HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Laboratory of Software Technology

Tuukka Lehtonen

Ontology-Based Diagram Methods in Process Modelling and Simulation

Master's Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Technology.

Espoo, May 7, 2007

Supervisor: Professor Eljas Soisalon-Soininen

Instructor: Tommi Karhela, D.Sc. (Tech.)

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author:	Tuukka Lehtonen
Name of the Thesis:	Ontology-Based Diagram methods in Process Modelling and Simulation
Date:	May 7, 2007 **Number of pages:** XII + 94
Department:	Department of Computer Science and Engineering
Professorship:	T-106 Software Technology
Supervisor:	Prof. Eljas Soisalon-Soininen
Instructor:	Tommi Karhela, D.Sc. (Tech.)

Configuration of process models often involves drawing of process flow diagrams (PFD) or piping and instrument diagrams (P&ID), which can be considered schematic illustrations of an underlying process information model. In the process industry, diagrams connected to a process information model are referred to as *intelligent diagrams*.

By integrating simulation facilities with an intelligent diagramming environment, a designer is allowed continuous testing of models and faster discovery of design flaws. Simulation results can also be used to drive intuitive and informative visualisations of process state. This combination of simulation and diagram visualisation has been referred to in the industry as *behaving diagrams*.

In this thesis, a graphics framework and an ontology-based flowsheet diagramming tool implementation are presented to serve as groundwork for a future implementation of behaving diagrams. Using this tool a case-specific simulation model can be kept synchronised with a modelled diagram through an ontology mapping mechanism. Ontologies were defined for simulation, 2D graphics, diagrams and ontology mappings. Everything is modelled in a unified graph data model, which allows highly versatile association of information. The defined ontologies, implementations and a simulation case are presented and mirrored against the requirements set for the work.

Keywords: diagram, ontology, process modelling, simulation

TEKNILLINEN KORKEAKOULU DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Tuukka Lehtonen	
Työn nimi:	Ontologiapohjaiset kaaviomenetelmät prosessimallinnuksessa ja -simuloinnissa	
Päivämäärä:	7.5.2007	**Sivuja:** XII + 94
Osasto:	Tietotekniikan osasto	
Professuuri:	T-106 Ohjelmistotekniikka	
Työn valvoja:	Prof. Eljas Soisalon-Soininen	
Työn ohjaaja:	TkT Tommi Karhela	

Prosessimalleja konfiguroidaan usein piirtämällä virtauskaavioita tai PI-kaavioita, joita voidaan pitää allaolevaa tehdastietomallia kuvaavina kaavamaisina piirroksina. Kaavioita, jotka on liitetty tehdastietomalliin, kutsutaan prosessiteollisuudessa *älykkäiksi kaavioiksi*.

Integroimalla simulaatiopalveluita älykkääseen kaavionpiirtoympäristöön voidaan tarjota suunnittelijalle mahdollisuus mallien jatkuvaan testaukseen ja täten suunnitteluvirheiden nopeampaan havaitsemiseen. Simulaatiotuloksia voidaan myös visualisoida kaavioilla, jolloin mallintaja saa paremman käsityksen prosessin käyttäytymisestä. Kyseisellä tavalla simulaatiota ja visualisointia yhdistävistä kaavioista on käytetty teollisuudessa nimitystä *käyttäytyvät kaaviot*.

Tässä työssä esitellään grafikkasovelluskehys ja ontologiapohjainen vuokaavionpiirtotyökalu, jotka tulevat toimimaan pohjana käyttäytyvien kaavioiden ympäristön kehitykselle. Määritellemällä ontologioiden välisiä kuvauksia, voidaan kyseisellä työkalulla pitää simulaatiomalli yhtenäisenä kaaviomallin kanssa. Ontologioita määritellään simulaatiomallille, 2D grafiikalle, kaavioille sekä ontologioiden välisille kuvauksille. Mallinnus pohjautuu yhtenäiseen graafitietomalliin, joka mahdollistaa tiedon vapaan yhdistelyn. Työssä esitetään määritellyt ontologiat, ohjelmistototeutukset, sekä simuloitava esimerkkiprosessimalli ja peilataan tuloksia määriteltyjä vaatimuksia vasten.

Avainsanat: kaaviokuva, ontologia, prosessimallinnus, simulointi

Acknowledgements

I want to thank my supervisor for his help and comments on my thesis.

I consider myself lucky to have had a most knowledgeable instructor and a good friend with me during this process and throughout the last two years — thank you.

My gratitude also goes to all the people at VTT who have helped me on my way, and especially my fellow thesis-workers for fruitful collaboration and pleasant company.

Finally, I would like to thank my parents, siblings and friends for all the possible support one could ever want.

I'll take the red pill, please. I want to see how deep this rabbit hole goes.

Espoo, May 7, 2007

Tuukka Lehtonen

Contents

Abbreviations

API	Application Programming Interface
CAD	Computer-Aided Design
CSG	Constructive Solid Geometry
DAML	The DARPA Agent Markup Language
DOM	Document Object Model
EAI	Enterprise Application Integration
EII	Enterprise Information Integration
EMF	Eclipse Modelling Framework
GEF	Graphical Editing Framework
GMF	Graphical Modelling Framework
GVT	Graphics Vector Toolkit
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
JDT	Java Development Tools (part of Eclipse SDK)
MOF	Meta Object Facility
MVC	Model-View-Controller
NIST	National Institute of Standard and Technology
OIL	Ontology Inference Layer

OMG	Object Management Group
OWL	Web Ontology Language
PFD	Process Flow Diagram
PGML	Precision Graphics Markup Language
P&ID	Piping and Instrument Diagram
RCP	(Eclipse) Rich Client Platform
RDF	Resource Description Framework
RDFS	RDF Schema
SVG	Scalable Vector Graphics
UML	Unified Modeling Language
URI	Uniform Resource Identifier
VML	Vector Markup Language
W3C	World Wide Web Consortium
XML	Extensible Markup Language

Glossary

Notation	Description	
conceptualisation	An abstract, simplified view of the world that we wish to represent for some purpose. [Gru93]	7
edge-labelled graph	A graph where the vertices are treated as indistinguisable and the edges are given distinguishable labels.	7
experiment	An test conducted on a process model in order to answer questions about the modelled process.	28
graph	A model consisting of vertices and edges between those vertices.	7
ontology	(1) Philosophically, the study of what might exist. [Flo03] (2) From the knowledge engineering point of view, an explicit specification of a conceptualisation. [Gru93]	7
process model	A computer interpretable representation of a process. For example process simulation models are often mathematical models.	28
process plant	Facilities and structures necessary for performing a *process*. [Int97]	26
process simulation	An experiment made with a process model.	28

Notation	Description	
process	Sequence of chemical, physical or biological operations for the conversion, transport or storage of material or energy. [Int97]	26

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Generally speaking, *an industrial process* can be viewed as a kind of *activity* or a network of related activities. An activity on the other hand is simply something that occurs. *Process modelling* is thus the act of designing such a network of activities, be that design a paper drawing, a CAD drawing or a geometric 3D model. Process models tend to grow too complex too quickly for a person to intuitively understand how they actually work. This is where process simulators come into play. They allow us to see how our process would function under given conditions. The main purpose of process modelling and simulation is to create more or less complete virtual models and to simulate and validate their behaviour before actually building the processes in real life.

Processes and modelling as such are enormously broad concepts. Even by examining modelling and simulation in the industrial plant modelling context there is a very large selection of disciplines in use. In an industrial plant project modelling gradually proceeds from highly conceptual design to actual implementation design. Each stage of this modelling process has different goals and each can benefit from simulation.

Different modelling disciplines create different kinds of data, i.e. different models. Take for example *piping and instrument diagram* (P&ID) design. P&I diagrams are a schematic illustration of functional relationship of piping, instrumentation and system equipment components. The keywords in this definition are "schematic illustration". Diagrams are, in essence, only an illustration of the topology and components of the underlying process. They are a model consisting of graphical symbols and their connections that is separate from the underlying process information model. Now, it is *possible* to draw a P&ID with

any common image manipulation software. Needless to say, the user experience will not be very optimized for the task. Even worse, the created schematic image will not contain any information about the process, its topology nor the involved process components *in computer interpretable form*. On the other hand, a software tailored for P&ID design could manipulate the underlying process information model simultaneously while the designer creates a P&I diagram. Even better, the diagram model contents could automatically be bound to corresponding elements in the process information model, making all this information accessible to the P&ID designer. In the process industry, diagrams bound to process information models are referred to as *intelligent diagrams*.

From the point of view of a P&ID designer it would be very useful to have simulation integrated into the used modelling environment. This would allow the designer to incrementally construct and test the process model being created. This way the designer could very quickly see that the model is broken and is given better chances of rapidly discovering the culprit. Having such tight data and simulation integration also enables many useful features in user interfaces. Simulation results can be used to drive different kinds of visualisations which can provide the user valuable information about the state of the process faster and more intuitively. This kind of transparency of simulation and diagram animation has been referred to in the industry as *behaving diagrams*.

The current reality is that intelligent diagramming features are offered by many commercial grade solutions. Alas, behaving diagrams, i.e. simulation integration is not very advanced in many domains - in fact its rather non-existent. An extreme example of very loose data integration is that often P&ID designs are completely recreated for the simulators by copying the design from printouts into the simulation software. In this case the simulation model is completely disconnected from the original model which amounts to even more administrative headache if the original design gets updated. Obviously this feels like a waste of time compared to the utopia of behaving diagrams. With proper data integration the simulation model could be constructed from the same underlying process information model that the diagram is bound to in intelligent diagrams.

To realize this integration utopia a highly expressive and versatile information model is needed. Based on the idea of the Semantic Web ontology-based modelling techniques seem like the current best bet for the future. Using a versatile, unified, computer interpretable base modelling language provides for complex data integration, more efficient user interfaces and hopefully more intelligent software.

1.2 Objectives and Scope

This thesis is part of the Semill and Simbiot Research Projects conducted by the Technical Research Centre of Finland (VTT), Tampere University of Technology and Helsinki University of Technology.

The Goal of the Semill project is to study the application of semantic web technologies in plant modelling in order to provide better semantic information for the plant lifecycle supporting services.

The Simbiot project studies computational and information technological methods for plant model based multi-scale process simulation. The goal is to define semantic plant model extensions based on an open source simulation platform that enables the utilisation of process simulation on a new basis.

Design and implementation effort related to this thesis is performed in an ontology-based environment which is introduced in chapter 4. The primary objectives of this thesis are to:

- *Define a semantic connection between graphical data models and information models.* This also involves the construction of ontologies for the graphical representations.
- *Create a framework and a tool for integrated flowsheet diagramming and simulation visualisation on an ontology-based platform.* The framework should be generic enough to be usable for most kinds of diagramming in the process modelling and simulation domain.

It seems obvious that there is no such thing as a single generic graphical editor that is optimally suited for all possible workflows in 2D design. Often the main difference is in the content that is being communicated with a drawing. Layout sketches emphasise geometry and measures whereas P&ID focuses on process topology. These differences affect the nature of the tools needed for the job.

In order to create valuable, user-friendly diagramming tools for specific domains, customisation may be needed. Therefore, any graphical editing framework created in this thesis is kept on a fairly generic level. Instead, the created diagramming tools are customised towards flowsheet-style process modelling because many of the problems related to this work can be expressed with flowsheet drawings.

1.3 Structure of the Thesis

This thesis is divided into 7 chapters.

Chapter 1 is this introduction which provides the motivation, objectives and scope for this thesis.

Chapter 2 reviews the technologies related to the diagramming domain in the scope of this thesis. This includes ontologies and ontological data storage and processing, two dimensional graphics, industrial process design standards and a round-up of existing software frameworks and tools closely related to this work.

Chapter 3 analyses the different users and their requirements for the diagramming tools and framework implemented in this work.

Chapter 4 gives a semi-detailed overview of the used software implementation environment.

Chapter 5 focuses on the design and use of the ontologies created for the implementation.

Chapter 6 takes a shallow dive into the actual implementation.

Chapter 7 presents the results and evaluation in general and through a flowsheet diagramming demonstration case.

Chapter 2

Technology Review

2.1 Ontology-Based Modelling

Although the concept of an ontology has quite a long history, ontologies and semantic ontology-based modelling have become a hot research topics only during recent years. With the emergence of the idea of the Semantic Web the interest has grown steadily, even among the industry. Before stepping into the ontology world, the current interoperability and integration issues deserve a review.

2.1.1 Interoperability and Integration

We live in a highly heterogeneous environment of data and applications. For almost every application domain there is a variety of software available, most of which use their own methods and formats for data storage and exchange. Naturally if such applications of a single domain would like to exchange data, they would need a common protocol for conversation. If no international or de facto standards are available, it is highly unlikely that these applications are able to talk to each other.

Interoperability is commonly understood as the ability of distributed system components to exchange services and data with one another. *Semantic interoperability* in turn refers to making this data exchange make sense according to a common understanding of the data or service requests. [Hei95, OS99]

Semantic integration has also been a much discussed topic among database researches as noted in a recent survey [DH05]. The terms semantic integration and semantic interoperability are used somewhat interchangeably when talking about enabling data exchange

between applications in a semantic way. The study of semantic integration began already in the early 1980's in the form of *schema matching* and *schema integration* in the database research community [BLN86]. Schema integration is about (semi-)automatically finding the similarities (matches) between a given set of structured data schemas and merging them into a global schema. Schema integration research has produced different kinds of rule-based and learning-based methods for doing this [DH05].

Data integration refers to the problem of combining data residing at different sources and providing the user with a unified view of these data [Len02]. A familiar example of a data integration application may be found in public libraries that often provide access to literature databases that are combined from multiple sources under a single search engine. Scientific article search engines on the internet are also applications of data integration.

As described in [HRO06], data integration issues started getting commercial attention in the late 1990's and today this application field is known as *Enterprise Information Integration* (EII). EII generally focuses on the data and making queries on it. A slightly more mature sector, *Enterprise Application Integration* (EAI) instead focuses on use of software and architectural principles to integrate a set of applications for supporting certain workflows in a certain application domain.

All this talk about interoperability and integration applies to the process industry sector as well. Naturally an organisation can provide itself application integration by forcing the use of a certain set of proven applications. However, along with the growth of global internetworking of organisations and companies, interoperability problems are bound to surface through the use of such vendor-lockdown integration tactics. Financially speaking, NIST interoperability studies [oSN04] conservatively estimate that 15.8 billion dollars were lost in 2002 on poor integration and information exchange problems in the capital facility industry alone. This is a clear indication that work remains to be done in the area of interoperability.

2.1.2 Ontologies

Ontology-based modelling is a fundamental building block of this thesis, but what it actually means is usually unclear to the general public. According to [Flo03] the word ontology originates from philosophy referring quite pretentiously to "the study of what might exist". It is often considered synonymous to *metaphysics* — a term used by students of Aristotle. In the philosophical sense, ontology is often thought of as an exhaustive classification of all existing entities, which obviously is an enormous task. In the context of this thesis though, ontologies are being looked at from a more practical onto-

logical engineering point of view. For this purpose a more recent definition of ontology by Gruber [Gru93] is more appropriate:

> *An ontology is an explicit specification of a conceptualisation.*

Gruber defines a *conceptualisation* as *an abstract, simplified view of the world that we wish to represent for some purpose*. An *explicit specification* could also be called a *formal description*. An example of such a formal description could be a file written in a machine-readable and -understandable encoding. The fundamental idea is that description actually stores the semantics of the conceptualisation being covered. The definition also emphasizes practicality by confining to a specific world to represent instead of classifying the whole universe.

Another, a bit more concrete definition for the word conceptualisation can be extracted from W3C's Web Ontology Language (OWL) specification [Wor04c] as describing "*the kinds of entities in the world and how they are related*". The two keywords here are ***entities*** and ***related***. They suggest that the data model used for describing ontologies is graph-like — just identify entities as graph vertices (nodes) and relations as graph edges. Consequently the data model for ontologies is not strictly structured, but rather semistructured where basically any entity can be associated with another entity by an identifiable relation. [Bun97] overviews the concept of semi-structured data and also states that the data representation is graph-like and in an example, refers to the model as an *edge-labelled graph*. This definition implies that the edges of the graph model are actually the creators of the formal description, i.e. the ontology.

The above talk about data models lacks a very important aspect — what are ontologies actually for? In [UG96] three main problem areas are identified that suffer from the lack of a shared understanding that ontologies could provide:

- *Communication* between people and organisations with different needs and viewpoints arising from their differing contexts.
- *Interoperability* between systems is severely hindered by the use of disparate modelling methods, paradigms, languages and software tools. Ontologies could be used as an inter-lingua to unify the field.
- *Systems engineering*, in particular, specification, reliability and reusability. Having a shared understanding can help people of different domains in IT system specification, potentially provide for more reliable software through automated consistency

checking, and enable better reusability of information and models through reusable formal encoding.

As [UG96, Roc04] point out, ontologies can actually take many forms depending on their intended use. Firstly, the formality of the specification can vary between highly-informal, semi-informal, semi-formal and rigorously-formal, i.e. anything from loose natural language specifications to meticulously defined terms with formal semantics, theorems and proofs of soundness and completeness. Secondly, the type of the represented knowledge varies. For example, the following types are identified in bottom-up order:

- *a meta-ontology*, also called ontology languages, specify the concepts and principles for defining other ontologies. Commonly this includes concepts like class and property.
- *a generic ontology*, also known as top or upper ontologies, which try to model general concepts of the surrounding world up to varying degrees without regard to particular domains or applications. Time, space, and mathematics are all examples of general concepts.
- *a domain ontology* embraces a certain domain in a generic fashion, thus aiming for reusability in that domain.
- *an application ontology* gathers more or less specialized knowledge for a particular task. In general these ontologies are not reusable.

Although this is just one possible categorisation, even a very rough one, a similar categorisation has been used by another author also [Gua98].

Whereas the previous section focused on semantic integration in the database world, [KS03] and [Noy04] review ontology-based approaches for semantic integration. *Ontology mapping* is advertised as the key technique for integrating data models of different domains in ontology-based modelling. Many mapping methods have been explored — some involving logic descriptions, some based on creating mapping ontologies on top of the mapped ontologies. In principle ontology mapping is about finding similarities between ontologies and creating mappings for transforming instance data of one ontology to instance data of another ontology, either manually or (semi-)automatically. For thorough reviews of previously researched methods, see [KS03, KHRS05].

2.1.3 Designing Ontologies

When ontologies are designed, decisions are made on how to represent things. In [Gru95], Gruber fittingly states on design evaluation as follows:

> *To guide and evaluate our designs, we need objective criteria that are founded on the purpose of the resulting artifact, rather than based on a priori notions of naturalness or Truth.*

Gruber also proposes a preliminary set of design criteria for ontologies with the purpose of knowledge sharing and interoperation among programs. These principles are only summarised here as deemed necessary for evaluation purposes — see the original article for more complete definitions.

Clarity: An ontology should effectively communicate the intended meaning of the defined terms. The definitions should be objective, formally as complete as possible and documented with natural language.

Coherence: A coherent ontology should sanction inferences that are consistent with both formal and informal definitions of terms. Inferences contradictory with given axioms make an ontology incoherent.

Extendibility: An ontology should be designed to anticipate the uses of its shared vocabulary. The representation should allow for extension and specialisation of the ontology monotonically. In other words, one should be able to use the existing vocabulary to define new terms without having to revision the existing definitions.

Minimal encoding bias: The ontology should be specified without dependence on a particular encoding. An encoding bias results when representation choices are made for the convenience of notation or implementation. Encoding bias should be minimized to maximize the usability of the ontology in different systems.

Minimal ontological commitment: An ontology should make as few claims as possible about the world being modelled, allowing the parties committed to the ontology freedom to specialise the ontology as needed.

2.1.4 Ontologies in Practice

In general, ontology modelling is about identifying the relevant entities and their properties (relationships) in the modelled domain. The purpose of this section is to examine

languages for metadata representation and ontology modelling relevant for this thesis. The discussion is therefore limited to W3C standards.

Resource Description Framework — RDF

In subsection 2.1.2 ontologies were gathered to be formulatable as semistructured, edge-labelled graph models. This abstraction is very close to one reality. The Resource Description Framework (RDF) [Wor99] by W3C is a Semantic Web standard for representing information about *named resources*, also known as metadata representation. In other words, RDF provides a way to make statements about resources, by creating *named relationships* between them. These statements are expressed as $(s\ p\ o)$ triples, where p (*predicate*) identifies the relationship between the resources s (*subject*) and o (*object*). In the RDF lingo the predicate is also known as the *property* of the triple. Therefore one way of reading a triple is: "s *has a property* p *with the value* o". To make these statements machine processable, the triple parts need to be made identifiable. RDF uses URI references [ea98] to supply these identifications. Thus in effect, the RDF data model is an *edge-labeled directed graph*, formed out of simple triples. Figure 2.1 shows how this model is generally illustrated.

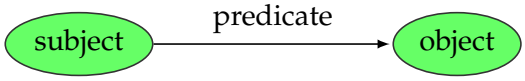

Figure 2.1: The basic RDF graph data model illustrated.

Representation of primitive data, such as strings and numbers, is done with *literals* in RDF. Literals can contain arbitrary primitive values encoded as unicode strings which are not confined by the URI specification. Literals do not have a separate identifier in addition to the literal value. Therefore a single literal can only be referred to once in a single triple, i.e. literals are not shared. As this implies, literal graph nodes never have outgoing edges and can only occur as the object of RDF statements. Figure 2.2 shows an example of an actual RDF graph with resources and a literal.

In order to store RDF graph models an encoding is needed. There are several serialization formats available, such as the XML-based RDF/XML [Wor04b]. Still, as [Her06] states, it is important to think of RDF models in terms of graphs and mind the serialization only as "syntactic sugar". Listing 2.1 shows how the RDF model in Figure 2.2 would look in RDF/XML.

