Mikael Kirkeby Fidjeland

Distributed Knowledge in Case-Based Reasoning

Knowledge Sharing and Reuse within the Semantic Web

Trondheim, 2006

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

Master's thesis
Programme of study: Master of Science in Informatics

Supervisor: Agnar Aamodt, IDI

NTNU
Innovation and Creativity

Abstract

The Semantic Web is an emerging framework for data reuse and sharing. By giving data clear semantics it allows for machine processing of this information. The Semantic Web technologies range from simple meta data to domain models using the Web Ontology Language (OWL). Large parts of the semantics of OWL stems from the Knowledge Representation field of Description Logics.

In Case-Based Reasoning (CBR) specific knowledge in the form of cases is in problem-solving. The Creek system is a Knowledge-Intensive approach to CBR that combines specific knowledge with general domain knowledge.

Part of this work is to define an OWL vocabulary for Creek. This vocabulary will contain concepts and relations used to describe case-bases and domain models. Using this vocabulary, a Creek knowledge model can be described in OWL and shared with others on the Semantic Web. We also examines how domain ontologies can be reused and imported into a Creek knowledge model.

We propose a design for Creek to operate in the context of the Semantic Web and its distributed nature of knowledge representation. Part of the design will be tested by implementation.

Problem description The Semantic Web is an emerging framework for data reuse and sharing. By giving data clear semantics it allows for machine processing of this information. The Semantic Web technologies range from simple meta data to domain models using the Web Ontology Language (OWL).

Case-Based Reasoning (CBR) uses specific knowledge in the form of cases to solve problems. The Creek system is a Knowledge-Intensive approach to CBR that combines specific knowledge with general domain knowledge.

This thesis project consists of three parts

1. Study knowledge representation related to the Semantic Web in general and OWL in particular
2. Define an OWL vocabulary for Creek.
3. Exemplify how this vocabulary can export and import to/from other ontologies.

The cand. scient project combines a theoretical study of Web ontologies with a design and implementation of an OWL ontology adapted to the Creek system.

Quotes The quotes at the beginning of each chapter is from the novel *Neuromancer* (Gibson, 1984). Gibson was one of the pioneers of the cyberpunk genre, and he is often credited to have coined terms like 'cyberspace' and 'the matrix'. Much of the action in this book centers around themes such as artificial intelligence and cyberspace. When the protagonist of the book is called Case, it all fits the theme of this report — Case-Based Reasoning and the Semantic Web.

Acknowledgements Thanks to professor Agnar Aamodt at IDI, NTNU, for good and patient supervising during the process of writing this master thesis.

Contents

List of Figures

Listings

Chapter 1

Introduction

> *'Cyberspace. A consensual hallucination experienced daily by billions of legitimate operators, in every nation, by children being taught mathematical concepts... A graphic representation of data abstracted from the banks of every computer in the human system. Unthinkable complexity. Lines of light ranged in the nonspace the mind, clusters and constellations of data. Like city lights, receding...'* William Gibson, Neuromancer

1.1 Goal

The goal of the research presented in this master thesis is to explore how to share the knowledge of a knowledge based system, and how to use reuse knowledge when building domain models. This will be done within the framework of Creek and the Semantic Web.

The goal will be approached by:

- studying the Creek architecture and knowledge representation
- studying knowledge representation in the Semantic Web
- extending the Creek architecture by
  - define an OWL vocabulary for Creek
  - represent Creek knowledge models in OWL
  - import OWL ontologies into a Creek knowledge model
- implementing the OWL/Semantic Web functionality for jCreek

1.2 Background and Motivation

Case-Based Reasoning (CBR) is a machine learning technique that relies on specific knowledge from previously solved cases to solve problems. Creek is a CBR architecture developed at the Department of Computer and Information Science at NTNU[1]. Creek uses general domain knowledge in addition to a case base for knowledge-intensive problem solving and learning (Aamodt, 2004). A general domain model is built of, among other things, concepts, classes, instances, properties and relations.

The Semantic Web is a World Wide Web Consortium activity concerning structuring information on the Internet. Part of it will be simple metadata describing resources on the Web, but more complex domain ontologies will also be in the Semantic Web. Domain ontologies will consist of classes, subclass, relations and so on.

As we see the Semantic Web's domain ontologies and the CBR domain models have much in common with regard to their representations. A Creek system will have knowledge, both specific and general, about a specific domain. It might be interesting to share this knowledge with others. The Semantic Web may be the medium for this. Other CBR systems might have use for the case-base as a whole or some specific case. The general domain knowledge may have broader interest.

The sharing of knowledge may be the other way around. When building a general domain model for a Creek system, there may already exist domain ontologies on the Semantic Web appropriate for the problem area at hand.

1.3 Methods Used

This report will begin with an analytical approach discussing Case-Based Reasoning and the Semantic Web. The background and Knowledge Representation aspects of both of these will be examined. Other related research will be presented. A design for sharing and reuse of knowledge of a Creek system using the Semantic Web will be presented. This design will be tested by implementation.

A Creek OWL Vocabulary and a namespace will be defined in the work presented. The publication and online availability of these should also be considered part of the work in this thesis. These resources can be retrieved at the location `http://creek.idi.ntnu.no/owl/`. A textual description of the Creek OWL Vocabulary is published, akin to what is presented here in section 5.2.

[1] Norwegian University of Science and Technology, `http://ntnu.no/`

1.4 Structure of Thesis

This thesis starts by going into the details of the background and motivation given above. Chapter 2 gives a short introduction of Case-Based Reasoning and the Creek system. A more detailed analysis of the Semantic Web follows in chapter 3. Three related research projects covering Case-Based Reasoning and the Semantic Web are described in chapter 4.

The approach and result will be described in chapter 5. A design for knowledge sharing and reuse for Creek is proposed in section 5.1. Section 5.2 presents the Creek OWL vocabulary, and the following sections goes into details of the design. The implementation done is described in chapter 6.

Finally, in chapter 7 we discuss the results and problems that might arise from our design. In the appendix A you find the Creek OWL Vocabulary and an example OWL domain model. Appendix B contains the API summaries of the implementation done.

Chapter 2

Creek

> *The density of information overwhelmed the fabric of the matrix, triggering hypnagogic images. Faint kaleidoscopic angles centered in to a sliver-black focal point. Case watched childhood symbols of evil and bad luck tumble out along translucent planes.* William Gibson, Neuromancer

This chapter gives a short introduction of Case-Based Reasoning and an overview of the Creek architecture. The focus is on Knowledge Representation in Creek and other aspect relevant to the work described in this thesis. Some familiarity with Creek and Case-Based Reasoning is assumed. For more elaborate background of Case-Based Reasoning and Creek see (Aamodt, 1991), (Aamodt and Plaza, 1994), (Aamodt, 1994), (Sørmo, 2000), (Tomassen, 2002) and (Aamodt, 2004).

2.1 Case-Based Reasoning

Case-Based Reasoning (CBR) is an approach to problem solving and machine learning that uses previously solved problems as the base for the reasoning and learning. A *case* is a problem situation described by a set of relevant findings. Knowledge is retained in past cases that also have attached the solution for each particular problem. New problems, or cases, are matched to the case-base to find a suitable solution. The typical four-step CBR cycle can be seen in figure 2.1.

When given a new problem, a relevant case is *retrieved* from the case-base. To be *reused*, the solution from this case might need some adaption for the new problem at hand. This possible solution is then tested and *revised*. Finally the new case and its solution is *retained* in the case-base for use in future problem solving.

