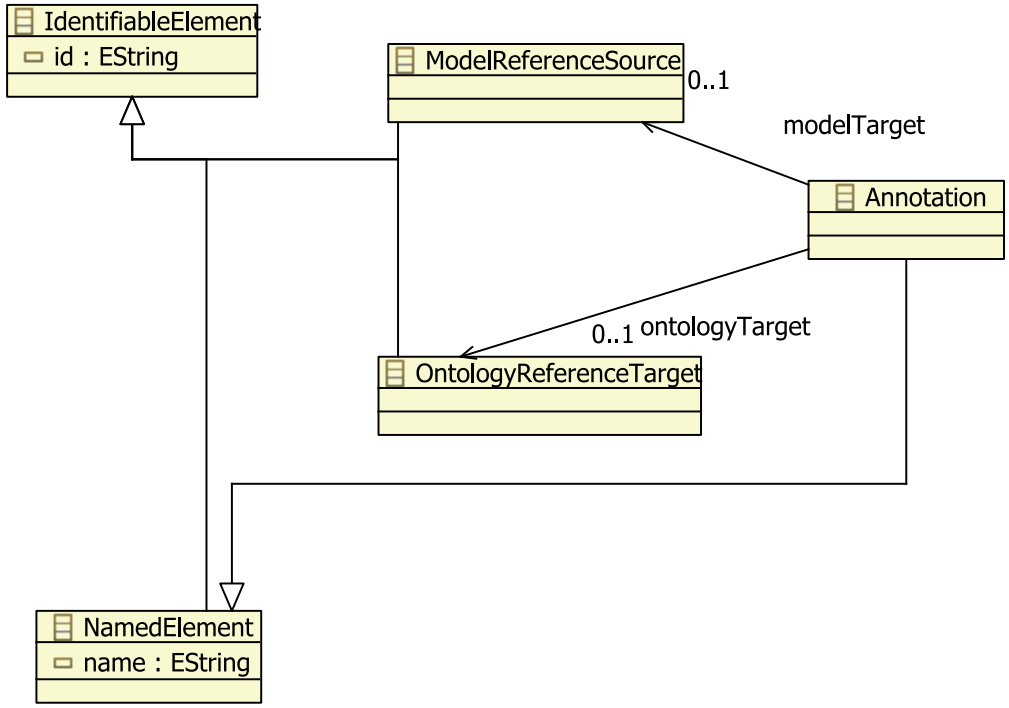
**Figure 4.5** SAM core annotation structure




Another way to approach that is to allow annotations to have multiple sources, something that would make working with the model hard, since it lower the distinctiveness of each annotation—rendering it ambiguous. Therefore, a *Annotation* is related to exactly **1** ModelReferenceSource and exactly **1** OntologyReferenceTarget, but both of these (their respective sources) are allowed to be related to several annotations. Further, the Annotation inherits from *NamedElement*, allowing it to have a distinct name. The NamedElement itself inherits from IdentifiableElement, so it can contain an unique id.

After these additions, the metamodel looks like figure 4.6.

**Figure 4.6** SAM core extended




**Metamodel core mapping structure** The second concern that needs to be addressed is that of *lowering* and *lifting* mapping. These are connected to the annotations, but should be as decoupled as possible from the annotation to allow for:

- interchangeability
- reusing similar mappings several places
- modify the mapping without affecting the rest of the annotation
- support of different mapping strategies

This leads us to a design where the *Annotation* element of the metamodel has an association to a *Mapping*. This Mapping is composed of a *Lowering* mapping construct and/or *Lifting* a lifting mapping construct. This is

