Ontology driven application prototyping in a smartspace environment

Jon von Weymarn

Master of Science Thesis
Supervisor: Professor Johan Lilius
Embedded Systems Laboratory
Faculty of Technology
Åbo Akademi University

Abstract

Today devices with a microchip or processor can be found anywhere. This leads to problems for developers to keep up with the development of these devices. Another issue that arises, from this abundance of devices, is interoperability between them. This thesis presents a method to aid in application development using smartspaces and agents. In this method a SIB device operates as a broker that other devices communicate with. To represent information in the smartspace RDF representation and ontologies are used. This makes the information more comprehensible. The solution presented in this thesis is a generator that creates a code framework, in the Python programming language, from a given OWL ontology. This generated framework enables an object oriented approach to data in the ontology as well as automates cumbersome tasks concerning device communication. As an example application, to illustrate the functionality and the benefit from using the generated framework, an application associated with calendar information is presented. First the application is created directly and after that the generator is used to create a framework for development of calendar applications. The application is created again, using the framework, to prove the potential the framework offers to simplify the task for the developer.

Keywords: Smartspace, Interoperability, Ontology driven, Application development, Distributed environment, Prototyping, Python framework

Contents

List of Figures

List of Tables

Listings

Abbreviations

API	Application Programming Interface
DIEM	Device and Interoperability Ecosystem
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IP address	Internet Protocol address
OWL	Web Ontology Language
OWL DL	OWL Description Logic
RDF	Resource Description Framework
RDFS	RDF Schema
SIB	Semantic Information Broker
UUID	Universally Unique Identifier
URI	Uniform Resource Identifier
W3C	World Wide Web Consortium
WQL	Wilbury Query Language
XML	Extensible Markup Language
ÅAU	Åbo Akademi University

Sammanfattning

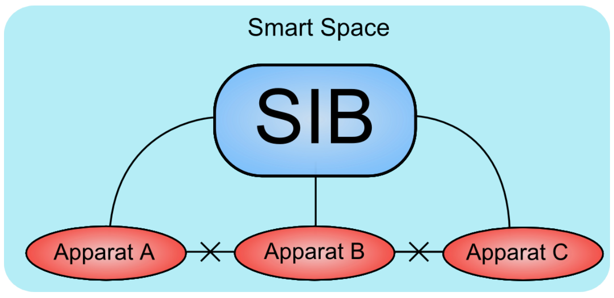
Ontologidriven applikationsutveckling i en smartspace-miljö

Under de senaste åren har inbyggda datorsystems popularitet ökat märkbart. Idag kan man hitta ett mikrochip eller en processor i vardagliga apparater såsom i en mikrovågsugn, garagedörr eller till och med i klädsel. Detta har lett till problem för tillverkaren, och systemutvecklaren, när de försöker hinna med i utvecklingen. Som ett resultat av detta lider ofta apparatens användarvänlighet. Ett annat problem som uppstår från detta överflöd av apparater är interoperabiliteten mellan dessa. Interoperabilitet betyder möjligheten att koppla samman flera apparater som klarar av att arbeta tillsammans.

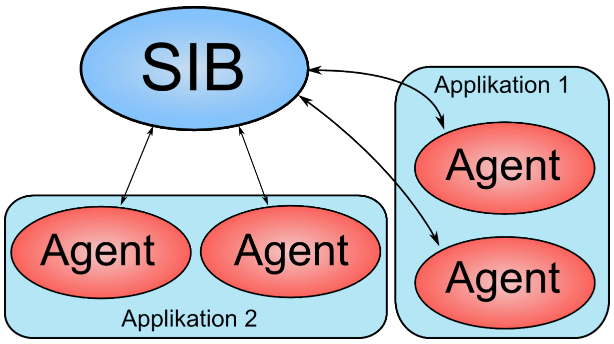
Detta diplomarbete är en del av *Device and Interoperability Ecosystem* (DIEM) projektet. Projektets mål är att skapa en plattform för systemutvecklare som skulle underlätta arbetet att skapa applikationer där flera apparater samverkar. Visionen är att plattformen skulle evolvera till ett sådant skede att en person utan teknisk bakgrund kunde använda den för att skapa applikationer där apparaterna i hans liv sammanverkar.

För att uppnå de ovannämda målen används en arkitektur med så kallade *smartspace*. Ett smartspace består av apparater och en *Semantic Information Broker* (SIB). SIB:ens uppgift är att hantera apparaterna samt att fungera som en databas med information som apparaterna delar. Varje apparat kommunicerar med, och endast med, SIB:en. För att utbyta information ansluter sig en apparat till SIB:en och skriver i den gemensamma databasen. Sedan kan en annan apparat anknyta sig till SIB:en för att läsa denna information. Ingen kommunikation sker direkt mellan apparater. Detta är illustrerat i Figur 1.

Ytterligare, är ett av målen i projektet att möjliggöra applikationer som består av flere apparater. För att uppnå detta används en modulbaserad modell där funktionaliteten av applikationen delas upp i så kallade agenter. En agent är definierad som en självständig helhet som utför endas en uppgift. Genom att



Figur 1: Smartspace arkitekturen bestående av en SIB och tre apparater

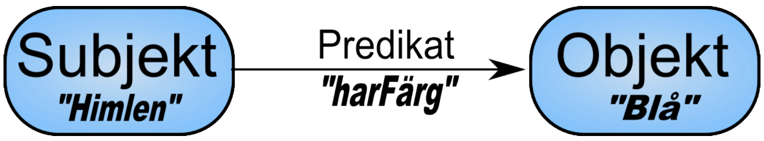


Figur 2: Två applikationer som produkten av deras tillhörande agenter

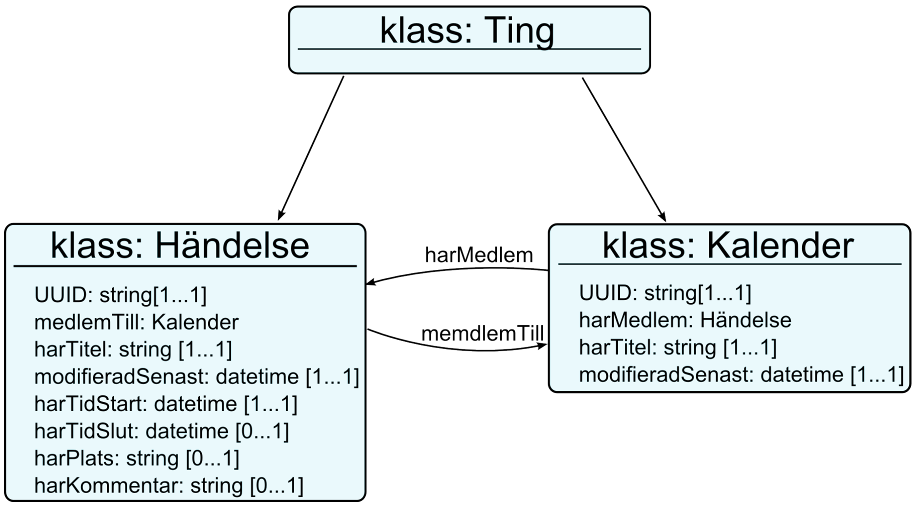
sedan kombinera agenter som utför de önskade funktionerna skapas applikationen. I Figur 2 illustreras två applikationer som är uppbygda av dess respektive agenter. I praktiken kunde detta illustreras med en e-post klient som bestod av, minst, en agent för att sända, en för att motta och en för att uppvisa e-post.

För att representera information, i DIEM projektet, används *Resource Description Framework* (RDF). RDF baserar sig på att skapa påståenden, i form av trippler, som beskriver informationen. En trippel i RDF har formen `subjekt-predikat-objekt`. Subjektet betecknar vad det är vi beskriver. Predikatet beskriver en egenskap eller en relation som subjektet har till objektet. Ett exempel, som kan ses i Figur 3, är `himlen-harFärg-blå`. En alternativ trippel som beskriver samma information är `blå-ärFärgTill-Himlen`.

En av orsakerna varför RDF används för att representera information är för att trippler kan förstås både av en människa och en dator. Dessutom om man använder en ontologi kan man utvidga, dela och kontrollera informatio-



Figur 3: En trippel som beskriver himlen och dess färg.



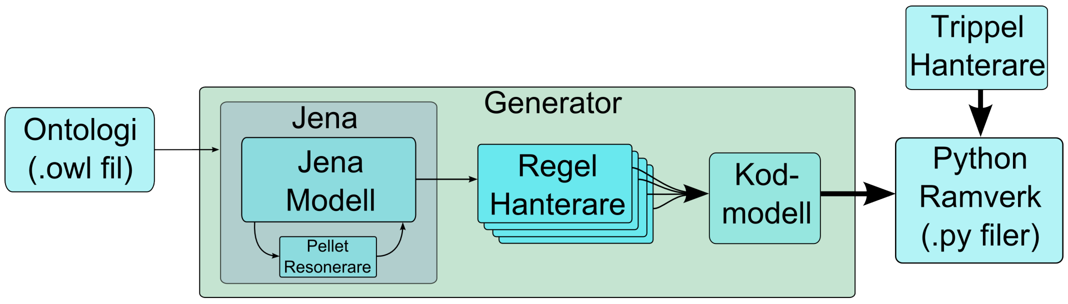
Figur 4: En enkel ontologi som beskriver kalenderinformation

nen. Till exempel, anta att många databaser innehåller medicinsk information. Om dessa databaser delar samma ontologi kan informationen samlas ihop av en dator och kombineras till en enda databas. För ett annat exempel, där informationen utvidgas, kan *friend of a friend* ontologin användas. Denna ontologi beskriver relationer mellan personer. Om databasen har tripplerna `John-harMor-Emma` och `Emma-harMor-Eva`, så kan en agent med ontologins regler resonera till en ny trippel `John-harMormor-Eva`.

En ontologi består av klasser, instanser och egenskaper. En klass är en abstrakt grupp eller samling av objekt. Klasser beskriver begrepp i en domän. Till exempel, klassen `vin` beskriver alla vin. Klasser är ordnade i en hierarkisk trädstruktur där klasser ovanför en specifik klass är *superklasser* och klasser som ligger under i hierarkin är *subklasser*. Klassen `vin` kan ha subklassen `vitvin`, `rödvin` och `rosévin`. En instans är en medlem av en specifik klass, till exempel vinet `Château Lafite Rothschild` är en medlem av klasserna `vin` och `röda viner`. Egenskaper delas upp i objekt- eller datatypsegenskaper. Objektegenskaper relaterar en medlem av en klass till en annan medlem. Ovan relaterades medlemmen `John` till medlemmen `Emma` med objektegenskapen `harMor`. Datatypsegenskaper relaterar en medlem till ett datavärde. `John` kan, till exempel, ha egenskapen `harÅlder` som relaterar till siffervärdet `30`.

Figur 4 beskriver ontologin som används i detta diplomarbete. Denna ontologi används för att beskriva kalenderinformation. Till ontologin hör klasserna `händelse` och `kalender`. Dessa är relaterade till varandra med objektegenskapen `medlemTill` (och inversen `harMedlem`). Båda klasserna har dessutom datatypsegenskaper som beskriver relevant information.

Den praktiska delen av detta diplomarbete är en generator som utav en definier-



Figur 5: Överblick av generatorns arkitektur

ad ontologi skapar ett objektorienterat Python ramverk. Detta rakverk underlättar, för systemutvecklaren, skapandet av applikationer som utnyttjar ontologin. Ramverket hanterar kommunikationen mellan apparaten och SIB:en samt innehåller en API[1] som underlättar arbetandet med informationen i ontologin. Dessutom upprätthåller rakverket reglerna inom ontologin.

Generatorns uppbyggnad är illustrerad i Figur 5. Det första steget, av genereringsprocessen, är att översätta ontologin till en Jena modell. Jena är ett ramverk av verktyg som är specifikt utvecklade för skapandet av applikationer för semantiska webben. Jena erbjuder verktyg som underlättar hanteraningen av ontologiinformationen samt moduler som kan förbättra den ursprungliga ontologin. Den viktigste modulen i detta diplomarbete är Pellet resoneraren. Pellet resoneraren läser Jena modellen, av ontologin, och utför resonering på den, det vill säga, den verifierar att alla påståenden är rimliga. Förutom detta kan resoneraren även finna information som kan utvidga ontologin, som i tidigare nämda *friend of a friend* ontologin (`John-harMor-Emma` och `Emma-harMor-Eva` kan resoneras till en ny trippel `John-harMormor-Eva`).

Nästa steg i genereringsprocessen, efter resoneringen, är regelhantering. Ontologins byggstenar är ett antal regler och dessa översätts till en objektorienterad kodmodell av deras motsvarande hanterare. Hanterarna bygger gradvis upp kodmodellen och efter det kan den serialiseras till ett objektorienterat programmeringspråk, i detta fall en Python. I samband med serialiseringen läggs även trippelhanteraren med i ramverket. Trippelhanteraren är en statisk modul vars uppgift är att handha kommunikationen mellan agenten och SIB:en. Dessutom innehåller den en lokal databas som innehåller alla relevanta trippler som används inom agenten.

Genom att använda det genererade ramverket istället för att manuellt manipulara information i SIB:en ger systemutvecklaren enorma fördelar. Den största

[1]En *Application Programming Interface*, är en regeluppsättning som kan underlätta kommunikationen mellan olika programvara.

fördelen är att informationen är uppbygd som objekt vilka lätt kan manipuleras med genererade lättförståeliga metoder. Förutom detta behöver systemutvecklaren inte sköta om kommunikationen mellan agenten och SIB:en då detta sköts automatiskt av ramverket.

Den nuvarande versionen av SIB:en och smartspace miljön är mycket primitiv. Detta kan ses från de dominerande problemen med den version som beskrivs i detta diplomarbete. Det största problemet ligger i informationens säkerhet. Alla agenter kan läsa och modifiera informationen i SIB:ens databas ifall de kan ansluta sig till den. Detta faktum, att informationen i databasen inte är pålitlig, gör att man inte kan använda systemet för praktisk tillämpning. För att lösa detta problem måste SIB arkitekturen utveklas vidare.

Ett annat problem med den nuvarande arkitekturen är resonering. Resoneringens fulla potential kan inte utnyttjas i detta skede, på grund av att detta endast sker då ramverket genereras. Nuvarande resoneringen stöder kontroll av att ontologin är giltig samt den drar slutsatser som kan leda till nya relationsegenskaper. Allt detta sker vid generering av ramverket så det kan inte garanteras att en agent inte under körtid sätter ontologin i en inkonsistent tillstånd. För att lösa detta problem borde resonering även ske i SIB:en under körtid.

Oavsett dessa problem är det den allmänna uppfattningen, inom medlemmarna i DIEM projektet, att SIB arkitekturen har en enorm potential. Fördelen av att använda en ontologi, för att representera information, är att man på kort tid kan få förståelse för hur apparaterna och deras information kan användas. Förutom detta möjliggör modulariteten som agentarkitekturen erbjuter ett fullständigt nytt synsätt för applikationsutveckling. Sista slutsatsen är att efter arkitekturen har utvecklats vidare, och bland annat de ovannämda problemen är lösta, kan platformen nå ett skede där den kan börja användas av icketekniskt kunniga för att skapa applikationer.

Chapter 1

Introduction

In recent years embedded systems in devices have become increasingly popular and widespread. In the past embedded systems were mostly associated with anti-braking systems for cars, aircraft electronics and plant control systems. Today we have moved toward a situation where embedded devices can be found almost everywhere. A microchip or processor can be found in everyday devices such as a microwave oven, garage door controller or even in clothing. This fact has resulted in problems for developers and engineers when trying to keep up with the development of these devices. Another issue that arises from this abundance of devices is the interoperability between them. What is meant by interoperability, in this context, is the ability to connect multiple devices, by different manufacturers, so that they can collaborate. To help developers and to find a solution to the interoperability problem are two of the goals of this work.