Creek is a theoretical CBR framework and *"a knowledge intensive approach to*

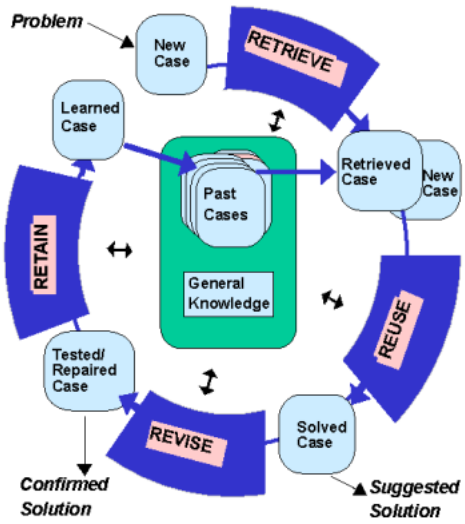

Figure 2.1: The CBR cycle. From (Aamodt and Plaza, 1994)

problem solving and learning" (Aamodt, 1991). Knowledge intensive here means that in addition to cases, denoted *specific knowledge*, Creek uses *general domain knowledge* in the reasoning and problem solving.

2.2 Knowledge Representation in Creek

Creek uses frames and semantic networks for knowledge representation. The general and the specific knowledge is tightly bound together in Creek. Cases are described in terms of findings that have relations to concepts in the general knowledge.

Figure 2.2 shows the structure and the interconnectivity of the knowledge base in Creek. The top-level *generic concepts* are special concepts and relations that Creek uses internally in its reasoning, like the concept of `Case` itself and the relations `hasFinding` or `hasSubclass`. This is known as the *ISOPOD model* (Aamodt, 1991). The rest of the knowledge model are subclasses or instances of entities here.

The *general domain concepts* part of the knowledge base contains concepts about the domain at hand and, among other, causal, structural and functional relations between these concepts. This part is also called the general knowledge.

The *cases* makes up the specific knowledge. The findings are, as mentioned above, relations to the general knowledge. The solution of a case will typically be a concept in the general knowledge, but it might also be another case.

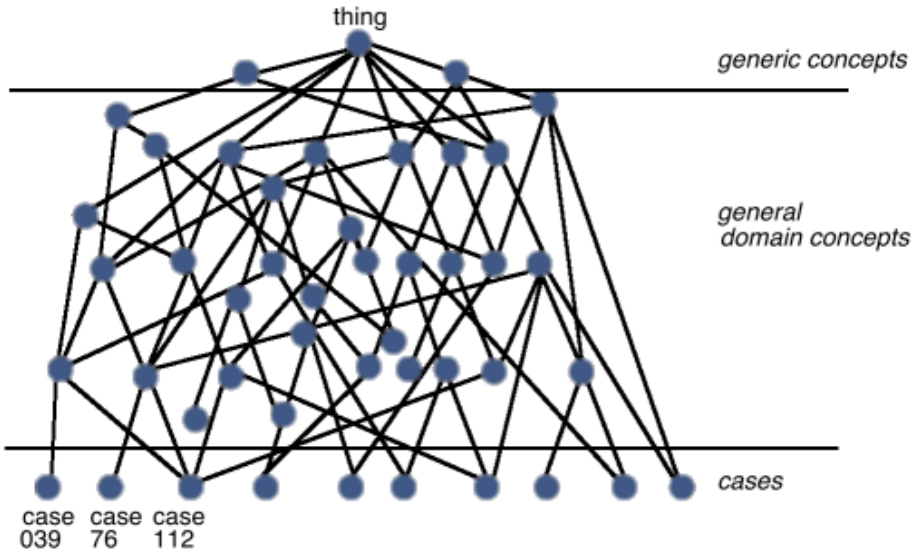

Figure 2.2: Integrating cases and general knowledge. From (Aamodt, 2004).

2.3 Reasoning in Creek

Creek is well suited to represent knowledge within open and weak theory domains, domains in which knowledge is uncertain and incomplete. The main inference type in Creek is abduction, *"inference to the best explanation"* (Aamodt, 1994). The reasoning in Creek follows a tree-step `ACTIVATE-EXPLAIN-FOCUS` process. Relevant parts of the semantic network are *activated*, a set of possible hypotheses with *explanations* are generated and the best solution is *focused* or selected. This is done in each of the retrieve, reuse and retain phases in figure 2.1

Inheritance is another reasoning mechanism used in Creek, and *default reasoning* is supported. With default reasoning properties that are inherited may be overridden at the local level. Another inheritance mechanism used in the abductive reasoning is *plausible inheritance* (Sørmo, 2000). This allows relations to be inherited or transferred over other than the usual `subclassOf` and `instanceOf` relations. Relations can be given weights to quantify its importance and strength. By adding up and comparing weights for relations that transfers other relations, Creek concludes whether it is plausible that some given relation should or should not be inherited.

Default reasoning, plausible inheritance and abduction are non-monotonic reasoning methods. In non-monotonic reasoning it is allowed to add new information such that previously drawn conclusions may be rendered invalid. By using default reasoning you may draw conclusions on the lack of contradicting evidence, that is as long as there is no local overriding of an inherited property. This conclusion may be changed if you later add new information by overriding this property, hence the non-monotonicity.

Overall, the integration of general domain knowledge and cases and the ab-

ductive reasoning makes Creek a flexible architecture for learning and problem solving.

The Creek architecture is implemented in Java in the jCreek[1] package.

[1] jCreek, see http://creek.idi.ntnu.no/

Chapter 3

The Semantic Web

> *He'd operated on an almost permanent adrenaline high, a byproduct of youth and proficiency, jacket into a custom cyberspace deck that projected his disembodied consciousness into the consensual hallucination that was the matrix.* William Gibson, Neuromancer

In this chapter we will go into detail about the Semantic Web and the technologies underlying it. The focus will be on the aspects being relevant to knowledge representation.

3.1 Background

The Semantic Web [1] is a World Wide Web consortium (W3C) [2] activity. It was initiated by Tim Berners-Lee, the man behind the WWW, HTTP and HTML technologies (Berners-Lee, 1998b).

While there is a seemingly endless amount of information on the Web today, it is often hidden in HTML pages. While good for presenting data for humans, the data is not structured for machine processing in a meaningful way. This kind of information presentation has efficiency and integrity limitations (Berners-Lee et al., 1999) with regard to computer processing.

By giving meaning — that is semantics — to content in documents, the Semantic Web aims to create an universal framework for information exchange.

> *The Semantic Web is an extension of the current Web in which information is given well-defined meaning, enabling computers and people to work in better cooperation.* (Berners-Lee and Miller, 2002)

[1] Semantic Web, `http://www.w3.org/2001/sw/`
[2] W3C, `http://w3.org`

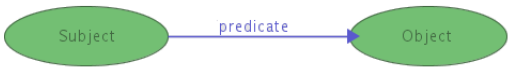

Figure 3.1: A RDF Statement. From (Klyne and Carroll, 2004)

Adding structure to meaningful content allows for automatic processing. In this environment any kind of resource in the Semantic Web, like web pages, programs or hardware devices, may publish its own data or use data published by others to its own needs.

> *[It] will provide an infrastructure that enables not just web pages, but databases, services, programs, sensors, personal devices, and even household appliances to both consume and produce data on the web.* (Hendler et al., 2002)

The spectre of information complexity in the Semantic Web spans from simple metadata for web pages to complex ontologies. The rest of this chapter will focus on parts of the Semantic Web that are relevant to building taxonomies, ontologies and domain models.