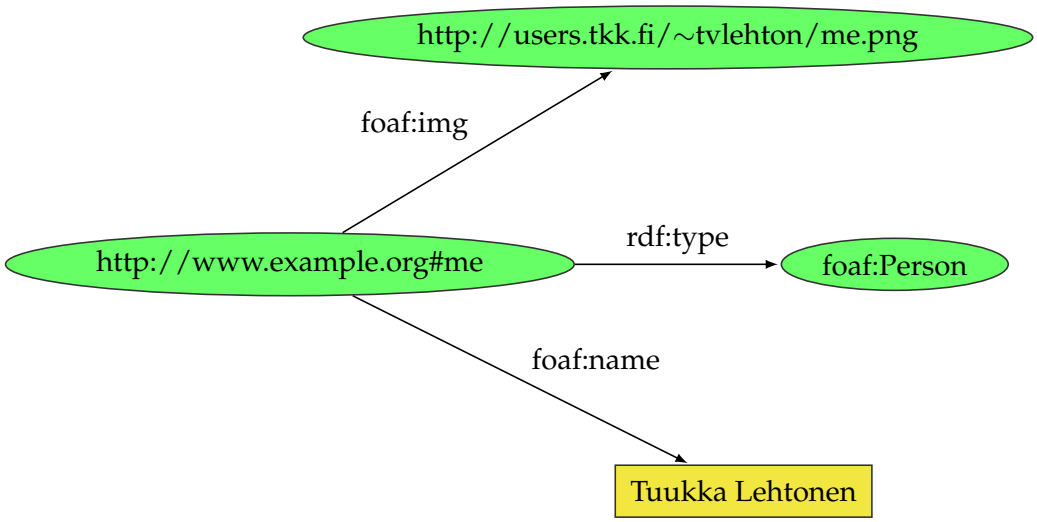

Figure 2.2: An example RDF graph illustrating three statements about a person. Resources are marked with ellipses and literals with rectangles.

Listing 2.1: RDF/XML serialization of Figure 2.2

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
  <!ENTITY foaf 'http://xmlns.com/foaf/0.1/'>
]>
<rdf:RDF xmlns:rdf="&rdf;" xmlns:foaf="&foaf;" >
  <rdf:Description rdf:about="http://www.example.org#me">
    <rdf:type rdf:resource="&foaf;Person"/>
    <foaf:name>Tuukka Lehtonen</foaf:name>
    <foaf:img rdf:resource="http://users.tkk.fi/~tvlehton/me.png"/>
  </rdf:Description>
</rdf:RDF>
```

RDF Schema

RDF only provides a model for expressing statements about resources with named properties and values. It does not provide any means to create application-specific vocabularies of types (classes) and properties and therefore is not a solution for semantic interoperability as such. RDF combined with RDF Schema (RDFS) [Wor04a] is a step in the direction of vocabulary description.

The core vocabulary of RDF and RDFS consist of *classes*, *properties*, *subclass* and *subproperty* relationships and property *domain* and *range* restrictions. Using these primitives, new class and property hierarchies can be specified and property domain and range can be defined. Note that these property domain and range definitions are universal for the re-

stricted property, not in the context of class definitions. See [Tuu06] for a more in-depth review of these features.

Web Ontology Language — OWL

OWL [Wor04c] has been W3C's recommended language for ontology definition since February 2004. It is based on the RDF and is derived from several earlier languages, especially DAML+OIL which was used as a starting point for design. OWL allows for more complex class description than RDFS, via for example class-specific property restrictions and property cardinality restrictions. OWL comes in three variants of increasing expressive power: OWL Lite $\subset$ OWL DL $\subset$ OWL Full. For details, see [Tuu06].

2.1.5 Semantic Data Storage

In most software systems, especially in useful and scalable ones, data persistence (storage) is needed. Semantic modelling tools are no different. As previously established, one way to express a graph model holding statements about resources is using triples for describing each statement separately. These triples are infact all it takes to describe data, metadata and ontologies.

General database systems can be used to store relations of arbitrary arity. A data model consisting of triples on the other hand only describes binary relations, i.e. relations with two participants, the subject and the object. This is the simplest possible relational model, but at the same time it is also highly versatile. The simple binary relation structure can be used to construct more complex relation structures. Compared to supporting arbitrary arity relations, sticking with binary relations only simplifies the storage layer. Implementation-wise, with this simple data model, everything boils down to storing and indexing a set of triples as efficiently as possible. Such components are often called *triple stores*.

As RDF has been a W3C recommendation since 1998, many triple stores have built-in support for the RDF data model and its semantics. These implementations often call themselves *RDF stores*. At the moment both open source and commercial RDF stores exist, which advertise scalability to hundreds of millions of triples or more. [Gea06, Ora05, Inc07]

2.1.6 Querying Semantic Models

The purpose of user interfaces is usually to offer a user a view into an underlying data model. To present that view the user interface needs to access that data model. In systems backed by relational databases SQL is the standard protocol, or query language for this purpose. In the semantic graph data model case, a mechanism for querying statement triples is needed. Generally speaking, the results of these queries are a subset of the set of triples in the whole model — usually a very small subset. If logical inferencing is supported, the query results may also contain statements entailing from the existing statements. For example, RDFS and OWL have a set of entailment rules which can be used to derive the new knowledge from existing statements.

There are several possible ways of querying the triple model. Since triple stores are still a fairly new concept compared to relational databases, a de facto query language standard has not really emerged yet. The strongest contender for the future is W3C's SPARQL [Wor06]. Instead of a whole query language, a more simple to implement way of accessing the graph data is browsing it relation by relation. In the following, these two query mechanisms are examined in more detail.

Browsing

People are familiar with browsing the internet through web pages. Because of the binary relational model of triple stores browsing the graph data model though named relationships is actually very similar to web browsing.

To browse the graph, a starting point is needed. This can be any resource in the graph. Triple stores generally support varying types of queries based on the three triple elements, subject, predicate and object. Generally the way to issue sensible queries is to constrain a part or parts of the result triples to certain values. The simplest example would be to query for all triplets which have the subject *http://www.example.org#me*. The result would be the set of triples in the current model that have the specified subject and anything as their predicate and object. To further constrain the query, the predicate could be fixed to *rdf:type*. In the example case of Figure 2.2 this would result in a single triple:

$$(\text{http://www.example.org\#me},\ \text{rdf:type},\ \text{foaf:Person}).$$

The following enumerates the types of browsing queries that make sense to perform:

- all triples with subject s,

- all triples with object o (resources, not literals),
- all triples with subject s and predicate p,
- all triples with object o and predicate p,
- all triples with predicate class p.

With the idea of subproperties and their logical entailments we can state that if resource s has a property p' which is a subproperty of p, s also has a property p. This logic should be employed in the browsing queries specifying predicates to make them more useful.

Query Languages

The concept of semi-structured data has spawned research on querying semi-structured models, such as [ea97]. Previously triple stores have offered their own alternative query languages, such as iTQL in Kowari or RDQL in Jena. Among others, RDQL has been used as a basis for the SPARQL specification. Recent triple store implementations have been mostly focusing on implementing SPARQL, such as Mulgara [Gea06] or AllegroGraph [Inc07].

SPARQL and its predecessors bear high resemblance to SQL. Both use a SELECT clause to identify the desired query results but differ in the use of WHERE clauses. In SPARQL the WHERE clauses consist triple patterns $(?s\ ?p\ ?o)$ and literal value constraints. Just as in simple graph browsing methods, these triple patterns can contain variables (e.g. $?s$) or constrained values and the variables can be used in the SELECT clauses for gathering the results. By chaining variables with multiple triple patterns (e.g. $\{(?x\ ?p_1\ ?y), (?y\ ?p_2\ ?z), \ldots\}$) a single query can traverse through multiple relations in the graph and collect results from there. In a sense, graph query languages simply offer syntactic sugar on top of a the simple step-by-step browsing API.

For the common developer, standardised query languages offer an easy to use protocol for versatile data access. Relational databases would probably be far less popular if SQL did not exist. The same goes for triple stores also.

2.2 Graphical Modelling

2.2.1 2D Graphics

Two-dimensional (2D) graphics is used in the process industry for many purposes, such as drawing diagrams on different levels of detail. 2D and 3D graphics are often used in close combination when different design disciplines are integrated.

Vector Graphics

A digital image is usually a regular 2D grid of pixels. Simple raster images do not quite cover what is interpreted as computer graphics. As Duce et al. describe in [DHH02], computer graphics is about more than just showing pixels but rather about creating, manipulating, analysing and interacting with pictorial data using computers. Interaction with raster images is fairly limited, especially for hyperlinking on the web. Still, raster images are a good representation for photographs and artistic drawings, which are really created just to be viewed and possibly manipulated without the need interaction. Using compressed image file formats photographs can generally be stored more efficiently than by using vector-based representations.

In principle vector graphics is about having a set mathematically defined *graphical elements* for constructing geometry, such as rectangles, ellipses, polylines, polygons and smooth curves. Since these elements are mathematically defined, they can be unlimitedly scaled (zoomed). To make the geometry visual, i.e. to rasterize the geometry representation, a kind of *paint* is needed. In 2D vector graphics, paint can generally be applied in two ways, *filling* and *stroking*. Filling means applying a specific paint (e.g. a flat color or gradient) to the interior area of a specific graphical element. Stroking refers to painting the *outline* of a graphical element with a specific brush where the brush attributes define how the outline will look.

Standards

Although there are several vector graphics content production applications on the market most of them have their own proprietary formats. The following only focuses on open standards, which are supported by most viable vendors.

WebCGM

Already in 1987, Computer Graphics Metafile (CGM) [Int99] first became an ISO standard for storing and exchanging graphics. It can handle vector, raster and text data and it has been used mainly for technical illustration (TI) in aerospace, defense, automotive and electronics industries among others. Technical illustration is a market of its own which is not that keen on stylability or other fancy aspects as it is on precise and complete specification of graphics. CGM is often used as the format of the final output that end users look at, not for engineering CAD data exchange. CAD drawings are also often used for creating the initial 2D illustrations. [Aut98].

Some industrial sectors have created their own CGM profiles. Profiles are subsets of the original specification and their purpose is to improve interoperability within a specific community. The problem was that these profiles lacked a vendor-neutral and interoperable hyperlinking mechanism. In 1999 the WebCGM profile [Wor01] was first released to fill this gap. The WebCGM profile adds additional constraints to improve interoperability, defines how hyperlinking works in accordance with current web technologies, and defines mechanisms for use in HTML. The WebCGM 2.0 specification [Wor07] was recently released, adding some important feature requirements that were left out of 1.0. See Figure 2.3 for WebCGM technical illustration examples. [1]

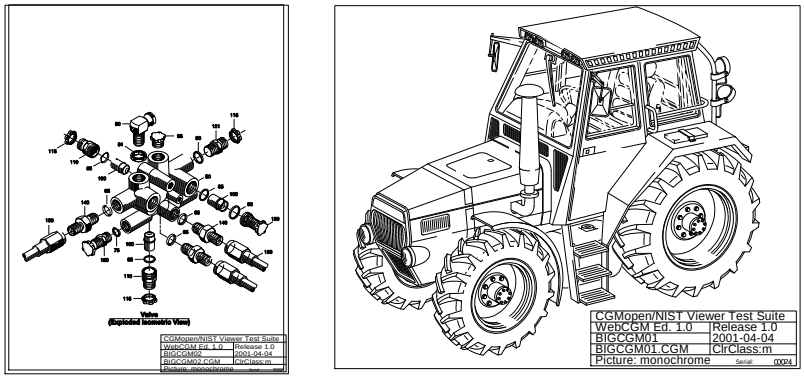

Figure 2.3: Examples of technical illustrations in WebCGM format. (Courtesy of WebCGM 1.0 Conformance Test Suite)

CGM itself allows for defining multiple encodings but the binary encoding is the most used one. CGM is optimized for small size since the amount of graphical primitives can

[1] Images courtesy of WebCGM 1.0 Conformance Test Suite: http://www.cgmopen.org/resources/test/index.html. Copyright ©2002, Lofton Henderson. All Rights Reserved.

easily grow up to tens of thousands.

An important characteristic of WebCGM is its completeness. It explicitly prohibits the inclusion of any and all private data. This forces both vendors and users to stay with a a rich but strictly limited set of functionality and features to achieve the common goal of interoperability.

CGM in general is designed to be self-contained, which means that it does not need data from outside the single file to produce a proper rasterized result image. Being self-contained is a requirement from the point of view of versioning and illustration management because in large projects the amount of separate drawings can grow to thousands or even more. Yet, linking to the outside world from the drawings is allowed, as creation of links between drawings is of high importance.

Requirements such as reliability, longevity and interoperability are of key value for technical graphics. Drawings need to stand the test of time, otherwise previous work is lost. For example the Boeing 747 aircraft has been developed and illustrated in the late 60s and early 70s and those illustrations are still in use today. The amount of illustrations needed to document such a Boeing (approximately 70000) is also something worth considering. [LW01, HW04]

SVG

SVG was born out of the desire and need for a high quality vector graphics standard on the web with extensibility in the late 90s. It is an application of XML and oriented for a different market than WebCGM. While CGM is intended purely for technical illustration, SVG tries to address graphic design for advertising, clip art, business presentations and general web use, requiring complex fills, restyling, image clipping and manipulation, reusable components and animation. It is designed to be stylable and to work well across platforms, output resolutions and color spaces. Good integration with XML and other W3C specifications is considered one of its key characteristics. SVG was not only an upgrade from raster graphics to vector graphics on the web, but also an attempt to achieve print-like quality on web pages. The specification was created based on several proposal submissions to the working group, such PGML [Ado98] and VML [Mic98]. The current recommended standard is SVG 1.1 [Wor03], 1.2 Tiny is a candidate recommendation and 1.2 Full is a working draft. [DHH02, HW04]

The current standard encoding of SVG is XML. Just as an XML document, an *SVG document* is made up of a hierarchy of *elements*. Each element in turn can have named *at-*

tributes. At the root of an *SVG document* is an <*svg*> element. Figure 2.4 shows an example of an SVG document along with the rendered output. SVG defines an element

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg width="12cm" height="4cm" viewBox="0 0 1200 400"
     xmlns="http://www.w3.org/2000/svg" version="1.1">
  <desc>Example rect02 - rounded rectangles</desc>
  <!-- Show outline of canvas using 'rect' element -->
  <rect x="1" y="1" width="1198" height="398"
        fill="none" stroke="blue" stroke-width="2"/>
  <rect x="100" y="100" width="400" height="200" rx="50"
        fill="green" />
  <g transform="translate(700 210) rotate(-30)">
    <rect x="0" y="0" width="400" height="200" rx="50"
          fill="none" stroke="purple" stroke-width="30" />
  </g>
</svg>
```

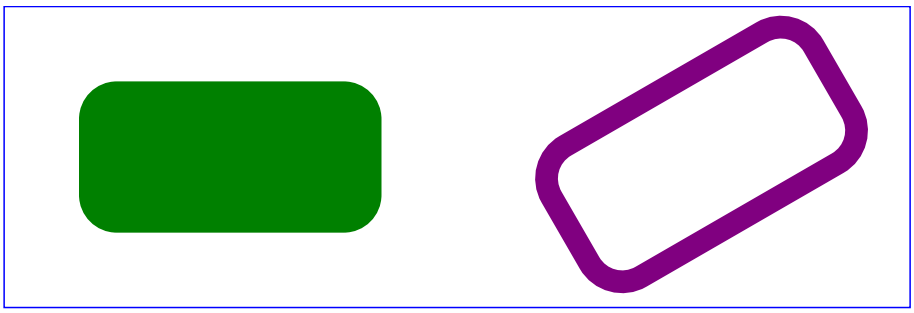

Figure 2.4: An example of a simple SVG file (top) with the rendered output (bottom).

for each supported graphical primitive, such as <*rect*>, <*circle*> and <*path*>. Primitive elements cannot have primitive subelements. SVG defines the <*g*> element for grouping together related graphical elements. Graphical element re-use is made possible through the <*use*> element. Styling can be added with specific attributes on graphical elements or standard CSS mechanisms. Rendering an SVG document is done by a pre-order traversal of the document tree. Rendering order can thus be controlled by reordering graphical and grouping elements in the document tree.

SVG content can be animated through modifying animatable attributes of graphical elements which can currently be done in two ways:

- Using SVG's animation elements, which allow the definition of time-based attribute value interpolations.
- Using the SVG DOM (Document Object Model) API to directly manipulate the

structure and content of an SVG Document.

In the context of process modelling and simulation, both animation methods can be considered useful, but for different purposes. Using animation elements is possible only if there is data available for attribute value interpolation. Consider a case where a simulation run had been completed which produced a series of result values along with an animated visual presentation of those results. SVG animation elements could be used for exporting a self-contained version of this animation for stand-alone presentation later on. Generally speaking, SVG animations are well suited for off-line 2D visualisation purposes in process modelling. Instead, for real-time simulation result or process data visualisation the only option is the DOM API because there simply are no values available for SVG attribute interpolation.

SVG has been applied both in normal applications and on the web. It allows for example desktop icons or application graphics to be stored more compactly than corresponding raster graphics. Some vector graphics editors support SVG as as a storage or graphics exchange format. SVG has also been applied in web-based mapping applications for diffent kinds of visualisations of geographic data [Car, uis, CS04]. During the recent years accelerated graphics has been pushing into the embedded device market. After the release of the OpenVG standard [Khr], SVG has also gained much interest among mobile developers for many of the same purposes as on the desktop.

SVG has also been applied for ontology-based visualisation purposes. [ILO06] presents an approach where SVG is used for visualisation of geographic data in OWL format. The geographic data is described according to OTN (Ontology of Transportation Networks) which the authors also developed. They achieved highly flexible and extensible configuration of visualisations through the use of a separate SVG transformation ontology which describes how to transform OTN into SVG. Configuration of this transformation is only a matter of changing the used instances transformation ontology concepts. [LK06] outlines the ontology-based diagramming implementation presented in this thesis.

Image Composition

Porter and Duff first introduced the idea of implementing image composition as a post-processing operationg in [PD84]. They separate *mattes* from the 3-channel RGB images which represent the coverage info of the RGB image, also known as the *alpha-channel* of the image. Using these mattes one can take several images and layer them on top of each other and composite their pixel values according to their original color coverage. A

useful set of basic composition operations were also introduced.

This approach works in image space, on rasterized pixels and their coverage information only, which makes the approach nicely uniform. One of the main uses of image space compositing is finalizing graphics for user viewing, by filtering and combining image layers.

The current SVG 1.1 standard recommendation has support for composition through combining *filter effects* into a DAG (Directed Acyclic Graph) where source images get passed through the filters of choice, in the end producing a composited result image. The SVG 1.2 Tiny candicate recommendation attempts to simplify the definition of compositing through simple composition operation (*comp-op*) attributes in graphical primitives. [Nor05]

In the context of diagramming and process model diagrams, this kind of composition could be utilized for visualisation purposes. For example, one could create animated symbols of process components, e.g. tanks with fill level visualisations by compositing a simple rectangular shape on top of a more complex base layer representing a tank. Figure 2.5 illustrates the idea.

Figure 2.5: Image space composition example.

Constructive Geometry

Another way of combining different graphical elements, is to use *constructive geometry* techniques. As the name suggests, it is about constructing new geometry by combining existing geometry with different operations, such as unions, intersections and differences. Correspondingly, in 3D this is referred to as Constructive Solid Geometry (CSG). One implementation for performing these operations in 2D is GPC [Mur] and another in pure Java is JTS [Inc].

Constructive geometry is highly useful in pretty much any geometric modelling application, from simple 2D layout sketching to 3D modelling. Generally geometric modelling starts from simple graphical primitives and through several cuts, extrusions, unions and other operations, the details are added to the geometry.

In practice one could use constructive geometry for composition purposes as described

in the tank visualisation example in the previous section. By intersecting a complex base geometry with a layer representing the tank fill level the geometry that visualises the fill level of the tank can be attained and rendered separately. The main difference between these approaches is that of whether new geometry is created or not.

The usefulness of constructive geometry techniques in diagramming naturally depends on the target application. Constructive geometry can be highly useful in floor plans or measurement drawings, which are more CAD-type applications. On the other hand, it is less useful in flow diagrams where topology is of the essence, not geometry.

2.2.2 Graphical Editing Frameworks

One of the objectives of this thesis is to create a framework for creating 2D process modelling and simulation result visualisation it is worthwhile taking a look at existing 2D frameworks.

Since editing (modelling) is a key desired feature, this review also focuses on *graphical editing frameworks* instead of just general graphical frameworks. Some of the things desirable from such frameworks may be:

(GFP1) Scene-graph Scene-graphs are generally used for controlling the renderer data, rendering methods and order of rendering. The concept of a scene-graph is more familiar in the 3D rendering context.

(GFP2) Efficient painting Despite the immensely powerful graphics cards of today, it is still beneficial with large graphical models to prune the amount of rendering via different visibility optimizations, such as spatial subdivision.

(GFP3) Picking Picking is used for determining the set of graphical elements under your pointing device at a given moment. Picking basically requires checking which graphical elements either intersect with or contain a given area. Often picking tends to be integrated into the scene-graph.

(GFP4) UI event handling and dispatching UI event handling and dispatching involves providing support for a forwarding lower level UI events to graphical elements as new events.

Other properties that affect the usefulness of a framework in the context of this thesis are:

Implementation Language The focus here is on Java frameworks, since that is the language of the thesis implementation.

Rendering Quality High quality anti-aliased rendering and support for image space composition (see subsection 2.2.1) are desirable features for a rendering engine.

Data Model One of the purposes of graphical editing frameworks is to provide means for representing an underlying data model. Some frameworks are indifferent to the data model whereas some frameworks enforce the use of their own base model. Indifference generally leaves the user more implementation burden while enforced models may be able to do more for the user. On the other hand, enforced models may prove to be unsuitable for ones purposes.