An another important issue for this work is to improve the usability of devices around us. In the past development of embedded devices has focused mostly on the device itself and on how to achieve all the desired functionality as efficiently as possible. This can be seen as vertical development when manufacturers make all the parts of the product themselves. Nokia is a good example of this as they make the hardware, the operating system and applications that are to be used with the device. For the end user vertical development results in frustration because there is a constant need to learn how devices should be used. On the other hand there are many problems for the manufacturer, like the lack of modularity and long time to market. In a perfect world manufacturers should be able to reuse everything that they create at an earlier stage, but today that is not a possibility.

This thesis is part of a project called *Devices and Interoperability Ecosystem*[1]

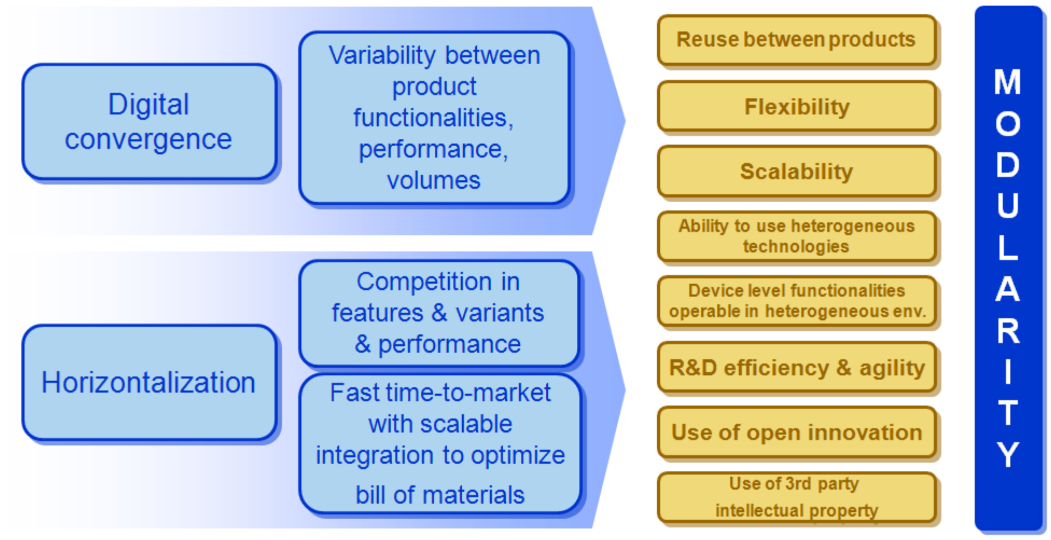

Digital convergence
Variability between product functionalities, performance, volumes
Horizontalization
Competition in features & variants & performance
Fast time-to-market with scalable integration to optimize bill of materials
Reuse between products
Flexibility
Scalability
Ability to use heterogeneous technologies
Device level functionalities operable in heterogeneous env.
R&D efficiency & agility
Use of open innovation
Use of 3rd party intellectual property
MODULARITY

Figure 1.1: Drivers for the DIEM project

(DIEM). The DIEM project targets to define and open a completely new domain for technology and service innovation in a global scale. The projects proposal for a solution to the above described issues is to enable a more horizontal view of development. In this solution a manufacturer, like Nokia, could focus on core areas of the product, while allowing third parties to develop other areas of the product. In Figure 1.1 the benefits of this approach are demonstrated.

The vision of the project is to have a platform, which will evolve to a standard. This platform would be aimed at the non-technically skilled person[1]. This person could use the platform to create applications where the devices in his life interoperate. He would be able to do this even if he did not have knowledge about how these devices function. Concretely, this will be achieved with a framework that handles all tasks concerning the underlying technologies.

For example, if a developer wanted to use a mobile phone as a remote control unit for a television nearby, he would start by scanning for devices with a tool provided in the platform. The tool would automatically find all nearby devices so that the developer could select the ones that he wants to interact. Then a list of available functionality between the devices would be presented and the developer could simply select those desired from the list. An advanced user could modify these to achieve more complex tasks. The goal is that the most time consuming part for the user would be if he wanted to design the appearance of the user interface instead of using a provided standard interface.

Actually a tool like this is not yet possible as there is still much development

[1]from now on called the *bedroom coder*

to be done. This thesis is about an overall view of the problem and the task of hiding as much of the underlying interoperative functionality. A generator, that generates a Python code framework, will be presented as a first solution in the later chapters.

1.1 Structure of thesis

The next chapter describes the necessary background information on what kind of technologies the tool utilizes. The chapter describes an approach using so-called smartspaces, with a broker device handling communication and information storage. It also covers the issue of how information in a space is represented in such a way that it is understandable to both humans and computers. The concept of using an ontology and a knowledge representation language to restrict the information will also be introduced.

In Chapter 3 a calendar case study will be presented, where different calendar standards represent different devices. A manually created first implementation will be demonstrated which synchronizes the contents of both calendars.

Chapter 4 will cover the generator that automatically translates an ontology into an object oriented Python framework. This framework will be used to connect devices and make them work with each other. Parts of the generated output will be described to gain an understanding of how the framework operates.

In Chapter 5 the manually created solution for the calendar example will be reviewed again, but now by using the output produced by the tool. This example will show the potential of the tool to considerably simplify the work of the developer.

Chapter 2

Background

Before presenting a more detailed view of the proposed tool for creating applications with interoperating devices, some technologies that are used in the development of the tool will to be covered. This chapter is dedicated to explain a number of concepts to facilitate the understanding of them. Each concept is placed in its context using examples of how the concept is used in the tool, or in other parts of this project.

2.1 Semantic Information Broker

In an environment where many devices are to work together, there are many issues that need to be considered before interoperability can be achieved. To start with, the devices need to be able to find each other in order to exchange information. Once they have found each other a connection is needed so that information can be shared across the network.

To facilitate these tasks so called *smartspaces* will be used. A smartspace consists of devices and a *Semantic Information Broker* (SIB)[2]. The role of the SIB is to manage the devices and store shared information that the devices can access and modify. Each device communicates with, and only with, the SIB. In order to exchange information one of the devices writes into the SIB and another can then read this information. An illustration of this can be seen in Figure 2.1. The scope of a smartspace can be scalable to very different domains, everything from a living room with five devices to an airport with hundreds of devices and thousands of users. Concerning devices the scope of the smartspace has to be scalable from simple energy limited devices, like sensors, to powerful centralized devices like servers or desktop computers.

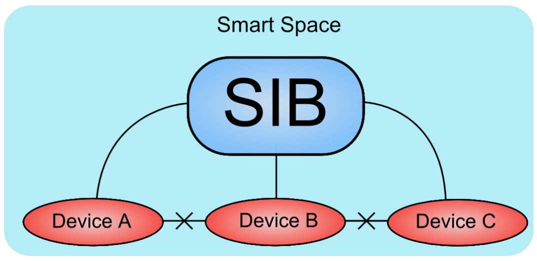


Figure 2.1: Building blocks of a smartspace

Having this space oriented approach with a SIB acting as a broker gives us many benefits. The greatest is that we do not need to worry about how the devices connect with each other. Instead, as long as every device can communicate with the common SIB, the system is functional. This results in flexibility where devices can be disconnected intentionally, or unintentionally, without affecting the overall system. Also, with the space concept carried out this way, it is easily extendable if SIBs are allowed to connect and merge. This also means that devices could, via an online SIB, be accessible, from for example, the web. At this stage of this project the smartspace is however limited to only one SIB and its surrounding devices.

On the other hand this approach also poses many potential challenges that need to be addressed. For instance, validation of the shared data. A device has no way of knowing if the device that wrote some information in the SIB is reliable or not. This is due to the fact that all devices that are connected to the SIB are able to write or modify information in the database. Consequently security becomes an issue. At this stage of this project there are no limitations on what parts of the data is visible. Other potential problems are transactions, information lifetime and data consistency. These are issues that will be addressed in the coming chapters.

The SIB used in this thesis has been developed and created by Nokia, that is a close partner in the project. This thesis will not go into great detail about the inner workings of the SIB, but a short description is however needed in order to gain an understanding of the capabilities of the SIB.

2.1.1 Capabilities of the SIB

The SIB offers a persistent data storage back-end, which is based on OWL (described in Section 2.5). The database can be seen as a common bulletinboard[1] where devices in a *push* manner share information. Push in this context means

[1]blackboard, greenboard, whiteboard

that devices write everything they know into the database. Devices are able to connect and disconnect to the SIB database. When connected the basic functions for manipulating and using information in the SIB are[2]:

- **Insert:** insert information atomically
- **Retract:** remove information atomically
- **Query:** synchronously query; returns information without delay
- **Subscribe:** asynchronously set up a subscription for a given query; return information when a change occurs
- **Unsubscribe:** terminate processing of the given subscription

Query

When a computer agent performs a valid query and the information requested exists in the SIB the requested information is simply returned to the agent. If the information does not exist, `NULL` is returned. In the current SIB implementation two kinds of queries are possible. The first is a simple triple query which looks for a pattern match in a given triple. The second is a more complex Wilbur[2] query. Using a Wilbur query it is possible to look for subgraphs within the RDF graph (described in Section 2.2). The query is built by using the following expressions as regular expressions[3]:

- **Sequence**: the operator `:seq` matches a sequence of n steps in the graph, consisting of subexpressions e_1,e_2,...,e_n; the operator `:seq+` is similar except any sequence e_1,e_2,...,e_k for k in [1...n] will match.
- **Disjunction**: the operator `:or` matches any one of n subexpressions e_1,e_2,...,e_n.
- **Repetition**: the operator `:rep*` matches the transitive closure of subexpression e; the operator `:rep+` is the same as `(:seq e (:rep* e))`.
- **Inverse**: satisfaction of `(:inv e)` requires the path defined by the subexpression e to be matched in reverse direction.
- **Container membership**: the atom `:members` will match any of the rdf:_1, rdf:_2, rdf:_3, etc. container membership properties.

[2]Wilbur is a toolkit for writing RDF-enabled software developed by Nokia Research Center

- **Wildcard**: the atom `:any` will match any label

Creating queries with these regular expressions is not the simplest of tasks. This is an example of a task that will be greatly simplified by the code framework seen in the later chapters.

Subscription

The case with a subscription is somewhat different compared to the query. A subscription can be seen as a query that is left in the SIB. Each time a change is made in the dataset that covers the subscription the query is performed and the results are sent to the creator of the subscription. The lifetime of the subscription is governed by the node that created it.

For example, an application can subscribe to a temperature value from a sensor device and every time the value changes the application gets notified thus enabling it to reacts somehow.

2.2 Describing data with the Resource Description Framework

To achieve interoperability devices must be able to find data from another device, but also be able to understand this data. Today this is a major problem due to the lack of a common standard for semantic interoperability. There is no understanding of a common protocol, or of how data should be described in such a way that it could be conveniently shared across devices. In order to tackle this problem the *Resource Description Framework* (RDF) standard will be used in this thesis.

Computers speak a language based on ones and zeros while humans like to use words in order to form sentences about what we want to communicate. RDF can be seen as something computers and humans both can understand. The RDF standard is based on making statements about resources in the form of a triple. A triple in RDF can be seen as an expression of the components `subject-predicate-object`. The *subject* denotes the resource that we are describing, and the *predicate* denotes traits or aspects of the resource. The predicate further expresses a relationship between the subject and the *object*. For an example, one way to represent the notion "The sky has the color blue" in RDF is as the triple: a subject denoting the `sky`, a predicate denoting

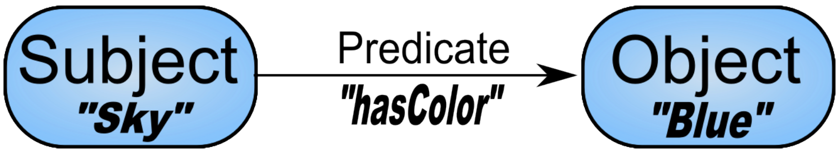


Figure 2.2: A simple triple with information about the sky and its color

`has the color`, and an object denoting `blue`. An illustration of this is seen in Figure 2.2. An alternative triple describing the same thing, but in reverse, is `Blue-isColorOf-Sky`[4].

2.2.1 Uniform Resource Identifier

To further understand RDF we need to understand the *Uniform Resource Identifier* (URI). As there can be more than one person who is called, for example, `John Smith` there needs to be a way to separate all these from each other. The solution for this is the URI. A URI consist of a string of characters and is used to identify, or name, a resource on the internet. The syntax of a URI offers a URI *scheme name* like HTTP, FTP, mailto, etc., followed by a colon, double slashes, and then the actual identifier. A website address is an example of an URI that most people are familiar with. For example, `http://www.abo.fi`. This kind of URI, describes a location, and will from now on be called the *namespace*.

If we wanted to use a URI to describe the object in the earlier example the URI would look like `http://www.abo.fi/#blue`. Here the beginning of the URI consists of the namespace followed by a # character. After this character the actual value for the object will reside. Even if this URI looks like something one could access with the HTTP protocol, this is not always feasible. A URI can be a reference to an existing resource but this is not required.

There are two possible ways for a subject in RDF to be represented. It is either a URI or a *blank node*, both of which denote the resource. If the subject is a blank node it is an anonymous node and it cannot be identified from the RDF statement. The predicate is also represented by a URI. The object can be represented by a URI, a blank node or a data type string. Data types are used to represent values such as integers, floating point numbers, Booleans and dates. The data type standard used by RDF is the XML-SCHEMA2 recommended by the World Wide Web Consortium[3]. Data types will be described in greater

[3]The World Wide Web Consortium (W3C) is the main international standards organization for the World Wide Web

detail in Section 2.5.4.

2.3 Ontology

The reason for describing information in RDF is that the data can be understood by both humans and computers. Additionally if we use an *ontology* with our data we can expand, share and control our information. An ontology is a way of defining rules for structuring our information. A computer can draw logical conclusion using these rules and create metadata which can be used to expand our data beyond its original scope.

For example, suppose we have a number of databases containing medical information. If these databases share and publish the same underlying ontology of the concepts they are based on, then computer agents can extract and aggregate information from these different databases. The agents can then use this aggregated information to merge the databases. This enables, for example, searching all databases by performing only one search[5].

For an example, with reasoning, the *friend of a friend*[6] ontology is suitable. This ontology is used to describe relationships between people. If we have an expression saying `"John- hasParent-Emma"`, and `"Emma-hasParent-Eve"`, then with the friend of a friend ontology rules a computer agent can infer that `"John-hasGrandparent- Eve"`. For this to work the ontology only needs to know that a parent of a parent is the same as a grandparent.

2.3.1 Structure of an ontology

Class

The most important building block of an ontology is the *class*. Classes can be seen as abstract groups, sets, or collections of objects. Classes describe concepts in the domain, for example a class of `wines` represents all wines. An ontology is built as a tree structure where *subclasses* are classes found below and *superclasses* are found above in the structure. Subclasses represent concepts that are more specific than the *superclass* in the tree structure. For example, we can divide the class of all `wines` into subclasses `red`, `white` and `rosé wines`. Alternatively, we can divide all wines into `sparkling` and `non-sparkling` wines. A superclass for the class `wine` is for example the class `Beverage`. The special `Thing` class is also a superclass and it will be described in greater detail later in this section. To improve readability, classes are written with the first letter capitalized in this thesis.

Instance or individual

An *instance* or *individual* is an actual member of a class. These may include concrete objects such as vehicles, mobile phones, persons or planets. An individual can also be an abstract individual such as a number or text string. Strictly speaking, an ontology need not include any individuals, but one of the general purposes of an ontology is to provide a means of classifying individuals, even if those individuals are not explicitly part of the ontology. In the example seen in Figure 2.3 the class `red wine` has the specific wine `Château Lafite Rothschild` as an individual.