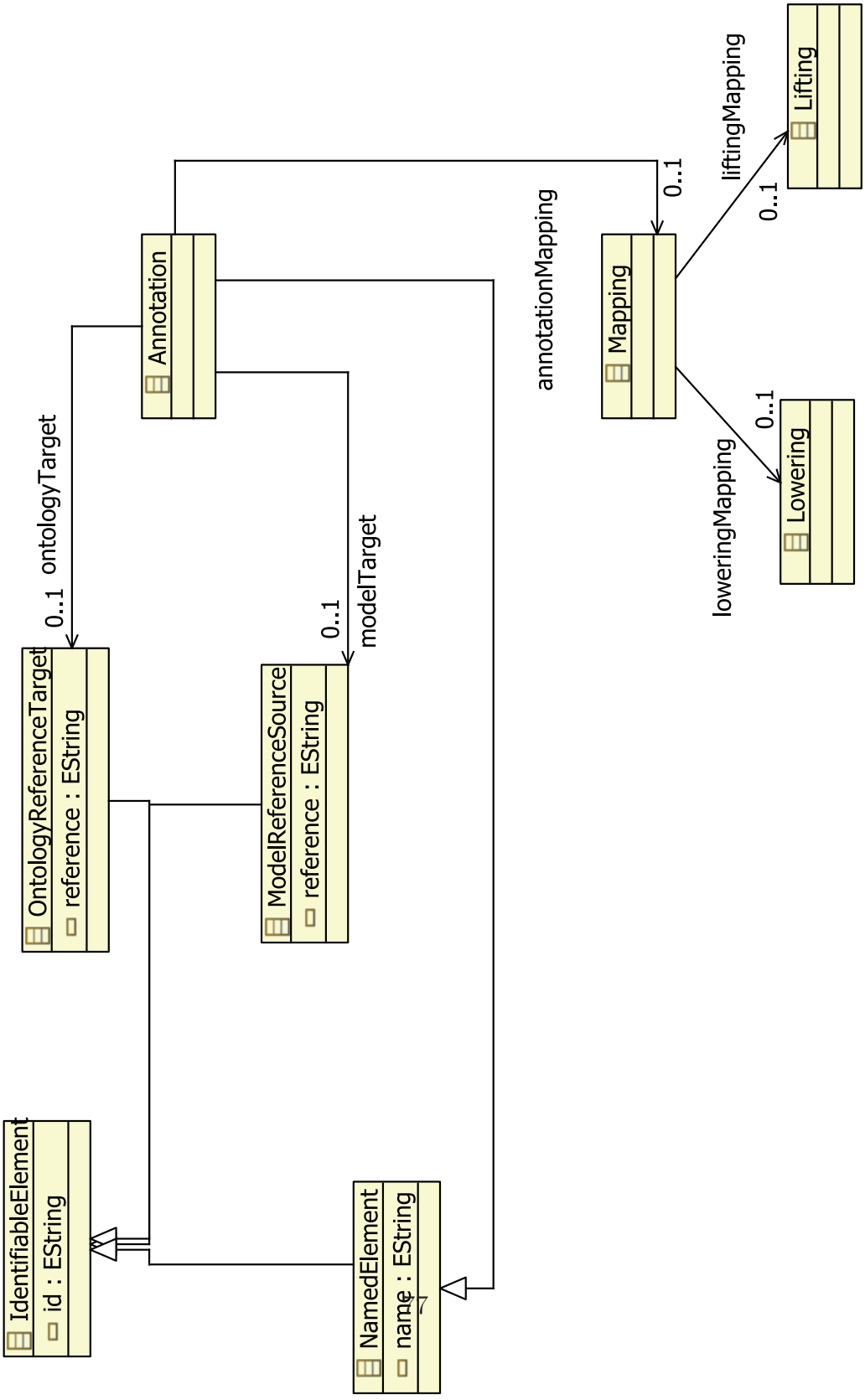
an abstract enough design that allows us to cater for several ways of doing the mappings. One problem that arises is that there i seemingly no direct association between the source and target the mappings are done for. This is intentional, since the source, target and the two transformations are implicitly highly couple, i.e. a change in either source or target will break the corresponding transformation(s). Because of this, the metamodel need to handle this implicit coupling in a way. The most straightforward way is to incorporate a simple versioning strategy, where the tool checks for changes in source and target since the last time the annotation was created or edited and warns the user that the transformation has changes, either in a general manner or by high detail. This is out of scope for the thesis, but a implementation suggestion is taken into the structure of the metamodel nonetheless. It is handled later in this section.

Thus the basic annotations with mapping structure looks like figure 4.7.

At this abstraction level there is of course not enough detail to support any mapping at all. To support a set of diverse mapping strategies the lowering and lifting mapping needs association to these concrete mappings. Since a complete mapping (both lifting and lowering) will map between model, ontology and model again we need to support some form of language to properly do this. The language can either be native of the SAM or be brought in from a part of the rest of the platform. One example of the former is to create a built-in language in the metamodel, with an associated DSL, and then transform its instances at *compile time*, meaning that the usage will be evaluated and transformed to a concrete mapping technology when the user desires to execute the mappings. This could be achievable with, for instance Atlas Transformation Language (ATL), which is a model to model language—primarily declarative in its form. The complexity of this solution really revolves around how complex the native language ends up being, but the upside is that creating it and the rules to compile it to a suitable execution format is done once per support transformation language.

The other alternative is to "out source" the transformation. This means that the lowering and lifting mappings only will hold enough information to be able to execute mappings that are non-native when they are executed. This solution adds complexity in the form of possibly many external dependencies, but this is also the strength of the approach; it is very flexible. Recognizing the benefits of both of these approaches, the design chosen is that of a small native transformation language, but with accommodations for future extensions of both approaches.

**Figure 4.7** SAM core mapping structure



In the case where a custom transformation file is preferred, e.g. an ATL transformation, the metamodel will need to contain both a reference to the file as well as a notion of which transformation engine that will execute it. This is known as the *TransformationFile* and *TransformationExecutionEngine* in the metamodel. Since mapping files themselves are applied to one specific mapping it limits what languages can be used. The most likely uses are together with XSLT or MOFScript modules. A possible expansion would to support some form of relation to specific regions of code in transformation code, this is further discussed in chapter 7.

**Metamodel details** To be useful, the metamodel requires some more details. Especially we need an abstract way of relating supporting the reference of models in the metamodel. To do this two datatypes are added to the metamodel. These are specific of EMF, and are an indirection to the abstract class *AbstractReference* that will hold the specific reference, using either EMFs API or some other API of choice.

The references for the Annotation is typed with the datatype *Reference*, while the mapping references are typed with *MappingReference*. This results in the updated metamodel structure shown in figure 4.8.

4.7.

**SMAT mapping language** The mapping language will be concise and support a set of operations that can be run procedurally to support most mapping scenarios. The operations needed was determined in section 4.3.4 and are:

- Split (String)
- Join (String)
- Substring (String)
- Replace (String)

The premise of these operations are that they operate on the presumption that we are dealing with string values in the class and attribute instances we transform. With this presumption we give up type safety, this can be remedied either by inferring the type from the ontology, or by expansion of the metamodel to support desired types. This is deferred to future work.