The *MVC architecture* (Model-View-Controller) has been widely adopted as a general solution for creating tools for controlling potentially large and complex data sets. The essential purpose of MVC is to bridge the gap between the human user's mental model and the digital model that exists in the computer. The ideal MVC solution supports the user illusion of seeing and manipulating the domain information directly. The structure is useful if the user needs to see the same model element simultaneously in different contexts and/or from different viewpoints. Figure 2.6 illustrates this idea. [Ree03]

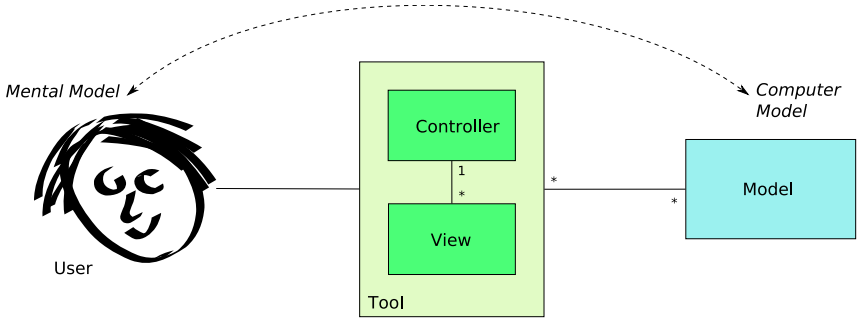

Figure 2.6: The Model-View-Controller architecture.

Batik

Batik is a Java based *toolkit* for applications or applets that want to use images in the Scalable Vector Graphics (SVG) format for various purposes, such as presentation, generation or manipulation. Batik features quite extensive support for the current SVG 1.1 standard and also some support for SVG 1.2 Full and Tiny drafts but these are subject to change.

Batik uses Java2D as its rendering engine. Java2D provides a good foundation for high-quality rendering and Porter-Duff composition, which Batik currently implements with

filter effects according to the SVG 1.1 standard.

Batik offers a Swing-based UI component for displaying and interacting with an SVG document. It can also be used for rasterizing SVG without using UI components.

Batik uses the standard SVG Document Object Model (DOM) as the user interface for feeding the rendering engine. Internally Batik converts the given DOM into its own tree of *graphics nodes*, a GVT tree (Graphics Vector Toolkit), which is more suitable for rendering and user interaction. The DOM and the GVT model are connected with a *Bridge* that listenes to both models and keeps them synchronised with each other. The DOM is a kind of scene-graph which in co-operation with the GVT tree provides fairly good support for all features **(GFP1)–(GFP4)**.

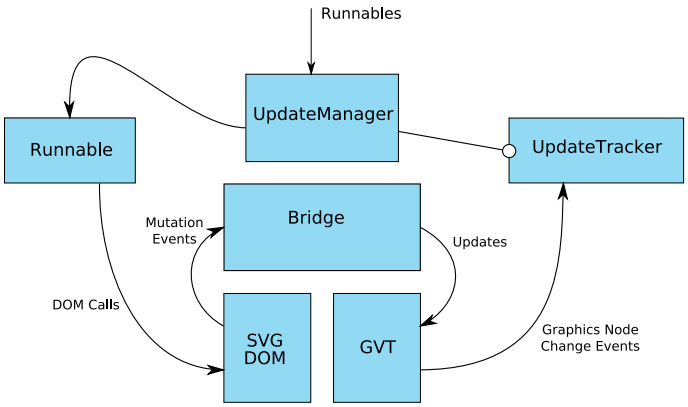

Figure 2.7: Updating the SVG DOM in Batik.

An example of how the DOM can be updated by the user and how the updates reflect to the other components is shown in Figure 2.7. All updates go through the *UpdateManager*. Its main purpose is to synchronise all DOM modifications. It runs as a separate thread and synchronously consumes Java *Runnables* that perform a batch of operations on the DOM. This circumvents the need for inefficient and complex fine-grained synchronisation in every DOM operation. Each DOM modification produces *mutation events* which the *bridge* listens to. The bridge updates the GVT model according to the received mutation events. Finally, by listening to GVT changes, the *UpdateTracker* keeps track of dirty regions that need updating. The UpdateManager uses this information to update the user view.

In conclusion, Batik provides a way for presenting and manipulating structured vector graphics according to the SVG standard and basic support for user interaction and picking for the graphical elements, either with JavaScript or through the SVG DOM. Just as

described, Batik is a *toolkit* that provides a basis for presenting and interacting with vector graphics, not a framework for graphical editing.

Piccolo

Piccolo [Uni] is a toolkit that supports the development of 2D structured graphics programs with the addition of Zoomable User Interfaces, coined ZUIs. It is implemented both in Java and C#. The idea of ZUI is to provide a framework for customized rendering based on the current "zoom level", i.e. view scaling.

Piccolo provides its own 2D scene-graph consisting of nodes with rectangular bounds through which it implements support for properties **(GFP1)–(GFP4)**. Piccolo does not provide an MVC framework for editing a model — it is model-agnostic and any editing facilities are left up to the user to build.

The authors of Piccolo regard its design as being *monolithic*, meaning that extending it is based on inheritance rather than composition. The approach and the reasons behind it are described in their article on graphical toolkit design [BGM04].

GEF

The Eclipse Graphical Editing Framework (GEF) [Ecla] is an MVC framework for creating customized graphical editors in the Eclipse RCP (Rich Client Platform) environment. The framework is model-agnostic, i.e. it only considers the model as a Java Object and it is up to the user to actually implement the model. The only requirement is that the model needs to have a change notification mechanism. This makes GEF a useful candidate for most editing scenarios. GEF provides Eclipse Workbench integration and an interaction layer which uses the *command* design pattern [GHJV95] for applying user interaction to modify the model. Despite the fact that GEF can be used to create any kind of custom graphical editor, the basic framework provides support for placing graphical elements on a canvas and creating connections between them. This editing pattern fits diagramming applications quite well, such as UML editors.

GEF uses a rendering framework called Draw2D which implements a lightweight graphical toolkit on top of Eclipse's own SWT (Standard Widget Toolkit) [Eclc]. Draw2D provides a *Figure* abstraction for representing a hierarchy of nested rectangular areas with customizable painting and propagating UI events to those figures in the same way as widget toolkit propagate events to their widgets. Draw2D can be viewed as a kind of scene-graph which provides support for all properties **(GFP1)–(GFP4)** listed above.

Recently successful attempts have been made to replace Draw2D's SWT backend with OpenGL [Cro07]. Performance-wise this is most intriguing and it should open up new possibilities for 2D and 3D integration in GEF-based tools. Although not the most up-to-date, probably the most comprehensive documentation of GEF is available as an IBM Redbook [MDG+04].

GMF

The Eclipse Graphical Modelling Framework (GMF) [Eclb] is an extension of GEF and Eclipse Modelling Framework (EMF). It uses GEF and Draw2D for its rendering purposes, but unlike Piccolo and GEF, it is not model-agnostic but is based on the use of EMF models.

EMF is the basis for all of the modeling tools built on top of the Model Development Toolkit (MDT) of the Eclipse platform. It's main contribution is a meta meta model, i.e. a facility for defining domain-specific meta models which in turn serve as the type system for models. The meta meta model is called Ecore. Ecore has evolved in parallel with OMG's Meta Object Facility (MOF) 1.4 model. It can be considered analogous to a meta-ontology.

The fundamental idea of GMF is to bridge EMF and GEF to allow graphical editing of Ecore domain models in a generative, model-driven fashion. The aim is that graphical editing tools can be constructed by modelling and code generation instead of manual programming. It also aims for reusability of graphics and tool definitions with different domain models.

The framework consists of two main components: *tooling* and *runtime*. The tooling is the generative part of the framework which is used by so called "toolsmiths" to model and generate the initial tools. The generated tools are then further customised by developers. The work of toolsmiths includes:

- modelling of a graphical definition for specifying the graphical elements used in graphical modelling
- modelling of a tool definition model for specifying runtime options supported, such as tool palette configuration and overview support.
- modelling of a mapping definition for binding the graphical definition elements to the domain model elements.

- modelling of a generator model for describing exactly how the tool code generation should be performed.

The runtime part of the framework consists of a set of plug-ins providing reusable components for the graphical editors, a standardised model for describing visual properties of diagram elements, an infrastructure for bridging EMF and GEF, and other services extensible via Eclipse extension-points. [Art07]

In summary, GMF allows model-driven creation of graphical editing tools faster and easier than previously by programming. Reusability of models also contributes to faster development times. GMF is thus a very attractive alternative when working with EMF models.

2.3 Process Modelling and Simulation

2.3.1 ISO 10628

The ISO 10628 standard [Int97] provides general rules and guidelines for the development of *flow diagrams* for process plants. Flow diagrams are used in multiple areas of the process industry, such as chemical, petrochemical, food, beverages and environmental industries. The main objective of the standard is to simplify the creation and understanding by domain specialists. Some the of the standard terminology in the process plant domain is defined in the standard. The most relevant ones are as follows:

Process: Sequence of chemical, physical or biological operations for the conversion, transport or storage of material or energy.

Process step: Part of a *process* which is predominantly self-sufficient and consists of one or several *unit operations*.

Unit Operation: Simplest operation in a process according to the theory of process technology.

Process plant: Facilities and structures necessary for performing a *process*.

Plant section: Part of a *process plant* that can, at least occasionally, be operated independently.

Equipment: Single parts of a plant, such as vessels, columns, heat exchangers, pumps, compressors.

A flow diagram is defined as a diagram representing the procedure, configuration and function of a *process plant* or *plant section*. The standard makes a distinction between three different types of flow diagrams depending on the level of detail of their content. The diagram type also dictates the type of the content the diagrams shall at least contain and also the additional information they may contain. The types are as follows from least detailed to most detailed:

Block diagrams: A block diagram depicts a process or process plant using rectangular frames including the relevant descriptions, interconnected by flow lines. The frames represent any of the above defined terminology and the lines may represent streams of materials or energy flows.

Process flow diagrams (PFD): A process flow diagram depicts a process or a process plant using graphical symbols, interconnected by flow lines. The symbols represent equipment and the lines represent flows of mass or energy or energy carriers. These are also referred to as *flowsheets*. An example of a PFD can be seen further ahead in Figure 2.9.

Piping and instrument diagrams (P&ID): The piping and instrument diagram is based on the process flow diagram. It represents the technical realization of a process using graphical symbols for equipment and piping combined with graphical symbols for process measurement and control functions.

Since the standard is about drawing diagrams, some rules for the actual drawing are also laid out. These include for instance flow line thicknesses, flow line spacing, equipment symbol dimensioning and diagram layout.

All in all, the standard and other drawing-related ISO standards serve as good guidelines for creating standard-compliant graphical editors for the process engineer and designer.

2.3.2 ISO 10303 AP-221

ISO 10303 Application Protocol 221 [Int05b], commonly referred to as AP221, concerns the functional and physical aspects of plant items. These aspects of a plant item are relevant to different activities, but both aspects are described by the same documents, e.g., P&IDs, data sheets and their electronic equivalents. The development of AP221 is currently suspended, pending harmonization with ISO 15926.

The principal focus of AP221 is the P&ID and property information about the plant items. The scope of AP221 includes e.g. plant system and equipment identification, connectiv-

ity, classification, definition of standard functional and physical classes, properties, materials and project data. [SP06]

AP221 is meant to be used in conjunction with the process plant Reference Data Library (RDL) which is being standardised as ISO 15926 part 4 [Int05a]. The RDL is a collection of process plant life-cycle data classes which are common to many process plants, for example activities, equipment and materials. In other words, the RDL attempts to gather knowledge about the functionality of process plants scattered around a large number of international process plant related standards into one place, modelled using a single data model (meta-model). As this single data model, the RDL uses the core data model defined in ISO 15926 Part 2 [Int03].

The relevance of AP221 for this thesis lies mainly in carrying across the idea of "intelligent plant data access", i.e. combining schematic illustrations (diagrams) with the actual process information models they represent. The standard generally regards all such illustrations as annotations of the information model. It also gives one possible view of what kind of concepts are needed to represent piping and instrument schematics as a graph data structure.

2.3.3 Process Simulation

Simulation in general refers to solving a mathematical model of any physical process, regardless of the amount of equations or the method of solution. Another widely accepted definition for simulation is the solution of a model of a system with a computer.

With these definitions, *process simulation* can be interpreted as solving a mathematical model of a *process* as per the definition of subsection 2.3.1, hereafter refered to as a *process model*. Another more pragmatic way of looking at process simulation is to think of it as an *experiment* performed with a process model where an experiment is a test conducted on a process model in order to answer questions about the modelled process. [BG02]

Steady-state Simulation

Steady-state is defined as a mathematical condition where the all properties of a system are constant with time. Although this condition very rarely happens in real life, steady-state simulation is very popular especially during the early design phases where systems are simulated on a higher level. Steady-state simulation is often applied for calculating mass and energy balances in chemical processes. Further applications include what-if analysis and process optimisation.

Dynamic simulation

Dynamic simulation describes the time-dependent behaviour of a system. This means that a simulation run produces a time-dependent data series instead of just producing a single set of result values. Generally it is not applied as widely as steady-state simulation, to some extent for the following reasons:

- Implementation of robust solvers for dynamic simulation is naturally harder than steady-state solver implementation due to the time-depdencies.
- Computational load and engineering complexity tend to be higher. More effort is required for setting up the model to perform correctly. For example dynamic solvers may require the user to input dimensioning information for the simulation to work properly which would not have been needed in corresponding steady-state simulations.

Yet this popularity issue is not quite as black and white as these reasons suggest. It is important to keep in mind that process simulation is an experiment — it is always performed in order to answer questions about a process. It is these questions and purposes that actually determine the needed simulations. Some questions are very high level where not that much details about the process are needed whereas other questions seek information about a very specific part of a process, such as the internals of a process equipment. Either way, questions that get asked the most also require simulation the most. [Kar07]

Optimisation

Optimisation is the mathematical process of finding such values for a set of independent variables that minimize a real-valued objective function. The objective function determines the goal of the optimisation, which can be based on for example operating cost or efficiency. For example optimisation methods can be applied in process plants to tuning operation either in real-time or with a simulation model when used in conjunction with steady-state or dynamic simulation.

2.3.4 Existing Tools

APROS / GRADES

APROS [VTTb] is a large-scale dynamic process simulator developed by VTT and Fortum. Large-scale means that it is intended for simulating complete processes, such as complete pulp and paper mills or power plants, including gas and liquid flow networks, process automation and electrical systems.

APROS is the simulator component of the system and only has a console user interface. APROS simulation models are configured graphically using a flowsheeting UI called *GRADES* (see Figure 2.8).

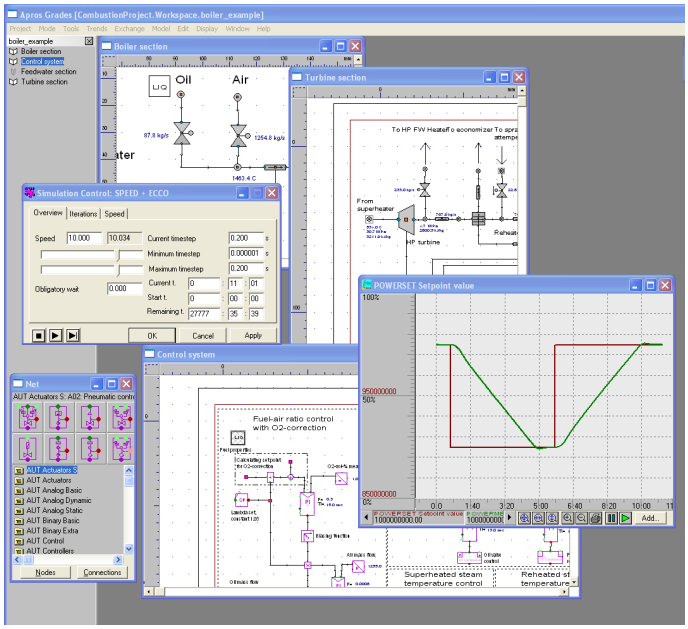

Figure 2.8: Apros Grades in action.

GRADES consists of two tools: a symbol editor and a net editor. The symbol editor is a fairly basic tool for constructing graphical composites (symbols) from graphical primitives and attaching terminals to them. A single simulation component can be given multiple different symbols. This is useful if the same simulation components are used for different domains or if the simulation component needs to be presented at different levels of detail.

The net editor in turn is the UI for simulation model configuration. It is used by drag-

ging component symbols from a palette onto the net and creating various connections between the inserted components. A project can also contain multiple nets. In case different nets need to be connected together at some point, GRADES employs a "slave copy" mechanism for doing this. It means that if a symbol S originally located on net A needs to be connected to another symbol on net B, the user needs to make a slave copy S' of S onto net B. Although the mechanism itself is fairly intuitive, it has received UI criticism about not being clear on which is the slave and which is the master [Pal07].

Simulation features are integrated into GRADES in multiple ways. First, the UI has a run/stop/step toolbar is for controlling simulation execution. Second, in order to actually see the results of the simulation the UI has support for textual monitors and 2D trends of single simulated values. Monitored values are shown directly on the net, while trends are shown in separate trend windows.

BALAS

BALAS® [VTTa] is a steady-state simulation package for chemical processes with emphasis on pulp and paper, also developed at VTT over the last 20 years.

The BALAS® environment is made up two separate programs: a graphical process design UI, *Flosheet*, and the simulator control UI of BALAS®. Both components attach to a runtime database component which is owned by the simulator UI process. The database contains all simulation model data, including units and streams along with their names and parameters, design functions, calculation cases and solver settings. The Database has a DCOM interface through which external parties, such as Flosheet or *Microsoft Excel* can access it.

Providing a mechanism for editing the simulation model properties is provided by the BALAS® process. The designer UI merely receives the UI events and tells BALAS® to show properties for a selected entity through the DCOM interface. The contents and functionality the property dialogue boxes are declaratively defined for each type of unit and stream in separate files.

In a more recent version, BALAS® successfully employs an existing diagram tool for graphical process design instead of Flosheet: *Microsoft Visio*. Just like Flosheet, Visio is only a used for creating a 2D visualisation along with the topology of the process to simulate. Process component symbols are provided as Visio *stencils*. Therefore, unlike GRADES, the Visio-based solution does not have a separate UI for drawing process component symbols. An example of a Visio-based BALAS® flowsheet model is shown in

Figure 2.9.

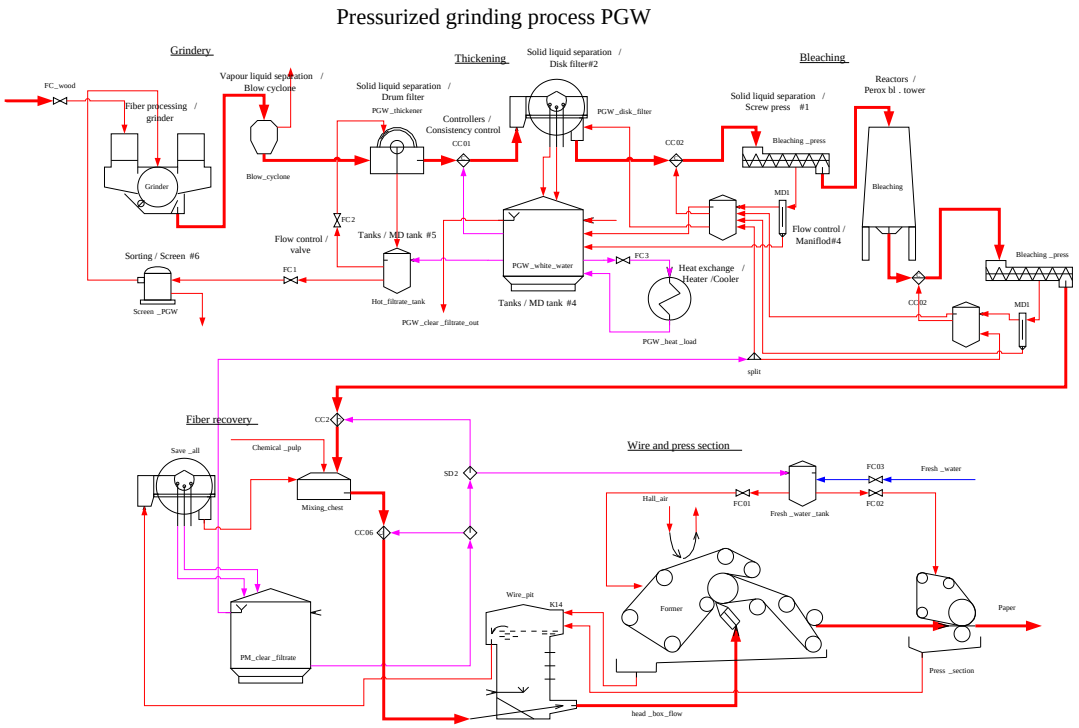

Figure 2.9: An example of a simulation model created with current Balas tools. It is very much representative of a process flow diagram (PFD).

On the visualisation front, the Visio interface supports monitoring of the simulation result values directly on the diagram through textual monitors. Another alternative is to use the ready-made Excel link that comes with BALAS®. The link allows manipulation of all BALAS® parameter and stream data and running any process from Excel. The advantage with Excel is that its built-in graphing features can be applied for constructing useful process visualisations and operation UI's. Another useful feature is the Excel-link's capability to take a series of input parameters for simulation and also produce a series of results.

All in all, BALAS® is a good example of adept use of existing tools for the construction of a simulation configuration and analysis environment. However, a more integrated model configuration UI could provide for a more streamlined user experience and better capabilities for simulation result visualisation.

It should be noted that in the future the work done in this thesis can serve as groundwork for a common 2D graphics configuration environment at VTT.

OpenModelica / MathModelica

Modelica is a declarative programming language specification for modeling and simulation of physical processes, developed by the Modelica Association [Mod, EM97]. OpenModelica aims to be a complete modelling, compilation and simulation environment for Modelica, released under an open source license [Pro].

Currently a trial version of a graphical editor called MathModelica Lite, developed by MathCore Engineering AB, is available for this environment [Mat]. The editor provides for creating Modelica models without actually writing the program code for the model. This is done by connecting basic graphical components, such as gain, integrator and derivator from libraries and adjusting their parameters to form those models. These composited graphical models can be parametrised and used hierarchically as a submodel in another, more complex model. The graphical representation of the models is stored along with the models as annotations, or typed comments, consisting of simple structured character data. MathModelica comes with a *Simulation Center* component which can be used for visual simulation of the constructed models.

SmartPlant P&ID

SmartPlant P&ID [Int] (SPPID) is an intelligent P&ID plant modelling tool developed by Intergraph Corporation. SPPID is just one of several plant modelling tools of different disciplines in the SmartPlant product family. It is included in this study only as an example of the current state-of-the-art in industrial P&ID modelling.