Properties

To describe information about an instance of a class we use *properties*. Properties can relate instances to other things, typically aspects or other instances. A property for the instance `Château Lafite Rothschild` can for example be that it has `countryOfOrigin` with the value `France`. Another property can be `bottleSize` with the values `0,75l` and `1l`. In an ontology it is possible to limit the values that a property can hold like limiting the property `bottleSize` to number values only. This will be described in greater detail in Section 2.3.1. To simplify separating classes from properties the first letter of a property is not capitalized in this thesis.

Relationships

A relationship is a type of property describing an instance and how it is connected to other instances. Typical relationships are `isSubclassOf`, `partOf`, or `memberOf`. All relationships have an inverse relationship which describes the same information in reverse. Inverses to the previous examples are `isSuperclassOf`, `hasPart` and `hasMember`. Much of the power of ontologies comes from the ability to describe relationships. Together, the set of relations describes the semantics of the domain. In our example describing wines we had a property `countryOfOrigin` = `France`. `France` could be a text string but in a proper ontology it would be an instance of the class `Country`. A relationship property for the instance `France` is for example `partOf` = `Europe` with the inverse

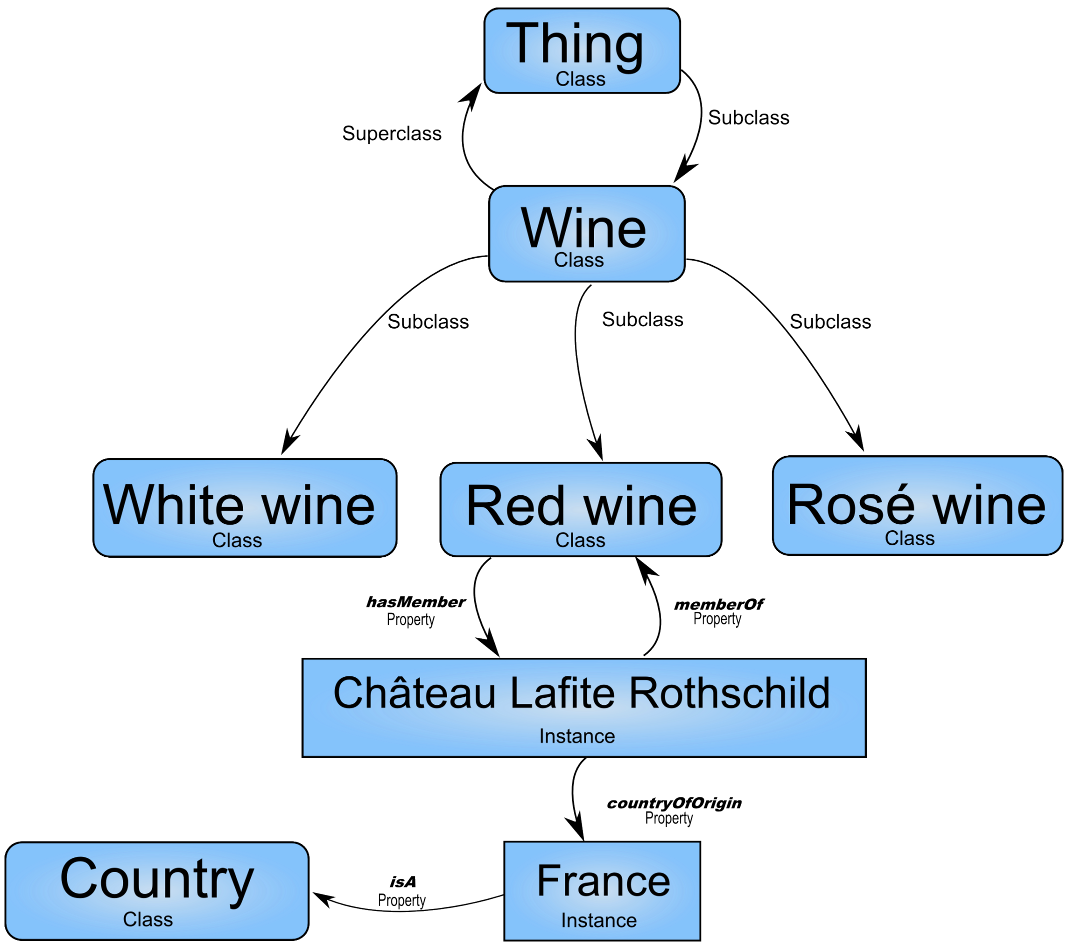


Figure 2.3: Structure of an ontology and an instance with a property

`Europe-hasPart-France`. Now the ontology knows that France is part of Europe. This means that the wine `Château Lafite Rothschild` is found by a search looking for French wines as well as a search looking for European wines.

Thing: the root class and the opposite Nothing class

In Figure 2.3 it is seen that the first class is called `Thing`. In fact `Thing` is always the first class in an ontology. This is needed so that everything described can be traced back to the same root regardless of what the ontology is used for. The `Thing` class also has an inverse class called `Nothing` which states that if something is not a thing it must be nothing. In a valid ontology the class `Nothing` must always be empty. If it is not empty the ontology is broken.

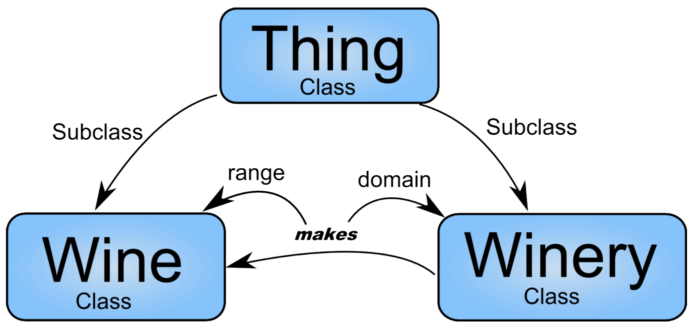


Figure 2.4: Structure of an ontology with a property "makes" illustrating domain and range.

Range and domain of a property

Each property can have a defined domain and range. The *range* of an instance means the allowed classes, or values, for the properties. The *domain* can be seen as the opposite, i.e. the classes that a property is allowed to describe. To illustrate this, an extended version of the wine example is used. In Figure 2.4 the extended example can be seen. In this example there are two separate classes, `Wine` and `Winery`, that have a relationship property linking them together. The `makes` property describes what the winery produces so the *range* of the `makes` property is `Wine`. Only a winery is allowed to make wine in this ontology so therefore the *domain* of the `makes` property is `Winery`

Ontology summarized

In practical terms, developing an ontology includes:

- defining classes in the ontology.
- arranging the classes in a hierarchy.
- defining properties and describing allowed values for these properties.
- creating instances as members of a class and defining properties for them.

In this section the following parts of an ontology have been described:

- **Classes**: sets, collections, concepts, types of objects, or kinds of things. Classes are hierarchical and form a tree structure.
- **Individuals**: instances or objects (the basic or "ground level" objects).

- **Properties**: aspects, properties, features, characteristics, or parameters that objects can have.
- **Relationships**: ways in which classes and individuals can be related to one another.
- **Thing** class: The class of all classes. Every class must be a subclass of Thing for the ontology to be valid.
- Simple **restrictions**: formally stated descriptions of what must be true in order for some assertion to be accepted as input.

The restrictions presented in this section (domain and range) are only the most basic of restrictions. In Section 2.5 the OWL ontology language will be presented and more restrictions will be covered.

2.4 Describing RDF graphs with XML representation

To describe RDF graphs and ontologies some kind of format which machines can read is needed. There are many possible formats available but the XM-L/RDF representation is by far the most popular, and therefore it has been chosen for this thesis.

XML (Extensible Markup Language) is an extensible specification designed for creating general purpose languages to store and share information. An XML file contains information stored between tags, as `<tag>`, which form a tree structure. Each tag is expected to have an end tag, `</tag>`. There are not many other syntactical requirements on an XML file except for that the file must have exactly one first tag, also called the root element. This root element tells us what type of information the XML file holds, and for the purpose of this thesis with XML/RDF, the tag is `<rdf>` or something similar. Before the root tag there can be a tag describing which version of XML and what encoding is used in the file. An example of such a tag is `<?xml version="1.0" encoding="UTF-8" ?>`.

To visualize the above described technologies, look at the following example using the data from Table 2.1. We have information about the country of origin and the type of grape in two popular wines. These wines are sold in an imaginary store called "The Wine Store".

Name	***Country***	***Grape***
Château La Couspaude	France	Merlot
Eileen Hardy Shiraz	Australia	Shiraz

Table 2.1: Two wines sold in "‘The Wine Store"’ and some information about them

Subject	http://www.winestore.fake/wine/**Château La Couspaude**
Predicate	http://www.winestore.fake/wine#**country**
Object	**"France"**
Subject	http://www.winestore.fake/wine/**Château La Couspaude**
Predicate	http://www.winestore.fake/wine#**grape**
Object	**"Merlot"**
Subject	http://www.winestore.fake/wine/**Eileen Hardy Shiraz**
Predicate	http://www.winestore.fake/wine#**country**
Object	**"Australia"**
Subject	http://www.winestore.fake/wine/**Eileen Hardy Shiraz**
Predicate	http://www.winestore.fake/wine#**grape**
Object	**"Shiraz"**

Table 2.2: Example RDF triples with namespaces

In Table 2.2 this information is described as RDF triples with their unique URIs. The namespace of the store is "*http://www.winestore.fake*" but in the rest of this thesis all namespaces will be generalized and shortened to `ns`.

The same information described in machine readable XML format can be seen in Listing 2.1.

2.5 Use of the Web Ontology Language to encode ontologies

In order to encode an ontology and its rules an ontology language is needed. There are a number of languages available, both proprietary and standard based. In this thesis the *Web Ontology Language* (OWL) family[7] is used. The OWL languages are built upon their predecessor *RDF Schema* which provides basic capabilities to describe ontologies. Both RDF Schema and the OWL languages are built on RDF so therefore they are the natural choice for this thesis. These languages are also enforced by the World Wide Web

Listing 2.1: Four triples written in XML format

```xml
<?xml version="1.0"?>

<rdf:RDF
xmlns:wine="http://www.winestore.fake/wine#">

 <rdf:Description
 rdf:about="http://www.winestore.fake/wine/Château La Couspaude">
   <wine:name>Château La Couspaude</wine:name>
   <wine:grape>Merlot</wine:grape>
   <wine:country>France</wine:country>
 </rdf:Description>

 <rdf:Description
 rdf:about="http://www.winestore.fake/wine/Eileen Hardy Shiraz">
   <wine:name>Eileen Hardy Shiraz</wine:name>
   <wine:grape>Shiraz</wine:grape>
   <wine:country>Australia</wine:country>
 </rdf:Description>
</rdf:RDF>
```

Consortium.

As described earlier in Section 2.3 an ontology consist of a set of axioms which place constraints on the data stored. These axioms also limit the permitted relationships between parts of the information. Earlier we had an instance `Château Lafite Rothschild` with the property `countryOfOrigin` holding the value `France`. OWL enables us to, for example, set the range restriction that the `countryOfOrigin` property can only hold values that are of class `country`. This results in that the ontology needs to have information about all countries in the world but on the other hand gives us the benefit of always getting a valid value for the `countryOfOrigin` property.

The OWL ontology languages are divided into three dialects that are tailored to suit different uses. The simplest dialect is OWL Lite followed by the similar but more complex OWL DL. After OWL DL there is the OWL Full dialect which is meant for very complex ontologies. All dialects have a different degree of restrictions and each of these dialects is an extension of their simpler dialect. This means that the following relations hold[7]:

- An OWL Lite ontology is a legal OWL DL ontology.
- An OWL DL ontology is a legal OWL Full ontology.

- A valid conclusion in OWL Lite is also valid in OWL DL.
- A valid conclusion in OWL DL is also valid in OWL Full.

2.5.1 OWL Lite

The OWL Lite dialect is meant for very simple use. One of the most basic things to restrict in an ontology is how many values a property can hold. This sort of restriction is a *cardinality constraint*. For example, a person must have at least one first name but it is not unusual to have more than one. In this case the cardinality is *"one or more"*. This becomes a problem with OWL Lite as it only supports limited cardinality. Limited in this context means that only the values 0 or 1 are supported. More complex cardinality is needed in this thesis so OWL Lite is therefore not suitable.

2.5.2 OWL DL

The OWL DL[4] dialect aims to achieve maximum expressiveness while still retaining computational completeness. This means that a reasoner agent (described in detail in Section 4.2) must be able to compute every conclusion using a practical reasoning algorithm. As reasoning is to be supported by the tool OWL DL becomes the perfect choice for this thesis.

A summary of the building blocks or *constructs* is given in Appendix A. The goal of the Appendix is to give an idea of what kind of ontologies can be created with the OWL DL language.

2.5.3 OWL Full

OWL Full is designed to be very flexible when compared to OWL Lite or DL. For example, a class can be treated simultaneously as a set of individuals or as an individual in its own right. This is not permitted in OWL DL. As a result of this flexibility, in OWL Full, it becomes impossible to perform reasoning on an OWL Full ontology. Therefore OWL Full is not suitable for this thesis.

2.5.4 Allowed data type values in OWL

Properties are distinguished according to whether they relate individuals to individuals (object properties) or individuals to data types (data type prop-

[4]where DL stands for "Description Logic"

erties). Data type properties may range over RDF literals, or simple types, defined in accordance with XML Schema data types[8][9].

OWL uses most of the built-in XML Schema data types and in Table 2.3 these can be seen.

string	normalizedString	boolean	
decimal	float	double	
integer	nonNegativeInteger	positiveInteger	
nonPositiveInteger	negativeInteger		
long	int	short	byte
unsignedLong	unsignedInt	unsignedShort	unsignedByte
hexBinary	base64Binary		
dateTime	time	date	gYearMonth
gYear	gMonthDay	gDay	gMonth
anyURI	token	language	
NMTOKEN	Name	NCName	

Table 2.3: Allowed XML Schema data types to be used with OWL

2.6 Architecture of an application

When creating the tool in this thesis a philosophical question arose on numerous occasions. This question was *"What is an application?"*. There is no unambiguous definition of what an application is but traditionally an application is seen as a program running on a computer or device. A word processor, on a personal computer, or a calendar on a mobile phone are examples of traditional, monolithic and single purpose applications. The main point being that traditionally an application runs on one single device. Data can be retrieved from a separate server but the main functionality is created on one device. This leads to difficulties in expanding the application or integrating it with other applications.

In this thesis the approach is somewhat different. The goal is to be able to have applications consisting of more than one device. The approach in this thesis, to enable this, is based on the *actor model* proposed by G. Agha[10] and H. Baker[11]. In the actor model all functions of an application are divided into so called *agents*. An agent is seen as an entity that performs only one specific simple task. The goal is to divide all functionality into as small parts as possible to achieve maximum modularity. This means that examples of

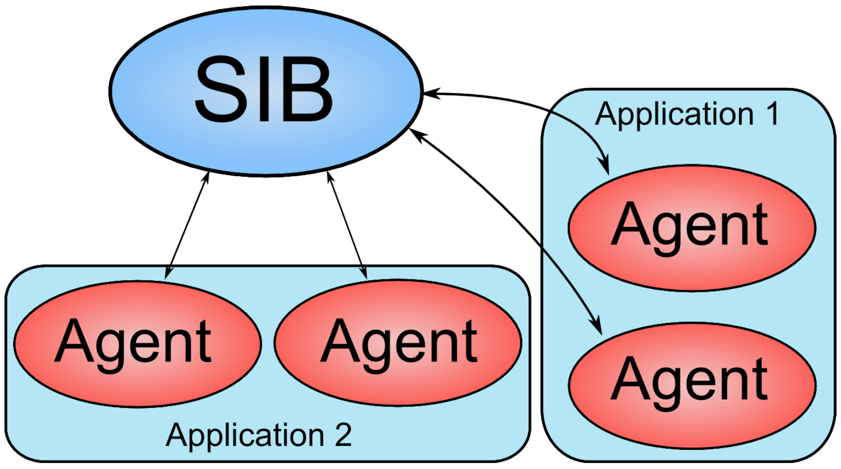


Figure 2.5: Application emerging from used agents

tasks in this context range from reading, writing or modifying information in the database to more complex task that cannot be modularized. By using this approach the actual application emerges from the combination of agents used. This is illustrated in Figure 2.5.