The Semantic Web is realized through a set of increasingly complex standard languages, namely XML, XML Schema, RDF, RDF-S and OWL. XML is a well known, general purpose markup language. It provides a syntax for structured documents, while XML Schema puts restrictions on the structure. RDF, RDF-S and OWL are discussed in more detail in the following sections.

3.2 Resource Description Framework

In the Semantic Web data and information are linked and defined. This is done by structuring data in descriptive statements. The language for doing this is RDF, Resource Description Framework (Klyne and Carroll, 2004) (Beckett, 2004). RDF grew out of a community need and design efforts for a framework for representing metadata on the web (Miller, 1998).

As its name implies, RDF describes resources. In the context of RDF and the Semantic Web, a resource is defined as anything that can be identified by an Uniform Resource Identifier (URI) (Masinter et al., 2005) (W3C, 2001). An URI is simply a string, and does not necessarily tell you how to access the actual resource or whether it is accessible online at all. Resources that can be located on the web, physical entities that exist in the real world, things that are entirely abstract — they can all be identified by URIs (Jacobs and Walsh, 2004). The work presented in this report will describe concepts and relations in domain models as RDF resources.

A simple RDF statement is a `subject-predicate-object` triple as seen in fig-

Figure 3.2: An example RDF graph

ure 3.1. A predicate, also called a property, describes a directed binary relation between a subject and an object. The subject and the predicate are resources, that is they are identified by an URI. The object is either a resource or a literal. Literals have atomic values like a string or a number. A set of RDF triples define a directed graph in which the nodes are the subjects and objects of the triples. There may be blank nodes, that is anonymous nodes without an URI. Figure 3.2 shows an example RDF graph, and listing 3.1 shows the equivalent RDF triples using N-Triples notation (Beckett and Barstow, 2001).

3.2.1 XML vs RDF

RDF itself is an abstract data model. There exists several notations for serializing a RDF model, the N-Triples notation is one of them. This report will use the RDF/XML notation (Beckett, 2004), which is the most widely used and the official notation endorsed by the W3C. Listing 3.2 shows the RDF/XML notation of the graph in figure 3.2.

Listing 3.1: N-Triples notation of RDF graph in figure 3.2

```
<http://creek.idi.ntnu.no/owl/example/car-model#Engine>
   <http://creek.idi.ntnu.no/owl/example/car-model#partOf>
   <http://creek.idi.ntnu.no/owl/example/car-model#Car> .

<http://creek.idi.ntnu.no/owl/example/car-model#Engine>
   <http://creek.idi.ntnu.no/owl/example/car-model#hasEngineStatus>
   <http://creek.idi.ntnu.no/owl/example/car-model#EngineRunning> .

<http://creek.idi.ntnu.no/owl/example/car-model#Car>
   <http://creek.idi.ntnu.no/owl/example/car-model#hasColor>
   "some color" .
```

Listing 3.2: RDF/XML notation of RDF graph in figure 3.2

```
<rdf:RDF
    xmlns = "http://creek.idi.ntnu.no/owl/example/car-model#"
    xml:base = "http://creek.idi.ntnu.no/owl/example/car-model"
    xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#">

<rdf:Description rdf:ID="Engine">
   <partOf rdf:resource="#Car"/>
   <hasEngineStatus rdf:resource="#EngineRunning"/>
</rdf:Description>

<rdf:Description rdf:ID="Car">
   <hasColor>some color</hasColor>
</rdf:Description>

<rdf:Description rdf:ID="EngineRunning"/>

</rdf:RDF>
```

There has been some discussion about how well suited the XML notation is. Some of the objections are that it is verbose, hard to write manually and that people focus on syntax rather than on the underlying data model and so on (Lassila, 2005). By comparing the N-triples notation in listings 3.1 with the XML notation in listing 3.2, you find that the actual RDF triples get somewhat lost in the XML syntax.

Some of this arise because a XML document defines a tree structure, while a RDF document defines a graph. Thus in XML the order of the elements is important and a XML tree can only be represented one way. In RDF it is not so. Order is not important and it is possible to have different RDF/XML representations of the same graph. So why bother with RDF, why not just use XML? No doubt the structure in figure 3.2 could be represented using XML. However, there is a lack of clear semantics in XML (Heflin and Hendler, 2000). The semantics is bound to the structure (van Harmelen and Fensel, 1999) of the XML document. While it is easy for humans to interpret it, machines need knowledge about this structure in order to understand the semantics. This is what RDF has to offer – clear semantics (Hayes, 2004) easily interpreted by machines.

3.2.2 RDF Documents, Namespaces and Vocabularies

RDF resources often have URIs in the form of `some-URI#local-name`. This kind of URI is called an URI reference, a *URIref* (Manola and Miller, 2004). Resources having the same `some-URI` part are said to belong to the *namespace* `some-URI#`. A XML namespace is by definition a collection of names, identified by an URIref (Bray et al., 1999). In the RDF/XML notation a namespace `http://www.some-URI.org/example#` is often assigned a prefix `ex:` such that `ex:term` is used as a shorthand for, and should be interpreted as, the full URIref `http://www.some-URI.org/example#term`.

The `rdf:ID` attribute identifies new resources, and only the `local-name` part is given. The resource's full URIref is obtained by adding the namespace or the URIref of the containing *RDF document*.

However, RDF and the Semantic Web is not only about identifying new resources. It is possible to make statements about other resources, about resources in arbitrary namespaces, using the `rdf:about` attribute. This makes the Semantic Web *decentralized*. This aspect will be more thoroughly discussed in section 3.4.

A RDF/XML document is a collection of RDF statements in RDF/XML notation, at some given location, organized in a XML file with one `rdf:RDF` root element and most often having its own namespace.

A set of URIrefs intended for some purpose or sharing some context is known as a *vocabulary*. A vocabulary will normally be defined in a namespace, and is often identified by its namespace URIref or the namespace prefix. Note that an URIref is itself an URI, as is a namespace name.

To exemplify; the RDF vocabulary is the set of terms

> `http://www.w3.org/1999/02/22-rdf-syntax-ns#RDF`,
> `http://www.w3.org/1999/02/22-rdf-syntax-ns#Description`,
> `http://www.w3.org/1999/02/22-rdf-syntax-ns#ID`,
> `http://www.w3.org/1999/02/22-rdf-syntax-ns#about`,
> `http://www.w3.org/1999/02/22-rdf-syntax-ns#resource`,
> and so on,

the namespace is `http://www.w3.org/1999/02/22-rdf-syntax-ns#`, by convention assigned the prefix `rdf:` such that it is identified as the `rdf:`-vocabulary. The RFD vocabulary is defined in (Beckett, 2004), which is a W3C Recommendation document, not a RDF/XML document.

3.2.3 RDF Schema

RDF Schema (RDF-S) (Brickley and R.V.Guha, 2004) is an extension to RDF used to create and specify application or user specific vocabularies. It is itself a vocabulary defined in the `http://www.w3.org/2000/01/rdf-schema#` namespace, assigned the `rdfs:` prefix.