SPPID is not a simulation tool in itself, but can be interface with various process analysis tools, such as AspenTech simulation software. One of the cornerstones of SPPID is that produced P&IDs always stay integrated with the same underlying plant data model that other SmartPlant family products integrate with. This means that there is ultimately a single source of data which all tools illustrate and use to their advantage. Another product called *SmartPlant Foundation* is built for serving this purpose of a single repository for all plant data. All the tools in the family are capable of sourcing data from the foundation and possibly integrating new data into it. Despite apparent implementational and terminological differences, this integration approach of the SmartPlant product family approach has similarities with what we hope to achieve with the ontology-based approach described in this thesis.

SPPID comes with large sets of standard symbol libraries to streamline design tasks and facilitate reuse of earlier data and designs. *SmartSketch* is a 2D sketching tool for de-

signing new symbols among other purposes. To aid the designer, SmartSketch employs many kinds of snapping techniques, reminiscent of constraint-based design traits made familiar by Ivan Sutherland's SketchPad system already in the 1960s [Sut03].

Chapter 3

Requirement analysis

3.1 User Roles

A high-level user categorisation based on [Kar02] is used in this work. Each are analysed separately in the following sections. The users are presented in low to high level order along with related use cases. Only use cases closely related to 2D modelling are analysed here.

3.1.1 Kernel Developer

Kernel developers are seasoned experts on modeling and simulation and are responsible for developing the modeling concepts and algorithms. They can be found in universities, research institutes and research departments of companies. Their use cases are shown in Figure 3.1.

Generally speaking, kernel developers are the users who do all the groundwork needed for higher level users to be able to work. They are responsible for creating and maintaining the base conceptualisations, i.e. ontologies for domain information models which are extended by library developers. In the 2D graphics context they define the ontologies used for graphical illustration of the information models, which allow higher level developers to create graphical elements as representations of domain model concepts.

In the used ontology-based environment one goal is to keep models of different ontologies as separate as possible, e.g. for better reusability of models. The separation is also somewhat inherently assisted by the use of a binary relational graph data model. Yet, something is needed to bind the two models together to describe how they correspond

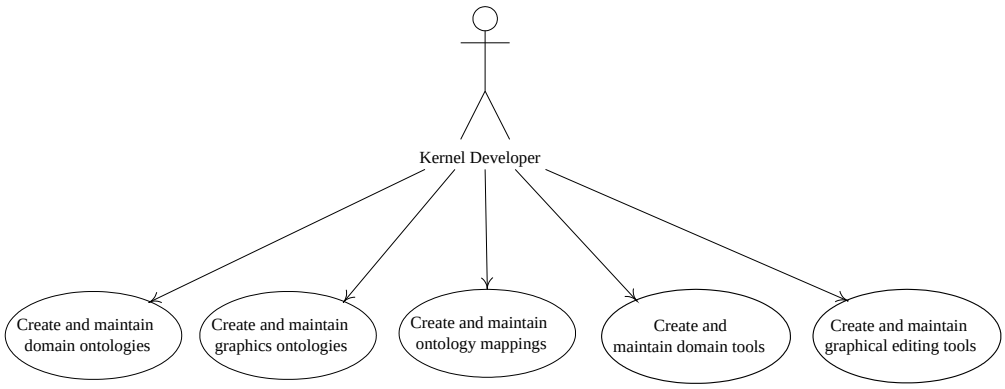

Figure 3.1: Use cases of Kernel developer.

to each other and furthermore, keep the models in synch with each other. An example of this case is keeping a domain information model and a graphical illustration in synch. To address this issue, kernel developers define ontology mappings. Ontology mappings are described further in subsection 5.1.5.

Kernel developers create or customise existing user interfaces for creating domain models. They also create and customise graphical user interfaces and tools for editing the graphical illustrations of the information models.

3.1.2 Library Developer

Library developers are domain experts who develop and maintain libraries of modeling constructs, which can be used to build models. Both domain information and graphics constructs need to be defined to support graphical modelling. The use cases are shown in Figure 3.2.

Domain library developers extend the domain ontologies created by kernel developers and other domain libraries. They define for example new process components for use in piping design. Companies may also want to define domain concepts for their own products.

In the flowsheet graphics context, symbol library developers use the graphical illustration ontologies and tools to define symbols. Symbols are considered two-dimensional graphical representations that may illustrate a domain concept or a part of a domain model. A single person may take on the tasks of both, domain library and symbol library developer. This separation simply emphasises that they are most likely not the same

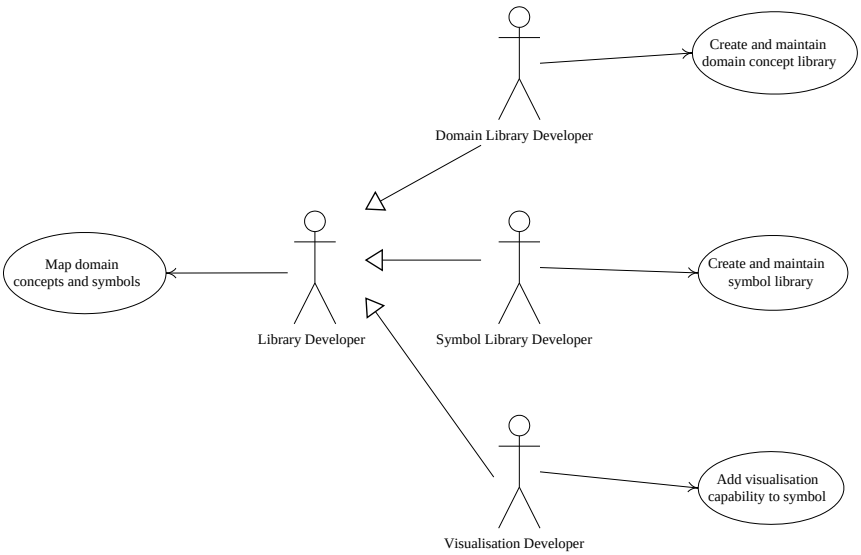

Figure 3.2: Use cases of library developer.

person.

Visualisation developers add support for visualisation capabilities in symbols to allow domain model information to be expressed graphically more intuitively for model configurators and model users. For example the fill level of a tank or the rotation speed of an engine could be depicted by the symbol itself in addition to seeing the values of these properties. Symbols are generally built for use in a particular domain, which also up to some degree dictates what kind of visualisations may be useful. The amount of useful visualisations is somewhat dependent on both the internal and external complexity of what is represented by the symbol. Again, a single person may be both a symbol library and visualisation developer, which is very likely indeed.

In order to keep domain models synchronised with graphics models, the ontology mappings defined by kernel developers may require extra information to be defined by library developers. However, this depends on how a particular ontology mapping works.

Domain concept design is most likely to consistute the largest part of all library development. This is because all domain concepts, such as different process components and different products of companies, need to be defined separately whereas symbols can be reused for many similar domain concepts. Often it is highly desirable for symbols to have a standard look to them so that domain experts quickly associate the graphics with particular concepts. It may also be that some library developers, such as companies, are

too busy to define symbols for their products, ending up reusing standard symbols. On the other hand it should be made possible for library developers, such as companies, to later create fancier and visually more capable symbols specifically for their own products. This implies that library developers need to be able to define multiple symbols for domain concepts and model configurators need to be able to use them interchangeably.

3.1.3 Model Configurator

Model configurators are engineers who are developing models from the building blocks and tools created by library developers and kernel developers. Their use cases are shown in Figure 3.3. In flowsheeting this development focuses on using the symbols and tools provided by library and kernel developers to build diagrams that may be mapped to domain models. Model configurators do not define ontologies nor ontology mappings.

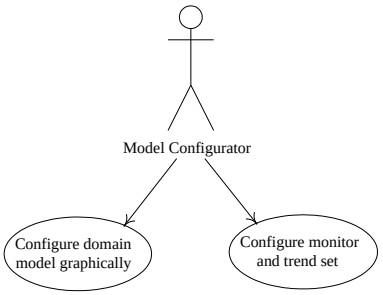

Figure 3.3: Use cases of model configurator.

For the created diagrams, model configurators may configure sets of monitors and trends. Monitors are a mechanism for graphically visualising selected values from the illustrated information model. Trends are used for graphically viewing the history and development of specified values.

If possible, model configurators may also want to test their models by simulating them from time to time. In this way they also act as model users. Especially when working with research simulators model configurators are often also model users.

There is a large amount of possible individual user groups for 2D drawing tools. These include for example mechanical designers, piping and instrumentation designers, building designers, district heating network designers and system dynamics modellers. All of these users need somewhat different tools to be able to configure models intuitively, although the created graphics models may prove to be highly similar or even generalis-

able.

3.1.4 Model User

Model users use existing models to acquire information using simulations or other types of analysis. In some sense they are end-users of everything that has been created and configured by the previous user groups.

Just as model configurators, model users are also not concerned with ontologies but use the tools provided for them. Any tools that require programming effort are created by kernel developers. It is also possible that some tools with simple form-type functionality could be created simply by means of graphical modelling. In this case they could also be created by model configurators.

As stated previously, a model configurator may often adopt the role of model user. However, there are also pure model users in the considered fields of 2D design. A common factor for model users of all domains is that they put already configured models to use by tuning and analysing them. It may even be that model configurators and model users use the same tools. On the other hand the user interfaces created for model users may also be completely customised. One example of a case for customisation are *plant operators*. Operators may be offered 2D views of e.g. gauges and meters for observing the state of a process more intuitively than by using a model configurator user interface.

Chapter 4

Implementation Environment

4.1 Simantics

The required implementation environment in this thesis is *Simantics* which is an open modelling and simulation environment, being developed by the Semantic Models research team at VTT [oF07]. The user interface parts of Simantics, called *ProConf*, are built on top of the Eclipse Rich Client Platform and Java technology. The Eclipse platform and plug-in architecture provide a solid foundation for building well integrated extensible applications.

The runtime system of the Eclipse platform allows dynamic loading and unloading of so called plug-ins, which are really nothing more than a mechanism for grouping, delivering and managing arbitrary content. In the Eclipse environment the content often consists of Java classes. The runtime also handles plug-in dependencies and Java classloader construction based on these dependencies, effectively limiting each plug-in's Java classpath to the paths exported by its dependencies. One of the key components of the platform is the *Extension Registry* which allows plug-ins to open themselves up for extension through declaratively defined (XML) extension-points. Extensions are defined by providing textual information, such as IDs and numbers or names of Java classes that implement interfaces defined by the extended extension-point. This mechanism allows contribution of Java interface implementations without imposing compile-time class dependencies between the extension-point and the extension. This is very important for dynamic extensibility. [ML05]

The idea of the Simantics environment is to serve as a platform of its own for building integrated and collaborative tools for different kinds of modelling and simulation needs,

such as process modelling or mechanical modelling both in 2D and 3D. The key building-block of the Simantics environment is its extensive use of a single unified RDF-like graph data model. All data in that graph model is based on a proprietary base ontology developed at VTT, called *Layer0*. This means that ontologies created in this environment are always based on at least the concepts defined in Layer0, just like in EMF everything is based on the Ecore meta meta model (see subsection 2.2.2).

The core of ProConf consists of plug-ins for connecting to the data model server and an Ontology Development Environment (ODE) for defining new ontologies and manipulating existing ones. In order to make any use of the defined ontologies, user interfaces are needed. This requires the creation one or more Eclipse plug-ins which take advantage of the Eclipse *Workbench UI* and its extension-points. Constructing UI code or any other code based on a set of ontologies leads to having that code depend on the existance of those ontologies in the data model at runtime. If the required ontologies have not been imported, the code will not work. Because all ontologies defined with ODE ultimately depend on Layer0, the data model to which ProConf connects must always contain at least a supported version of Layer0 or otherwise no UI can function.

Using the Eclipse Workbench UI forces the use of certain UI concepts, most importantly *perspectives*, *views* and *editors*. Views and editors are both mechanisms for showing UI components (e.g. trees, lists, text areas) inside the Eclipse Workbench with slightly different characteristics. Perspectives on the other hand can be compared to virtual desktops. They allow the customisation of the workbench layout for different working disciplines and workflows. All of these mechanisms are available to the user as extension-points. For example, the creation of a new view is a matter of defining a *view* extension for the *org.eclipse.ui.views* extension-point in a plug-in. Minimally this requires (1) an ID for the extension, (2) a name for the extension, and (3) the fully qualified name of a class implementing the interface *IViewPart*. For more information on these issues, see [Fou07] or [ML05].

4.2 Layer0

As already introduced in the previous section, Layer0 is the base ontology used in Simantics as the basis of modelling. It has minor similarities with RDFS/OWL but has been developed with different requirements. It has been often asked throughout the short history of Layer0 why OWL and RDF were not applied directly. The main reason has been our desire to rapidly steer its development in a direction more suitable to Simantics

environment requirements. Uschold and Gruninger suggest the following for ontology capture in section 5.2.1 of [UG96]:

> *Initially, do not commit to any meta-ontology. Doing so may constrain thinking and potentially lead to inadequate or incomplete definitions.*

They also state that this principle should only be applied before programming begins. This is probably true in some cases, but the distinction becomes more blurred along with modern agile software engineering methodologies such as extreme programming. In these methodologies designs and implementations evolve iteratively through refactoring, starting from the simplest solutions. Ontologies too need to evolve as it's very hard to construct ontologies that cater to all needs, even the unforeseen ones.

In the end, one of the reasons for departing from OWL boils down to the path that has been taken in Simantics environment for triple storage and graph data access and its suitability for OWL reasoning. Furthermore, the development of Simantics and ProConf has not yet reached the point of real applicability for reasoning services. Thus no particular attraction in embracing OWL is seen, regardless of its somewhat standard status in semantic web research. There's little sense in embracing it just because we can, not because of real value. Another issue is that OWL is designed for the semantic web and is rooted tightly to description logics. It emphasises the *open world assumption*, which implies that lack of knowledge does not imply falsity where reasoning is concerned. Yet, in the territory of simulation configurations the *closed world assumption* is more desirable because we cannot simulate a model only by the information about things it contains, we also have to know what it does not contain.

The following sections shed light on the basic data model of Layer0 and all the things it contains.

4.2.1 Data Model

The data model used in the environment can be thought of as a directed graph. In this graph the nodes have unique identifiers. For each edge an identifier is also assigned. The identifiers assigned to the edges of the graph have a corresponding node in the graph. Such a graph data structure can be represented by listing the set of edges between the nodes. In this case a single edge can be represented by a triple of identifiers $(start, edge, end)$. The nodes of the graph are called resources and represent the entities described in the model. The edges of the graph represent relations between the resources thus partially defining their semantics.

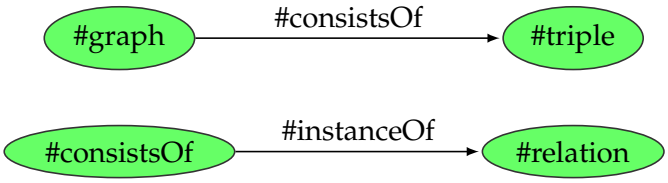

Figure 4.1: Triples in the semantic graph

Figure 4.1 models a resource with identifier *#graph*, which is partially defined by establishing an edge identified by *#consistsOf* to another resource *#triple*. The resource *#consistsOf* is further defined by connecting it to a resource identified *#relation* and identifying the edge by resource *#instanceOf*. The identifiers here are given names only for simplification purposes. In reality the identifiers for the nodes are used only internally in the system. The actual textual identification of the resources requires that the graph of resources is connected to some elements of primitive data. For this a special set of resources called variables are introduced. For each variable an item of primitive data is associated. The variable identification provides the type of the primitive data.

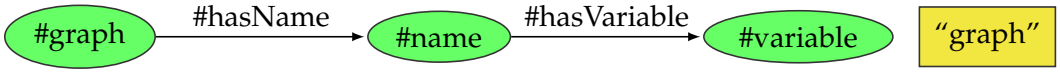

Figure 4.2: Primitive data associated with the graph of resources.

Figure 4.2 shows the assignment of a textual name to the resource *#graph*. The *#graph* resource is associated with a *#name* resource, which is associated with a *#variable* resource. For the *#variable* resource the system stores a primitive text string “graph”.

Building on this model of a graph of resources and associated primitive data some modeling standards need to be established. The standard concepts defined by the system are collected in the Layer0. Semantics of the Layer0 concepts are documented and reflected into the environment framework code. Layer0 concepts provide the most basic building blocks for modeling in the environment.

4.2.2 Part Division

Layer0 is divided into 6 parts. The contents of these parts are listed up to a reasonable level and Figure 4.3 shows the part dependencies. The parts that build upon the core concepts can be considered separate ontologies. As Layer0 is still under development,

only the more clearly defined parts are elaborated on in the following.

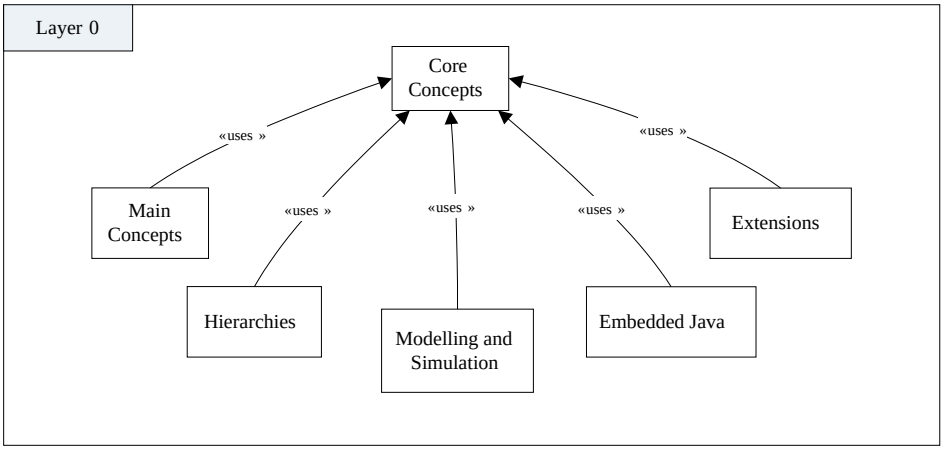

Figure 4.3: The division of Layer0 into parts and the dependencies between the parts.

Part 1: Core concepts

- Types and Instances
- Restrictions
- Relations
- Properties
- Enumerations
- Primitive Properties
- Built-in properties
- Acquires
- Linked Lists
- Views

Part 1 constitutes the core ontology that can be used to define typing systems (classifications) by three basic mechanisms: instantiation, restrictions and inheritance. Instantiation is used for classification of resources. Inheritance is used for inheriting restrictions from supertypes. Finally, restrictions are used to describe the allowed outgoing relations, along with their multiplicities and allowed targets. Restrictions are specified for types and they should apply for all instances of that type. Restrictions can be used to validate models: a model is considered valid if and only if restrictions apply for all instances in the model. These concepts are illustrated in Figure 4.4.

Also the concepts of property and relation are defined which are very fundamental types to the system and as such carry around implicit semantics about their use — purely by

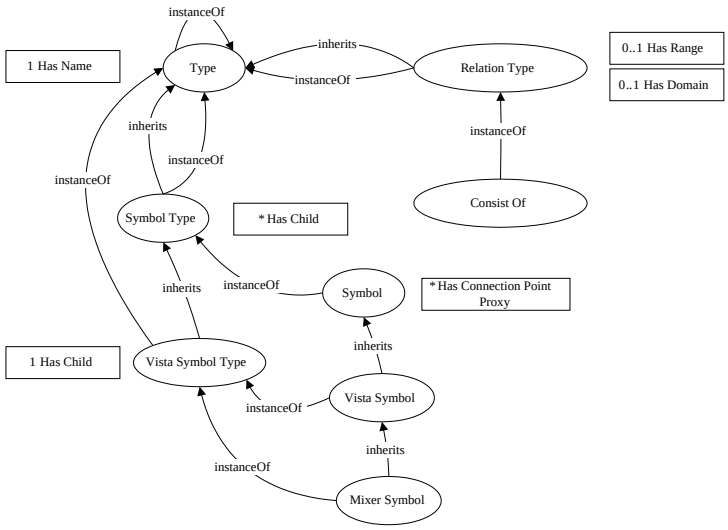

Figure 4.4: An example of defining types with Layer0. Every resource is an instance of something. Some of these instances are types, which is described by being an instance of something that is inherited from *Type*. Instance Of relations classify resources into classes of other instances, which can be types. Inherits relations are used to acquire restrictions from the inherited instances. Both multi-instantiation and multi-inheritance are allowed as long as restrictions do not conflict.

specification. An example of defining relation types is shown in Figure 4.5. *Primitive* and *built-in properties* allow the user to construct more complex composite properties out of primitive ones and *linked lists* allow ordering of resources. Enumerations are considered properties with a set of possible values. An example of defining an enumeration is shown in Figure 4.6. *Acquire* is a special mechanism for acquiring relations from other resources through specialised relations, which is basically just a mechanism on top of a primitive graph access interface. Finally *views* are used for defining how to traverse the graph model by triple rules. In the RDF world, Fresnel [BLP05] is an attempt to create a display vocabulary for defining *what to display* from the graph by *lenses* and *how to display* formats it with *formats*. Views also try to answer the question of "what to display".

Part 3: Hierarchies

- Libraries
- Ontologies
- URIs

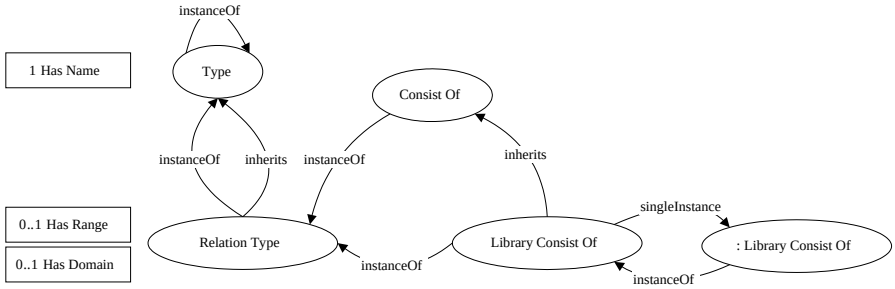

Figure 4.5: An example of defining relation types with Layer0.