The benefits with this approach are that agents can run on separate devices and as long as they share a common SIB they can be part of the same application. With this approach great flexibility and modularity is also achieved. Agents can of course be reused and by adding or removing agents in an application the functionality will change. Reasoner agents can also be inserted into the application to validate and extend the data.

In Figure 2.6 a traditional email client is presented. If this same application was created using the agent model it would consist of, for example, the following agents:

- Agent for displaying received emails as a list. The agent displays information like size, topic and date about the message.
- Agent for displaying folder structure and the number of unread messages.
- Agent for creating and sending a new message.
- Agent for replying to selected message.

The granularity of the agent structure may vary so that the same application could also be formed from much smaller agents. For example, the agent displaying the list of emails could be broken down to separate agents for each piece of information.

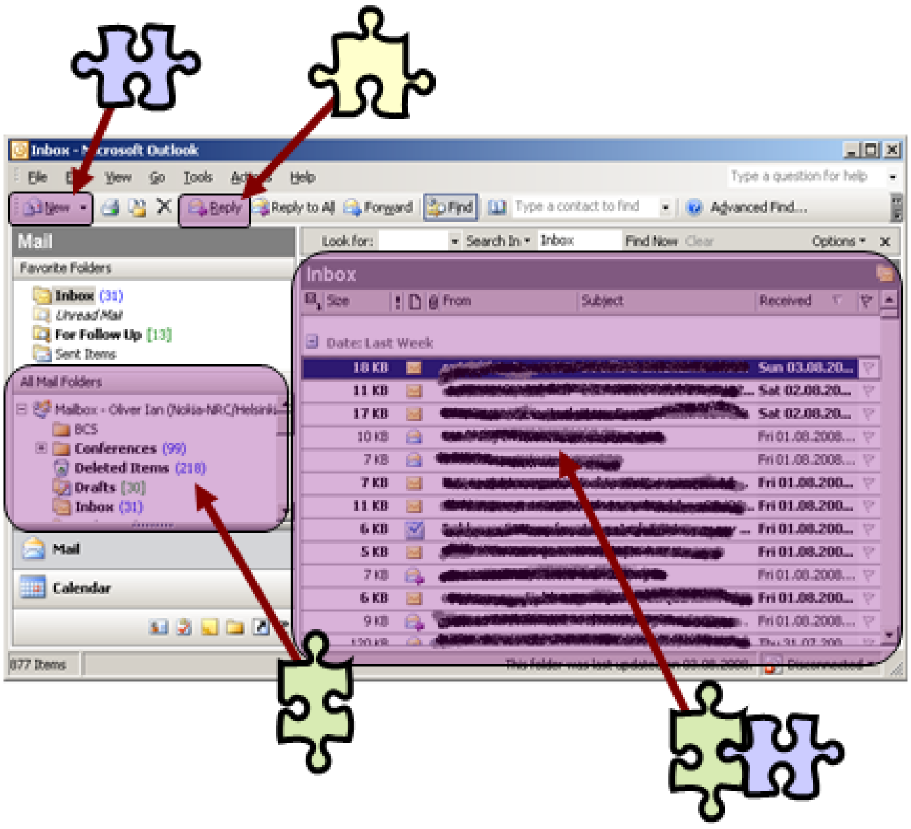


Figure 2.6: Traditional email client application using agents

2.7 Limitations with the SIB and agent model

The architecture described in the previous section enables a variety of applications but it is however not perfect. As agents are allowed to join and leave the smartspace there are a number of issues that arise.

2.7.1 Security

One major limitation with the current SIB, from a practical point of view, is security. The current version of the SIB has no enforcement of security which is a vital part in a system where any agent can write, modify and remove information. This fact also means that an agent can put the ontology in an inconsistent state which can lead to that the whole system becomes unstable. It can be compared to a file system on a computer where anyone is allowed to access the system and create chaos.

An initial solution to this is to only allow trusted agents to join the SIB. If an agent is not recognized and trusted it would be blocked until it presented the needed credentials. Another way to improve security is to limit parts of the ontology in a number of ways. The simplest way would be to only allow the agent that wrote the information to access it. Other agents would only have *read* rights to this information. This restriction would however exclude applications where multi-agent data updating would be the backbone of the application.

2.7.2 The transaction problem

A transaction is a group of write and read operations which can be rolled back in case any of the operations fail. One real life use case of a transaction is transferring money from one account to another. In this scenario if either the withdrawal or the deposit operation fails, both will be rolled back and the accounts will be returned to their initial states.

When working within a given OWL ontology, the assertions in form of triples must form a valid whole. In an example ontology there is a `MaxCardinality` restriction on a property. The restriction is set to four property instances (`MaxCardinality = 4`). Then lets imagine that two agents want to add a property instance. They query the current instances from the back-end and the SIB returns information about three instances in existence. Then both agents write their respective new property instance, and thus set the ontology in an inconsistent state.

To avoid this situation, both the read and the write operation should be contained in a single transaction, which then can be rolled back on errors. The need for transaction atomicity is mainly needed to be able to guarantee consistency of the database, when several concurrent agents are storing information in the SIB.

At the time of writing this thesis, the current version of the SIB only has limited transaction support. Therefore transaction atomicity is impossible to guarantee at this stage.

2.7.3 The information lifetime problem

In a distributed computer environment it is not possible to know which entity needs what information. As many agents may be using the same data in the SIB it becomes important that nothing is deleted that another agent might need. For example, an agent can be interested in old calendar information and therefore agents may only delete data that they know is incorrect.

If an agent shuts down, or leaves the smartspace, the data it produced for the SIB may not be removed as another agent may be dependent on this data. This data may, however, be outdated at this stage as the agent that produced it is no longer available to update it. This issue creates a need for a sophisticated method for information versioning and clearing data from the SIB. The current version of the SIB lacks this which leads to some limitations in how complex smartspaces can to be implemented.

2.8 Summary

The aim of this chapter was to cover technologies that the tool described in the next chapters utilizes. Details were presented of how data can be represented in such a way that both a computer agent and a human can understand. This is achieved by depicting information as triples that form readable sentences. Further we used an ontology and the OWL ontology language to restrict, and place rules, on how data is used.

An architecture where many devices can contribute to an application was shown. This architecture uses a information broker, or SIB, as a hub that stores information and manages connections between devices. The actual application emerges from the combination of agents and the operations performed on the information provided from the devices.

Lastly some limitations, and issues to resolve before practical implementation, were discussed. Most of these limitations are resolved by developing the SIB. As this is not a central part of this thesis, those issues will not be investigated further.

Chapter 3

Calendar Application Example

This chapter explains the calendar application which was chosen for the practical part of this work. Working with calendars is suitable because they can be seen as basic devices performing for example read, write and subscribe operations. A calendar also has a great potential to create new functionality when connected to various other devices. The goal of the calendar application is to showcase the technologies and the agent-based model, described in chapter 2, and how it can be applied to a traditional application like synchronizing calendars. Two calendar standards, *iCal* and *Google Calendar*, are used in the example in order to bring forth the problem with interoperability between different devices or standards.

The functionality wanted from the application is synchronizing two different calendar standards (iCal and Google Calendar). After the synchronization both calendars should have the same event data. The functionality is separated into agents in order to achieve modularity and other benefits as described earlier in Figure 1.1. The selected architecture is to use four agents to achieve the desired functionality. The four agents needed are:

- Read an iCal standard file and write information to the SIB.
- Read an Google Calendar standard feed and write information to the SIB.
- Read information from the SIB and write to a iCal standard file.
- Read information from the SIB and write to a Google Calendar standard feed.

In Figure 3.1 an illustration of the needed agents can be seen (agents represented by ellipses).

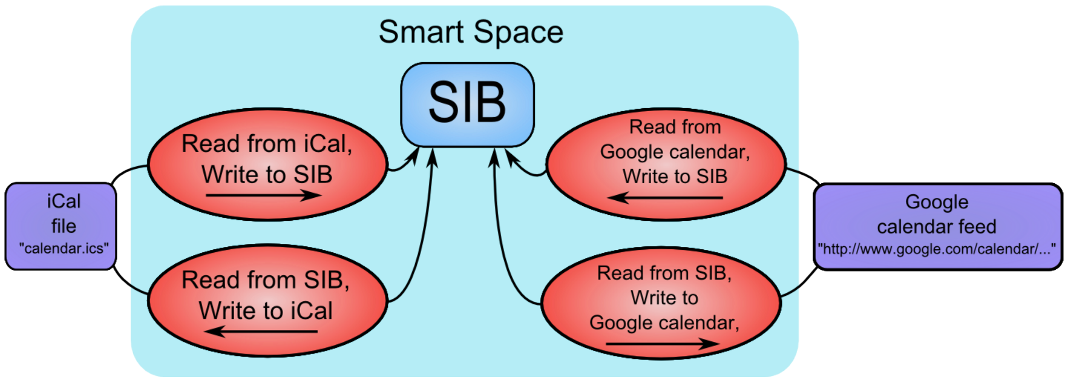

Smart Space
SIB
Read from iCal, Write to SIB
Read from SIB, Write to iCal
Read from Google calendar, Write to SIB
Read from SIB, Write to Google calendar,
iCal file "calendar.ics"
Google calendar feed "http://www.google.com/calendar/..."

Figure 3.1: Agent architecture in calendar synchronizing application

Every agent independently handles connecting and disconnection to the SIB. This agent-based model leaves us the option of easily adding more functionality to the application. If we wanted to synchronize with another standard, for example, *Microsoft Outlook*, we only need to create the agents to handle the read and write of the Outlook data with the SIB.

3.1 The Calendar Ontology

The ontology used for the calendar is in OWL DL format, as disjoint classes are needed. Disjoint classes means that if two classes are defined as disjoint an individual cannot be a member of both classes (see Appendix A for more details). For example it is not possible for an event to also be a calendar. The ontology is detailed in Figure 3.2.

The root class `Thing` has the subclasses `Calendar` and `Event`. The `memberOf` relationship property enables individuals of the `Event` class to belong to an individual of the `Calendar` class (`Event-memberOf-Calendar`). The `memberOf` property has the inverse property `hasMembers` (`Calendar-hasMembers-Event`).

Both the `Event` class and the `Calendar` class must have exactly one *Universally Unique Identifier* (UUID). This is shown by the cardinality constraint [1...1] (from 1 to 1). This identifier is used, together with the namespace, as an ID number to separate instances from each other. This identifier is in fact a long (128-bit) number but it has been shortened is this work for improved readability.

A title property is also needed for every instance of the `Event` and `Calendar` class for the ontology to be valid. The `hasTitle` property is defined to be a text `string` data type. The `wasModifiedLast` and `hasDtStart` (start time of event) are like the `hasTitle` property except for that they are of type

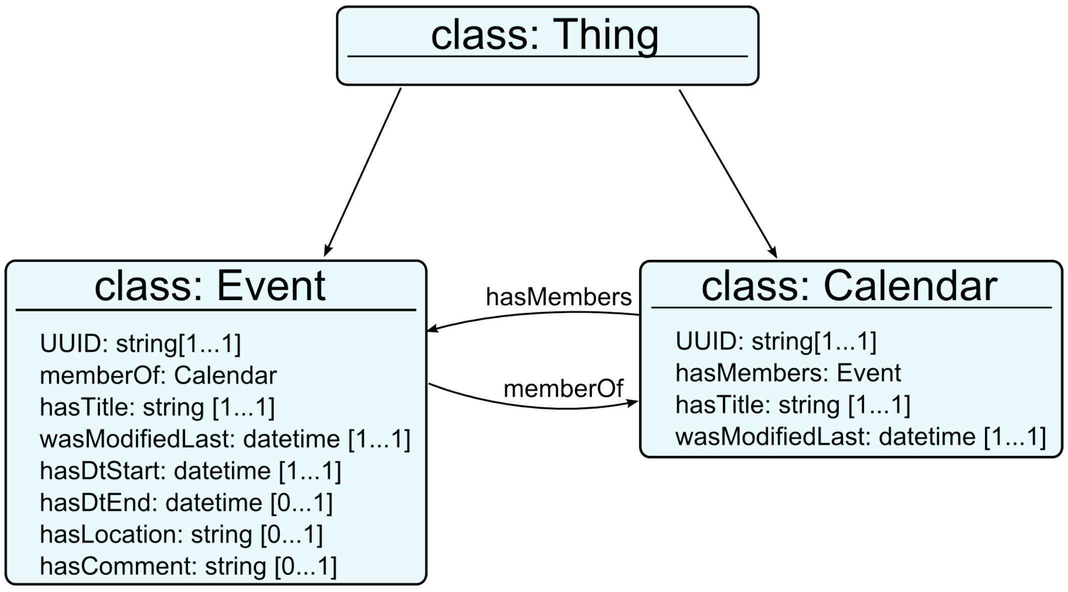


Figure 3.2: Ontology used in calendar application

`datetime` (a date and a time of day value). Furthermore, the `Event` class has a few voluntary properties (`hasDtEnd`, `hasLocation` and `hasComment`) with cardinality [0...1] (empty or 1 value).

3.1.1 Example data using the calendar ontology

In Figure 3.3 the calendar ontology is used to represent some calendar information. The `Calendar` class has two instances, `JonsCalendar` and `BobsCalendar`. The `Event` instance `ThesisMeeting` belongs to `JonsCalenadar` and the `LunchMeeting` instance belongs to both `JonsCalenadar` and `BobsCalenadar`. When implemented in this fashion, with the `LunchMeeting` instance shared, any changes that are made to the event will be seen by both Jon and Bob. If this functionality was not desired then both calendars need their own `LunchMeeting` event instance with different UUIDs.

Figure 3.3 also illustrates, with the `member` relationship, a feature that is difficult to map into a object oriented programming language, in a straightforward way. Due to issues with multiple inheritance and dependencies between classes relationships, in an OWL graph, pose a challenge.