The simple metamodel for the mapping language will look like figure 4.9.

**Figure 4.8** SAM core mapping structure with datatypes

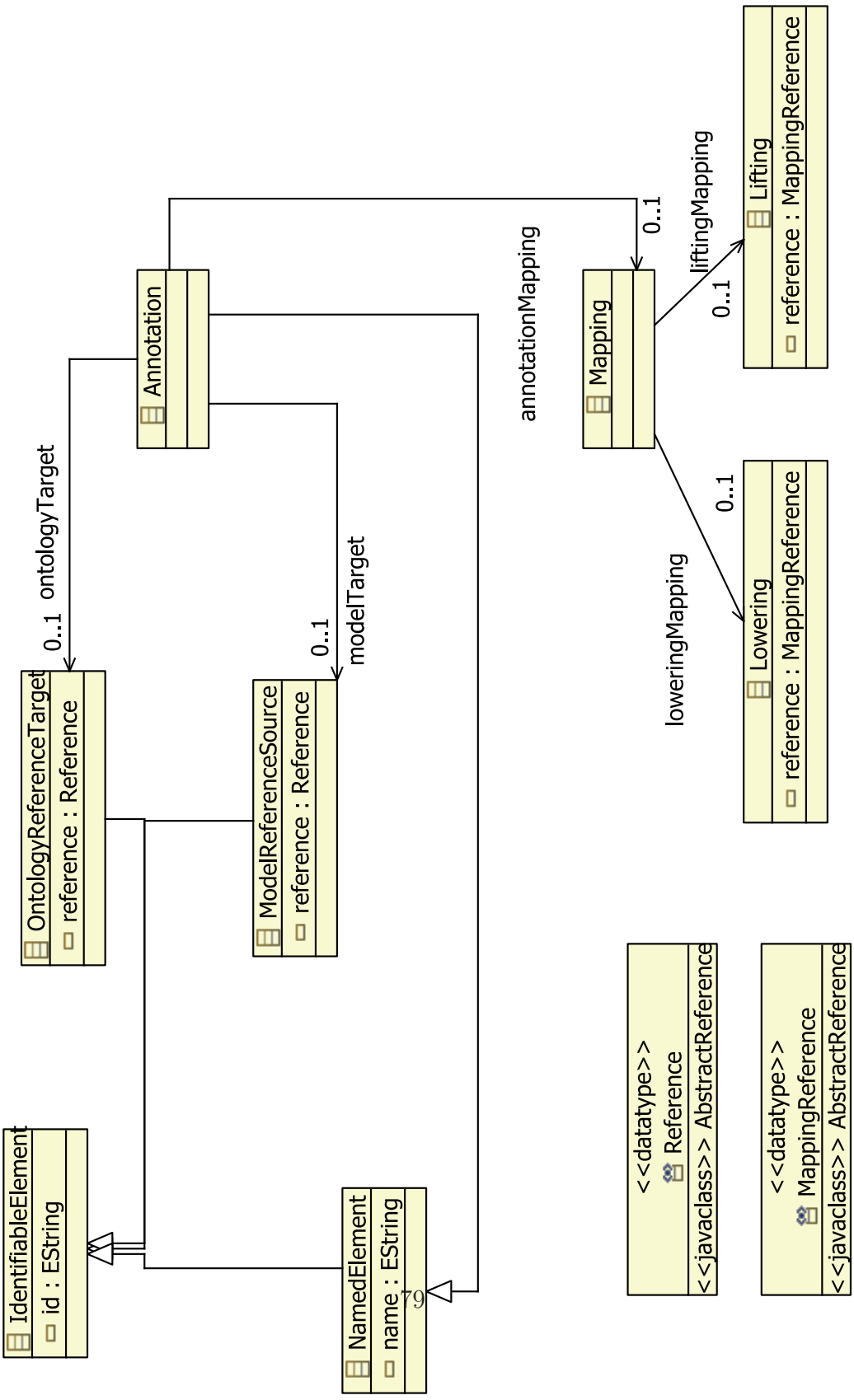
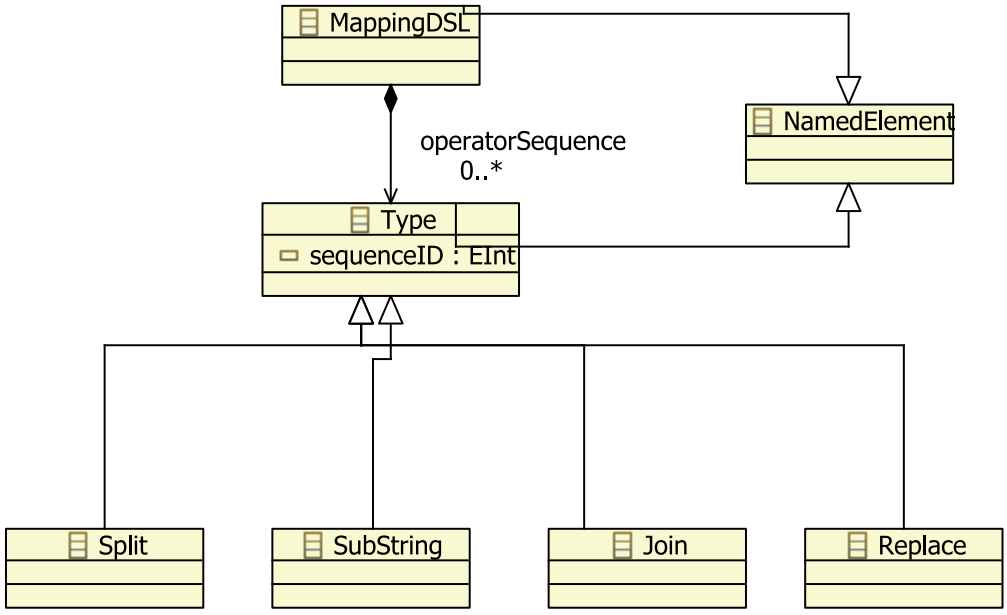