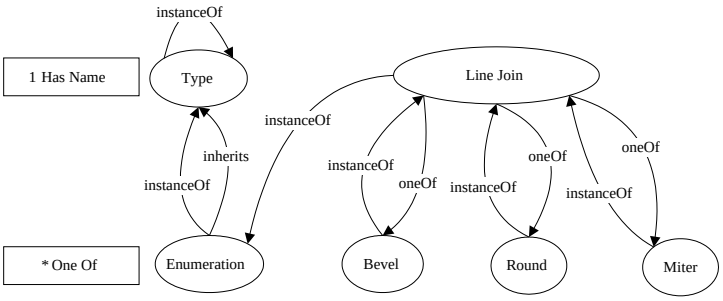

Figure 4.6: An example of defining enumerations with Layer0.

Part 3 defines types and relations to be used for creating different kinds of hierarchies and libraries out of the graph resources.

These hierarchies are often needed for creating hierarchical representations of the things contained, for example in an ontology.

Part 4: Modelling and Simulation

- Units and quantities
  - Expressions
- Experiments
- States

Quantities and their measurement units are vital in all kinds modelling and simulation. Modellers need to be able to define the quantities and units used in their models and also be able to discern the same information from other models. They also enable conversions between different units through expressions.

Experiments and states contain key concepts related to supporting the configuration and control of collaborative and distributed simulation with multiple clients under a server.

4.3 Client-Server Model

4.3.1 ProCore

The Simantics environment consists of clients and a hierarchy of ProCore servers. The clients can be, for example, instances of the ProConf user interface or *extensions* that operate autonomously or by command to produce a meaningful service, either internal or external to the environment. Together these elements form a client-server model as shown in Figure 4.7.

Excluding the topmost root server in the hierarchy, each server has a parent server from which it sources all graph data and allocates resource identifications for its own use. From the point of view of the parent server, the sourcing servers act as clients. The parent servers keep track of the data that the clients have requested, i.e. uphold a *working set* of each client. Each server creates its own version history, shrinking in granularity when walking away from the root server. Each server follows the versioning of its parent, i.e.

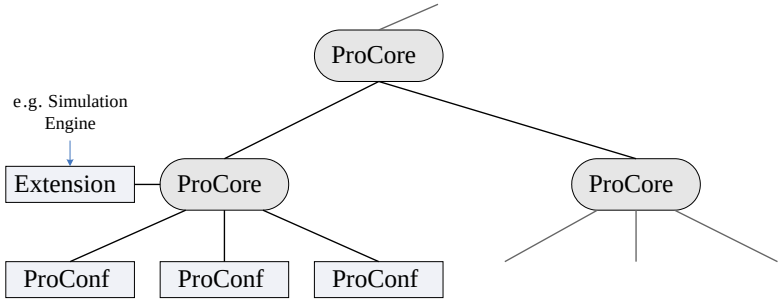

Figure 4.7: The Simantics environment consists of a hierarchy of ProCore servers and their clients, such as instances of ProConf or for example simulation engine extensions.

the servers track their parents for committed changes. Here, the server can use client working set information to infer which clients need to be informed of the changes and which don't. The servers are active in dissemination of up-to-date data to all their clients with an open session, emphasising the system's role in supporting active collaboration of multiple users in a distributed environment. General version control systems are almost always based on a *pull* communication model whereas this system tries to also apply a *push* model to its advantage. Joining and importing data into a server hierarchy after working outside of it is also possible via an export/import sequence.

Fully distributed peer-to-peer architecture has also been considered but is at the moment regarded as future research topic.

4.3.2 Transactions

ProCore is essentially a triple store. It is transaction-based and versioned. Each transaction effectively creates a set of changes to the graph model, called a *change set*. Therefore versioning in this context roughly means that the history of change sets is recorded as *revisions* and data from any given revision can be served at request. A change set consists of three things: added triples, removed triples and changed variable values.

A session must be opened to a ProCore server in order to communicate with it. Each client has its own session. All access to the graph data goes through read and write transactions. There is no support in ProCore for query languages such as SPARQL, but the graph is read inside transactions through browsing methods, as explained in subsection 2.1.6.

Looking at the ACID-properties (Atomicity, Consistency, Isolation and Durability), it can

be said that ProCore provides all but durability of transactions:

- Atomicity is provided through enforced use of read transactions and exclusive write transactions.
- Consistency instead is not an implicit property of ProCore but can be implemented by attaching a validator client to ProCore that monitors all write transactions and cancels them if, for example, restrictions defined in ontologies are violated.
- Isolation is supported by the transaction model also. Intermediate transaction state is only visible to clients that specifically request to be included in the progress of a write transactions, such as validators.
- Durability is not provided because the implementation is not journaled, i.e. transactions are not logged into persistent storage before they are actually performed for performance priority reasons.

Multiple read transaction are allowed to take place at the same time. Instead, write transactions need exclusive locking. Read transactions are necessary to guarantee that the model is not changed while a client is performing a batch of read operations on it.

The lifecycle of a read transaction differs from a write transaction in multiple ways. Read transactions are simply started and completed later, disregarding how the transaction was completed whereas write transactions must always be either committed or cancelled. Furthermore, write transactions need not be performed by a single client alone — they are collaborative events. A write transaction basically works as follows:

1. A client starts a write transaction, an exclusive lock is acquired.
2. The server asks the clients C_i of all open sessions whether they want to participate in the transaction.
3. If a client C_i chooses not to participate:
   (a) If the transaction is canceled, nothing happens.
   (b) If the transaction is committed, the client C_i will receive a change set containing the transaction changes to update itself.
4. If a client C_i chooses to participate it will get notifications about transaction changes and also be given the chance to make its own changes.

5. The transaction will run until all participants are willing to commit to the changes or some participant wants to cancel.

6. When the write transaction ends, the exclusive lock is released.

4.3.3 Undo

Generally undo facilities are implemented using certain recurring design patterns. These patterns include *command* and *memento*. The command pattern is employed for describing reversible and irreversible operations that can be used to construct and walk a history of undoable commands. Memento in turn is applied for serialising and deserialising the internal state of objects without breaking encapsulation [GHJV95], which can prove useful with objects referenced by commands. This is also the basic approach taken in Eclipse's own frameworks.

Despite of this, ProConf attempts to take a different approach. In addition to having a hierarchy of ProCore servers and ProConf clients, a similar server to ProCore is used for facilitate undo for the user interface. This server, called *UndoCore*, is used for performing graph undo/redo for an instance of the ProConf user interface by utilizing its graph model versioning features. This approach is based on the principle that all user interfaces in ProConf should be direct illustrations of the *current state* of the graph model. To implement this, in principle, one only needs to source data from the graph model in addition to listening and reacting to incoming changes. Hence, UndoCore was introduced.

There are important differences between versioning a graph data model and versioning a bunch of documents, i.e. files. The main difference is that files are considered inherently separate entities of data which can only be bound to each other by their content. Generally version control systems need not care about contentual dependencies between files. Thus the versioning model of plain files is fairly simple. Instead, in the graph model, resources are bound to other resources by relations directly in the data model. Versioning this model is a bit heavier since making small changes can have highly global effects due to the semantic implications of the changes. Large changes are naturally more prone to cause conflicts in collaborative work. This necessitates the ability to resolve conflicting changes made by different parties.

Contextual undo is an important feature when editing separate files in a single editor. It would be very irritating for the user to not have separate undo histories for the open files. One consequence of having a single UndoCore that every part of the UI is using simultaneously, is that contextual undo becomes considerably harder to implement. For

this reason, undo in ProConf is currently not contextual. To support contextual undo, conflict resolution needs to be resolved.

4.4 Plug-ins

Eclipse plug-ins created for the Simantics environment which access and manipulate the graph data model are always bound to a certain ontology or a set of ontologies. Among other data those plug-ins bundle code that operates on the graph model solely from the point of view of the ontologies the code understands. Any such code generally works properly only with a particular version (or versions) of the ontology that the code was built on. The key point here is that ontologies and related code need to evolve hand in hand. Plug-in extensibility is also very much tied to the level of the related ontologies: the more generic the ontologies are, the more likely it is for both the ontologies and the related code to be extended elsewhere.

4.5 Simulation

Simulation engines are generally pieces of software whose sole purpose is to take a model as input and produce some meaningful simulation results as its output. Exactly how this input/output process works depends on implementations.

As defined in subsection 2.3.3, an experiment is a test conducted on a process model. The concept of experiments is adopted in ProCore as a management mechanism for running controlled simulation sequences. Roughly speaking, an experiment consists of a model to simulate, the *state* of the simulation model, a number of simulation engines and a *sequence* according to which the simulation proceeds. The simulated model used to initialize the simulation engines is called configuration data, because it is the model created and configured by the modellers. The property values inside the configuration data get modified during simulation, which is done inside the state used in the experiment. The sequence determines which simulation engines participate in the simulation and in which order.

In the Simantics environment, simulation engines are to be connected to the environment as separate clients, so called extensions (not related to Eclipse Extensions and Extension-points). Simulation extensions are pieces of software dynamically managed inside an experiment that can access the graph data model. Extensions are ran as separate processes to make the environment more fault tolerant incase any extension exhibits unexpected behaviour.

Simulation engines generally need their own data structures for simulation. Therefore the graph model offered by ProCore is used only for initialisation. To attach an existing simulator to the platform requires creating an extension that mediates data between ProCore and the simulator.

4.6 Trending

Webmon is a web service based trending package for the Eclipse RCP developed internally at VTT. It can be used to create extensible 2D and 3D visualisations of data coming from various extensible sources. For example one can view a 2D trend can of a single real-valued time-dependent variable $f(t)$, where f could be for example a pressure or temperature value coming from a real process or a dynamic process simulator. This kind of trending is highly useful for monitoring a process or a dynamic simulation in a more intuitive way than just by looking at a set of numbers. Webmon could be utilized in ProConf for exactly these purposes, especially in process flowsheet diagrams.

Chapter 5

Design

5.1 Ontologies

The next sections describe the domain ontologies developed and used in this thesis, which are aimed for reusability. They focus on the basics of graphical and structural modelling. More case-specific application ontologies, defined for testing graphical modelling, ontology mapping and simulation are briefly covered later on in section 7.1.

5.1.1 Vector Graphics Ontology

Vector Graphics ontology is a simplified version of SVG modelled according to Layer0.

Figure 5.1 shows the inheritance hierarchy of the types defined in Vector Graphics ontology. A set of most useful geometric *shapes* have been lifted from the SVG 1.2 Tiny candidate recommendation. In principle the general *Path* shape could be used to define the basic shapes, i.e. *Rectangle*, *Ellipse*, *Polygon* and *Polyline*, but these are kept separate mainly for preserving the implicit semantics of the shape. The shapes defined by the ontology are further generalised as *Graphics Nodes* which are structurable as scene-graphs, essentially for defining the rendering order of the Graphics Nodes.

Vector Graphics ontology defines a fairly minimal amount of new relations — most of them are subordinate property relations. Normal relations are only needed for representing the scene-graphs. The *Has Previous Sibling* and *Has Next Sibling* are used to give a rendering order to sibling nodes whereas *Has Child* and *Has Parent* relations provide the hierarchy. Figure 5.2 shows an example of a Graphics Node hierarchy using the defined graphics nodes and relations.

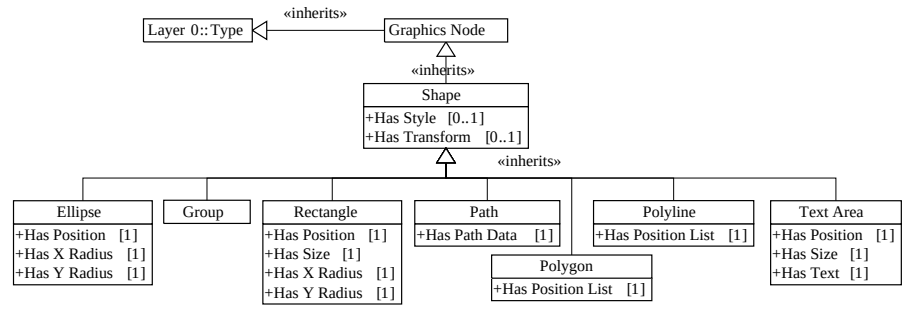

Figure 5.1: The type inheritance hierarchy of the Vector Graphics ontology. The UML generalisation arrows are used to symbolise Inherits relations. Attributes depict restrictions defined in the ontology. For example an ellipse is defined by a position and two radii for both coordinate axes.

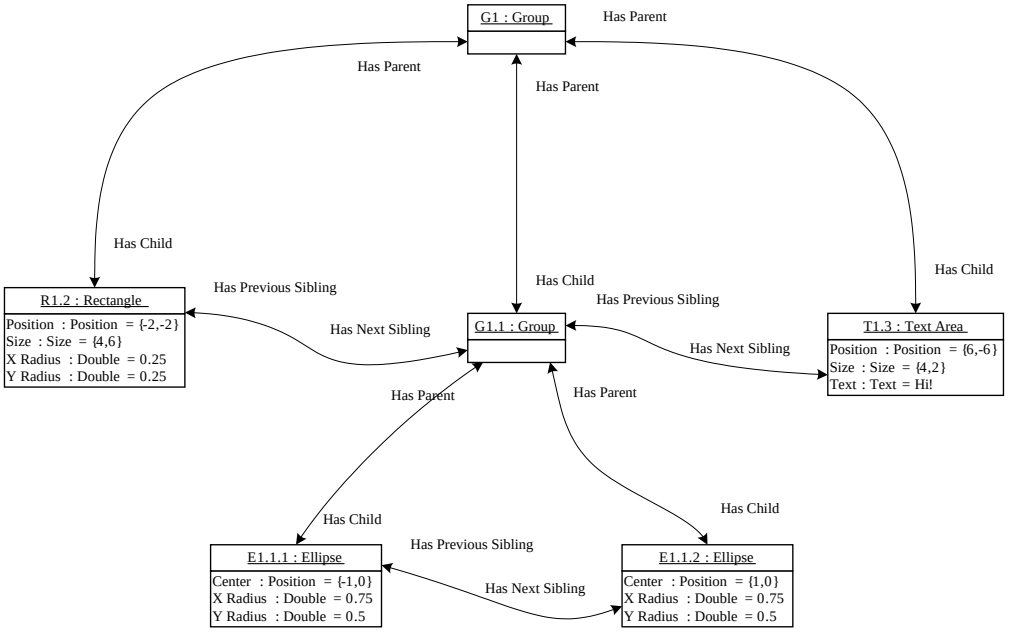

Figure 5.2: An example of a Graphics Node instance hierarchy defining a simple grouping of Ellipses, a Rectangle and a Text Area. UML attributes are used as a short notation for the corresponding property values in the graph.

A significantly larger amount of *properties* are defined for the purposes of representing shape properties, styling and composition. Each shape needs a set of properties of various types to define its geometry, which are shown in Figure 5.1. Styling of shapes works according to the description of subsection 2.2.1. A *Style* property consists of a *Stroke* property and a *Fill* property. Stroke defines a paint (i.e. color or gradient), opacity, width, line join, line cap and a dashing for drawing the the outline of a particular shape. Fill simply defines a paint, opacity and a fill rule for describing how to fill self-overlapping paths. Any Shape can have a style property and style is inherited by child Graphics Nodes. One more commonly used property is the *Transform* property. It is used to describe the position, scale and rotation of shapes and other graphical elements as shown in Figure 5.3.

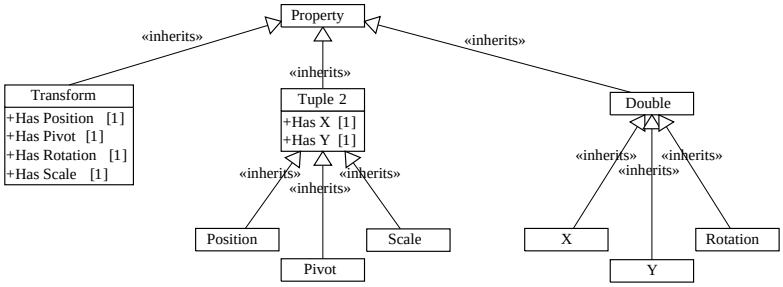

Figure 5.3: The type hierarchy related to the Transform property in the Vector Graphics ontology. All shown elements are types, inherited from Property. Attributes depict restrictions defined in the ontology. For example "*Position [1]*" states that a Transform property instance must have one Position property.

The purpose the vector graphics ontology has not been to cover the whole SVG specification. Instead, it has been developed with the principle of low hanging fruit in mind, i.e. only concepts that have been easy, useful or absolutely necessary have been included at this time.

5.1.2 Structural Modelling Ontology

The structural modelling ontology is a basis for defining models consisting of so called *structural objects*, their *terminals*, and *connections* between these terminals. The ontology supports hierarchical decomposition of models by *terminal mappings* between different hierarchy levels, i.e. having entities with substructure, as is shown in Figure 5.4. Most of the other ontologies developed during this work are in some ways based on the structural modelling ontology.

In summary, the goal of this ontology is twofold: firstly to provide an abstraction for

connecting entities and secondly to provide a mechanism for hierarchical decomposition of entities.

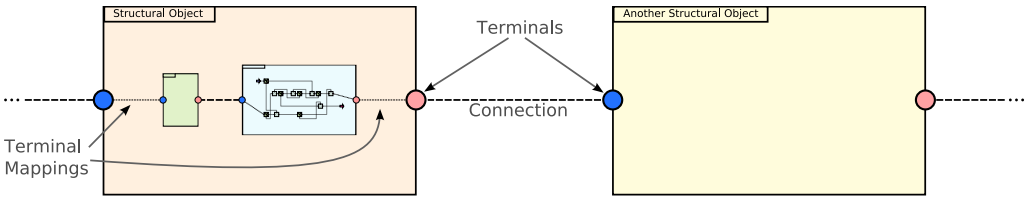

Figure 5.4: An abstract example illustrating the use of structural modelling ontology showing all four elements of the ontology: structural objects, terminals, terminal connections and terminal mappings.

5.1.3 Flowsheet Diagramming Ontology

Flowsheet diagrams are used to describe a process for the process engineer. Most importantly flowsheet diagrams define the topology of the flowsheet rather than containing accurate geometric representations. Generally flowsheets contain symbolic representations of flowsheet elements and connections, also known as flow lines, to define the topology, i.e. how the symbolic elements are attached to each other. With these descriptions, flowsheet diagrams can be seen as structural models with structural objects and terminals.

Here are the most central types of the ontology:

- A *diagram* is a structural object that consists of *symbols* and *connections*.
- A *symbol* is a structural object type that has a two-dimensional graphical representation. Symbols are instantiated as a part of diagrams. A symbol can have *connection points*. Graphical representations are created according to the Vector Graphics ontology.
- A *connection point* is a terminal with a graphical representation.
- A *connection* is a structural object that creates topology between symbols by connecting connection points together. A connection is a network of *connection branches* to enable branching of connection lines instead of always connecting two connection points. Connections also consist of connection points that need to be added when a connection is branched.
- A *connection branch* is the most basic part of a connection that connects two connection points.

Symbols and connection points need to be positioned on diagrams. The transform property from Vector Graphics ontology is used for this purpose. Connection points are positioned relative to the symbol. Figure 5.5 shows two examples of simple diagrams with the above entities highlighted. The reason for the possibility of more than one branches in a connection derives from PFD's and P&ID's. Pipes or flows are generally represented with lines, such as the connections discussed here. Sometimes these flowlines are split or merged together simply by connecting lines to each other which this ontology supports inside a single connection. It should be noted that for the user the same effect could be produced without multi-branch connections by instantiating new symbols and connections on-demand when the user attempts to branch a connection.

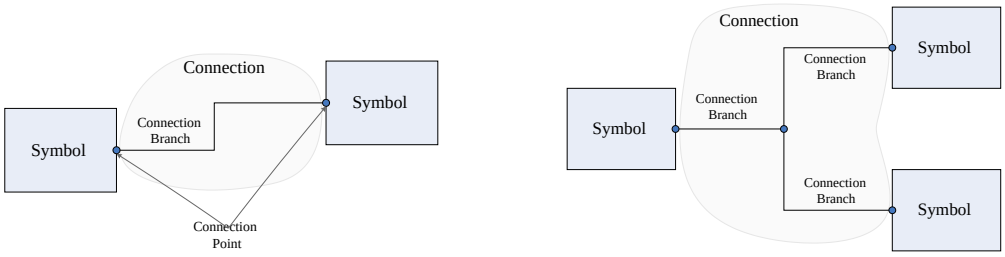

Figure 5.5: The basic model of a flowsheet diagram consisting of symbols and connections (left) along with an example of a connection with multiple branches (right).

The flowsheet diagramming ontology only defines abstract symbol types which cannot be used in diagrams. Any concrete symbols with actual graphics need to be defined in domain-specific ontologies.

5.1.4 Domain-specific Flowsheet Diagramming Ontologies

Domain-specific flowsheet diagramming ontologies are basically about customizing the diagram and symbol types by inheritance. Connection points also use symbols as their graphical representation, and therefore they too can be customized in this manner. Connections and connection branches can be customized by giving each their own style properties or by using a shared style. With these mechanisms, diagrams, symbols, editors, and tools can be customized or a given domain.

As described in section 4.4, domain-specific flowsheet diagramming ontologies are to be bundled with plug-ins that also contain the code for the customised editors and tools using the domain-specific ontologies. The ontologies and code go pretty much hand in hand, i.e. where one ontology extends another, the corresponding code also extends the

other.

5.1.5 Ontology Mappings

The basic idea of ontology mapping was outlined in subsection 2.1.2. In the context of ProConf ontology mappings are created manually by domain experts. They utilize a simple mapping framework which allows the creation of Java code rules. Rules are executed during write transactions when performed graph data changes match the conditions specified by a rule.

Creating a mapping between two ontologies requires the creation of another, so called *mapping ontology*, which need to define everything necessary to make the mapping work. In our current cases these include:

- Specialised *mapping relation* types, which are used by the mapping framework to relate mapped entities.
- Correspondences between structural objects and their terminals (see next section).