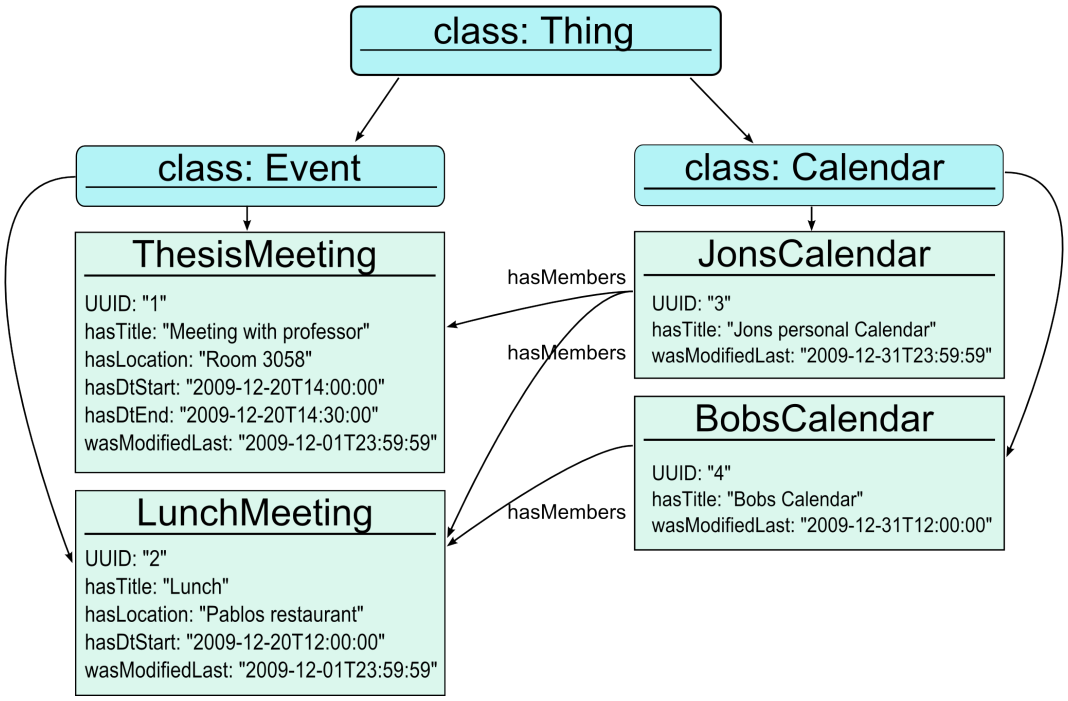

class: Thing
class: Event
class: Calendar
ThesisMeeting
UUID: "1"
hasTitle: "Meeting with professor"
hasLocation: "Room 3058"
hasDtStart: "2009-12-20T14:00:00"
hasDtEnd: "2009-12-20T14:30:00"
wasModifiedLast: "2009-12-01T23:59:59"
LunchMeeting
UUID: "2"
hasTitle: "Lunch"
hasLocation: "Pablos restaurant"
hasDtStart: "2009-12-20T12:00:00"
wasModifiedLast: "2009-12-01T23:59:59"
JonsCalendar
UUID: "3"
hasTitle: "Jons personal Calendar"
wasModifiedLast: "2009-12-31T23:59:59"
BobsCalendar
UUID: "4"
hasTitle: "Bobs Calendar"
wasModifiedLast: "2009-12-31T12:00:00"
hasMembers
hasMembers
hasMembers

Figure 3.3: Example instances using calendar application ontology

3.2 First implementation of the calendar synchronizing application

The first implementation of a calendar synchronizing application using the agent based model was written as fairly "low-level" code. What this means is illustrated in the next section with code listings showing parts of one of the four agents in the synchronizing application.

3.2.1 Agent: Write iCal data to SIB

The functionality of the agent described is to write event data from an iCal file to the SIB. The first task of any agent is to create a connection to the SIB and the code for this is seen in Listing 3.1.

Firstly a node is created on line three and four. This node represents the connection to the SIB. The nodes UUID (or `nodename`) is the unique ID number which is used to identify the agent. On line six a local smartspace object is created and on line seven the `SS_User_TCP_Discovery_- IPADDR()` method gathers information from the user about the name of the SIB , the IP address and port to use when establishing a connection to the smartspace. The last line joins the node to the smartspace.

Listing 3.1: Find and join a SIB

```
def joinSIB(self):

    self.nodename = str(uuid.uuid4())
    self.theNode = ParticipantNode(self.nodename)

    dm = SS_User_TCP_Discovery_IPADDR()
    self.theSmartSpace = dm.discover()

    self.theNode.join(self.theSmartSpace)
```

Listing 3.2: Defining simplified calendar ontology

```
1 def defineCalendarOntology(self):
2     t = RDFTransactionList()
3     ns="http://www.it.abo.fi/calendarsync#"
4
5     t.add_Class(ns+"Calendar")
6     t.add_Class(ns+"Event")
7
8     p = self.theNode.CreateInsertTransaction(self.theSmartSpace)
9     p.send( t.get() )
```

Next, in Listing 3.2 the ontology is defined piece by piece. In the first application the ontology is greatly simplified to only the two classes. No relationships or restrictions are used. First, a transaction list, that will contain the ontology, is created. This list will be sent as a whole when completed. The transaction list is made out of RDF triples and as every triple is to contain the namespace this is defined on line three.

On line five and six the classes of the ontology are added to the transaction list. For the `Calendar` class the `add_Class` method creates the triple:

- **Subject**: `http://www.it.abo.fi/calendarsync#Calendar`
- **Predicate**: `http://www.w3.org/1999/02/22-rdf-syntax-ns#type`
- **Object**: `http://www.w3.org/2000/01/rdf-schema#Class`

Line eight prepares an insert transaction operation and defines the SIB while line nine performs the insert operation using the transaction list.

Listing 3.3: Adding an event to the SIB

```
1  def createEvent(self, title, start, end, lastMod):
2      t = RDFTransactionList()
3      ns="http://www.it.abo.fi/calendarsync#"
4      u1 = ns+str(uuid.uuid4())
5
6      t.setType(u1,ns+"Event")
7
8      t.add_literal(u1, ns+"hasTitle", title)
9      t.add_literal(u1, ns+"hasStartTime", start)
10     t.add_literal(u1, ns+"hasEndTime", end)
11     t.add_literal(u1, ns+"wasModifiedLast", lastMod)
12
13     p1 = self.theNode.CreateInsertTransaction(self.theSmartSpace)
14     p1.send( t.get() )
```

How an event, and some event information, is added to the SIBs RDF graph is seen in Listing 3.3. The `title, start, end` and `lastMod` variables have been fetched from the iCal file and contain the event information.

To start with, the transaction list is created and the events namespace and UUID are defined like in the previous examples. On line six the event is defined to belong to the class `Event`. Next the `add_literal` method is used to create the triples describing the properties of the event as literals.

To conclude, the insert transaction operation is created and sent to the SIB.

3.2.2 Issues with the first implementation

The first implementation of the agent based application had many shortcomings of which some can be seen from the above listings.

The most severe issue was that no relationships or restrictions were enforced. All the properties of the event that were added were added as strings. No, for example, data type or cardinality checking was done which lead to that there were no longer any guarantee that the ontology in the SIB was valid. Due to this most the benefit of using an ontology language is lost.

Also the coding style presented was not very intuitive or simple. This is a major problem as the goal is to target the bedroom coder. To solve these issues it was decided to introduce an object oriented approach and an API to work with the objects. It was also decided to simplify agent development, by hiding as much of the cumbersome tasks as possible, into a framework package.

3.3 Summary

The aim of this chapter was to describe the calendar synchronizing application. This application is used to illustrate an example of what the bedroom coder can use the framework to create. A calendar ontology was presented as well as data using this ontology. An architecture of four agents was chosen to give the desired functionality of calendar synchronizing. After this a first version of one of the agents using the SIB was covered. It was concluded that this implementation could not make use of the full functionality of an ontology language and that it was far from easy to create an understand.

Chapter 4

Python Framework Generator

This chapter describes the generator mentioned in the earlier chapters. The goal is to provide the bedroom coder with an, easy to use, *toolbox* for working with information in a specific ontology. The toolbox is also supposed to handle all functionality concerning connectivity between agents and the SIB.

The generator is a tool that takes, as input, an OWL ontology in the form of an RDF/XML formatted `.owl` file. The generator then translates this information to an object oriented Python code framework. This framework contains an API to assist working with information from devices as well as functionality to enforce the rules of the provided ontology.

Introducing the object oriented thinking enables the bedroom coder to see the information in the ontology as objects. For example, when using the calendar ontology a Python `Event` object is very close to the OWL class `Event`. The Python object has variables for all the properties in the OWL class; `name, startTime, endTime` and so forth. For a complete list see Figure 3.2. These variables can be modified as any Python variables. This is easier compared to directly modifying triples in the SIB. The framework also provides a platform for information retrieval with, for example, intuitive *get methods*. Without these methods every query would have to be defined with regular expressions as explained in Section 2.1.1.

The next sections discuss the architecture of the generator and give a brief description of its parts. The generator consists of a number of modules that gradually create an object oriented code model which is then used to create the code framework. In Figure 4.1 an overview of the process is shown.

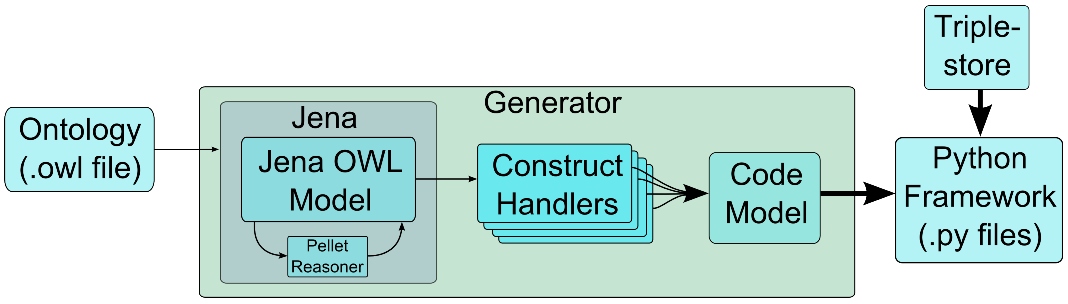


Figure 4.1: Overview of the code generators architecture

4.1 Jena

To be able to create an ontology specific Python framework from an OWL ontology directly is challenging. A parser can be used to help work with, and navigate RDF graphs and in this work the *Jena framework*[12] is used for this.

Jena provides a multitude of tools for building Semantic Web applications in the *Java programming language*. It has been specifically designed for Java which means that it takes advantage of, and enables, features from the programming language. Apart from the parser Jena also provides other useful functions. It has support for several reasoners and it creates an *RDF model* of the ontology which aids the API generation in this work.

Jena starts by reading the ontology, from the `.owl` file, and uses it to create the RDF model. The model contains the classes, properties and restrictions of the ontology organized as a tree structure. The model can then be used by a number of modules which each extend the model. For this work the *Pellet reasoner*, described in Section 4.2, is the most relevant.

The model is implemented in such a way that it understands anonymous resources, i.e. resources do not need to have a known URI. Jena internally tracks the identity of resources so that it can determine if two resources are in fact the same resource. The Jena framework also understands an attempt to add an already existing resource to the RDF graph. If this is the case the operation will be ignored in order to avoid duplicates.

A query API module is also provided in the Jena framework. This query API is quite flexible in the sense that it provides three different types of responses to queries. A query can return either a new graph, which is a sub-graph of the original, an iterator containing all the statements matching the query or a table of values matching the query.

4.2 Pellet Reasoner

Pellet is an open source reasoner developed to work with OWL DL. It has extensive support for debugging ontologies as well as reasoning with individuals. The Pellet reasoner contains a module created to work with a Jena code model. It starts by reading the Jena model; it then performs reasoning on it and writes back to the Jena model. The actual reasoning is done by performing consistency, concept satisfiability, classification, realization and data type checking[13].

Consistency checking ensures that the ontology does not contain any contradictory facts. This check steps through every node in the ontology and if it finds something that can be inferred it adds this knowledge to the ontology. For example, a property has an `inverseProperty` it makes sure that the corresponding property is defined. An example of an inverse property is the `hasChild` property with the `inverseProperty` `hasParent` or `childOf`.

The output of this checking is the word "*consistent*", "*inconsistent*" or "*unknown*". While this does not allow us to do anything valuable with the ontology it is still an important task to perform due to the fact that the following checks rely on the ontology to be consistent. The formal definition of ontology consistency can be seen in the *OWL Abstract Syntax & Semantics*[14] document by W3C.

Concept satisfiability checking sees if it is possible for a class to have any instance. If the check finds an unsatisfiable class then defining an instance of this class will cause the whole ontology to be inconsistent.

Classification checking computes all the subclass relations for all classes and creates a class hierarchy. This hierarchy ensures that the answer to queries, getting all, or only the direct, subclasses of a class, is complete and correct.

Realization checking computes the direct type of every individual, i.e. finds the most specific class that the individual belongs to. This ensures that the answer to a query, finding all the classes an individual is a member of, is complete and correct.

Data type checking checks all defined data types in the ontology. It does this by checking if the intersection of data types is consistent or not. What this means is that an intersection is inconsistent if they have no data values in common, e.g. the intersection of `xsd:positiveInteger` and `xsd:negativeInteger` is empty.

4.2.1 Summary and reasoning limitations

After the reasoner has stepped through the model successfully it is guaranteed that the ontology is valid, i.e., the classes, properties and restrictions are constructed properly, all properties have their correct counterparts and the properties data type restrictions are feasible.

Reasoning executed in this fashion creates one drawback on the Python model of the OWL ontology. The problem is that the reasoning is only performed before the actual Python framework is used. In theory it is possible for the bedroom coder to create an application that would need runtime reasoning to ensure that a complicated ontology stays consistent. To not perform runtime reasoning is, however, acceptable at this stage as the goal is not to convert a full model of OWL into Python, but to create a platform for application development.

If runtime reasoning was vital two potential solutions have been identified. Firstly, runtime reasoning could be performed by the agent within the device. This however is not ideal due to the fact that the agent needs knowledge of the ontology that resides in the SIB to be able to perform reasoning. This would need a great deal of bandwidth and storage space. As the goal is to allow agents to run on very limited devices this becomes impractical. The other way to stop incorrect usage of the ontology at runtime is to let the SIB handle reasoning. This is the reasonable choice as the SIB has all the information needed as well as the performance to fulfill such a task. Runtime reasoning is however not available in the current version of the SIB.

4.3 Construct Handlers

The actual mapping from an OWL DL ontology to an object oriented model is done by a number of *construct handlers*. The handlers work according to a set of static mappings that generate an object oriented code model. This model is then converted to native Python classes, methods and variable declarations which can be used by the bedroom coder to access data in the SIB. As described above, some OWL constructs need runtime reasoning in order to verify the correctness of the data in the ontology. Therefore the implementations of these constructs is somewhat limited.

The code model is built gradually by the construct handlers as they work through the Jena RDF model. The handler that starts the process is the *class handler*.

4.3.1 Mapping of owl:Class and rdfs:subClassOf

The construct handler for mapping classes is fairly straightforward. An OWL class by the name `event` generates a Python class with the same name (`event`). The `rdfs:subClassOf` property is also added to the model, by the handler, so that the Python class extends its superclasses.

This construct handler also adds the *class variables* and the *class methods* to the model. Class variables and methods are associated with the class so that all the instances of the particular class share them. For example, with the `event` class, the variable `memberCount` which stores the number of instances of the type `event`. A usage example, from the bedroom coder´s point of view, is seen in Listing 4.1. First an instance of the type `Event` is created. On line two a check is performed which proves that the `lunchEvent` is a subclass of `Thing`. On the last line the `memberCount` class variable is used to print the number of instances belonging to the `Event` class.

Listing 4.1: Usage example of an instance of a class

```
1 lunchEvent = Event()
2 assert issubclass(lunchEvent, Thing)
3
4 print "Number of events stored is " , Event.memberCount
```

At this stage the model contains the most basic information about the classes. An illustration of the model is seen in Figure 4.2.

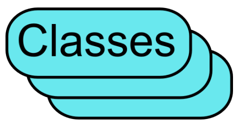


Figure 4.2: Code model after class construct handler

4.3.2 Mapping of properties

Properties are used to state relationships between individuals (`owl:ObjectProperty`) or from individuals to data values (`owl:DatatypeProperty`). The property handler adds the *instance variables* as well as their corresponding *get* and *set* methods to the classes created above.

For the calendar ontology the handler will create all the instance variables, and get and set methods, for the properties that were defined in the ontology (Figure 3.2). For example, for the `Event` class these were `UUID`, `title`, `location`, `comment`, `dtStart`, `dtEnd` and `modifiedLast`.

Listing 4.2: Usage example of a calendar event with a property and the corresponding get and set methods

```
1 location = "Pablos restaurant"
2
3 lunchEvent = Event()
4 lunchEvent.setLocation(location)
5
6 value = lunchEvent.getLocation()
7
8 assert(location == value)
```

A usage example, from the bedroom coder´s point of view, where a location value is added to an event, is seen in Listing 4.2. On line four the information is stored in the variable with the set method. After that the get method can be used to retrieve the information. The last line confirms that the inserted and retrieved information is the same. In a multi-agent system this assertion should be done with caution as another agent could have changed the information. Performed like this, however, is safe, as all the data is local (nothing sent to the SIB at this point).