**Figure 4.9** SAM mapping language



The usage of the language is to build directed graphs of operations that are executed in a serial manner, in effect a ordered list of operations. This is done by composition of *Type*, with ordering persisted in *sequenceID*, by *MappingDSL*. The sequenceID has to be assigned programmatically at initialization. The metamodel is not sufficient detailed to be implemented as is, but it can be reasoned about on a conceptual level. The detailing and implementation is deferred to future work.

**Handling differences between source, target and their mappings** As previously stated there is an implicit coupling between the lowering and lifting mappings and their source and target references that is part of the annotation that is mapped. The suggestion solution to this was to incorporate some form of versioning. This if course a broad topic that have many solutions, but the metamodel will cater for one of the simpler:

- The *ModelReferenceSource* and *OntologyReferenceTarget* will contain a timestamp that is set on creation and subsequent updates (if any).
- The *Annotation* holds the timestamp of the latest update to these two for easy comparison with
- A timestamp held in the *Mapping* that also is a result of updates to timestamps in the *Lowering* and *Lifting* mappings.

This solves the part of the problem when updates happened within the tool, e.g. the user edits a annotation or mapping. It allows the tool to issue warnings where the may be changes that do not correspond. Another part of the problem is when the model or ontology changes. This is a more complicated problem to solve, since it is unlikely that metainformation about these changes will be persisted anywhere. It is also on the edge of what the tool should be responsible for. One way of solving it could be to store a hash of the source and target that the annotation relate, e.g. a MD5 hash[10] of text values within these elements.

Combined with also making a MD5 hash of the file itself, stored at the root element of the metamodel it would allow the tool to detect if the model/ontology file(s) has changed since the last time the model (SAM) was opened and if so use the md5 sums in the annotations to check the if the annotations has been affected by the changes. If there is any changes

[10]A 128 bit hash using the Message-Digest algorithm 5 (MD5). Often used to check integrity of files or other information. See: http://tools.ietf.org/html/rfc1321

the affected annotations are flagged in the editor and/or the problems view (part of the Eclipse platform).

Summarized the basic change support is accomplished by:

1. incorporating the additional model/ontology change support constructs in the metamodel. Namely:
   - a file hash stored in the root of the SAM.
   - hashes of the contents references the annotation has.
2. Implementing algorithms that can walk the changed models and/or ontologies and check if the annotations in SAM are affected.
3. Implementing timestamping of references and mappings to detect possible problems with mappings after updates to the annotations are done.

### 4.3.5 SAM editor

The SAM Editor is built as a tree view editor, providing the user with the constructs needed to do both mapping and relations of models and ontologies.

#### Supporting automatic detection of mappings

### 4.3.6 SAM Mapping Service

The mapping service will in principle be open for use with any transformation language, provided that they can be plugged into the extension point provided by the mapping service. In other words this is the service that allows SMAT to function with external transformation languages.

The goal of the mapping service is provide a abstraction between the actual transformations and the abstract mappings done in the SAM Editor. This is realized in a component called the SAMMS, SAM Mapping Service. It is a common service that provides the necessary interfaces needed to provide provided transformation or plug-ins with the infrastructure needed to execute mapping rules on the mappings associated with the SAM.

The SAMMS will feature service interfaces that provide the following public functionality:

- Transformation directory service interface

- Transformation of SAM and source/target models interface
- Transformation registration (extension point) interface

The transformation directory service provides a means to access all available transformations in SAMMS that the developer wants to make use of, allowing the SAM Editor to provide easy means of accessing these transformations through the transformation model interface. The service offers basic query functionality. The transformations registration extension point interface is a Eclipse construct that allow the user to register other transformation services than those provided with the transformation engine.

## 4.4 Realization of SMAT with EMF and GMF

SMAT is realized through the use of the Eclipse projects EMF and GMF. EMF is the Eclipse Modeling Framework and provides facilities for metamodeling, generating code for model interaction and simple GUI editors.

Graphical Modeling Framework is at the other end of the spectrum and provides facilities for a descriptive way of building graphical notations for metamodel instances. GMF also relies on and provides an abstraction to the lower level library GEF, Graphical Editing Framework.

These two frameworks in combination give the possibility to quickly model the target domain and define a tailor-made domain specific language of choice.