The main purpose of separate mapping ontologies is to promote the extensibility of ontologies as described in subsection 2.1.3. They should create a "bridge" on top of the mapped ontologies. In other words the bridging definitions need to be kept in the mapping ontology and out of the mapped ontologies (see Figure 5.6). This separation of models also bears resemblance to the approach of GMF (see subsection 2.2.2).

Structural Model Mapping Ontology

Recall that structural models consist of structural objects and their terminals. The purpose of structural model mapping ontology is to provide a method for defining how the terminals of two corresponding structural objects correspond to one another (Figure 5.7). This is done by using a *Mapping Context* to bind two structural objects. To this mapping context, multiple *Terminal Mapping Contexts* are attached which create a correspondence between two terminals in the given context. Obviously this is a very simple mechanism, only suitable for defining one-to-one correspondences between structural objects. Still, it succeeds well in doing exactly that while at the same time keeping the mapping data separate from the mapped ontologies. The reason for covering this ontology is in the case created for demonstrating the results of this work. Structural type mappings are needed for the related ontologies to perform the instance side ontology mapping.

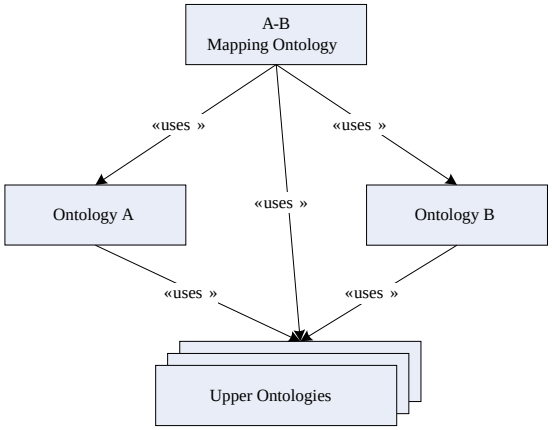

Figure 5.6: An abstract example of ontology dependencies when creating a mapping between two ontologies, A and B. The mapping ontology creates a bridge between A and B without imposing dependencies between them.

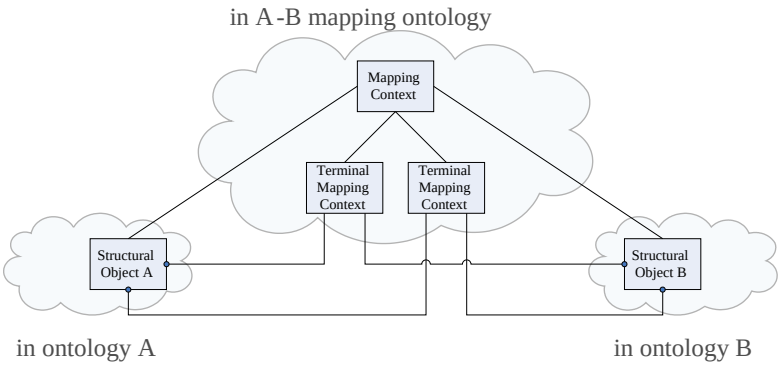

Figure 5.7: An example of mapping two structural objects. A mapping context bridges structural objects with the correspondences of their terminals are in turn bridged by related terminal mapping contexts.

5.2 Symbol Design

Designing graphical symbols involves constructing a hierarchy of graphics nodes for the symbol and adding styling properties to these nodes, according to the vector graphics ontology. This information can be used to create a static model for the renderer. SVG animations allow the creation of timed animations, which can be used to create preset animations which should be performed automatically by the SVG rendering engine. Although this would provide for a more dynamic rendering model, preset animations are not really useful when animation parameters are being created by a dynamic process, such as simulation or user interaction. Therefore externally modifying the rendering model seems to be a suitable approach for animation in these cases. SVG's internal animation facilities can still be utilized e.g. for viewing or exporting animations of completed simulation runs.

5.2.1 Parametrisation

Symbol parametrisation is a mechanism for parametrising the graphics of a symbol with a set of named parameters $p_i = (name, value)$. The purpose of the parameters is to make the symbol animatable by animating the parameter values. Ideally the parametrisations should be designed in such a way that the parameters correspond to some internal state or functionality that the symbols try to express. The parametrisations should also make it possible for the user to intuitively associate the parameter value changes with changes in the visual appearance of the symbol.

An example is a parameter $(size, 1)$ which could be used for scaling a symbol in a custom way. Another example is a symbol that needs to express its internal chemical state, such as a pH value. The pH value could be associated with a general colour parameter of the symbol, effectively hinting the user of what is going on. An example of a tank symbol with three parameters is shown in Figure 5.8.

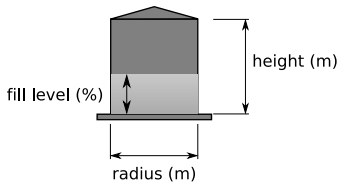

Figure 5.8: An example of a tank symbol with 3 parameters: height, radius and fill level.

Notice that in order to visualise anything inside a domain model, values from that model are needed. For example, a symbol does not automatically know that it is supposed to use a pH value from this model as an input for defining its colour parameter — it has to be told so by the visualisation developer who adds visualisation capabilities to the symbol.

One more thing to note about this approach is that it can be used to create highly customizable parametrisations if necessary. As noted before, the rendering model of a symbol can be initialized from a hierarchy of graphics nodes with styling in the graph. In this case the most simple way to parametrise is to map the parameter values to attributes of the SVG rendering model elements. Yet, with this mechanism, one could dynamically create a complex rendering model without ever creating a graphics node model in the graph. The problem with this approach is that one cannot create relations in the graph model to the graphics node hierarchy of the symbol since it does not exist.

5.3 Diagram Typing

Diagrams are typed in order to customize things related to them. Typing means the creation of a new diagram type which is inherited from the base type Diagram or its subtype. This creates a context for customization or extension. At least the following components can benefit from the typing:

Extension points Eclipse extension points can use a diagram type URI to be associated with it. Although there currently are no extension points which do so, several can be devised in the future.

Editors In the Eclipse environment editors often need to be customised for specialised inputs which in this case are different diagram types. Diagram editors are associated with a set of tools that can also be customised and extended for each diagram type.

Symbols Symbols or symbol libraries can be attached to diagram types in order to associate them to be usable with those particular diagram types.

Ontology mappings Ontology mappings are rules made of custom pieces of code that transform instances of one ontology into instances of another ontology. The mappings do their work based on types and therefore the context created by the diagram type is necessary to be able to perform the correct mappings for each diagram.

Chapter 6

Implementation

In justifying the implementation choices that have been made, it is beneficial to review the primary objectives of this thesis, originally introduced in section 1.2. The objectives have been rephrased to reflect the terminology used in the implementation environment.

- Define ontologies for graphics and diagrams.
- Create a framework for creating extensible graphical editors in the Simantics environment.
- Create concrete tools for editing flowsheet-type diagrams.
- Define proof-of-concept ontology mappings between a domain-specific diagram ontology and an information model ontology.

6.1 Ontology Design

Defining the initial graphics and diagramming ontologies served as a starting point for the whole development process. They were vital for initiating the development of concrete tools. Another good reason to start off with these ontologies is that they should be kept free of any binding to specific graphics frameworks.

The domain information ontologies were created later on as a test case for modelling multi-phase chemical systems in pulp bleaching processes.

Work on the ontology mappings could naturally proceed only after the graphics and domain information ontologies reached some point of stability.

6.2 Graphical Editing Framework

The design of the graphical editing framework design has been left slightly into the background although it is an important factor regarding the future of this work. In the beginning of the work, the main delaying issue was the choice of the graphics framework to be used. After that, little by little, the graphics framework has been constructed and extracted from existing code into what is described in the following sections.

6.2.1 Supporting Technologies

Widget Toolkits

As already explained in chapter 4 the user interfaces are based on Eclipse RCP. Eclipse natively uses a widget toolkit called SWT (Standard Widget Toolkit) for all UI purposes. Standard Java offers the AWT (Abstract Window Toolkit) and its lightweight companion Swing for the same purposes.

SWT differs from standard Java AWT in that it uses the native widgets in its implementation for each supported platform in an attempt to maximally preserve the native UI look and feel that users are used to. AWT and Swing in turn strive for a unified look and feel on every supported platform.

Another key difference between SWT and AWT is in rendering model versatility. The Java2D API of AWT supports porter-duff image compositing (subsection 2.2.1) while SWT does not. This heavily affects the choice of supporting graphics technologies.

The problem is that one cannot use AWT components in an SWT application completely transparently or vice versa. SWT provides a bridging mechanism for embedding AWT components inside an SWT widget. Performance-wise this seems to be the best solution for integration at the moment.

One annoyance resulting from the mixed use of both toolkits is more complex synchronisation. Both toolkits apply single-threaded rendering, naturally each in its own thread. This means that the programmer needs to be careful not to introduce race conditions in concurrent UI event handling.

Batik

The two most promising graphics toolkit contenders seemed to be Batik and GEF at the moment of choice. The decision boiled down to a trade-off between two features: rendering model versatility and amount of reusable framework available.

For rendering model versatility, Java2D (AWT) based Batik wins over the Draw2D (SWT) based GEF. Lately Java2D has also seen promising developments on the hardware acceleration front, giving more rendering performance and enabling rather nifty 2D and 3D graphics interoperability with OpenGL.

GEF was the strongest contender for a featureful graphical editing framework that could be put to use in the Simantics environment. GMF was not considered an option due to its close ties with the Ecore meta meta model. GEF has many desirable qualities such as general maturity, good support, seamless integration with Eclipse, and naturally, the included editing framework. The only negative thing was the fact that it is currently tied to SWT.

In the end, despite the SWT/AWT integration burden, Batik was chosen as the toolkit for this work. It's backend, Java2D, was a major contributor to this decision. This means that the a graphical editing framework needs to be created on top of the graph data model, Java2D and Batik.

Graph Data Access

Graph data access and manipulation facilities are provided by the Simantics environment. Ultimately all graph data access by kernel developers happens through two interfaces called *Graph* and *Resource*. A Graph is used for getting a hold of Resource instances which refer to a single resource in the graph. Resources are used for reading and manipulating the graph. Ontologies are used to generate so called *stubs*, which operate on top of the previous interfaces and provide an easier method for manipulating the graph as specified by ontologies. Stubs are generated based on restrictions defined for types in ontologies.

6.2.2 Dissection of the Framework

The framework can be divided into several seemingly separate pieces, in the end forming the basis of the editing framework.

The main goal here is to shed some light on the interfaces and vital implementations to

give the reader a hint of how to create graphical editors in the Simantics environment. The framework can be separated into the following conceptual parts:

- User interface components, i.e. canvases
- Customisation of painting
- Integrating SVG into the canvas:
  - for specifying a runtime graphics model, i.e. DOM
  - for rendering that model
  - for interacting with that model
- Interacting with the canvas (interactors, tools)
- Bringing all of this together as an editor of a single domain

Each component is briefly reviewed with accompanying UML diagrams in the following. Note that the UML diagrams do not contain all details about available operations, only the most relevant ones.

Canvases

One of the first things needed for graphical editing is a canvas to render on. Java2D was decided to become the 2D rendering API in this framework. Therefore AWT components need to be used in order to harness the performance advantage of direct rendering. Also the canvas needs to be embedded into ProConf which is inherently an SWT application.

There are two important coordinate spaces used in this kind of rendering. The first one is the so called *user coordinate space* wherein the positions and orientations of all renderable objects are defined. User space is inherently infinite. It is limited only by the precision and size of floating-point numbers. The other one is the *device coordinate space* or *canvas space*. Device refers to the output medium at hand, which may be for example the display device or a printer. In any case, during rendering the transformation from user space to device space must be performed. It boils down to transforming an area of user space into an area on the output device, such as a rectangular window on a display. As the device space is used for presenting a portion of user space, it is always limited. Naturally any concrete display device is also ultimately limited. See Figure 6.1 for an illustration of the coordinate spaces.

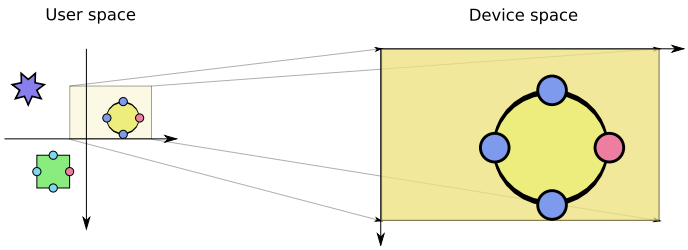

Figure 6.1: Coordinate spaces in 2D rendering. Colored area in device space represents the display device area.

Figure 6.2 shows the interfaces used for the created canvases and other closely related listener interfaces. The purpose of the *IVectorCanvas* interface is to offer only the most basic functionality, i.e. the possibility to manage painting of the canvas (*IPainterManager*) and to control the user space area shown by the canvas. *ISVGCanvas* extends IVectorCanvas to add support for viewing SVG.

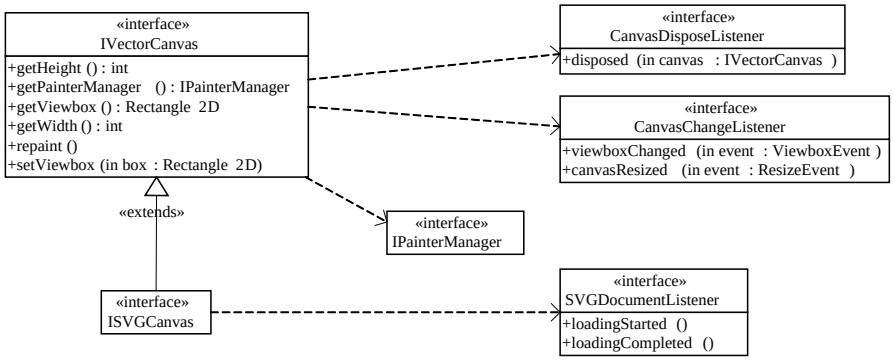

Figure 6.2: User interface canvas component interfaces.

Figure 6.3 shows the current corresponding implementations for these canvases. They are used in the construction of graphical editors inside an Eclipse RCP application.

Customised Painting

A rendering canvas is not very useful if one cannot paint on it. Therefore the previously introduced IVectorCanvas interface offers an *IPainterManager* (see Figure 6.4). This interface allows the user to manage a set of *IPainter* instances, which can perform arbitrary painting using Java2D's *Graphics2D* API. The painters can be added to the canvas either as *underlays* or *overlays*. The only difference between these is that underlays are applied

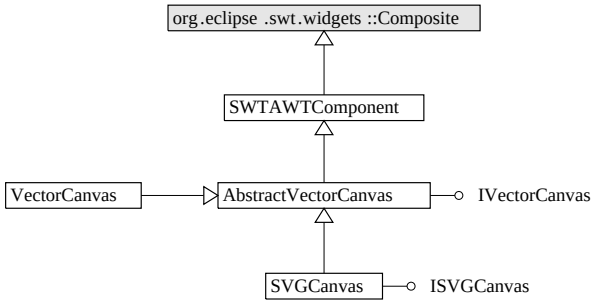

Figure 6.3: Canvas component implementations.

before overlays.

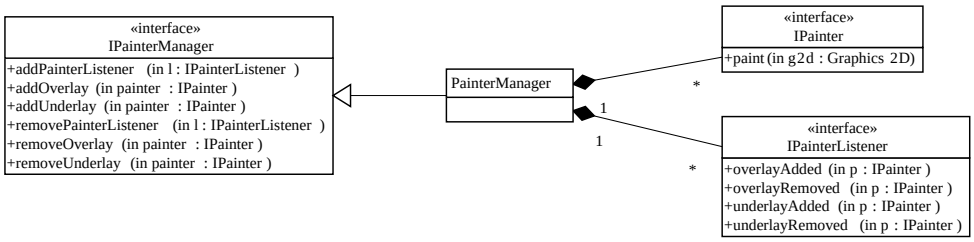

Figure 6.4: Canvas painting customisation interfaces.

Integrating SVG

Graphical editing frameworks are generally based on some kind of MVC architecture as stated in subsection 2.2.2. It is very common for the view and the controller to also have its own runtime data model. The essential idea behind integrating SVG and Batik into this framework is to exploit Batik's data models and interfaces (DOM, GVT) in the implementation of the data models for the controllers and views. Figure 6.5 illustrates the idea.

Batik is integrated into the *ISVGCanvas* and its implementation *SVGCanvas* as a painting layer between the *underlays* and *overlays* mentioned previously. Whereas the customisable *IPainter* layers require programming to be customised, the SVG layer provides a data-driven alternative for painting customisation through a well defined structure. Batik's own *JSVGCanvas* is built for showing a single *SVGDocument* at a time. SVGDocument is the SVG extension for the standard DOM *Document* interface.

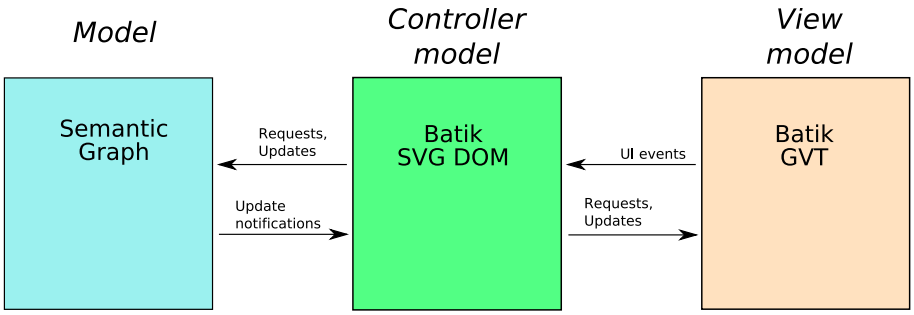

Figure 6.5: An MVC view of the integration of SVG and Batik into the Simantics graphical editing framework. The arrows represent program control flows between the components.

The following abstractions were created to support this integration:

SVG Context The interfaces *ISVGContext* and *ISVGContextProvider* play a vital role in the integration work. Essentially, ISVGContext contains a single SVGDocument. ISVGContextProvider is used for synchronised distribution of a single SVGContext to several parties. This is accomplished allowing interested parties to use context change listeners (*SVGContextProviderListener*) to commence necessary updates at context changes. One example of such a party is SVGCanvas. It is always initialised by attaching to a specified ISVGContextProvider. This effectively removes the need for separate management of the SVG context of the canvas by relying on the context change listeners to do their job. Figure 6.6 illustrates these interfaces.

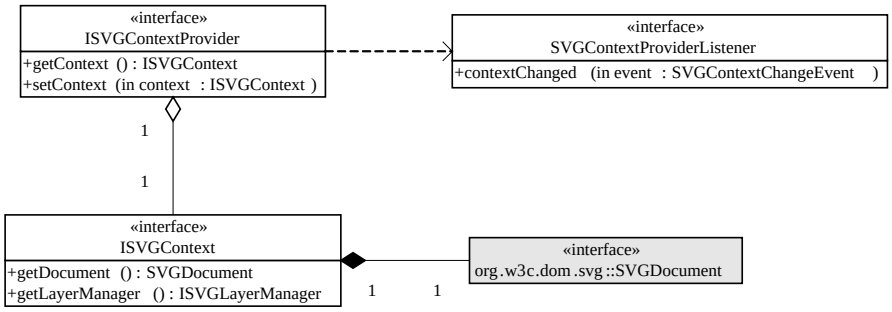

Figure 6.6: The SVG context abstraction illustrated.

SVG Layering While *IPainterManager* provided a way for layering programmatically customisable painters, SVG also inherently provides this option in a data-driven way. SVG layering can be done simply by using non-rendering group elements

(<g>) as root elements of each layer. Ordering of layers is then only a matter of ordering the group elements representing layers. The *ISVGLayerManager* interface offered by ISVGContext is simply an abstraction for obtaining and controlling the layer elements through a *ISVGLayer* interface. Another purpose of the manager is to control the *active SVG layer*, which marks the layer used for editing at any given moment. *ISVGLayerListeners* can also be attached to the ISVGLayerManager to listen to changes in the layer structure. Figure 6.7 illustrates these interfaces.

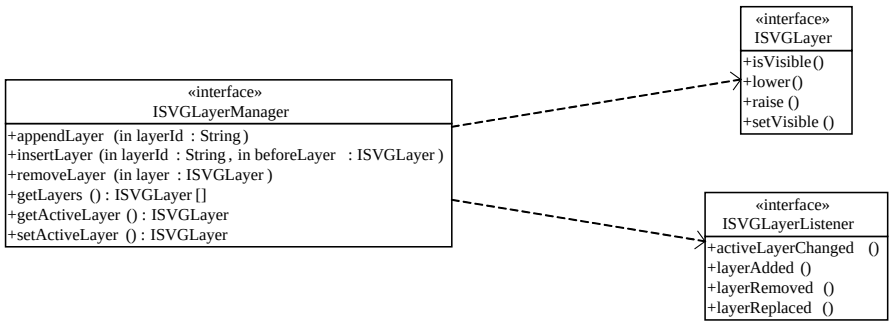

Figure 6.7: The SVG layering abstraction illustrated.

Canvas Interaction

All user interaction produces UI events that need to be interpreted as the intent of the user. For this framework canvas interaction has been separated into two categories: modal interactors and tools. Both are considered interactors in the sense that their purpose is to react to user input.

A modal interaction is one that is started by some user input combination after which the modal interactor will consume all user input until the end of the interaction is triggered. They are well suited for simple interaction tasks, such as panning or zooming the current view of the canvas.

Tools on the other hand are more closely related to *edit domains* and editors. An edit domain serves the purpose of managing and distributing the state of an editor instance. This includes the current *input* of the editor, i.e. the resource that is being edited, and the currently active tool. Only one tool can be active at any moment. The main difference between tools and modal interactors is that modal interactors have input precedence over tools once they get triggered. One final piece in the tool puzzle is the ToolDelegateLis-

tener which listens to the actual UI events and delegates them to the correct receiver at all times. See Figure 6.8 for an illustration of the described components.

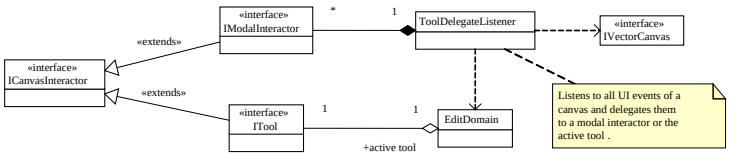

Figure 6.8: Interfaces and classes related to canvas interaction.