At this stage the code model contains the main building blocks but it is still very simple (as seen in Figure 4.3). After this however, the restriction and relationship handlers start adding more complexity to the model.

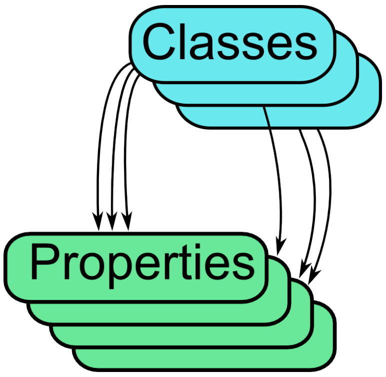


Figure 4.3: Code model after property handler

Syntax of generated methods

Depending on a properties cardinality the syntax used differs somewhat. A popular property in OWL is a property like `hasMember`. If this was translated directly to a Python variable `hasMember` some problems would appear with its get and set methods. This is due to that in OWL the `hasMember` denotes that a certain OWL domain has a member object. In object oriented programming, a direct mapping, e.g. `getHasMember()` could falsely denote that a Boolean return value is expected. Therefore, for example, the "has" -prefix on an OWL property is removed in this thesis (`hasMember` is changed to simply `member`). This way the method name becomes `getMember()` which will return a member object, as expected. In Table 4.1 all the naming conventions are shown.

Syntax	Cardinality	Object oriented example
get	1 or more	getElement(), getElements()
set	1	setElement()
add	more than 1	addElement()
remove	more than 1	removeElement()

Table 4.1: Naming conventions from OWL to object oriented Python

4.3.3 Mapping of cardinality

The cardinality handler is one of many handlers that add restrictions to the model. For example, imagine an ontology with a cardinality constraint on how many events may occur in a calendar. In order to enforce this cardinality class variables are used. Depending on what kind of cardinality is set, a combination of the class variables `eventCardinality`, `eventMaxCardinality` and `eventMinCardinality` is created. The handler also creates `if` statements that check the cardinality when a new event is added to the calendar. If the calendar is full the `if` statement will raise a `CardinalityException`.

In Listing 4.3 an example is presented where three events may belong to a calendar. First the calendar and four separate events are created. After this the events are added to the calendar surrounded by a `try` statement. On line ten an attempt is made to add the fourth event to the calendar. This attempt will trigger the `CardinalityException` which in this case is a simple `print` notification.

Before illustrating how the cardinality handler modifies the model one more handler will be described.

4.3.4 Mapping of owl:inverseOf

Properties have a direction, from domain to range. It is useful to define relations in both directions: persons own cars, cars are owned by persons. The `owl:inverseOf` construct can be used to define such an inverse relation between properties. In order to implement this the construct handler creates an

Listing 4.3: Usage example with a cardinality restriction

```
calendar = Calendar()
event1 = Event()
event2 = Event()
event3 = Event()
event4 = Event()
try:
    calendar.addMember(event1) # Ok
    calendar.addMember(event2) # Ok
    calendar.addMember(event3) # Ok
    calendar.addMember(event4) # Will trigger the cardinality exception
except CardinalityException:
    print "Cardinality exception!"
```

`if` statement for the property that checks if the inverse property is defined. If not, it adds the inverse property to the code model.

A usage example, from the bedroom coder´s perspective, is seen in Listing 4.4. First a calendar and an event is created. On line four the event is added to the calendar. When this is done the `if` statement that was added to the `addMember` method, recognizes that the corresponding property is not set and handles this. The last line checks if the inverse relation was set successfully.

Listing 4.4: Usage example of a property and a corresponding inverse property

```
1 calendar = Calendar()
2 event = Event()
3
4 calendar.addMember(event)  # relates the calendar to event
5
6 # an inverse relation has been automatically created from event
    to calendar, see Listing 4.9.
7 assert containsElement(event.getMemberOf( ), calendar)
```

4.3.5 Construct Handlers summarized

Above some of the construct handlers were described. These construct handlers all contribute to the object oriented code model which will become the Python framework.

Figure 4.4 shows how the handlers assemble the final code model. First the class handler adds the most basic information about classes to the model. Then the properties handler does the same for properties. The properties handler

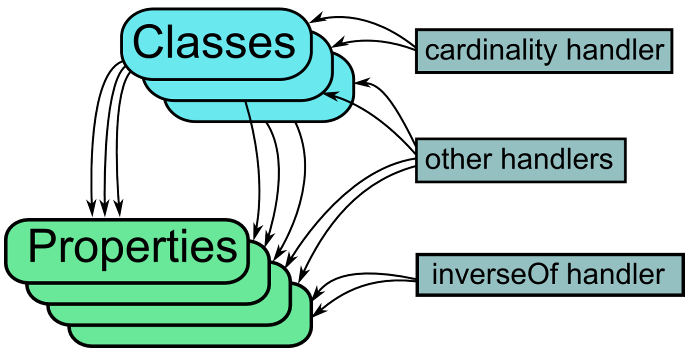


Figure 4.4: Complete code model assembly process

also links the properties to the correct classes. After this, as described in Section 4.3.3, the cardinality handlers adds further information to the classes in order to enforce cardinality. The `inverseOf` handler, described in Section 4.3.4, adds information to the properties within the model. Apart from the described handlers there are also a number of additional handlers that were left out from this thesis. These include handlers for owl:differentFrom, owl:allDifferent, owl:allValuesFrom, owl:equivalentClass, owl:equivalentProperty, owl:hasValue, owl:sameAs, owl:someValuesFrom as well as handlers for property data types. Depending on the qualities of these constructs they modify either the classes or properties of the code model.

This section described some of the construct handlers and how they create the code model. In Section 4.5 the code model will be serialized into the actual Python API.

4.4 Triple Store

When planning the architecture of an agent an issue with updating data was realized. For example, when should a sensor device update its sensor value in the SIB? Information updating can be performed in a number of ways, e.g. on a regular basis with a time interval, when a change in the data has occurred, or when another agent shows interest in the particular piece of data. The goal of this thesis is to allow all these ways of information updating as different agents have different needs. In some applications it might be vital that the information in the SIB is constantly updated. On the other hand, some devices limited bandwidth or battery power can lead to a need for a more economical way of updating data. To tackle this issue the *triple store* is used.

The triple store is basically a local database, within the agent, containing pointers to every piece of local information. When a new object is created in the agent it is automatically added to the local triple store. If an already existing object is changed within the agent it will get tagged in such a way that the triple store knows that the change has occurred. This makes it possible for the triple store to monitor at which state the agent´s data is compared to the SIB.

The triple store handles read and write operations between the agent and the SIB. To enable the different ways of updating information mentioned above, the timing of the writing and reading is left to the bedroom coder. When the bedroom coder wants to update the agent data *toward* the SIB the triple stores `commit` method is run (as seen in Listing 4.5). This method steps through every piece of information in the agent and finds the data that needs to be updated. Then this information is converted from Python objects to RDF triples and sent to the SIB.

Listing 4.5: Writing agent data to SIB

```
    tripleStore.commit()
```

If the agent uses information that might have been modified by another agent the process is similar. The bedroom coder can ensure that the information is up to date before it is locally used by calling the `update` method. This method contacts the SIB and if another agent has changed the information the agent´s local copy is updated. An example, where a calendar event object `lunchEvent` is updated, is seen in Listing 4.6.

Listing 4.6: Updating data from the SIB

```
    lunchEvent.update()
```

4.4.1 The triple store and the transaction problem

In Section 2.7 a problem with transactions was described. The triple store has a potential to help with this issue and therefore it further fortifies its place in this work.

To be able to guarantee transaction atomicity transaction rollback needs to be supported. What this means that if a transaction, between the SIB and an agent, fails for some reason the SIB and agents local database need to return to their initial states. Furthermore this exception needs to be handled in some way within the agent. Due to limitations in the capabilities of the SIB this is not yet possible. However, when the SIB is updated the triple store module can be extended to provide the needed support to achieve transaction rollback.

Listing 4.7: Part one of the generated calendar.py API

```
from TripleStore import *

class Calendar:
  def __init__(self):
    self.sib = None

  def setupOntology(self, sib):
    self.sib = sib

    t = RDFTransactionList()
    ns="http://www.abo.fi#"
    t.add_Class(ns+"Calendar")
    t.add_Class(ns+"Event")
    if not self.sib.isJoined:
      self.sib.join()
    p=self.sib.node.CreateInsertTransaction(self.sib.smartSpace)
    p.send(t.get())

    Thing.ts = self.sib.tripleStore
```

4.5 Serialization from code model to Python framework

4.5.1 The calendar.py file

Now that the code model is complete with all the information from the ontology it is ready to be translated into the Python API. First the classes are serialized to Python code and after that all methods are added to the classes. The result of this is a `.py` Python file with the same name as the ontology.

For the calendar ontology the `calendar.py` file contains over 700 lines of code. To get an idea of what the file contains parts of it will be described in the following listings. The parts showed concern the registering of the ontology, the class `Thing`, the class `Calendar`, and the properties `hasTitle` and `hasMember` of the `calendar` class. The rest of the calendar classes´ properties, as well as the whole `Event` class are excluded from the listing.

Listing 4.7 shows the beginning of the `python.py` file. First the triple store (described in Section 4.4) is imported. This is needed throughout the API. After that the ontology is sent to the SIB by the `setupOntology` method. The method creates a list of transactions to send to the SIB, defines the namespace, Listing 4.8: Part two of the generated calendar.py API

```
20 class Thing(object):
21         ...
22     def onNew(cls, hnd):
23         ...
24     ...
25 class DPTitle(DatatypeProperty):
26     range = None
27     def __init__(self, instance):
28         DPTitle.range = XSDString
29         DatatypeProperty.__init__(self, instance)
30     ...
31     def getRange(self):
32         return DPTitle.range
33 
34     def __validateRange(self):
35         if(isinstance(self.instance, self.getRange()) == False):
36             raise RangeException(self.instance)
37 
38 class OPMember(ObjectProperty):
39     ...
40     def __init__(self, uuid, instance = None):
41         OPMember.range = Event
42         ObjectProperty.__init__(self, uuid, instance)
43   ...
```

adds the classes to the transaction list and checks if a connection to the SIB exists. If not, it creates a connection. On line 17 the ontology is sent to the SIB. On the last line the triple store is registered to the `Thing` class. This is needed because every instance needs to be able to access the triple store and as every class extends the `Thing` class they are now able to do so.

In Listing 4.8 the file `calendar.py` continues. Irrelevant parts of the code have been removed for improved readability. The class `Thing` contains an abundance of code needed to facilitate the framework and to enable the SIB technology. This will not be detailed in this thesis. The `onNew` class method is, however, mentioned on line 22, as it contains the functionality to enable subscriptions toward the SIB. The method is used to start a subscription and it stores a pointer to what action to take when the subscription is triggered. A usage example of a subscription is seen in the next chapter in Listing 5.2. After the `Thing` class the `title` property is defined. The `__init__` method initializes an instance of the property and defines that it is of range `XSDString`. The `getRange` class method enables us to query a property for its range. On line

34 the __validateRange method is created in order to enable enforcement of the range restriction. It checks if the range of an instance is set correctly and if it is not, it raises an exception so this can be handled. Lastly the member property is show. The class is similar to DPTitle apart from that the member property is an ObjectProperty.

Listing 4.9 contains the third and last part of the calendar.py file. This listing demonstrates the operation of the class Calendar as well at its properties. The ontology defines that the Calendar class is a subclass of the Thing class. This is achieved on the first line (line 44) in the listing by stating that the Calendar class extends the Thing class. To demonstrate cardinality the calendar, in this ontology, is restricted to contain a maximum of three events. This is defined, with a class variable, on line 45. The ontology also defined that every calendar must have a title. This cardinality is set on line 46. The following __init__ method sets the member and title property to their correct types. After this, on line 53, the class adds itself to the triple store. Now the triple store has knowledge about the class so that if the bedroom coder called the triple stores commit method the Calendar class would be sent to the SIB.

On line 55 the addMember method begins. This method handles how new events are added to the calendar and it is one of the most important methods in the Calendar class. The first line of the method handles the actual linking of the Event instance to the Calendar instance. After this the if statement, that was created by the inverseOf construct handler (described in Section 4.3.4), can be seen. This if statement checks if the inverse relation is set. If not, this is done on line 58. The next if statement is created by the cardinality construct handler (described in Section 4.3.3). Here the cardinality check is performed and if the cardinality slots are already filled, the CardinalityException will be raised to correct, and handle, this. Lastly, an if statement checks that the added instance is of the correct type (Event). If not, the RangeException is raised on line 62 to handle this. After the addMember method some important *get* methods are shown that are needed in order to retrieve information from the Calendar instance.

4.5.2 Additional files

Apart from the calendar.py file, containing the ontology as an object oriented API, some other files are statically added by the generator. These include the tripleStore.py module which works as the local database and handles the connection to the SIB (described in Section 4.4). Also the files node.py, Listing 4.9: Part three of the generated calendar.py API

```
class Calendar(Thing):
    eventCardinality = 3
    titleCardinality = 1
    ...
    def __init__(self, id=None):
        self.member = []
        self.title = []
        Calendar.memberRange = OPMember
        Calendar.titleRange = DPTitle
        Thing.ts.addObject(self)
        ...
    def addMember(self, member):
        self.member = [OPMember(member.getUUID(), member)]
        if(member.getMemberOf() != self):
            member.setMemberOf(self)
        if (eventCardinality > length(self.member))
            raise CardinalityException()
        if(isinstance(member, self.getMemberRange()) == False):
            raise RangeException(obj)

    def getMember(self):
        retVal = []
        if(len(self.member) > 0):
            retVal = self.member
        return retVal
    def getTitleCardinality(self):
        return Calendar.titleCardinality
    def getEventCardinality(self):
        return Calendar.eventCardinality
    def getTitleRange(self):
        return Calendar.titleRange
    def getMemberRange(self):
        return Calendar.memberRange
    ...
```

`uuid.py` and `RDFTransactionList.py` are added to the package. These are needed in order for both `calendar.py` and `tripleStore.py` to work as intended.

Some template agents are also included for the bedroom coder to give a starting point for agent development. These include examples of how regular tasks are performed.

4.6 Summary

This chapter described the code generator that generates the object oriented Python code framework from and OWL ontology. The generator takes as input an `.owl` file and generates as output a number of `.py` Python files. The process of the code generation was described step by step by explaining each of the modules of the generator.

The first module was the Jena module. Jena was used in the generator due to its capable parser and its code model, which simplified the succeeding tasks. After the Jena model was created, the Pellet reasoner performed reasoning on the Jena model. This guaranteed that the ontology was valid and that all relationships stated in the model contained their corresponding counterparts. After this the construct handlers stepped through the Jena model creating a new object oriented model, which would become the framework after the serialization module. In the serialization section, parts of the generated output was illustrated, to give an impression of its contents. The triple store module was also described, which functions as a local database and connection to the SIB.