### 4.4.1 SMAT and the Eclipse Modeling Framework

SMAT is realized at the meta level using EMF. This involves activities with metamodeling using a 3. party plug-in called Topcased[1] that provides a somewhat better graphical metamodel editor than the default included in the current release of EMF (at the time this thesis was written that was version 2.3).

Using EMF ODM is implemented with basis in the OMG document[13] that details this metamodel. OCL constraints are implemented using the subproject of EMF that provides an OCL framework.

The SAM metamodel is implemented in EMF using this thesis as specification. The code is generated without any customization of the genmodel that EMF generates.

### 4.4.2 SMAT and the Graphical Modeling Framework

#### ODM editor realization

As just the implementation of models is of little value in the context of this thesis there is a need to develop further means of interacting with the model. As stated in 2.3.2 the tool is required to support a graphical notation that is the same as the one found in Concept Maps. This forms the base of the domain specific language, DSL, in SMAT. This DSL is used in the ODM editor, and provides a simple editor supporting modelling concepts and their relations.

The concrete implementation needed here is by unifying the ODM metamodel and the GMF constructs. Specifically a GMF node in a ODM editor diagram refers to a class in ODM. To model the relationship, a link in GMF, one has to instantiate a *ObjectProperty*. This ObjectProperty holds a *Domain*, that is the allowed classes on the left-hand-side (lhs) and a range, the allowed classes on the right-hand-side(rhs).

#### SAM editor realization

SMAT itself involves two metamodels, and subsequently two DSLs has to be developed. The first will need to support the *mapping operations* outlined in the mapping language outlined in 4.3.4. The resulting DSL will be an editor that provides easy construction of the mapping language graphs.

The core metamodel will require an editor that is easy to use and provides the easy access and a clear visualization of the populated model instance. Since a tree view editor is simple, easy to comprehend and navigate, the proposal chooses this method of visualizing the model.

The tree view editor need to display:

- The annotations of ontology and models
- The mappings between the ontology and a model element

We propose the following hierarchical structure to present the information in the metamodel instance:

- Root element
  - A ontology Concept
    - Some element from a model

        * Some other element from a model
    - Another ontology Concept
        * Some element from a model
            · Lowering mapping
            · Lifting Mapping
        * Some other element from a model

In the case of classes, these should be represented with Classname:class, e.g.: *ShippingCompany:class*. In the case of class attributes, these should be represented with:*attributeName:ClassName*, where attributeName is the name of the attribute, and ClassName the name of the class it belongs to. Additionally, an icon should be put before all the nodes in the tree view, to make distinction between different types easier. These icons should have contrasting colors for easy identification.

In addition, we need to present some more information about each unit in the hierarchy. This is done with a properties view that present e.g. a button to edit the mapping definition for a mapping, metainformation about a model element and more information about a concept.

### 4.4.3 Proposal summary

This proposal has set out to provide a tools et that will include the necessary parts to annotations, and inherit mappings, of models and ontologies together. To accomplish this there has been developed two metamodels:

- the SAM metamodel
- the Mapping metamodel

These two metamodels are suggested implemented with the Eclipse technologies EMF and GMF. Subsequently their conceptual implementations has been described in the latter part of the thesis.

CHAPTER

# FIVE

# SMAT APPLIED

This chapter apply the SMAT solution to the problem definitions from chapter 2. The goal is to show how the solution proposal would act when tasked with a scenario mimicking a real world problem, albeit our test case is a bit limited. The application will also serve to reveal any shortcomings and specifically if the solution doesn't' pass the requirements stated in the problem definition chapter. These requirements were stated in table 2.1.

Since the proposal isn't fully implemented, the evaluation is done with the metamodel constructs and the conceptual solution in mind.

## 5.1 Buyer/Seller evaluation case

As described in the problem definition chapter the evaluation case consists of two domain models described with UML class diagrams and a reference ontology. The ontology reference uses the same language as model A, to show that we can do 1-1 mappings between the model and ontology. Model B on the other hand expresses the domain in different terms with differences in the model structure compared to the ontology and model A.

### 5.1.1 Annotation and mapping from model A to ontology

Model A and the ontology have the same structural features. The evaluation in this case is therefore to asses if the metamodel will support the annotation

from *model A* to the *ontology*. First, we observe that our metamodel has the capability to annotate different resources through its *Reference* attribute in the respective classes *ModelReferenceSource* and *OntologyReferenceTarget*. This allows to connect the classes in model A with its counterpart in the ontology. In the proposed tree view this will equate to presenting the user with the following hierarchy (excerpt) in the proposed tree view editor from 4.4.2:

- root
  - Order
    - Order:class
  - ShippingCompany
    - ShippingCompany:class
    - Customer:class
    - name:Customer

This shows us that the metamodel can cater to annotation of concepts in the ontology and of model elements. There's an ambiguity in this application though. A small ambiguity in this example is that of the name property of Customer. Since ODM is a representation of OWL the name property can both be a concept related to ShippingCompany through an objectProperty, or it can be a value property that is contained with the concept itself. This distinction should be made explicit in the properties of the part of the annotation concerning name:Customer where we are dealing with equivalent information. In the case where we are mapping between the ontology and model, this distinction will be inherit in the mappings.