RDF-S provides a *type system* (Manola and Miller, 2004) to RDF, making it possible to structure resources and properties in class hierarchies. A *class* is a resource describing a type or a category of things. Being a resource, a class has an URIref identifying it. A class has a *class extension* associated with it, which is the set of *instances* of the class. The resource `rdfs:Class` is the superclass of all RDF classes, and new classes are instances, or class members, of this class. RDF-S also introduces the class `rdfs:Resource`, which is the class of everything. All resources are instances of this class. A class may be an instance of itself, for example is `rdfs:Class` the class of RDF-S classes and is defined as an instance of itself:

```
<rdfs:Class rdf:about="http://www.w3.org/2000/01/rdf-schema#Class"/>
```

A resource may be an instance of more than one class. Class membership is expressed with the `rdf:type` property, so the RDF triple `<myClass> <rdf:type> <rdfs:Class>.` defines a new class `myClass`. The RDF/XML syntax allows for an abbreviation for such instantiation, such that

```
<rdfs:Class rdf:ID="myClass"/>
```

is equal to

```
<rdf:Description rdf:ID="myClass">
 <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
</rdf:Description>
```

and further makes it possible to declare a resource `myInstance` an instance of `myClass` writing

```
<myClass rdf:ID="myInstance"/>
```

Classes may be organized taxonomically using the `rdfs:subClassOf` property. The meaning, or the semantics, of a class B being a *subclass* of class A, is that all instances of class B will also be instances of class A. `rdfs:subClassOf` is *transitive*, so given class A and B above and that C is a subclass of B, all instances of C will also be instances of B and A.

RDF-S also introduces constructs to organize and describe properties. The `rdf:Property` resource is the class of all RDF properties, and properties are instances of this class. Note that `rdf:Property` is not a super-property of all properties. `rdf:Property` is itself an instance of `rdfs:Class`. Properties may be organized hierarchically with the `rdfs:subPropertyOf` property. The semantics of `rdfs:subPropertyOf` is that given that a property Q is a sub-property of a property P, all resources related by Q are also related by P.

The use of a given property may be restricted using the `rdfs:range` and the `rdfs:domain` properties. `rdfs:domain` is used to state that a property may be applied to only instances of a certain class, that the subject of a RDF statement in which the property is used is an instance of a certain class. Likewise, `rdfs:range` restricts the values, or the object of a RDF statement, to an instance of some class.

For example, declaring a property `myProperty` with a domain and range

```
<rdf:Property rdf:ID="myProperty">
  <rdfs:range rdf:resource="#A"/>
  <rdfs:domain rdf:resource="#B"/>
</rdf:Property>
```

allows for consistent use on some instances `mySubject` and `myObject` such as

```
<A rdf:ID="myObject"/>
<rdf:Description rdf:ID="mySubject">
  <myProperty rdf:resource="#A"/>
</rdf:Description>
```

given that A and B are `rdfs:Class`'es. This also allows to infer that `mySubject` is an instance of B.

A property may have more than one class in its range and/or domain. In this case, the resource being the subject/object must be an instance of all the stated classes.

As `rdfs:range` and `rdfs:domain` are properties, that is they are instances of the `rdf:Property` class, they may be applied to themselves. The range for both of them is the class `rdfs:Class`, and their domain is the class `rdf:Property`.

In RDF-S properties are independent of class definitions and thus will generally have global scope. The use of a given property may be restricted as described above, but such restrictions describe how properties are applied to given classes and what might be inferred from their usage. They do not put any other limitations on the use of these classes.

Even if RDF-S is an extension of RDF, all legal RDF-S documents are also legal RDF documents. By this is meant that an RDF-S document define a RDF graph, and even if some program have no understanding of the RDF-S concepts such as `rdfs:subClassOf`, `rdf:Property`, `rdfs:range` and so on, it can still see a RDF-S document as a set of RDF statements.

3.3 Web Ontology Language

What is an ontology? *"An ontology is a specification of a conceptualization"* (Gruber, 1993). In other words, an ontology is about describing and defining meaning of terms and concepts and interrelationships between them, in some specific domain of interest. (McGuinness, 2002) defines an ontology at least to have these properties:

- Finite controlled (extensible) vocabulary
- Unambiguous interpretation of classes and term relationships
- Strict hierarchical subclass relationships between classes

As we see, RDF-S allows us to make simple ontologies. A more expressive and powerful language for defining ontologies is the Web Ontology Language, another W3C Recommendation (McGuinness et al., 2004) (McGuinness and van Harmelen, 2004) (Dean and Schreiber, 2004).

The Web Ontology Language is known under the acronym OWL. Reasons for choosing this rather than the more 'acronymically correct' WOL might be the easier pronunciation of 'owl' and the wisdom often ascribed to owls. It is also said to be a tribute (wikipedia.org, OWL) to a Knowledge Representation (KR) project at the MIT in the 70's by Bill Martin called "One World Language", also known as OWL.

OWL was not the first ontology language developed for use on the web. It is built upon work on other web-oriented ontology languages such as SHOE (Heflin et al., 1999), DAML-ONT (Ogbuji and Ouellet, 2002), OIL (Horrocks, 2000) and DAML+OIL (van Harmelen et al., 2001). SHOE introduced the idea of using URIs for names and terms in ontologies. DAML-ONT was part of a DARPA program working on Semantic Web ontology languages, and later merged with OIL, an European initiative, into DAML+OIL. DAML+OIL is the direct predecessor of OWL, and was the starting point when the development of the OWL language began.

3.3.1 Description Logics

Much of the formal basis and semantics for OWL come from the field of Description Logics (DL). This section will give a short introduction to and background of DL. See (Nardi and Brachman, 2003), (Donini et al., 1996) and (Horrocks et al., 2003) for more details about DL, its logical foundations and its relation with OWL.

The motivation behind DL was the lack of clear semantics in the frames and semantic network (SNF) approaches to Knowledge Representation (KR). For example, most of SNF systems have some kind of `is-a` relation, but it can sometimes be unclear if this means something is an instance of something, if it is a subclass, if it is a prototype and so on. DL combines the naturalness of SNF with the clear semantics of logical KR systems. One of the ancestors of DL is the KL-ONE (Brachman and Schmolze, 1985) system, which started to move the semantic networks approach towards a more logical foundation.

DL is a subset of first order predicate calculus (FOPC). An important point was to have the clear formal semantics of FOPC, but still maintain computational tractability. With respect to FOPC, DL sacrifices some of the expressive power to effectiveness. DL also has more primitives than FOPC. DL is one of the most active fields of research within KR, and is one of the best understood formalisms in terms of complexity, decidability and expressiveness.

In DL knowledge is divided in two distinct "domains", the *TBox* and *ABox*. The TBox, the terminological box, contains taxonomical hierarchies and definitions of concepts. In DL terminology, a *concept* corresponds to a class, and a *role* corresponds to a property. The knowledge in the TBox is *intensional*, it refers to the meaning of the underlying concept. The ABox, the assertional box, contains facts about individuals and their relations to concepts in the TBox. DL use the term *individual* for an instance. The knowledge in the ABox is *extensional*, it points to things in the real world.

Description Logics has the ability to express the following:

Conjunctions, disjunctions and negations. New concepts can be defined as conjunctions, disjunctions and/or negations of other concepts. For example, if given the concepts of Car and Red, one might express the concept of 'cars that are not red' by

$Car \sqcap \neg Red$

Or given the concepts of $EngineNotRunning$ and $EngineNotStarting$ the expression

$EngineNotRunning \sqcup EngineNotStarting$

refers to the concept of the problem of an 'engine not running or not starting'.