Chapter 5

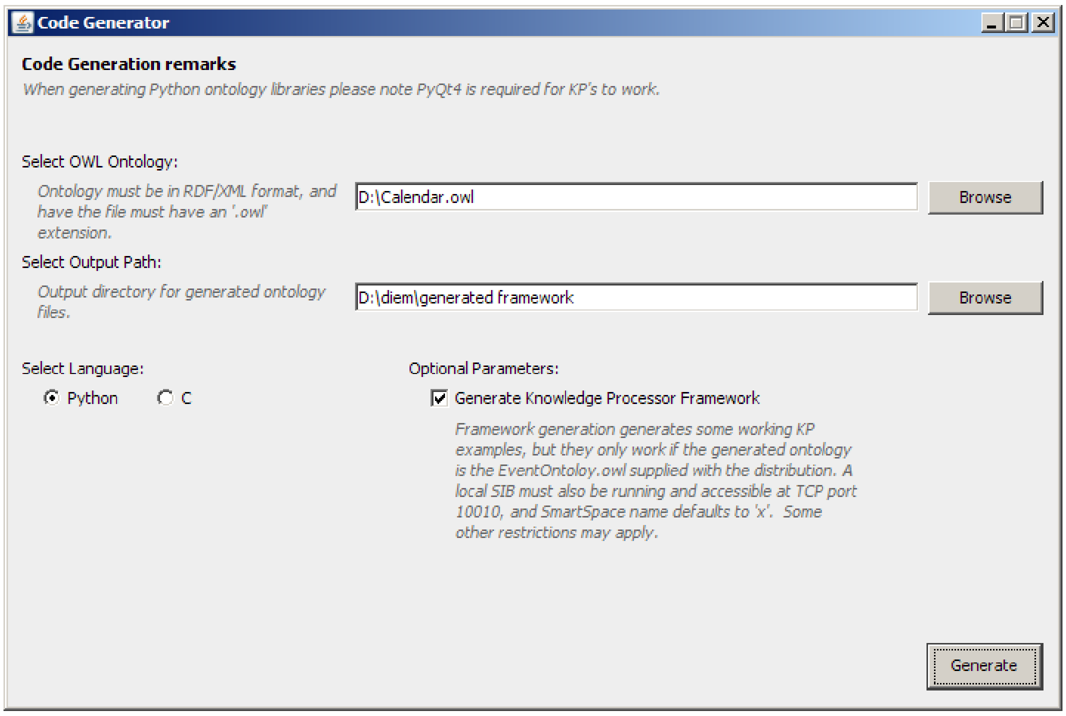
Generation and usage of Python code framework

This chapter is dedicated to demonstrating the process of how a bedroom coder would use the tools in this thesis. The bedroom coder´s intention is to create an application that works with calendar information. For this we will use the ontology and example data from Chapter 3.

The first task for the bedroom coder is to obtain a calendar ontology. The vision of the DIEM project is that the bedroom coder could download the ontology from a library containing standard ontologies. At this stage no such library exists so the bedroom coder must create the ontology himself. In the calendar case the ontology is simple so this should not be a problem. There exists a number of software that can help the bedroom coder with the creation of the ontology, for example, *Protégé Ontology Editor*[15] or *TopBraid Composer*[16].

5.1 Generating the framework

After the bedroom coder has obtained the ontology it can be used to create the Python framework. This is done by simply giving it to the generator as input and defining a location for where the generated output should be saved. The user interface of the generator can be seen in Figure 5.1.


Code Generator

Code Generation remarks

When generating Python ontology libraries please note PyQt4 is required for KP's to work.

Select OWL Ontology:

Ontology must be in RDF/XML format, and have the file must have an '.owl' extension.

D:\Calendar.owl

Select Output Path:

Output directory for generated ontology files.

D:\diem\generated framework

Select Language: [✓ Python] C

Optional Parameters:

[✓ Generate Knowledge Processor Framework]

Framework generation generates some working KP examples, but they only work if the generated ontology is the EventOntoloy.owl supplied with the distribution. A local SIB must also be running and accessible at TCP port 10010, and SmartSpace name defaults to 'x'. Some other restrictions may apply.

Figure 5.1: The code generators graphical interface

5.2 Agent development with generated framework

After the generator has generated the framework the bedroom coder can use it to create an agent. To illustrate an agent the ontology from Figure 3.2 and data from Figure 3.3 is used. The same information can be seen in Appendix B in RDF/XML `.owl` format. This `.owl` file is what the generator has used to generate the framework in this example case.

Listing 5.1 shows the agent, as a whole, that describes the example data. This agent is similar to the agent that writes to the SIB in the calendar synchronizing application described in Section 3. The choice to not describe this particular agent was made due to that it contains code that is specific for the *iCal* standard, which is irrelevant in this context.

In Listing 5.1, on the first, second and last line the frameworks strength is seen. Firstly, on line one the triple store is created. Behind the scenes, a connection to the SIB is also established, and the triple store registers the agent in the SIB. On line two the calendar ontology is defined, and sent to the SIB. If the bedroom coder used multiple ontologies in this agent these would all be imported like the calendar ontology. After this the actual calendar objects are created, first *Jons calendar* and then *Bobs calendar*. When the calendars

Listing 5.1: Calendar agent created using generated framework

```
from TripleStore import *
from calendar import *
# Calendars
jonsCal = Calendar()
jonsCal.setTitle("Jons personal Calendar")

bobsCal = Calendar()
bobsCal.setTitle("Bobs Calendar")
# Events
lunchMeeting = Event()
bobsCalendar.addMember(lunchMeeting)
jonsCalendar.addMember(lunchMeeting)

lunchMeeting.setTitle("Lunch")
lunchMeeting.setLocation("Pablos restaurant")
lunchMeeting.setDtStart("2009-12-20T12:00:00")

thesisMeeting  = Event()
jonsCalendar.addMember(thesisMeeting)
thesisMeeting.setTitle("Meeting with professor")
thesisMeeting.setLocation("Room 3058")
thesisMeeting.setDtStart("2009-12-20T14:00:00")
thesisMeeting.setDtEnd("2009-12-20T14:30:00")

tripleStore.commit()
```

are instantiated their UUIDs are automatically created. The calendar objects are also added to the triple store so that they will be added to the SIB when the bedroom coder chooses to commit the data. On line five and eight the set methods for the calendars titles are used to add title information. The `wasModified` property of the calendar is automatically updated when properties of the calendars are changed.

On the lines 10-23 the information about the two events is created. First the `lunchEvent` is created and as it is defined to be a member of both calendars this relation is set on lines eleven and twelve. In the ontology `hasMember` was defined to have the `owl:inverseOf` property `memberOf`. This is done automatically by the `addMember` method (as seen in Listing 4.9). Like with the calendar, the `wasModified` property of the events is also automatically updated, when a property in them is changed. After this, some data properties for the event are set. On line 18 the `thesisMeeting` event is created followed by the addition of some properties to it.

The last line is important as this is when the local information is committed to the SIB. As described in Section 4.4, the triple store knows about all the information that is not yet committed. The triple store converts this information to RDF triples, places them in a transaction list and sends it to the SIB.

5.2.1 Simple subscription example

Listing 5.2: Subscription example created using generated framework

```
1  Event.onNew(newEventCreated())
2
3  def newEventCreated(self, newInstance)
4      print "New event has been created"
```

One of the strengths of the chosen agent architecture is that agents can subscribe to pieces of information that they are interested in. In Listing 5.2, an example of such a subscription is seen. The idea of this code listing is that it runs in the same smartspace as the previous (Listing 5.1). The subscription can either be a separate agent or it could also run on the same agent that ran Listing 5.1. On line one, in Listing 5.2, the subscription is created and sent to the SIB. It is of type `onNew` which means that if a new event instance is added to the SIB it will run a defined method, in this case the `newEventCreated` method. On line three and four the `newEventCreated` method prints a notification to let the user know about the new event. This, however, could be much more complex as the new instance as passed to the method. At this stage `onNew` is the only supported subscription type.

5.3 Summary

This chapter demonstrated how the bedroom coder uses the generator. After that, an example agent was created using an ontology and data from Chapter 3. One of the goals was to hide as much as possible concerning the connectivity and the SIB. Another goal was to make the development process intuitive and easy to understand. The above two listings show that these two goals were achieved, as long as the bedroom coder remembers to import the correct modules and to commit the changes to the SIB.

Chapter 6

Conclusions

This last chapter discusses the DIEM project and the conclusions from this thesis. Some suggestions to future work are also given.

The current version of the SIB and the smartspace environment is very primitive. This can be seen from the types of problems that are dominant at the moment. The largest issue at this stage is security. Any agent can read or modify the information in the database, as long as it is able to connect to the SIB. This fact, that the information in the database cannot be trusted, renders the whole system useless. To solve this issue the SIB architecture has to be developed. One solution is to divide agents into agent-groups that have different permissions on the data in the database. The simplest permission based restriction on the data would be to only let the creator modify or remove what it has itself created. Also the creator would define which agent-groups are allowed to read the created data. This solution would, in a limited way, solve the problem but it would also present such restrictions that one of the strengths of the architecture would be lost. As the vision is to enable multiple agents to create, modify and use information in the space this solution has to be further developed; for example, the SIB would have a list of trusted agents that have permissions to handle certain pieces of data.

Another strength of the architecture that cannot be fully utilized at this stage is reasoning. Reasoning is the feature that has the potential to set the DIEM projects approach apart from other solutions. Now, as reasoning is only performed before the actual usage of the framework only a limited benefit is achieved. The current reasoning support enables checking if the ontology is valid and it adds relationships that can be inferred from the existing information. However, as no runtime reasoning is performed, there is no guarantee that an application created using the framework will not set the ontology in an inconsistent state. For example, cardinality is only statically checked by the framework and not by the SIB. If the bedroom coder wanted to break the ontology in the SIB he could easily remove this check from the framework and write data that exceeds the set cardinality. Instead, to solve this problem, the SIB should perform runtime reasoning. In that case it would simply deny the transaction if an agent tried to write inconsistent data. Runtime reasoning is also needed to enable more dynamic applications. What is meant by dynamic applications is for example an application that suggests alternative solutions if the proposed solution is not feasible. An example of such an application is an application used to book meetings. The runtime reasoner could check the participants' calendars to see if the proposed time is free. If not, it could suggest another time that is free for all the participants. The reasoner could also include a check to confirm a room is available at the proposed time.

Apart from the above issues there is also a challenge when it comes to manufacturers of devices. The goal is to allow a non-technically skilled person to create applications. To enable this not only the SIB architecture, but also the device-side of the application has to be simple and intuitive. An example of this in the DIEM project has been the calendar standards in the calendar synchronizing application. The framework and Python API simplify the communication between agent and SIB greatly but the same cannot be said about the agent to device (or agent to calendar) communication. *Google calendar* have their own API for working with their calendars and with the *iCal* standard a third party API was used. This meant that a total of three APIs were combined in the calendar synchronizing application. Apart from this, *Google calendar* and *iCal* use different representations for dates and times, so conversions had to be made within the application. Issues like this abolish the possibility that a bedroom coder could create a calendar synchronizing application at this stage. One proposed solution to this problem is that manufacturers would describe the functionality of their devices using standardized ontologies. This way, the bedroom coder could visually, by looking at the ontology, see how an agent can communicate with the device.

Regardless of the extent of the existing problems the general view, about the smartspace approach, remains positive within the members of the DIEM project at ÅAU. It is felt that gaining an overall understanding of how an ontology is used is achieved in a short time. The benefit from using an ontology is that it gives an insight to how the devices and their data can be used. Also, the modularity, the architecture offers, with the agent based applications, opens a whole new approach to application development.

The smartspace architecture has a great potential to revolutionize device interoperability and application development. As more devices appear in our homes and surroundings the need for a solution is constantly growing. The final conclusion is that the DIEM platform is a very promising first solution for a smartspace architecture. After the above described problems are solved and some additional improvements are done it is believed, by the DIEM project members, that the architecture can reach such a stage where the bedroom coder can start creating applications.

Appendix A

OWL DL constructs

This appendix is dedicated to giving a brief explanation of what the chosen OWL DL language enables us to do with an ontology. The language is built out of many parts, called *constructs* and these are covered in this Appendix. The goal is to show what kind of ontologies can be created with the OWL DL language. Some of these constructs originate from OWL:s predecessor RDF Schema and these can be recognized by their prefix `rdfs:`[17].

owl:Class

With `owl:Class` it is possible to create basic classes. A class defines a set, or group, of individuals that belong together and share some properties. For example, `Mia` and `Bob` are both members of the class `Person`.

rdfs:subClassOf

In an ontology we want to put our classes in a hierarchical order and the `rdfs:subClassOf` construct enable us to do this. Every class is in the tree structure has the property `subClassOf` the root class `Thing`. For example, the class `Man` could be stated to be a subclass of the class `Person`. From this a reasoner can deduce that if an individual is a `Man`, then it is also a `Person`. This construct is inherited from RDF Schema.

rdfs:Property

Properties are used to state relationships between individuals or from individuals to data values. For examples the properties `hasChild`, `hasRelative`, `hasSibling`, and `hasAge` could be properties for individuals of the class `Person`.

The first three are used to relate two instance that both are of the class `Person`, and the last (`hasAge`) is used to relate a `Person` instance to an instance of the data type `Integer`. See section 2.5.4 for more information about data types. If we want to further limit what properties can relate to we can use `owl:ObjectProperty` and `owl:DatatypeProperty` which both are subclasses of the RDF class `rdf:Property`. `owl:ObjectProperty` relates individuals to individuals and `owl:DatatypeProperty` relates individuals to data types.

rdfs:subPropertyOf

Properties can also be put in a hierarchical order by making one or more statements that a property is a subproperty of one or more other properties. An example is `hasSibling` that may be stated to be a subproperty of `hasRelative`. From this a reasoner can deduce that if the individual `Mia` is related by the `hasSibling` property to `Bob`, then `Mia` and `Bob` are also related to each other by the `hasRelative` property.

rdfs:domain

The domain of a property limits the individuals to which the property can be used. If a property relates an individual to another individual, and the property has a class as one of its domains, then the individual must belong to the class. For example, a property `hasSibling` may be stated to have the domain of `Person`. From this a reasoner can deduce that if `Bob-hasSibling-Mia`, then `Bob` must be a `Person`. Note that `rdfs:domain` is called a global restriction since the restriction is stated on the property and not just on the property when it is associated with a particular class. In Figure 2.4 domain is illustrated.

rdfs:range

Compared to the `domain` restriction, the range restriction limits a property in the other direction. The range of a property limits the individuals that the property may have as its value. If a property relates an individual to another individual, and the property has a class as its range, then the other individual must belong to the range class. For example, the property `hasSibling` may be stated to have the range of `Person`. From this a reasoner can deduce that if `Mia` is related to `Bob` by the `hasSibling` property, then `Mia` is a `Person`. Range is also a global restriction as is domain above. In Figure 2.4 range is illustrated.

Individual

Individuals are instances of classes, and properties may be used to relate one individual to another. For example, an individual named `Bob` may be described as an instance of the class `Person` and the property `hasEmployer` may be used to relate the individual `Bob` to the individual `Åbo Akademi University`.

owl:equivalentClass

Equality can be used to create synonymous classes. To do this two classes are stated to be equivalent. Equivalent classes have the same instances. For example, `Car` can be stated to be `equivalentClass` to `Automobile`. From this a reasoner knows that any individual that is an instance of `Car` is also an instance of `Automobile` and vice versa.

owl:equivalentProperty

Two properties may be stated to be equivalent. Equivalent properties relate one individual to the same set of other individuals. Equality may be used to create synonymous properties. For example, `hasCoach` may be stated to be the `equivalentProperty` to `hasTrainer`. From this a reasoner can deduce that if `X` is related to `Y` by the property `hasCoach`, `X` is also related to `Y` by the property `hasTrainer` and vice versa. A reasoner can also deduce that `hasCoach` is a subproperty of `hasTrainer` and `hasTrainer` is a subproperty of `hasCoach`.