Editors

At this point constructing a graphical editor remains a matter of putting the introduced pieces together. Take a canvas of your choice and possibly add decorators such as a ruler. If using SVG, implement an SVG context provider for the editor. Create a new EditDomain for the current editor input. Attach a ToolDelegateListener to the editor and the EditDomain and the canvas. Finally register any necessary modal interactors and tools.

This dissection has focused mainly on introducing interfaces. For many of these interfaces, abstract implementations and different helpers have been created, which are intended to be extended and used instead of reimplementing the interfaces fully.

6.3 Flowsheet Editors

Two editors needed to be created for flowsheeting. The first one is a *symbol editor* which is used for define *symbols* according to the ontologies defined in subsection 5.1.1 and subsection 5.1.3. The second one is a *diagram editor* that uses defined *symbols* to construct *diagrams* according to the ontology defined in subsection 5.1.3. Both editors have been built on top of Eclipse APIs and the graphical editing framework components previously described. Both editors also provide a simple zoom-level adaptive grid which can be used for grid snapping in editing activities.

As described in section 4.1 using the Eclipse Workbench UI forces the use of certain UI concepts. The flowsheet tools have been implemented as editor extensions along with multiple view extensions to support them. Separate perspective extensions have been created for both symbol and diagram modelling tools in an attempt to support more

natural user workflow.

6.3.1 Symbol Editor

The purpose of the symbol editor is to support use cases of library developers, particularly the symbol library and visualisation developers. Basically symbol editing is about constructing and manipulating a hierarchy of graphical primitives to become an illustration of something that needs to be represented in 2D, such as process components. The editor is uses for editing a single symbol at a time. The tasks of creating and managing libraries of symbols are supported by ProConf's ontology development environment. Figure 6.9 shows an example of the symbol editing environment.

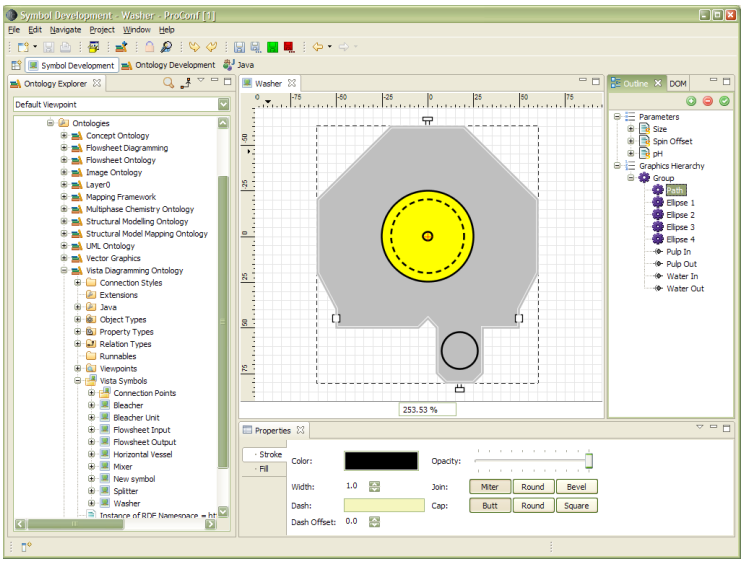

Figure 6.9: Washer process component symbol editing with the symbol editor. On the left is the Ontology Explorer tree view used for library creation and management. In the middle is the symbol editor along with an outline view of the editor contents on the right. At the bottom is the property view for editing different properties of the current selection.

Defining Graphics

Ideally defining graphics for symbols should closely resemble the use of a quality vector graphics editor. The user should be able to define and composite graphical primitives graphically, without worrying about the details of how these primitives are actually defined. Realistically speaking, creating such an editor is no small task.

The current implementation is still in its infancy but can nevertheless be used for constructing hierarchies of graphics nodes. Graphical primitives can be transformed graphically and their style properties can be edited via property views. The lack of graphical primitive specific editing tools is probably the biggest weakness at the moment. Because of this the symbol developer often has to resort to using the ontology explorer tree to edit the properties of the graphical primitives to construct the graphics. The styling properties of graphical primitives can be edited with special property views as seen in Figure 6.9.

Composition of Graphics

In this work image space composition of graphical primitives has been considered useful for creating good looking and intuitive visualisations for symbols. Most importantly, it allows clipping the rendering of any graphical element to any other graphical element, regardless the geometric complexity of the graphical elements. The performance of composition should only depend on the amount of image space covered by the rendered output. Sadly, although Batik was tested to support the necessary mechanisms, there was no time to implement these features.

Symbol Parametrization

Symbol parameterisation is implemented by associating Java code to a symbol that performs the mapping from its named parameters to the graphics nodes of the rendering model. The implementation takes advantage of the JDT (Java Development Tools) bundled with the Eclipse SDK to facilitate user friendly Java code editing.

In this implementionm symbol parametrisation has been seen as a mechanism for supporting the definition of visualisation capabilities for symbols. The idea is simple and seemingly powerful but there are difficulties involved. The main challenges lie in supporting simultaneous programmability and graphical editing of the symbols. If the visualisation developer is allowed completely unrestricted programmability in parametrisation, it can become very difficult to support graphical editing. These challenges are highly analogous to the ones encountered in existing UI design software where a designer uses graphical tools to define UI layout and programming tools to define UI functionality simultaneously. The problems arise from the fact that both tools edit the same model, i.e. the code that constructs the UI. Typically, when the UI code gets more complex or no longer conforms to the format automatically constructed by the graphical tools, the graphical tools stop working properly.

6.3.2 Diagram Editor

The implemented diagram editor supports the general editing traits of diagram models constructed according to the flowsheet diagramming ontology introduced in subsection 5.1.3.

The diagram editor allows adding symbols on a diagram by instantiating them. Symbols can also be removed in which case any connections attached to the connection points of the removed symbol are detached. Symbols can be graphically positioned and oriented on the diagram. Connections can be added by drawing a continuous line segment between two connection points, i.e. terminals. Only single-branch connections are supported since multi-branches connections were not a top priority feature and would have only introduced unnecessary complexity. Connections can also be removed which detaches possible terminal connections at both ends. Connections can currently only be manipulated by editing the segmentation and the corner points of a connection line. Detaching and reattaching existing connections is not supported although it would be useful. At the moment changing connecting connectivity implies removing the original connection and adding a new one. Figure 6.10 shows an example diagram constructed with the these tools.

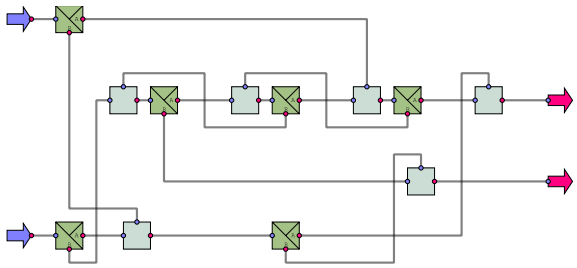

Figure 6.10: The internals of a washer created with the diagram editor.

Configuring a Domain Model Graphically

As gathered in subsection 5.1.5, a central idea in this environment is that the logic of configuring domain models graphically should not be coupled to the user interface code. Instead, the logic should be implemented in ontology mappings, effectively bridging the graphics and domain models. In this implementation and the chemical multiphase modelling test case this objective was achieved. The graphical modelling tools are being used unmodified, with the exception of adding a button for enabling the mapping. After

that the topology of the domain model is configured along with the graphical diagram model. Figure 6.11 shows an example of the model configuration user interface built for the proof-of-concept mapping case.

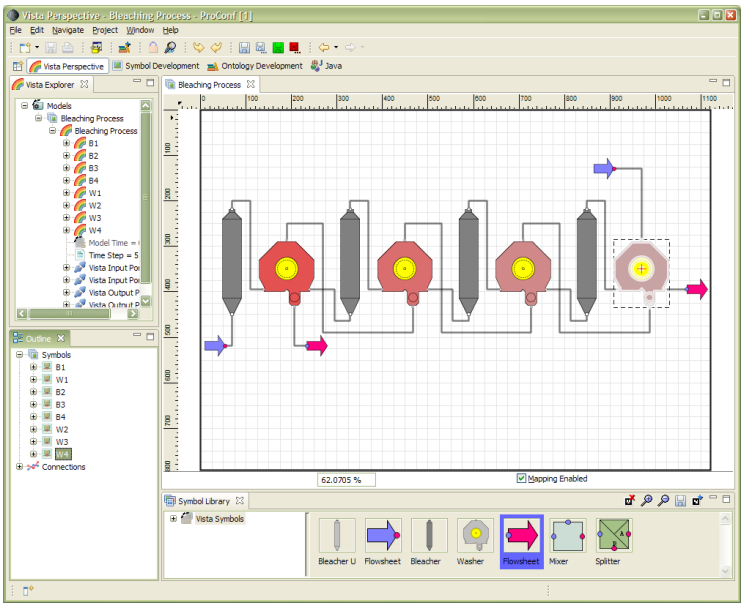

Figure 6.11: Editing a bleaching process diagram with an application specific diagram editor. Two tree views on the left are showing structure of both the domain model (top) and the diagram model (bottom). The symbol library view at the bottom is used by the modeller to add new symbols to the diagram.

Configuring Monitors and Trends

Monitors and trends are important features for supporting graphical visualisation of simulation results. Sadly support for these features was not finished in time to be included in this thesis work.

Textual monitors could have been easily implemented using symbol parametrisation and text flow functionality provided by Batik. Unfortunately these text facilities were discovered too late in the development process.

Chapter 7

Results and Evaluation

Looking back to the beginning of this work, the main set objectives can still be considered realistically implementable in the given timeframe. In the end, most goals were achieved up to a certain level and some were left as future research. In the following sections, the different aspects of this work are evaluated and criticised by subjective assessment of the author and with regard to requirements and viability for real-world applications. In section 7.1 the results are evaluated from the point of view of the implemented modelling case. In section 7.2 the created domain ontologies are analysed from a more general point of view. Finally section 7.3 briefly analyses the ontology-based modelling approach as a whole.

7.1 CASE: Flowsheet modelling of a multi-phase chemical process

The implemented tools and mapping mechanisms were put to the test in a proof-of-concept use case involving modelling and simulation of a pulp bleaching process using structural modelling methods, including hierarchical decomposition. Pulp bleaching is a part of a fiber line in pulp mills. Its purpose is to improve the brightness and cleanliness of the pulp by removing residual lignin from it. The case involved integrating an in-house steady-state simulator which includes balancing both mass flow and multi-phase chemical state, developed as part of a project called Vista [BLH+]. This simulation engine was integrated into the Simantics environment as an extension client.

The test case required the definition of multiple ontologies for specifying the symbols used for diagramming, the simulator domain model and the mapping between these

two. These ontologies are considered application ontologies and are thus not defined with reusability in mind.

Vista Diagramming ontology is extended from the Flowsheet Diagramming ontology to contain the diagramming-related customisations required for the case. It includes the Vista Diagram type for allowing UI customisation based on the diagram type and all the needed symbols, such as Mixer, Splitter, Washer and Bleacher. Some of these symbols can be seen in Figures 6.9, 6.10 and 6.11.

Flowsheet ontology contains base concepts for modelling flowsheets consisting of flowsheet objects, such as units, their terminals, and streams which are used to connect the units by their terminals. These concepts are extended from structural modelling ontology.

Multi-phase Chemistry ontology defines the basic properties related to describing multi-phase chemical state. A *Multi-phase System* consists of several *phases* which in turn consist of several *species*, i.e. the components of matter in a particular phase.

Vista ontology brings the flowsheet and multi-phase chemistry ontologies together to define the simulation model concepts corresponding to the created symbols, i.e. Mixer, Splitter, Washer, and so on. These are the concepts based on which the simulation engine builds its internal calculation structures. The ontology also defines additional phases and species for a case-specific *Vista Multi-phase System* property type.

Vista Mapping ontology As specified in subsection 5.1.5, this ontology bridges the Vista Diagramming ontology and the Vista ontology to enable graphical modelling of Vista simulation models.

For modelling and the involved ontology mapping, this approach can be said to work as planned. The only mandatory customisation for the diagram editor was a mechanism (i.e. a button) for enabling the ontology mapping for an edited diagram. The customised code for performing the mapping was not implemented rigorously and completely enough to be able to keep the simulation model in sync with the graphics at all times. Another thing to note is that these mappings are not bidirectional as there was no real need for that with these ontologies. Anyhow, these deficiencies are mostly due to programmer sloppiness, not the mapping mechanism itself.

Another addition to the diagram editor was the ability to show the properties of the simulation model from the diagram. All the Vista simulation model properties are contained in the streams that connect the flowsheet objects. Since diagram connections were

mapped to these streams, the simulation model properties can easily be shown in an Ontology Explorer tree by selecting a particular connection on a diagram.

The biggest problems with this particular mapping implementation are concerned with *model reuse* and *scalability*. Model reuse comes into play along with hierarchical modelling. In this test case, the bleaching process was modelled on two levels: the top level, i.e. the bleaching process, and the lower level, i.e. the internals of a Washer and a Bleacher. For example, a bottom-up modelling process should work as follows:

1. The model configurator creates a new diagram, Washer, and enables mapping. He uses the Splitter, Mixer and Input/Output symbols to construct the internals of a Washer as to form what is shown in Figure 6.10. The Washer diagram therefore contains both, the diagram model and the mapped simulation model.

2. The model configurator makes this diagram a possible subdiagram of the Washer symbol which is shown in Figure 6.9. This allows the modeller to use the single Washer symbol as an abstraction for the whole underlying washer model. Another way to think of this situation is to consider the Washer internals diagram as a type and instances of the Washer symbol as an instance of that type — as if one possible template was given for the Washer.

3. The model configurator creates a new diagram and enables mapping for it. He adds a Washer symbol onto that diagram. This is where the hierachical modelling comes into play and there are several alternatives for what can happen. The most ideal scenario would seem to be having copy-on-write semantics for this shared diagram. This means that at first all Washer symbol instances should share the washer's internal structure, i.e. the diagram and the mapped simulation model. If that shared diagram is modified, all symbol instances sharing it are affected. At some point a configurator may want to customise the internal structure of a certain Washer symbol instance. This implies that the system cannot modify the shared version of the Washer internals diagram, but a copy has to be made to which the modifications should be made.

However, reality is more complicated than the above. Although the underlying Washer diagram model could in principle be shared among multiple Washer symbol instances, the mapped simulation model cannot be shared. This is because from point of view of the simulation engine, the property values of the internal simulation model of each Washer need to be separate although the simulation model topology can remain the same. Sadly,

achieving this with the current modelling conventions is rather difficult. So far no real solution for this has been devised. The only thing that could be done within the timeframe of this thesis was to resort to cloning everything, immediately when a new symbol with substructure is instantiated. An obvious drawback is that massive amounts of triples get generated very quickly. Another one is that editing of shared substructure diagrams is not possible.

Although support for generating the simulation model through mappings was achieved and the simulation engine was integrated, integrating the simulation results back to the diagram model for visualisation purposes was left as future work. Therefore the visualisation developer's use case of defining "visualisation capabilities" by mapping simulation model property values back to symbol parameters is not supported.

7.1.1 Scalability

Scalability problems in mapping-enabled diagram modelling arise from the very large number of triples generated already with very small models. There are multiple underlying reasons for this bloating behaviour. The first one is of course the cloning mechanism described above, which creates massive amounts of data by doing theoretically unnecessary cloning purely for making the mapping implementation easier. The other reasons for bloat are ontology-related. Currently the definitions of certain property types in certain ontologies generate most of the triples both in the diagram and the simulation model. Recall that the Vista ontology contained the "Vista Multi-phase System" property type. It describes a case-specific chemical state as a composite property with three phases and several species in each phase, which makes instances of this property type very large. Its instances contribute to approximately $90 - 95\%$ of all data in the simulation model. The other bloating culprit is the Transform property defined by the Vector Graphics ontology. A more detailed and precise analysis of it is presented in section 7.2. Table 7.1 shows the amount of triples generated by modelling the internal Washer structure diagram and also the amount of triples generated by the bleaching process diagram and simulation model. The simulation model seems to be taking almost 80% of the whole triple mass. Combining this observation with the $90 - 95\%$ property percentage above shows that over 70% of the whole data mass is in the simulation model property structures. Emphasising that the case model is still really small, this is a clear indication that optimisation work remains to be done.

Since the simulation result visualisation use cases were not implemented, it's hard to say anything conclusive about the real-time visualisation performance of the diagramming

Table 7.1: The amount of triples and literals created during mapping-enabled diagramming.

Model	# Triples	# Literals
Washer simulation model	13525 (79%)	1923 (78%)
Washer diagram model	3501 (21%)	537 (22%)
Washer total	17026	2460
Whole Bleaching Process model	83837	12198

framework or the viability of the symbol parametrisation mechanism.

7.1.2 Usability

The importance of usability and good looks of the graphical editing tools cannot be underestimated. As already stated in subsection 6.3.1 the symbol editor is far from real-world usability, mainly because the integration of parametrisation into symbol modelling was more difficult than was originally anticipated. The diagram editor in turn is most importantly missing different kinds of snapping mechanisms, such as ruler guidelines or geometry-based snapping. The connection manipulation mechanisms could really use different kinds of auto-routing facilities to streamline the editing process.

7.1.3 Case Conclusions

Based on the previous evidence, this implementation as such is not really viable for real-world use. The positive thing is that the ontology mapping does its job pretty much as planned while remaining transparent to the user although UI usability can still be greatly improved. There are multiple areas of improvement with regard to do with making the produced models more compact. Firstly, the related ontologies need to lose some weight. Secondly, the mechanisms and methods for hierarchical modelling must be developed further to enable better reuse of models. Finally, mechanisms and methods for allowing sharing of simulation model topology are key to this desired weight loss.

7.2 Ontologies

The ontologies used in this thesis have been mostly defined with the principle of low-hanging fruit in mind. This means attention has been paid to defining only concepts that have been mandatory for satisfying requirements or easy to include.

Referring to principles defined in subsection 2.1.3, it can be said that all ontologies created in the Simantics environment contain a certain amount of encoding bias. This bias originates from the way the environment handles identification of resources and also from the way Layer0 ontology is defined. At this point it is unclear whether our ontologies are representable in OWL for example, without any modification of semantics.

Naturally bad naming decreases ontology clarity, but currently the largest clarity issues in the created ontologies arise from the lack of attached documentation. Currently there is not enough natural language documentation to allow an outsider to easily comprehend the ontologies. One important deficiency in the current ProConf version is that even if this documentation is available, it is not utilised at all. Granted, adding documentation to ontology entities as Layer0 *Comment* properties does not currently require too much effort. However this documentation could be made accessible in more immediate ways than by opening an editor, such as tooltips and descriptive text boxes in selection dialogue boxes. Although this criticism mainly concerns ODE, it applies for other UI's also. A major point in having a versatile expressive information model is being able to associate and access useful information where it is needed.

The extendibility of ontologies is worth considering for at least the Vector Graphics (VG) and Flowsheet Diagramming ontologies (FD). Since VG is a fairly straight-forward ontology which merely defines different 2D shapes and depends on no other ontologies, there are very few hindrances to its extendibility. Instead the FD ontology contains some clear flaws due to the legacy reasons. Originally these two ontologies were one and the same, until they were sliced in two along with the realisation that vector graphics definition and diagram definition can be considered separate problems. The main flaw is that currently the FD ontology is bound to using nothing else but the VG ontology. This is because VG ontology concepts are specifically defined to be used for defining symbol level matters, more precisely connection points of symbols. There are alternatives which do not include this particular coupling, thus allowing FD ontology to use alternative forms of graphical definition for symbols and connection points besides VG. Extendibility is only hindered incase a person extending FD cannot use VG, so the issue is not really that big. Besides extendibility, the issue is also related to minimising ontological commitment.

Generally speaking, minimising ontological commitment is more important when dealing with domain ontologies or general ontologies. Instead, application ontologies are by nature more customised for a particular purpose and are likely to contain more strict commitments. The domain ontologies presented in chapter 5 fare quite well in this sense.

7.2.1 Scalability

It is natural for the triple-based graph data model to produce more data than normal rigidly structured data models. Complete data structures consist of large amounts of triples that need to be interpreted instead of having a single block of memory carry the information of that data structure. This is why care needs to be taken not to make ontologies more complex than necessary.

The scalability of diagramming and vector graphics ontologies can generally be considered fairly good, except for one poor devil: the *Transform* property of the Vector Graphics ontology. It is not a very complex property per se, it just combines the basic 2D transformations, i.e. translation, rotation with a pivot point and scaling into a composite property. It does so by defining each of these properties separately to preserve the semantics and values of each transformation type instead of just combining them all using a 3×2 affine transformation matrix. This preservation also allows changing the preferred order of the primitive transformations, if necessary.

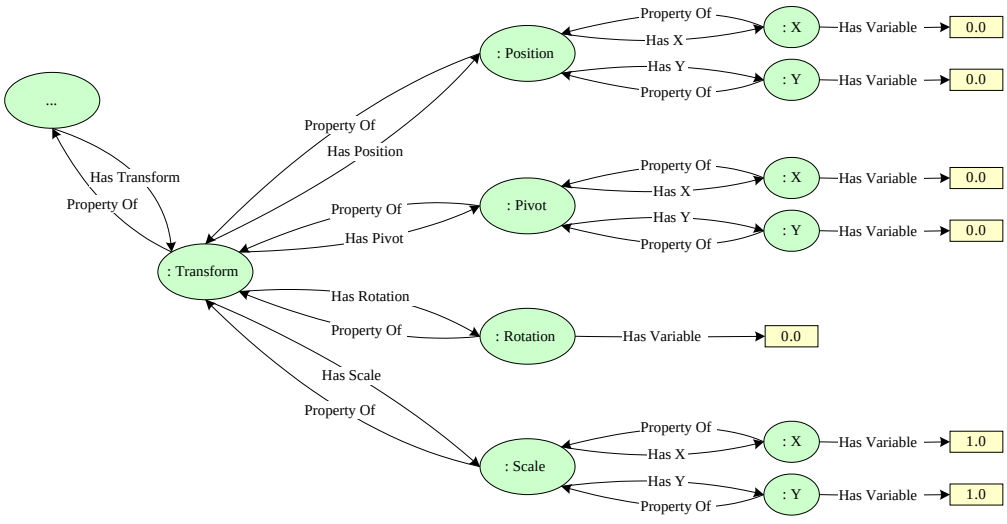

Figure 7.1: The amount of triples and literals produced by a single Transform property instance illustrated.