Quantified restrictions. Roles might be restricted with *existential quantifications* and *value restrictions*. The concepts of 'individuals having the color red' can be described by

$\exists hasColor.Red$

and the concept of 'cars all of whose passengers are male' by

$Car \sqcap \forall passengers.Male$

Number restrictions. Cardinality for roles is set using number restrictions. All cars having three or more doors and four or less seats, can be expressed with

$Car \sqcap (\geq 3 hasDoors) \sqcap (\leq 4 seats)$

The main inference method in DL is *subsumption*. A concept D subsumes another concept C if D is considered more general than C, written $C \sqsubseteq D$. In other words, if C is a subset of D. A classification task for a given concept can be solved by finding the most specific concepts subsuming it, and the most general concepts subsumed by it, thus finding the right place in the hierarchy. Another inference method is *satisfiability*, which is to test if a given expression do not necessarily denote the empty set. *Instance checking* is an inference testing if an individual is an instance of a given class. It is used for *instantiation*, which is to find the most specific class for a given individual. In DL being a subset of FOPC, inference in DL is sound and complete. Thus, inference in DL is monotonic. The monotonic nature of DL and the the implications for reasoning in the Semantic Web in general will be discussed in section 3.5.

The meaning of the OWL language and its constructs is formalized through a model theory based on DL (Patel-Schneider et al., 2004).

3.3.2 OWL Vocabulary

OWL builds on top of and uses constructs from RDF and RDF-S. OWL uses a RDF syntax, which means an OWL document is also a RDF graph — and is serialized with the RDF/XML notation. The OWL vocabulary is defined in the `http://www.w3.org/2002/07/owl#` namespace, assigned the prefix `owl:`.

3.3.2.1 Classes and Individuals

A class is a set of *individuals*, where individual is OWL jargon for instance. The set of a class' members is known as the *extension* of the class. Every individual is defined to be member of the class `owl:Thing`. OWL also specify the empty class `owl:Nothing`.

A resource is defined as an OWL class by declaring it to have `owl:Class` as the value for the property `rdf:type`. Class hierarchies are expressed using the RDF-S construct `rdfs:subClassOf`. Individuals are defined by declaring them to be member of a class. For example

```
<owl:Class rdf:ID="Status"/>
<owl:Class rdf:ID="EngineStatus">
   <rdfs:subClassOf rdf:resource="#Status"/>
```

```
</owl:Class>
<EngineStatus rdf:ID="EngineRunning"/>
```

declares a class `Status`, a subclass `EngineStatus` and an instance `EngineRunning` of the class `EngineStatus`. The semantics for the subclass relation is the same as in RDF-S. For the example above, every instance of `EngineStatus` is also an instance of `Status`. Remember that the `rdfs:subClassOf` relation is transitive. In OWL, a class is defined *intensionally*, that is the meaning of the underlying concept. So two classes may have the same set of members, but still be two different classes.

Two classes may be defined to have exactly the same members, that their class extensions are equal, by using the property `owl:equivalentClass`. Similarly, two instances may be defined to be identical by the `owl:sameAs` property. And opposite, two instances might be asserted to be distinct using the property `owl:differentFrom`. OWL does not make a *unique names* assumption. In other words, several URIrefs or several OWL individuals can refer to the same thing. Unless `owl:sameAs` or `owl:differentFrom` is specified, OWL does not make an assumption of either.

OWL allows classes to be described using the set operators *union*, *intersection* and *complement* using the OWL properties `owl:unionOf`, `owl:intersectionOf` and `owl:complementOf`. These applies to classes' extensions, the set of their members, and can describe new classes as, say, an union of two other classes. These work as the conjunction, disjunction and negation operators in Description Logics. Other ways to describe a class is by *enumerating* exactly and only its members, or by declaring it to be *disjoint* with other classes.

Further, OWL allows for *anonymous* classes which contains individuals which properties match certain *restrictions*. These restrictions and how they are used with and apply to properties are described below.

3.3.2.2 Properties

OWL separates between two property classes; `owl:DatatypeProperty` and `owl:ObjectProperty`. New properties are defined as instances of these two OWL classes. Object properties are relations between instances of classes. Datatype properties are relations between instances and RDF literals or XML Schema datatypes (XSD) (Biron and Malhotra, 2004). The XSD datatypes `xsd:integer` and `xsd:string` are required by the OWL specification to be supported by OWL reasoners.

OWL uses the constructs `rdfs:subPropertyOf`, `rdfs:domain` and `rdfs:range` from RDF-S to describe and organize properties. These have the same meaning in OWL as in RDF-S. Properties can further be described to be *transitive*, *symmetric* and *functional*. A property might be defined as an *inverse* of another property. Given

```
<owl:ObjectProperty rdf:ID="hasEngineStatus">
```

```
  <rdfs:range rdf:resource="#EngineStatus"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="engineStatusOf">
  <owl:inverseOf rdf:resource="#hasEngineStatus"/>
</owl:ObjectProperty>
```

if an individual or class X is related to an individual or class Y by `hasEngineStatus`, then Y is related to X by `engineStatusOf`.

Two properties may be declared to be the same using `owl:equivalentProperty`.

While `rdfs:range` and `rdfs:domain` are global in their scope, OWL allows to restrict properties locally. `owl:allValuesFrom` can restrict all the values for a property of instances of some class to be from a given class. `owl:someValuesFrom` make a similar restriction that at least one value be from a given class. These work as the existential quantifier and the value restriction in Description Logics. A property can be required to have as value an instance of a given class by `owl:hasValue`. The number restrictions from Description Logics are expressed by defining a cardinality constraint using `owl:cardinality` or setting an upper and/or lower bound using `owl:maxCardinality` and/or `owl:minCardinality`.

All the OWL constructs described above might be used to define arbitrary complex classes. For example, to define the class of all red cars, given the classes `Car`, `Red` and the property `hasColor` one might write

```
<owl:Class rdf:ID="RedCars">
   <owl:intersectionOf rdf:parsetype="Collection">
       <owl:Class rdf:about="#Car">
       <owl:Restriction>
           <owl:onProperty rdf:resource="#hasColor"/>
           <owl:hasValue rdf:resource="#Red"/>
           <owl:cardinality> 1 </owl:cardinality>
       </owl:Restriction>
   </owl:intersectionOf>
</owl:Class>
```

Note that the part within the `owl:Restriction` tag defines an anonymous class which class extension is all individuals having exactly one `hasColor` property with the value being exactly `Red`. The class $RedCars$ is the intersection of this class and the Car class.

All OWL classes can be described using Description Logics syntax. The definition of the $RedCars$ class above is the same as $RedCars \equiv Car \sqcap (\geq 1 hasColor.Red)$ in DL syntax.

Listing 3.3 shows an simple example OWL ontology with some classes, subclasses, properties and restrictions.