The `equivalentClass` and `equivalentProperty` are especially useful when combining ontologies where classes and properties have the same meaning but different names. They also enable translation of ontologies made in other languages.

owl:sameAs

Two individuals may be stated to be the same. These constructs allow us to create a number of different names that refer to the same individual. For example, the individual `Bob` may be stated to be the same individual as `Bob Smith` or `Bobby`.

owl:differentFrom

Sometimes it is needed to state that an individual is different from another individual. For example, the individual `Bob` may be stated to be different from the individuals `Mia` and `Steve`. The point with this is that as OWL permits many names for the same individual there also has to be a way to separate them. This becomes valuable when using functional properties (meaning the property has, at most, one value). For example if both `Bob` and `Mia` are values for a functional property, then there is a contradiction. Without the `differentFrom` statement a reasoner would not find this contradiction and instead conclude that `Bob` and `Mia` must be the same individual.

owl:AllDifferent

If we want to say that a set of individuals are all different it becomes cumbersome using the `differentFrom` statement. Instead we can use the `AllDifferent` statement which handles all combinations between individuals. For example, the individuals `Monday` thru `Sunday` could be stated to be mutually distinct using the `AllDifferent` construct.

owl:inverseOf

One property may be stated to be the inverse of another property. An example is the `hasOwner` property with the inverse `owns`. If `Volvo-hasOwner-Ford`, then a reasoner can deduce that `Ford-owns-Volvo`.

owl:TransitiveProperty

Properties may be stated to be transitive. What this means is best described with an example and for this the property `ancestor` is used. The person `Bob` is an ancestor of `Mia` which in turn is an ancestor of `Steve`. If the `ancestor` property is set to be transitive a reasoner can deduce that `Bob` is also an ancestor of `Steve`. The nature of a transitive property sets the side condition that it cannot have a `maxCardinality = 1` restriction. Without this side-condition OWL DL would become undecidable language.

owl:SymmetricProperty

`Friend` is a good example of a symmetric property. For example, if `Bob` is a `friend` of `Steve` then a reasoner can deduce that `Steve` is a `friend` of `Bob`.

owl:FunctionalProperty

Properties may be stated to have a unique value. If a property is a `FunctionalProperty`, then it has no more than one value for each individual (`minCardinality = 0` and `maxCardinality = 1`). For example, `hasPrimaryEmployer` may be stated to be a `FunctionalProperty`. From this a reasoner may deduce that no individual may have more than one primary employer. This, however, does not imply that every `Person` must have at least one primary employer.

owl:InverseFunctionalProperty

The social security number is a good example of an `inverseFunctionalProperty`. A person can be stated to have exactly one social security number, i.e. the property `hasSocialSecurityNumber` has `cardinality = 1`. The inverse of this property (which may be referred to as `isSocialSecurityNumberFor`) has at most one value for any individual in the class of social security numbers. This leads to that any one person's social security number is the only value for their `isTheSocialSecurityNumberFor` property. From this a reasoner can deduce that no two different individual instances of `Person` can have the identical social security number. Consequently, a reasoner can deduce that if two instances of `Person` have the same social security number, then those two instances refer to the same individual.

owl:minCardinality, owl:maxCardinality, owl:cardinality

OWL DL allows cardinality statements for arbitrary non-negative integers. For example the class `Person` could restrict the cardinality of the property `hasFingers` to a cardinality of the integer 10.

owl:disjointWith

Classes may be stated to be disjoint from each other. This means that an individual cannot be a member of both classes. For example, `Man` and `Woman` should be stated to be disjoint classes. From this `disjointWith` statement, a reasoner can deduce an inconsistency when an individual is stated to be an instance of both and similarly a reasoner can deduce that if `Bob` is an instance of `Man`, then `Bob` is not an instance of `Woman`.

owl:oneOf (enumerated classes)

A class can be described by enumeration of the individuals that make up the class. In an enumerated class the members of the class are exactly the set of individuals; no more, no less. A typical example is the class of `daysOfTheWeek` which can be described by simply enumerating the individuals `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday` and `Sunday`. A reasoner may deduce from this that the maximum cardinality is seven for any property that has `daysOfTheWeek` as its `allValuesFrom` restriction.

owl:hasValue (property values)

A property can be required to have a certain individual as a value. For example, instances of the class of `finnishCitizens` can be characterized as those people that have `finnish` as a value of their `nationality`.

owl:unionOf, owl:complementOf, owl:intersectionOf (Boolean combinations)

OWL DL and allows arbitrary Boolean combinations of classes and restrictions: `unionOf`, `complementOf`, and `intersectionOf`. For example, `citizenship` of the class `EuropeanUnion` could be described as the `unionOf` the `citizenship` of all member states.

Appendix B

Calendar ontology XML listing

Listing B.1: Ontology and instances from Figure 3.2 and 3.3 as XML. Input for example codegeneration process

```
<?xml version="1.0"?>

<!DOCTYPE rdf:RDF [
     <!ENTITY www "http://www.abo.fi/#" >
    <!ENTITY diem "http://www.abo.fi/diem.owl#" >
    <!ENTITY owl "http://www.w3.org/2002/07/owl#" >
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" >
    <!ENTITY owl2xml "http://www.w3.org/2006/12/owl2-xml#" >
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#" >
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#" >
]>

<rdf:RDF xmlns="http://www.abo.fi/diem.owl#"
      xml:base="http://www.abo.fi/diem.owl"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:owl2xml="http://www.w3.org/2006/12/owl2-xml#"
      xmlns:owl="http://www.w3.org/2002/07/owl#"
      xmlns:www="http://www.abo.fi/#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:diem="http://www.abo.fi/diem.owl#">
    <owl:Ontology rdf:about=""/>

    <!-- ****************
         OBJECT PROPERTIES
         **************** -->

    <!-- http://www.abo.fi/diem.owl#hasMember -->

    <owl:ObjectProperty rdf:about="#hasMember">
```

```
        <rdfs:domain rdf:resource="#Calendar"/>
        <rdfs:range rdf:resource="#Event"/>
    </owl:ObjectProperty>

    <!-- http://www.abo.fi/diem.owl#memberOf -->

    <owl:ObjectProperty rdf:about="#memberOf">
        <rdfs:range rdf:resource="#Calendar"/>
        <rdfs:domain rdf:resource="#Event"/>
        <owl:inverseOf rdf:resource="#hasMember"/>
    </owl:ObjectProperty>

    <!-- ***************
         DATA PROPERTIES
         *************** -->

    <!-- http://www.abo.fi/#UUID -->

    <owl:DatatypeProperty rdf:about="&www;UUID">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:range rdf:resource="&xsd;string"/>
        <rdfs:domain>
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="#Calendar"/>
                    <rdf:Description rdf:about="#Event"/>
                </owl:unionOf>
            </owl:Class>
        </rdfs:domain>
    </owl:DatatypeProperty>

    <!-- http://www.abo.fi/diem.owl#hasComment -->

    <owl:DatatypeProperty rdf:about="#hasComment">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="#Event"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>

    <!-- http://www.abo.fi/diem.owl#hasDtEnd -->

    <owl:DatatypeProperty rdf:about="#hasDtEnd">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="#Event"/>
        <rdfs:range rdf:resource="&xsd;dateTime"/>
    </owl:DatatypeProperty>

    <!-- http://www.abo.fi/diem.owl#hasDtStart -->

    <owl:DatatypeProperty rdf:about="#hasDtStart">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="#Event"/>
        <rdfs:range rdf:resource="&xsd;dateTime"/>
    </owl:DatatypeProperty>

    <!-- http://www.abo.fi/diem.owl#hasLocation -->

    <owl:DatatypeProperty rdf:about="#hasLocation">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:domain rdf:resource="#Event"/>
        <rdfs:range rdf:resource="&xsd;string"/>
    </owl:DatatypeProperty>

    <!-- http://www.abo.fi/diem.owl#hasTitle -->

    <owl:DatatypeProperty rdf:about="#hasTitle">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:range rdf:resource="&xsd;string"/>
        <rdfs:domain>
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="#Calendar"/>
                    <rdf:Description rdf:about="#Event"/>
                </owl:unionOf>
            </owl:Class>
        </rdfs:domain>
    </owl:DatatypeProperty>

    <!-- http://www.abo.fi/diem.owl#wasModified -->

    <owl:DatatypeProperty rdf:about="#wasModified">
        <rdf:type rdf:resource="&owl;FunctionalProperty"/>
        <rdfs:range rdf:resource="&xsd;dateTime"/>
        <rdfs:domain>
            <owl:Class>
                <owl:unionOf rdf:parseType="Collection">
                    <rdf:Description rdf:about="#Calendar"/>
                    <rdf:Description rdf:about="#Event"/>
                </owl:unionOf>
            </owl:Class>
        </rdfs:domain>
    </owl:DatatypeProperty>

    <!-- **************
```

```
            CLASSES
        ************** -->

    <!-- http://www.abo.fi/diem.owl#Calendar -->

    <owl:Class rdf:about="#Calendar">
        <rdfs:subClassOf rdf:resource="&owl;Thing"/>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#wasModified"/>
                <owl:cardinality
                    rdf:datatype="&xsd;nonNegativeInteger">1
                <owl:Class rdf:resource="&xsd;dateTime"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="&www;UUID"/>
                <owl:cardinality
                    rdf:datatype="&xsd;nonNegativeInteger">1
                <owl:Class rdf:resource="&xsd;string"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#hasTitle"/>
                <owl:cardinality
                    rdf:datatype="&xsd;nonNegativeInteger">1
                <owl:Class rdf:resource="&xsd;string"/>
            </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>

    <!-- http://www.abo.fi/diem.owl#Event -->

    <owl:Class rdf:about="#Event">
        <rdfs:subClassOf rdf:resource="&owl;Thing"/>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#wasModified"/>
                <owl:cardinality
                    rdf:datatype="&xsd;nonNegativeInteger">1
                <owl:Class rdf:resource="&xsd;dateTime"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Restriction>
```

```
                <owl:onProperty rdf:resource="#hasDtEnd"/>
                <owl:maxCardinality
                    rdf:datatype="&xsd;nonNegativeInteger">1
                <owl:Class rdf:resource="&xsd;dateTime"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#hasTitle"/>
                <owl:cardinality
                    rdf:datatype="&xsd;nonNegativeInteger">1
                <owl:Class rdf:resource="&xsd;string"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="&www;UUID"/>
                <owl:cardinality
                    rdf:datatype="&xsd;nonNegativeInteger">1
                <owl:Class rdf:resource="&xsd;string"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#hasComment"/>
                <owl:maxCardinality
                    rdf:datatype="&xsd;nonNegativeInteger">1
                <owl:Class rdf:resource="&xsd;string"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#hasLocation"/>
                <owl:maxCardinality
                    rdf:datatype="&xsd;nonNegativeInteger">1
                <owl:Class rdf:resource="&xsd;string"/>
            </owl:Restriction>
        </rdfs:subClassOf>
        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#hasDtStart"/>
                <owl:cardinality
                    rdf:datatype="&xsd;nonNegativeInteger">1
                <owl:Class rdf:resource="&xsd;dateTime"/>
            </owl:Restriction>
        </rdfs:subClassOf>
    </owl:Class>

    <!-- http://www.w3.org/2002/07/owl#Thing -->

    <owl:Class rdf:about="&owl;Thing"/>

    <!-- **************
          INDIVIDUALS
         ************** -->

    <!-- http://www.abo.fi/#BobsCalendar -->

    <Calendar rdf:about="&www;BobsCalendar">
        <wasModified rdf:datatype="&xsd;dateTime"
            >2009-12-31T12:00:00</wasModified>
        <www:UUID rdf:datatype="&xsd;string">4</www:UUID>
        <hasTitle rdf:datatype="&xsd;string">Bobs
            Calendar</hasTitle>
        <hasMember rdf:resource="&www;LunchMeeting"/>
    </Calendar>

    <!-- http://www.abo.fi/#JonsCalendar -->

    <Calendar rdf:about="&www;JonsCalendar">
        <wasModified rdf:datatype="&xsd;dateTime"
            >2009-12-31T23:59:59</wasModified>
        <www:UUID rdf:datatype="&xsd;string">3</www:UUID>
        <hasTitle rdf:datatype="&xsd;string"
            >Jons personal Calendar</hasTitle>
        <hasMember rdf:resource="&www;LunchMeeting"/>
        <hasMember rdf:resource="&www;ThesisMeeting"/>
    </Calendar>

    <!-- http://www.abo.fi/#LunchMeeting -->

    <Event rdf:about="&www;LunchMeeting">
        <www:UUID rdf:datatype="&xsd;string">2</www:UUID>
        <wasModified rdf:datatype="&xsd;dateTime"
            >2009-12-01T23:59:59</wasModified>
        <hasDtStart rdf:datatype="&xsd;dateTime"
            >2009-12-20T12:00:00</hasDtStart>
        <hasTitle rdf:datatype="&xsd;string">Lunch</hasTitle>
        <hasLocation rdf:datatype="&xsd;string"
            >Pablos restaurant</hasLocation>
        <memberOf rdf:resource="&www;BobsCalendar"/>
        <memberOf rdf:resource="&www;JonsCalendar"/>
    </Event>

    <!-- http://www.abo.fi/#ThesisMeeting -->

    <Event rdf:about="&www;ThesisMeeting">
        <www:UUID rdf:datatype="&xsd;string">1</www:UUID>
        <wasModified rdf:datatype="&xsd;dateTime"
            >2009-12-01T23:59:59</wasModified>
        <hasDtStart rdf:datatype="&xsd;dateTime"
            >2009-12-20T14:00:00</hasDtStart>
        <hasDtEnd rdf:datatype="&xsd;dateTime"
            >2009-12-20T14:30:00</hasDtEnd>
        <hasTitle rdf:datatype="&xsd;string"
            >Meeting with professor</hasTitle>
        <hasLocation rdf:datatype="&xsd;string">Room
            3058</hasLocation>
        <memberOf rdf:resource="&www;JonsCalendar"/>
    </Event>
</rdf:RDF>
```

Bibliography

[1] "Devices and Interoperability Ecosystems project homepage - www.diem.fi."

[2] I. Oliver and J. Honkola, "Personal semantic web through a space based computing environment," *arXiv*, vol. cs.NI, Aug 2008. [Online]. Available: http://arxiv.org/abs/0808.1455v1

[3] O. Lassila, "Enabling semantic web programming by integrating rdf and common lisp," in *The First Semantic Web Working Symposium.* Citeseer, 2001, pp. 403–410.

[4] J. J. Carroll and G. Klyne, "Resource description framework (RDF): Concepts and abstract syntax," *W3C recommendation*, Feb. 2004, http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[5] N. Noy and D. McGuinness, "Ontology development 101: A guide to creating your first ontology," *Knowledge Systems Laboratory*, pp. 01–05, 2001.

[6] D. Brickley and L. Miller, "FOAF vocabulary specification," *Namespace Document*, vol. 3, 2005.

[7] F. van Harmelen and D. L. McGuinness, "OWL web ontology language overview," *W3C recommendation*, Feb. 2004, http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[8] P. V. Biron and A. Malhotra, "XML schema part 2: Datatypes second edition," *W3C recommendation*, Oct. 2004, http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[9] A. Malhotra and P. V. Biron, "XML schema part 2: Datatypes," *W3C recommendation*, May 2001, http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.

[10] G. Agha, “Actors: A model of concurrent computation in distributed systems,” vol. AITR-844, Oct 1985.

[11] H. Baker, “Actor systems for real-time computation,” vol. MIT-LCS-TR-197, Jan 1978.

[12] B. McBride, “Jena: Implementing the rdf model and syntax specification,” in *Proc. of the 2001 Semantic Web Workshop*, 2001. [Online]. Available: http://ftp1.de.freebsd.org/Publications/CEUR-WS/Vol-40/mcbride.pdf

[13] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz, “Pellet: A practical owl-dl reasoner,” *Web Semantics: science, services and agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, 2007.

[14] P. F. Patel-Schneider, I. Horrocks, and P. Hayes, “OWL web ontology language semantics and abstract syntax,” W3C, W3C Recommendation, Feb. 2004, http://www.w3.org/TR/2004/REC-owl-semantics-20040210/.

[15] N. Noy, M. Sintek, S. Decker, M. Crubezy, R. Fergerson, and M. Musen, “Creating semantic web contents with protege-2000,” *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 60–71, 2001.

[16] T. Inc, “Topbraid Composer,” *Getting Started Guide Version*, vol. 1.

[17] D. L. McGuinness, C. Welty, and M. K. Smith, “OWL web ontology language guide,” *W3C recommendation*, Feb. 2004, http://www.w3.org/TR/2004/REC-owl-guide-20040210/.