### 5.1.2 Annotation and mapping from model B to ontology

The annotation and mapping between *model B* and the *ontology* is a harder task. To recapitulate from chapter 2, we have the following problems at hand:

1. Concatenated information in strings. will need to be split according to a separator or by index. Examples are: *ShippingAddress* and *OfficeAddress* in *Customer* from *model b*

2. Classes that are in-cohesive, e.g. they model more than one concern will need to be split up. An example is the class *Transport* from *model b*. It contains both type information, company and address. To conform with *model a* it would need to transformation to that structure.

3. Naming differences need to be resolved according to the references ontology. Example is *Stock* in *model b* and *Inventory* in *model a*. The reference uses the word *Inventory*.

Our conceptual mapping language supports the mapping of all of these operations. A possible annotation structure would look like this:

- root
  - Inventory
    - Stock:class
      - Lowering mapping
      - Lifting mapping
  - ShippingCompany
    - company:Transport
      - Lowering mapping
      - Lifting mapping

The mappings in this example for the *Inventory* would be that the replace operation is run at the ontology name when we lower from the ontology, and vice versa replaced with inventory when we lift to ontology level.

The operations from the lowering of *company:Transport* uses the *split* operation to attain the name of the company and instantiate as the company of a Transport instance class. Lifting also uses split to extract the company name into a instance of the separate concept ShippingCompany.

### 5.1.3 Execution of mappings between A, B and ontology

When executed the mapping graphs will be transformed to a transformation language and executed, leaving the user with the desired output (A transformed to B or vice versa.). Other possible uses of the tool includes generating mappings for e.g. ESBs.

## 5.2 Discussion

This chapter has shown a rough outline how of the tool would be applied to a problem. This conceptual application is not elaborate, but it serves to tell that it is indeed possible to construct this tool in a viable manner with the proposed conceptual solutions from chapter 4. There exists some problematic aspects regarding the mappings that the proposal has not addressed properly, the most prominent is that of how to performed mappings between properties in ontologies and classes/attributes in models.

CHAPTER

# SIX

# EVALUATION OF SMAT

This chapter does an evaluation of SMAT in light of the proposal, the previous chapter with application of the proposal and the solution requirements. Each requirement is gone through and a grade ranging from 0 to 2 is assigned based on tool fulfillment of the requirement. A table with the aggregated scores is presented in the final section of this chapter.

## 6.1 Evaluation of criterions

In chapter 2 we listed **6** requirements that the solution should achieve. The subsections here are the same as in that chapter for easier reference.

### 6.1.1 Shared model driven vocabulary

This requirement prescribed the support of different types of input models, ontology infrastructure and the use of a properly documented metamodel to achieve this.

The conceptual solution support different types of input models, it uses ODM/OWL as infrastructure and metamodel. The score here is *2*.

### 6.1.2 Visual tool for vocabulary development

The second requirement was: *Supply visual tooling to develop/edit the vocabulary*, from section2.3.1. The proposed solution fulfill the requirements by

proposing an ODM backed editor that uses concept maps as the graphical formalism. The score here is *2*.

### 6.1.3 Horizontal mappings between models

This requirement described the need to be able to *lower* and *lift* model elements to ontology concepts they were related to. This was prescribed to be done with mapping operations.

The conceptual solution achieves this. The score here is *2*.

### 6.1.4 Vertical mappings to PSM/Code level

This requirement described the need to be able to map the SAM model down to PSM/Code. The conceptual solution can be extended in this direction, but doesn't provide all the necessary parts. Score *1*.

### 6.1.5 Open implementation environment

The use of an open environment is fulfilled. The Eclipse platform uses an open source licence. Score *2*.

### 6.1.6 Extensible implementation

This requirement described the necessity of providing an implementation that is extensible. The conceptual solution outlines the use of OSGi for modularization. The solution does not address the use of extension points, but this is certainly possible to include. Score *1*.

## 6.2 Summary

As we see from the previous sections the *conceptual* solution fared quite good. Table 6.1 summarizes the results.

Compared to table 3.1 in section 3.3.2 we see that SMAT scores equal to ModelCVS and 5 points higher than Semaphore. ModelCVS has currently better support for code generation from the vertical mappings, while SMAT has the advantage of an open implementation environment.

| No. | Requirement | Score |
|---|---|---|
| 1. | Support a shared, model driven, vocabulary for multiple models | 2 |
| 2. | Supply visual tooling to develop/edit the vocabulary | 2 |
| 3. | Support horizontal mapping operations between the models | 2 |
| 4. | Support code generation from the vertical mappings | 1 |
| 5. | Use an open implementation environment | 2 |
| 6. | The implementation has to be extensible wrt. future additions | 1 |
| | Final score: | **10 out of 12** |

Table 6.1: Main requirements fulfillment score table

CHAPTER

# SEVEN

# CONCLUSION AND FUTURE WORK

## 7.1 Conclusion

In this thesis we have explored the area of MDA and semantics, by doing so we have also approached areas that are related, either directly or indirectly. We have seen that there can be substantial benefits gained from combining semantics and a tool that conform to the MDA idea. Requirements for a tool supporting annotations of models, with the intent to relate them to ontologies, was presented in chapter 2. This chapter also introduced to an imaginary, although realistic enough, test case to evaluate such a tool.