Listing 3.3: Example OWL Ontology

```
<rdf:RDF
    xmlns = "http://creek.idi.ntnu.no/owl/example/car-model.owl#"
    xml:base = "http://creek.idi.ntnu.no/owl/example/car-model.owl"
    xmlns:owl = "http://www.w3.org/2002/07/owl#"
    xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#">
<owl:Ontology rdf:about="">
    <rdfs:label>Car example domain model</rdfs:label>
</owl:Ontology>

<owl:Class rdf:ID="Car">
  <rdfs:label>a car</rdfs:label>
</owl:Class>

<owl:Class rdf:ID="Engine">
  <partOf rdf:resource="#Car"/>
  <rdfs:subClassOf>
    <owl:Restriction>
       <owl:onProperty rdf:resource="#hasStatus"/>
       <owl:allValuesFrom rdf:resource="#EngineStatus"/>
    <owl:Restriction>
  <rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Battery">
  <partOf rdf:resource="#Car"/>
  <rdfs:subClassOf>
    <owl:Restriction>
       <owl:onProperty rdf:resource="#hasStatus"/>
       <owl:allValuesFrom rdf:resource="#BatteryStatus"/>
    <owl:Restriction>
  <rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Status"/>

<owl:Class rdf:ID="EngineStatus">
   <rdfs:subClassOf rdf:resource="#Status"/>
</owl:Class>
<EngineStatus rdf:ID="EngineRunning"/>
<EngineStatus rdf:ID="EngineNotRunning">
   <owl:differentFrom rdf:resource="#EngineRunning"/>
</EngineStatus>

<owl:Class rdf:ID="BatteryStatus">
   <rdfs:subClassOf rdf:resource="#Status"/>
</owl:Class>
<BatteryStatus rdf:ID="BatteryOK"/>
<BatteryStatus rdf:ID="BatteryEmpty">
   <owl:differentFrom rdf:resource="#BatteryOK"/>
</BatteryStatus>

<owl:ObjectProperty rdf:ID="partOf"/>

<owl:ObjectProperty rdf:ID="hasStatus">
   <rdfs:range rdf:resource="#Status"/>
</owl:ObjectProperty>

</rdf:RDF>
```

3.3.2.3 Importing

An OWL ontology may import another ontology using the `owl:imports` construct. All the content of the importee ontology is considered part of the importer ontology. `owl:imports` is a transitive statement, meaning that if the ontology being imported is itself importing a third ontology, this one is also imported to the first and so on.

3.3.2.4 OWL Variants

OWL comes in three sub-languages; OWL Full, OWL DL and OWL Lite. These differ in expressiveness and computational tractability.

OWL Full is an extension to and compatible with RDF-S. All RDF constructs may be used, and in general, all RDF documents will also be OWL Full documents. In OWL Full, `owl:Class` is equivalent to `rdfs:Class`, `owl:Thing` is equivalent to `rdfs:Resource` and `owl:ObjectProperty` is equivalent to `rdf:Property`. OWL Full have maximum expressiveness, for example might a class be both a collection of instances and an instance itself. However this comes at the cost that there is no guaranties about computational tractability.

OWL DL is a sub-language of OWL Full. It support all OWL constructs, but puts constraints of how some of these might be used. For example, the sets of classes, instances and properties must be disjoint. In OWL DL, `owl:Class` is a proper subclass of `rdfs:Class`. The DL part of the name of course stems from OWL's relation to Description Logics. The constraints in OWL DL are based on the work on Description Logics so that reasoners, inference engines and other tools made to work with Description Logics can be utilized. OWL DL has maximum expressiveness while at the same time the reasoning being computational complete and decidable.

OWL Lite is a sub-language of OWL DL. It has the same constraints as OWL DL, but does not support all the language constructs. The objective is to have a minimal set of useful features with limited expressiveness, but more easy implementation for tools and inference engines.

Every legal OWL Lite ontology is a legal OWL DL ontology, and every legal OWL DL ontology is a legal OWL Full ontology. The same holds for valid inferred conclusions. For the rest of this report, when referring to OWL we refer to OWL Full unless the DL or Lite variants are specified.

3.4 Knowledge Representation in the Semantic Web

The Semantic Web as a whole is not an Artificial Intelligence (AI) or Knowledge Representation (KR) project. It is about making information on the Web processable for machines. On the other hand, OWL is very much based on AI and KR research from fields such as Description Logics — and is indeed a KR language. This section will discuss some KR related aspects about OWL.

OWL is not just another ontology language, it is a *web* ontology language. It is designed to to operate in the framework of the Internet in general and the Semantic Web in particular. This has some implications that give OWL some distinct properties compared to more traditional KR languages. As the web itself, KR in OWL is *distributed* and *decentralized*.

3.4.1 Decentralized Representation

> *"The Semantic Web is what we will get if we perform the same globalization process to Knowledge Representation that the Web initially did to Hypertext. We remove the centralized concepts of absolute truth, total knowledge, and total provability, and see what we can do with limited knowledge."* (Berners-Lee, 1998a).

One key aspect of KR in the Semantic Web is the global scope of the names of concepts, relations and so on. 'Global' here means global in the widest sense of the word. By using URIrefs for identification, a name is universally unique. A concept described or defined in one ontology can be referred to in any other ontology. Because an URIref is identifying the concept, there is no ambiguity of what concept it is. There is no central control of the URI/URIref naming scheme. Anyone or anywhat can name any concept one wish, even though it is good practice using URIrefs from domains one has control over. So everyone is free to make whatever ontology one might wish, naming the terms freely and make it available over the Semantic Web.

As important is the fact that anybody might make assertions about concepts defined by others. One can use terms from other ontologies, say things about them, extend them and so on. This together with the OWL constructs to import other ontologies, extend them, assert that a concept is equal to another concept — all this makes knowledge distributed.

> *It's about creating things from data you've complied yourself, or combining it with volumes [...] of data from other sources to make new discoveries.* (Updegrove, 2005)

Even though decentralized and distributed Knowledge Representation is possible on the Semantic Web, it is not necessarily a precise description of its current state. It is a field of continuing development and active research. The size and

number of available ontologies is somewhat limited. It is still an open question if the Semantic Web will be as successful and widespread as the World Wide Web and Internet in general.

3.4.2 Open World

OWL makes an *open world assumption*. This is the assumption that anything not currently known to be true is considered unknown. This in contrast to the closed world assumption where such things are considered false. In closed worlds, one assumes all relevant information is specified. Database systems is an example of such closed worlds, and also many traditional KR systems makes this assumption.

Because OWL allows concepts or classes defined in one ontology to be extended by other ontologies, and because of the size and possible rate of change on the Web, one cannot assume something to be false based on lack of any assertion.

3.5 Reasoning in the Semantic Web

The OWL specification states when extending an ontology, the consequences is *monotonic*. Reasoning is monotonic if; **S** and **T** are sets of propositions, A is a valid inference of **S** and $\mathbf{S} \subseteq \mathbf{T}$, then A is a valid inference of **T** (O'Donnell, 1993) In other words, the addition of new information cannot make a previously drawn conclusion invalid. With regard to the open world assumption, an extension of a class or a concept cannot retract information declared elsewhere. Assertions might be added, but never deleted.

All reasoning based on logics such as FOPC are monotonic. Thus the reasoning endorsed by Description Logics are monotonic — which again leads to reasoning with OWL DL resources being monotonic. The attitude of the W3C Semantic Web Working Group is quite clear in its support for monotonic reasoning. The actual reasoning, if there is such a thing, will be manifested as a wide range of various web services, differing in complexity, what language constructs is supported and competence.

The monotonicity of the Semantic Web does not make it immune to inconsistencies. This is not necessarily a problem though,

> [...] *criticize the Semantic Web because they think that everything in the whole Semantic Web will have to be consistent, which is of course impossible. In fact, the only things I need to be consistent are the bits of the Semantic Web I am using to solve my current problem.* (Updegrove, 2005).

Conflicting and inconsistent data might possible be a characteristic of the Semantic Web as it is of the Web as a whole. Validation and establishing trust is therefore an important task faced by reasoners operating in the Semantic Web.