Table 7.2: The transform properties' percentage share of the triples and literals generated by instantiating symbols containing different amounts of connection points.

$N_{cp}(s)$	$N_p(s)$	$T(s)$	$L(s)$	$T_{tr}(s)$	$L_{tr}(s)$	$\mathbf{p}(\frac{\mathbf{T_{tr}(s)}}{\mathbf{T(s)}})$	$\mathbf{p}(\frac{\mathbf{L_{tr}(s)}}{\mathbf{L(s)}})$
1	1	95	16	80	14	84%	88%
2	1	139	23	120	21	86%	91%
3	1	183	30	160	28	87%	93%
4	1	227	37	200	35	88%	95%

Figure 7.1 shows what a Transform property looks like as triples and literals. The image does not show the *Instance Of* relations attached to each of the resources labeled with a leading ':'. By counting these in we find that a single transform property takes a total of 38 triples and 7 literals (variables) to express plus two triples to attach the transform property to another resource. Comparing this with a normal programming language representation which would require a mere $7 * 8 = 56$ bytes to represent in double-precision floating-point numbers, the amount of data in the graph is greater by over an order of magnitude.

By extending this analysis into the diagramming ontology, where the Transform property is used extensively for positioning symbols and connection points, we discover that:

$$\begin{aligned} T(s) &= 47 + 44N_{cp}(s) + 4N_p(s), \\ L(s) &= 1 + 7(1 + N_{cp}(s)) + N_p(s), \\ T_{tr}(s) &= 40(1 + N_{cp}(s)), \\ L_{tr}(s) &= 7(1 + N_{cp}(s)), \end{aligned}$$

where $T(s)$ is the amount of triples and $L(s)$ the amount of literals it takes to represent symbol s. $T_{tr}(s)$ and $L_{tr}(s)$ represent the amount of triples and literals needed by the transform properties in symbol s. $N_{cp}(s)$ and $N_p(s)$ are the amounts of connection points and parameters in the symbol respectively. Table 7.2 shows that the transform properties own a major part of the generated data — over 80% and only grows by increasing by the amount of connection points.

There are multiple ways to reduce this bloat. The first, and also the easiest alternative would be to change the Vector Graphics ontology to allow the different transform parts to exist only if necessary (multiplicity [0..1]) instead of requiring that a position, scale, rotation and pivot always exist (multiplicity 1). Another way would be to take out the

semantics from the transform property and just model it as a double vector of size $3 \cdot 2 = 6$ to store an affine transform matrix. Yet another way is to again model the property as a vector of size 7 and store the translation, rotation, pivot, and scale in some agreed and documented order. One deficiency with the former two approaches is that they make it impossible to create more references to the translation, scale, and other properties from outside of the property hierarchy shown in Figure 7.1. Then again, by using a simple value vector, the amount of triples and literals would be greatly reduced.

7.3 Ontology-based Modelling and Mapping

The taken ontology-based modelling approach, i.e. the graph data model along with the modelling semantics of Layer0 allows us to describe data in unprecedented ways. Layer0 defines basic law and order into the otherwise structureless graph by specifying the most basic modelling conventions. Being able to relate data (resources) to other data in unlimited ways is really a great source of expressive power despite of the simplicity of the underlying mechanism (triples).

The basic idea of this ontology-based approach is much the same as in Eclipse's EMF-based modelling toolkits: a model-driven approach where everything, starting from your own ontologies (meta models), should be done in a modelling-supported fashion instead of pure programming. The model-driven approach allows support for model validation, supporting ontology-based code generation for model processing, and automated generation of user interfaces based on the semantics of ontologies. Most importantly the employed graph data model allows us to do all modelling of different disciplines in a single environment. The key distinctive features of the Simantics environment are its support for versioning of the graph model and its focus on real-time data support, such as simulation results or real process data.

Being ontology-based is not something that is or should be somehow immediately apparent to the user. The only difference is that if the model is put to proper use, one can have versatile linkedness between different data models. The user-visible differences or even enhancements only become apparent when the user interfaces also truly embrace this data model.

As already stated, we were able to reach the original goal of getting a proof-of-concept mapping implementation to work. In the future, the main goal will be figuring out ways to make ontology mapping both easier and more robust by adding syntactic sugar and helping user interfaces. Syntactic sugar could be added by raising the level of expres-

sion out of Java code into something more high-level, possibly a Prolog-style declarative form or even into a specially crafted UI. The current Java code mechanism seems too complicated for just any kernel developer to be able to comprehend and hold together.

Another area of examination in the future could be spatio-temporal diagramming, which basically means adding a time-interval of existence into every aspect of diagram modelling. This might prove highly useful in any modelling cases where lifetime issues need to be considered, such as construction planning or district heating network simulation over long time periods.

Chapter 8

Conclusions

Currently, in many cases process modelling and simulation still tend to be rather separate modelling disciplines. Semantic integration research is looking for mechanisms to map models of different domains in both manual and (semi-)automatic fashion but so far no silver bullet has been discovered. An extreme example of bad data integration is that often simulation models are manually recreated for simulation software based on printouts of original designs.

Real-world process design most often involves drawing 2D diagrams, such as P&IDs to illustrate the process. Computer-assisted diagramming is nothing new under the sun — it has been done for ages already. Also several intelligent design systems are available nowadays, which combine different assets into a unified design model, such as diagrams, geometric 3D models or other documents. This is also nothing new.

The fundamental idea behind the Simantics environment is creating a framework for the integration of multiple modelling disciplines with a clear focus on real-time data and simulation support. The desire is to have diagramming tools specifically crafted for this architecture and the graph data model, without external sanctions or unwanted compromise. Especially important is the utopia of behaving diagrams which we hope to achieve in a natural way through real-time data support and symbol parametrisation. So far the use of visualisation in 2D process simulation has mostly involved use of textual monitors and trends of selected values. The symbol parametrisation mechanisms created in this thesis provide groundwork for more advanced visualisation.

The unified data representation and modelling scheme along with the capability for free association of information can be considered the greatest virtues of the ontology-based graph data model approach. The graph model is a very fine-grained way of representing

information and therefore also inevitably more computationally intensive than regular more coarse-grained models. However, with scalable and optimised architectural data storage solutions and careful ontology modelling, the graph-based ontology approach seems feasible for real-world applications.

Bibliography

[Ado98] Adobe Systems Incorporated et al. Precision Graphics Markup Language (PGML), April 1998. http://www.w3.org/TR/1998/NOTE-PGML. Last checked 2007-02-27.

[Art07] Artem Tikhomirov and Alexander Shatalin. GMF Best Practices. EclipseCon 2007, March 2007. Available at http://www.eclipsecon.org/2007/index.php?page=sub/&id=3739. Last checked 2007-05-01.

[Aut98] Auto-trol Technology. CGM: A Non-Proprietary, Editable 2D Graphics Format That Handles Vector, Raster, and Text Data. *Engineering Automation Report*, 7(8), August 1998. Available at http://www.tech-illustrator.com/TI/Articles/ear_cgm.pdf. Last checked 2007-03-08.

[BG02] Bertrand Braunschweig and Rafiqul Gani, editors. *Software Architectures and Tools for Computer Aided Process Engineering, 11*, volume 11 of *Computer-Aided Chemical Engineeirng*. Elsevier, first edition, 2002. ISBN 0-444-50827-9.

[BGM04] Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. Toolkit design for interactive structured graphics. *IEEE Transactions on Software Engineering*, 30(8):535–546, 2004.

[BLH+] Anders Brink, Daniel Lindberg, Mikko Hupa, Timo Fabritius, Jaana riipi, Jouko Härkki, Ari Kankkunen, Mika Järvinen, Carl-Johan Fogelholm, Seppo Louhenkilpi, Shenqiang Wang, Petteri Kangas, Pertti Koukkari, Reijo Lilja, Risto Pajarre, Karri Penttilä, Fredrik Bergström, and Kenneth Eriksson. Multiphase Chemistry in Process Simulation (MASIT04 (VISTA)). MASI Technology Programme 2005-2009, Yearbook 2007. Technology Review xxx/2007. Tekes. To be published.

[BLN86] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Comput. Surv.*, 18(4):323–364, 1986.

[BLP05] Chris Bizer, Ryan Lee, and Emmanuel Pietriga. Fresnel - Display Vocabulary for RDF, April 2005. http://www.w3.org/2005/04/fresnel-info/. Last checked 2007-03-22.

[Bun97] Peter Buneman. Semistructured Data. In *PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 117–121, New York, NY, USA, 1997. ACM Press.

[Car] carto.net — Cartographers on the net. http://www.carto.net/. Last checked 2007-05-02.

[Cro07] Thomas Crockett. GL2D: Overhauling Draw2D's graphics engine with OpenGL, March 2007. http://www.eclipsecon.org/2007/index.php?page=sub/&id=4138, presentation material not available. Last checked 2007-05-01.

[CS04] Ida K.L. Cheung and Geoffrey Y.K. Shea. Visualizing and Editing GIS data with SVG for Internet and Mobile Users. In *SVG Open 2004, 3rd Annual Conference on Scalable Vector Graphics*, September 2004. Available at http://www.svgopen.org/2004/papers/VisualizingEditingGISdatawithSVG/. Last checked 2007-05-02.

[DH05] AnHai Doan and Alon Y. Halevy. Semantic integration research in the database community: A brief survey. *AI Magazine*, 26(1):83–94, 2005.

[DHH02] David Duce, Ivan Herman, and Bob Hopgood. Web 2D Graphics File Formats. *Computer Graphics Forum*, 21(1):43–64, 2002.

[ea97] Serge Abiteboul et al. Querying Semi-Structured Data. *Proceedings of the 6th International Conference on Database Theory*, pages 1–18, 1997.

[ea98] Tim Berners-Lee et al. Uniform Resource Identifiers (URI): Generic Syntax. Request for Comments, August 1998. http://www.ietf.org/rfc/rfc2396.txt. Last checked 2007-02-15.

[Ecla] Eclipse Foundation. Graphical Editing Framework (GEF). http://www.eclipse.org/gef/. Last checked 2007-02-26.

[Eclb] Eclipse Foundation. Graphical Modeling Framework (GMF). http://www.eclipse.org/gmf/. Last checked 2007-02-26.

[Eclc] Eclipse Foundation. SWT: The Standard Widget Toolkit. http://www.eclipse.org/swt/. Last checked 2007-02-27.

[EM97] Hilding Elmqvist and Sven Erik Mattsson. An Introduction to the Physical Modeling Language Modelica. In *Proceedings of the 9th European Simulation Symposium, ESS'97*, October 1997. Available at http://www.modelica.org/publications/papers/ESS97Modelica.pdf. Last checked 2007-05-03.

[Flo03] Luciano Floridi, editor. *Blackwell Guide to the Philosophy of Computing and Information*, chapter Ontology by Barry Smith (preprint version), pages 155–166. Oxford: Blackwell, 2003. Available at http://ontology.buffalo.edu/smith/articles/ontologies.htm. Last checked 2007-03-29.

[Fou07] The Eclipse Foundation. The Official Eclipse FAQs, 2007. http://wiki.eclipse.org/index.php/The_Official_Eclipse_FAQs. Last checked 2007-04-26.

[Gea06] Paul Gearon. Mulgara — an open source scalable RDF storage database in Java. Website, 2006. http://mulgara.org/. Last checked 2007-02-19.

[GHJV95] Erich Gamma, Richard Helm, Ralph Jonhson, and John Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1995.

[Gru93] Thomas R. Gruber. A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Gru95] Thomas R. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. *International Journal of Human-Computer Studies*, 43(5):907–928, November/December 1995.

[Gua98] Nicola Guarino. Formal ontology in information systems. In Nicola Guarino, editor, *Proceedings of FOIS'98*, pages 3–15. IOS Press, Amsterdam, June 1998.

[Hei95] Sandra Heiler. Semantic Interoperability. *ACM Comput. Surv.*, 27(2):271–273, 1995.

[Her06] Ivan Herman. Questions (and answers) on the semantic web, September 2006. Available at http://www.w3.org/2006/Talks/0927-Berlin-IH/. Last checked 2007-02-16.

[HRO06] Alon Halevy, Anand Rajaraman, and Joann Ordille. Data Integration: the Teenage Years. In *VLDB'2006: Proceedings of the 32nd international conference on Very large data bases*, pages 9–16. VLDB Endowment, 2006.

[HW04] Lofton Henderson and Dieter Weidenbrück. Applicability of CGM versus SVG for technical graphics, April 2004. http://www.cgmopen.org/technical/cgm-svg-20040419.html. Last checked 2007-05-01.

[ILO06] Frank Ipfelkofer, Bernhard Lorenz, and Hans Jürgen Ohlbach. Ontology Driven Visualisation of Maps with SVG — An Example for Semantic Programming. *Proceedings of the conference on Information Visualization*, 0:424–429, 2006.

[Inc] Vivid Solutions Inc. JTS — Java Topology Suite. http://www.vividsolutions.com/jts/jtshome.htm. Last checked 2007-04-03.

[Inc07] Franz Inc. AllegroGraph 64-bit RDFStore, 2000–2007. http://www.franz.com/products/allegrograph/. Last checked 2007-02-21.

[Int] Intergraph. SmartPlant P&ID. http://www.intergraph.com/smartplant/pid/. Last checked 2007-05-02.

[Int97] International Organization for Standardization, Geneva, Switzerland. ISO IS 10628:1997(E): Flow diagrams for process plants — General rules, April 1997.

[Int99] International Organization for Standardization, Geneva, Switzerland. ISO IS 8632:1999: Information technology — Computer graphics — Metafile for the storage and transfer of picture description information, 1999.

[Int03] International Organization for Standardization, Geneva, Switzerland. ISO IS 15926-2:2003: Industrial automation systems and integration — Integration of life-cycle data for oil and gas production facilities — Part 2: Data model, 2003.

[Int05a] International Organization for Standardization, Geneva, Switzerland. ISO CD TS 15926-4:2004: Industrial automation systems and integration — Integration of life-cycle data for oil and gas production facilities — Part 4: Initial reference data, January 2005.

[Int05b] International Organization for Standardization, Geneva, Switzerland. ISO DIS 10303-221:2005(E): Industrial automation systems and integration — Product data representation and exchange Part 221: Application protocol: Functional data and their schematic representation for process plant, 2005.

[Kar02] Tommi Karhela. *A software architecture for configuration and usage of process simulation models. Software component technology and XML-based approach*. VTT Publications 479, Espoo, 2002. ISBN 951-38-6011-6.

[Kar07] Tommi Karhela. Personal communication, 2007.

[Khr] Khronos Group. OpenVG - The standard for Vector Graphics Acceleration. http://www.khronos.org/openvg/. Last checked 2007-05-02.

[KHRS05] Y. Kalfoglou, B. Hu, D. Reynolds, and N. Shadbolt. Semantic integration technologies survey. Technical Report 10842, School of Electronics and Computer Science, University of Southampton, Southampton, UK, 2005.

[KS03] Yannis Kalfoglou and Marco Schorlemmer. Ontology Mapping: the State of the Art. *Knowl. Eng. Rev.*, 18(1):1–31, 2003.

[Len02] Maurizio Lenzerini. Data Integration: a Theoretical Perspective. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, New York, NY, USA, 2002. ACM Press.

[LK06] Tuukka Lehtonen and Tommi Karhela. Ontology approach for building and visualising process simulation models using 2d vector graphics. In Esko Juuso, editor, *SIMS 2006 Proceedings of the 47th Conference on Simulation and Modeling*, pages 141–146. Finnish Society of Automation, SIMS — Scandinavian Simulation Society, September 2006.

[LW01] Chris Lilley and Dieter Weidenbrück. WebCGM and SVG: A Comparison, May 2001. http://www.gca.org/papers/xmleurope2001/papers/html/s12-1.html. Last checked 2007-05-01.

[Mat] MathCore Engineering AB. MathModelica. http://www.mathcore.com/products/mathmodelica/. Last checked 2007-05-03.

[MDG+04] William Moore, David Dean, Anna Gerber, Gunnar Wagenknecht, and Philippe Vanderheyden. *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*. IBM, February 2004. Available at http://www.redbooks.ibm.com/abstracts/sg246302.html. Last checked 2007-03-05.

[Mic98] Microsoft Corporation et al. Vector Markup Language (VML), May 1998. http://www.w3.org/TR/NOTE-VML. Last checked 2007-02-27.

[ML05] Jeff Mcaffer and Jean-Michel Lemieux. *Eclipse Rich Client Platform — Designing, Coding, and Packaging Java Applications*. The Eclipse Series. Addison-Wesley

Professional, pap/cdr edition, October 2005. See also http://eclipsercp.org. Last checked 2007-03-15.

[Mod] Modelica Association. Modelica. http://www.modelica.org/. Last checked 2007-05-03.

[Mur] Alan Murta. GPC — General Polygon Clipper library. http://www.cs.man.ac.uk/~toby/alan/software/. Last checked 2007-03-08.

[Nor05] Craig Northway. Understand Compositing and Color extensions in SVG 1.2 in 30 minutes! In *SVG Open 2005, 4th Annual Conference on Scalable Vector Graphics*, August 2005. Available at http://www.svgopen.org/2005/papers/abstractsvgopen/. Last checked 2007-03-08.

[Noy04] Natalya F. Noy. Semantic Integration: a Survey of Ontology-Based Approaches. *SIGMOD Rec.*, 33(4):65–70, 2004.

[oF07] VTT Technical Research Centre of Finland. Simantics — Open modelling and simulation environment, 2007. https://www.simulationsite.net/simantics. Last checked 2007-03-15.

[Ora05] Oracle. RDF Support in Oracle. Whitepaper, February 2005. Available at http://www.oracle.com/technology/tech/semantic_technologies/pdf/semantic_tech_rdf_wp.pdf. Last checked 2007-02-19.

[OS99] Aris M. Ouksel and Amit Sheth. Semantic Interoperability in Global Information Systems. *SIGMOD Rec.*, 28(1):5–12, 1999.

[oSN04] National Institute of Standards and Technology (NIST). Cost Analysis of Inadequate Interoperability in the U.S. Capital Facilities Industry, August 2004. Available at http://www.bfrl.nist.gov/oae/publications/gcrs/04867.pdf. Last checked 2007-04-20.

[Pal07] Matti Paljakka. Personal communication, 2007.

[PD84] Thomas Porter and Tom Duff. Compositing Digital Images. *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, 18(3):253–259, 1984.

[Pro] Programming Environment Laboratory, Department of Computer and Information Science, Linköping University. The OpenModelica Project. http://www.ida.liu.se/~pelab/modelica/OpenModelica.html. Last checked 2007-05-03.

[Ree03] Trygve Reenskaug. The Model-View-Controller (MVC) Its Past and Present. *JavaZONE, Oslo, 2003. JAOO, Århus, 2003*, September 2003. Available at http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/HM1A93.html. Last checked 2007-04-25.

[Roc04] Christophe Roche. Ontology: a survey. Technical report, University of Savoie — Equipe Condillac, 2004.

[SP06] Pekka Siltanen and Antti Pärnänen. Comparison of data models for plant life-cycle information management. In *Proceedings of the 13th ISPE International Conference on Concurrent Engineering: Leading the Web in Concurrent Engineering.*, pages 346–353, 2006.

[Sut03] Ivan Edward Sutherland. Sketchpad: A man-machine graphical communication system. Technical Report UCAM-CL-TR-574, University of Cambridge, Computer Laboratory, September 2003.

[Tuu06] Kimmo Tuukkanen. Representing Industrial Data models in OWL Web Ontology Language. Master's thesis, Helsinki University of Technology, November 2006.

[UG96] Mike Uschold and Michael Grüninger. Ontologies: principles, methods, and applications. *Knowledge Engineering Review*, 11(2):93–155, 1996.

[uis] uismedia Lang and Müller. Web Mapping with SVG Technology. http://www.mapviewsvg.com/index.html. Last checked 2007-05-02.

[Uni] University of Maryland. Piccolo Toolkit. http://www.cs.umd.edu/hcil/piccolo/. Last checked 2007-02-27.

[VTTa] VTT Technical Research Centre of Finland. BALAS Process Simulation Software. http://balas.vtt.fi/. Last checked 2007-02-26.

[VTTb] VTT Technical Research Centre of Finland. The Advanced Process Simulator Environment. http://apros.vtt.fi/. Last checked 2007-02-26.

[Wor99] World Wide Web Consortium. Resource Description Framework (RDF), 1999. http://www.w3.org/RDF. Last checked 2007-02-01.

[Wor01] World Wide Web Consortium. WebCGM 1.0 Second Release, December 2001. http://www.w3.org/TR/REC-WebCGM/. Last checked 2007-03-07.

[Wor03] World Wide Web Consortium. Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation, January 2003. http://www.w3.org/TR/SVG/. Last checked 2007-02-27.

[Wor04a] World Wide Web Consortium. RDF Vocabulary Description Language 1.0: RDF Schema, February 2004. http://www.w3.org/TR/rdf-schema/. Last checked 2007-02-01.

[Wor04b] World Wide Web Consortium. RDF/XML Syntax Specification (Revised), February 2004. http://www.w3.org/TR/rdf-syntax-grammar/. Last checked 2007-02-16.

[Wor04c] World Wide Web Consortium. Web Ontology Language (OWL), February 2004. http://www.w3.org/2004/OWL/. Last checked 2007-02-01.

[Wor06] World Wide Web Consortium. SPARQL Query Language for RDF, October 2006. http://www.w3.org/TR/rdf-sparql-query/. Last checked 2007-02-20.

[Wor07] World Wide Web Consortium. WebCGM 2.0, January 2007. http://www.w3.org/TR/webcgm20/. Last checked 2007-03-07.