Chapter 3 introduced us for several solutions that exist in the same problem domain that this thesis have. Neither of the two tools evaluated solved what we posed as requirements, and the other solutions in the technical space could not provide more than parts of the solution. This obviously created a need to pose a solution that would solve the requirements. This was done in chapter 4. The rest of this section is devoted to an evaluation of this tool in the light of the results from the previous chapters and will also present the answers to the three questions posed in chapter 2:

*Will integration of semantics, through the means of ontologies, allow greater interoperability?*

and:

*Will the integration of semantics allow easier transformation of models?*

and:

*Will the integration of semantics provide better validation of models?*

It is the conclusion of this thesis that the integration of semantics will allow for greater interoperability. This statement is in the light of the discussion in chapter 2 and especially in the light of the proposal and application of the conceptual solution that have been done. It shows that it is possible to create reference models using ontologies and annotate UML models with the concepts contained within the ontologies. Also it has been shown that with a mapping operation scheme it is possible to lift and lower the representation to the ontology level and thereby creating a "bridge" between two models.

With regards to allowing easier transformation of models, the answers is yes. This is due to the fact that the involved models will be described according to a reference ontology and their mapping operations will be transformed to a common transformation language. The transformation of the operations can be interchanged, creating a versatile tool.

The validation aspect has not been addressed to the fullest extent, but it is clear that the definition of formal semantics and the connection to models that doesn't have the possibility to define so rigid formal descriptions will allow for greater validation of these models. Of course, a chain is not stronger than the weakest link; in all cases we are dependent on developer defining these semantics in the correct way and annotating the models correctly.

Summing up we see that the construction of a tool like the proposed SMAT is:

1. doable
2. adding value to the annotated models
3. making more extensive validation possible
4. making interoperability between models better

The ending conclusion is that SMAT as a conceptual solution should be further refined and implemented.

### 7.1.1 Shortcomings

An obvious shortcoming is of course lack of a proper implementation. Besides this there are shortcomings regarding the annotation themselves. One special one is that there is no distinction on what one annotates. Potentially one could create an annotation that related a class in a model and a relation

between concepts in the ontology. While this is not entirely wrong, as the model class could be an association class, it doesn't provide the distinction needed to see that this is what is meant.

The tool as of now doesn't have proper support for relating associations to concepts or relations in the ontology either. This is left for future work.

## 7.2 Future work

There are several areas to enhance or extend within this thesis. These are:

1. Annotation and ontology repository
2. Automatic model matching in the repository
3. Extensible validation
4. Relations between specific regions in transformations and models
5. Expanding the tool to support source code
6. Extending the ODM editor
7. Detailing the mapping language

These areas are further discussed in the forthcoming subsections.

### 7.2.1 Annotation and ontology repository

The most interesting is possibly the development of a repository for ontologies and the annotation models that can be distributed over a network using web services or similar technologies.

This repository would enable server-side reasoning and querying of both the annotation models and ontologies. This would make the process of annotating models more open and accessible, and would also allow greater findability within the models. The repository would also enable the users to collaborate on the development of ontologies, and make it possible to search for models matching ones own, perhaps most specifically service interfaces.

The repository would, or should, also enable for:

- versioning of ontologies
- cross validation of models and ontologies

### 7.2.2 Automatic model matching in the repository

Another area to further explore is one of those that would also be contained within a repository—automatic model matching. Some schemes have been outlined in this thesis, but several more ways of doing this should be possible. Since models have annotations to respective ontologies, it would be possible to match with other models in the repository when they annotate the same thing in the ontology. If enough models is contributed to this "annotation space", it would create possibilities of easier reuse and ease the integration effort. One

### 7.2.3 Extensible validation

The validation as described in this thesis is done passive, e.g. it has to be invoked by the user. SMAT should support both live validation of local models and also the models in the repository. The validation should in all instances be plug-in-based so that is would be possible to add new validators as need. For instance one could imagine a validator that checked the meaning and relations of nouns in a domain model against the data in WordNet [1].

### 7.2.4 Relations between specific regions in transformations and models

A smaller, area mentioned in the proposal chapter(4) is the relation between transformation files and mappings in the model. The suggestion is to allow specific regions of code to be related to specific mappings. This can again be utilized to provide the developer with support when changes occur in the associated model, prompting him or her to check if the mappings still hold. This feature would require special adaptions to the languages that would be supported. One would require specific parsers that can identify and present viable code regions, e.g. functions, to the developer for relation to the mapping. At the execution of the mapping, depending on support, the transformation service could choose to only execute certain mappings or execute all.

[1] http://wordnet.princeton.edu/

### 7.2.5 Expanding the tool to support source code

A last area that could be investigated as part of future work is to include source code in the model part of the tool, meaning that the tool would treat source code as PIMs.

### 7.2.6 Extending the ODM editor

The ODM editor should be extended to support more advanced uses. The thesis only proposes a simple editor that handles concepts/classes and their relations, to support quickly building a domain language. An extension of the editor should allow a second *mode of operation* that support the more advanced parts of ODM/OWL. This includes creation of logical expressions, defining cardinality and the addition of value properties.

### 7.2.7 Extending the mapping language

The mapping language model in the model needs to be further detailed to support the operations needed. Specifically there will be a need to include index positions for string operations, references for split and join operations (both String and class references).

# BIBLIOGRAPHY


[1] Topcased [online]. Available from: `http://www.topcased.org/` [cited 15-01-2008].

[2] Ieee standard computer dictionary: A compilation of ieee standard computer glossaries., 1990.