Chapter 4

Other related research

> *'You're the other AI. You're Rio. You're the one who wants to stop Wintermute. What's your name? Your Turing code. What is it?'*
> William Gibson, Neuromancer

This chapter gives an overview over research related to Case-Based Reasoning in the context of the Semantic Web. Three projects are described; CaseML, a Web Case Language; DzCBR, a framework for decentralized Case-Based Reasoning; and Colibri, a framework for software reuse in building Case-Based Reasoning systems.

4.1 Case Markup Language

The Case Markup Language (CaseML) (Chen and Wu, 2003) is a RDF based vocabulary to describe cases for sharing them over the Semantic Web. Cases are defined as ordered pairs $(p, s) \in P \times S$, where P is the problem space and S is the solution space. Individual problems are of the form $P(x_1, x_2, ..., x_n)$, where x_i are features describing the problem. Individual solutions are similar, but from S rather than P. Cases can be represented in several ways; described textually in free-text, described as attribute-value pairs or described structurally by a set of objects, which again is a set of attribute-value pairs.

The CaseML namespace is `http://grid.zju.edu.cn/caseml#`, using `caseml:` as prefix.

The CaseML vocabulary defines the following classes; `caseml:CaseBase`, `caseml:Case`, `caseml:Problem`, `caseml:Feature`, `caseml:Solution` and `caseml:SimilarityAssessment`.

The following properties are defined; `caseml:domainOntology`, `caseml:hasProblem`, `caseml:hasDescription`, `caseml:hasSolution`, `caseml:hasSimilarityAssessment`, `caseml:measureMethod` and

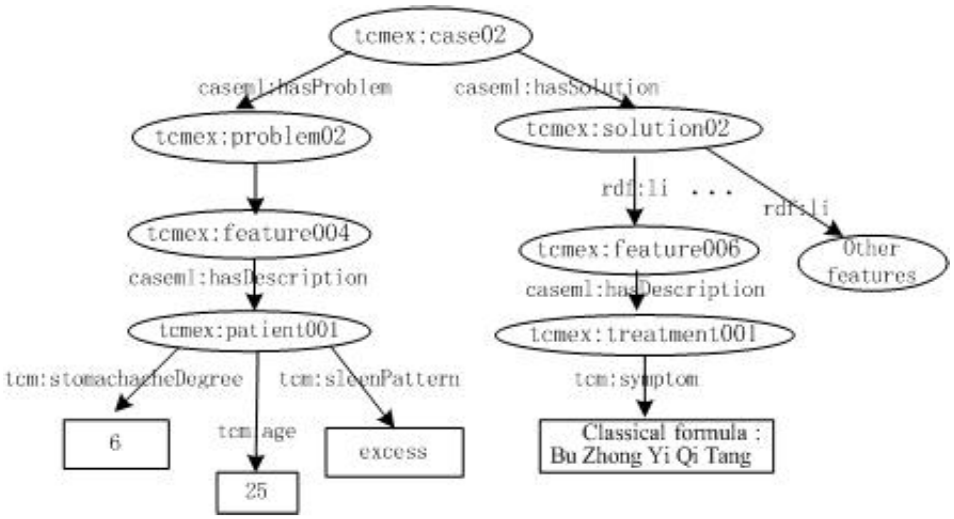

Figure 4.1: RDF graph of a case described in CaseML. From (Chen and Wu, 2003)

`caseml:hasAdaptionRule`.

The definition of the case, problem, solution and feature classes and properties are as described above. The CaseBase is the set of cases. A CaseBase can have one or more SimilarityAssements, which is the class of assessment algorithms used to match a problem to a solution. A CaseBase belongs to one domain, in which the vocabulary used to describe the cases are defined. The domainOntology property is used to declare the URI to the domain ontology.

Figure 4.1 shows an example case described structurally.

CaseML is a small vocabulary to describe cases. It puts some restrictions for how cases can be represented, and on what can actually be represented. The use of RDF gives it clear semantics, but limited expressiveness compared to OWL.

4.2 Decentralized Case-Based Reasoning

DzCBR (Decentralized Case-Based Reasoning) (d'Aquin et al., 2005) is a framework for distributed reasoning. It is a knowledge-intensive approach to CBR (KI-CBR) that relies on knowledge from domain knowledge during the problem solving process. DzCBR is designed for use within the Semantic Web, but uses knowledge from 'contextualized ontologies' using *Context OWL* (C-OWL) (Bouquet et al., 2003).

C-OWL differs between *shared* knowledge and *local* knowledge. An *ontology* is a shared model having a common representation. A *context* is a local model, having its own language and its own interpretation. Even though knowledge

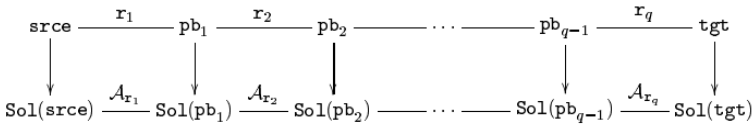

Figure 4.2: A similarity path path from `srce` to `tgt` and an adaption path from `Sol(srce)` to `Sol(tgt)`. From (d'Aquin et al., 2005)

is local and thus not shared, it is still compatible with other representations. C-OWL is an extension of OWL that *"allow to relate [...] concepts, roles and individuals in different ontologies"* using *bridge rules*. A bridge rule is a semantic relation between entities in different local ontologies, that is contexts. A set of bridge rules is a *context mapping*. A contextual ontology is an OWL ontology that by the use of context mappings is related to other OWL ontologies. In this way, C-OWL facilitates modular ontologies, where each context is a different *viewpoint* on the same domain. DzCBR processes are distributed among these viewpoints, collaborating through the bridge rules constructs of C-OWL.

In DzCBR, a case is a problem `pb` and the solution to that problem `Sol(pb)`. A case base is a set of source cases `(srce, Sol(srce))`. The CBR process is the solving of a target problem `tgt`, and has two steps:

Retrieval Find a matching case `(srce, Sol(srce))` from the case base.

Adaption Adapt the found solution into `Sol(tgt)`. This solution is tested, repaired and possibly retained for future problem solving.

4.2.1 Adaption Knowledge

In addition to domain knowledge, ontologies in DzCBR may contain *adaption* knowledge. The basic elements of the adaption knowledge are *reformulations*. A reformulation is a pair $(r, \mathcal{A}_r)$ where r is a relation between two problems and $\mathcal{A}_r$ is an *adaption function* such that if r relates $\texttt{pb}_1$ to $\texttt{pb}_2$, then $\texttt{Sol(pb}_1\texttt{)}$ can be adapted into $\texttt{Sol(pb}_2\texttt{)}$ using $\mathcal{A}_r$.

DzCBR uses the decentralized knowledge in a C-OWL contextualized ontology. Each viewpoint has local knowledge about how to solve some particular kind of problem and also context dependent adaption knowledge. The retrieval step in the CBR process consists of finding a *similarity path*, a set of relations r_i that relate a source case `srce` to the target problem `tgt`. Through a reformulation each relation has an according adaption function $\mathcal{A}_{r_i}$. These adaption functions makes up an *adaption path*, forming `Sol(srce)` into `Sol(tgt)`. See figure 4.2 for how these to paths relate to each other.

4.2.2 Reasoning with C-OWL Ontologies

DzCBR defines two OWL classes `Problem` and `Solution`, which members are the problems and the solutions in the domain of discourse. The `hasSolution` property relates a problem `pb` to its solution `Sol(pb)`. OWL axioms are used to relate problems and solution to the domain knowledge. Further, reformulations and adaption functions are formalized using OWL constructs. Not much specific is said about this, other than *" [...] by Web services implementing transformation operations like specialization, generalization and property substitution on OWL individuals"*.