[3] Maarten Van Steen Andrew S. Tanenbaum. *Distributed Systems - Principles and Paradigms*. Pearson Education, 2006. 2. edition.

[4] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. Owl web ontology language reference [online]. W3C Recommendation 10 February 2004. Available from: `http://www.w3.org/TR/owl-ref/` [cited 06-12-2008].

[5] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, May 2001.

[6] David A Carlson. Semantic models for xml schema with uml tooling, 2006.

[7] Gary Cernosek. A brief history of eclipse [online]. Available from: `http://www.ibm.com/developerworks/rational/library/nov05/cernosek/` [cited 07-16-2008].

[8] Michael C. Daconta, Leo J. Obrst, and Kevin T. Smith. *The Semantic Web: A Guide to the Future of XML, Web Services, and Knowlegde Management*. Wiley Publishing, 2003.

[9] Kheiredine Derouiche and Denis A. Nicole. *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, chapter Semantically

Resolving Type Mismatches in Scientific Workflows. Springer Berlin / Heidelberg, 2007.

[10] Eclipse.org. The eclipse modeling framework (emf) overview [online]. This paper presents a basic overview of EMF and its code generator patterns. Available from: `http://tinyurl.com/5vx8fv` [cited 06-12-2008].

[11] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, August 2003.

[12] Cheng Hian Goh, Stuart E. Madnick, and Michael D. Siegel. Semantic interoperability through context interchange: Representing and reasoning about data conflicts in heterogeneous and autonomous systems. Available from: `http://citeseer.ist.psu.edu/191060.html` [cited 06-10-2008].

[13] Object Management Group. Ontology definition metamodel - omg adopted specification, 2007. OMG document number: ptc/2007-09-09.

[14] Tom Gruber. Ontology [online]. "to appear in the Encyclopedia of Database Systems, Ling Liu and M. Tamer Özsu (Eds.), Springer-Verlag, 2008.". Available from: `http://tomgruber.org/writing/ontology-definition-2007.htm` [cited 03-19-2008].

[15] Gerti Kappel, Elisabeth Kapsammer, Horst Kargl, Gerhard Kramler, Thomas Reiter, Werner Retschitzegger, Wieland Schwinger, and Manuel Wimmer. *MoDELS 2006*, chapter Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. Springer-Verlig Berlin Heidelberg, 2006.

[16] Vitaly Khusidman. Adm transformation. Technical report, Object Management Group, 2008. Draft 1 - Transformation Whitepaper. 17-Jun-08.

[17] Vitaly Khusidman and William Ulrich. Architecture-driven modernization: Transforming the enterprise [online]. Whitepaper. Draft v0.5. Available from: `http://www.omg.org/docs/admtf/07-12-01.pdf` [cited 05-12-2008].

[18] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. On-demand merging of traceability links with models. In *3rd ECMDA-Traceability Workshop*, 2006.

[19] Andreas Limyr, Tor Neple, Arne-Jørgen Berre, and Brian Elvesæter. Semaphore - a model-based semantic mapping framework, 2006.

[20] Deborah L. McGuinness. Ontologies come of age. In *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, 2003. From pre-print version found at: http://www-ksl.stanford.edu/people/dlm/papers/ontologies-come-of-age-abstract.html.

[21] Deborah L. McGuinness and Frank van Harmelen. Owl web ontology language overview [online]. W3C Recommendation 10 February 2004. Available from: `http://www.w3.org/TR/owl-features/` [cited 06-12-2008].

[22] Merriam-Webster. Ontology [online]. Available from: `http://www.merriam-webster.com/dictionary/ontology` [cited 03-24-2008].

[23] Jishnu Mukerji and Joaquin Miller. Mda guide v1.0.1, 2003.

[24] Protégé project. What is protégé? [online]. Available from: `http://protege.stanford.edu/overview/` [cited 05-28-2008].

[25] Eclipse Model Development Tools. Eodm [online]. Available from: `http://www.eclipse.org/modeling/mdt/?project=eodm` [cited 03-24-2008].

[26] W3C. Ontology driven architectures and potential uses of the semantic web in systems and software engineering [online]. Available from: `http://www.w3.org/2001/sw/BestPractices/SE/ODA/` [cited 03-21-2008].

[27] W3C. Rdf vocabulary description language 1.0: Rdf schema. W3C Recommendation 10 February 2004. Available from: `http://www.w3.org/TR/rdf-schema/` [cited 04-21-2008].

[28] W3C. Semantic annotations for wsdl and xml schema [online]. W3C Recommendation 28 August 2007. Available from: `http://www.w3.org/TR/sawsdl/` [cited 05-23-2008].

[29] H. Wache, V. ogele, T. Visser, U. Stuckenschmidt, H. Schuster, G. Neumann, and H. ubner. Ontology-based integration of information - a survey of existing approaches [online]. 2001. Available from: `http://citeseer.ist.psu.edu/article/wache01ontologybased.html` [cited 05-14-2008].

APPENDIX

# A

# RETRIEVING SMAT CONCEPT MODELS

The conceptual models can be download from:

*http://www.brovig.org/smat/smat.zip*

The models were built using Eclipse Ganymede.

The zip file is a Eclipse project. You can either unzip the file as is, or import it as a project in Eclipse itself using the Eclipse *import* dialog under *file*.