DzCBR uses reasoning mechanisms from Description Logics. Classification is used to structure the case base, and a class represents an index for a source problem. If a source case `srce` is an instance of some index class, then `Sol(srce)` is considered to be a reusable solution for all instances of this class. During the retrieval step, instantiation is used to find index classes for source problems. New information are inferred about an individual during this process.

4.2.3 Distributed Reasoning

The use of C-OWL and modular ontologies are responsible for the 'decentralized' part of DzCBR. Reasoning is divided into *local* and *global* reasoning. The local reasoning is the classification and instantiation tasks as described above, but within a specific context or viewpoint. Global reasoning is the *global subsumption* and *global instantiation* with the use of bridge rules. Information known about some individual in a given context can be completed using inferences made in other contexts.

The case-based reasoning in DzCBR has a *global target* problem which is a set of *local target* problems. Each of the local target problems are solved by a DzCBR process within some context. This is an approach taken from decentralized AI. It consists of agents, each having its own goal. In this case, this goal is to solve a local problem case. The agents collaborate through knowledge sharing to find a global solution.

The rationale behind this approach is among other things that some domains are better represented as a modularized ontology, and the use of contexts in C-OWL is a simpler way of doing this than having all such knowledge in one place.

4.3 Colibri

Colibri (Díaz-Agudo et al., 2005) is a framework for software reuse in CBR systems. It is designed to be of aid in building knowledge-intensive CBR (KI-CBR) systems. As mentioned before, KI-CBR is an approach to case-based reasoning that use both specific knowledge in the form of cases, and general domain knowledge. The reuse aspect covers both software, that is Problem-Solving

Methods as described in section 4.3.1 below, and reuse of domain knowledge in the form of ontologies. There is a clear separation between reasoning and the knowledge base in this approach. The reasoning tasks are described in units called Problem-Solving Methods. The knowledge base is declared using terms defined in an ontology; CBROnto. Also there is a separation between domain-independent CBR concepts and the knowledge in the domain model used in KI-CBR systems.

Colibri is implemented in Java in the jColibri[1] package.

4.3.1 Problem-Solving Methods

Problem-Solving Methods (PSMs) (Fensel and Motta, 2001) are application and domain independent, reusable reasoning components. A PSM is a reasoning task, and is decomposed into a number of subtasks. A task is a goal that must be achieved, and a PSM contains the problem-solving behaviour needed to do that. Related to this is the IBROW (Benjamins et al., 1998) project, aimed at intelligent brokering service for reuse of PSMs, and UPML (Fensel et al., 1999), a language developed to describe and implement such components.

A PSM consists of three parts; *competence*, *operational specification* and *requirements*. The competence is a declarative statement of what can be achieved, the operational specification a description of the reasoning process and the requirements is a description of what knowledge is required to achieve the competence of the PSM.

4.3.2 CBROnto

CBROnto (Díaz-Agudo and González-Calero, 2001) is an OWL ontology describing a common CBR terminology. The work on CBROnto is still an active field of research, but the current vocabulary is defined in the `http://gaia.sip.ucm.es/cbronto.owl#` namespace. We use the prefix `cbronto:` for this namespace.

CBROnto is used to describe the CBR process (Díaz-Agudo and González-Calero, 2002), such as the PSMs described below and also an ontology used to describe knowledge models used in CBR (Díaz-Agudo and González-Calero, 2000). We describe these two aspects in the two following subsections.

4.3.2.1 A task/method ontology

The jColibri approach describes PSMs in CBR. There is four main tasks; the *retrieve*, *reuse*, *revise* and *retain* tasks in the CBR cycle in figure 2.1. A PSM is a method to achieve the goal of a task, so there can be several alternate PSMs that solve the same task. Each of these PSMs are decomposed into subtasks.

[1] http://sourceforge.net/projects/jcolibri-cbr/

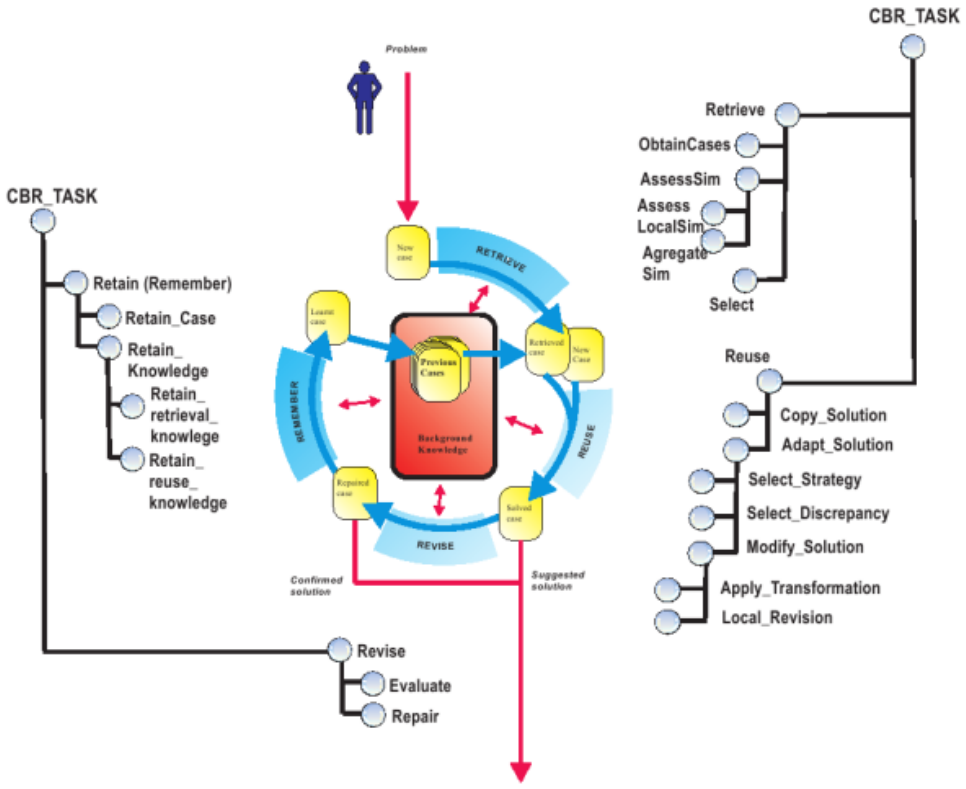

Figure 4.3: CBROnto task structure. From (Díaz-Agudo et al., 2005).

Figure 4.3 shows how these four tasks are decomposed into methods and subtasks in the Colibri framework.

The ontology defines the class `cbronto:Task` and its four subclasses:

```
<owl:Class rdf:ID="Task"/>
<owl:Class rdf:ID="Reuse">
  <rdfs:subClassOf rdf:resource="#Task"/>
</owl:Class>
<owl:Class rdf:ID="Retrieve">
  <rdfs:subClassOf rdf:resource="#Task"/>
</owl:Class>
<owl:Class rdf:ID="Retain">
  <rdfs:subClassOf rdf:resource="#Task"/>
</owl:Class>
<owl:Class rdf:ID="Revise">
  <rdfs:subClassOf rdf:resource="#Task"/>
</owl:Class>
```

These tasks can be solved by a PSM. The methods are of the class `cbronto:Method`:

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

The following properties can be used to describe a method:

```
<owl:DatatypeProperty rdf:ID="hasName"/>
<owl:DatatypeProperty rdf:ID="hasTextualDescription"/>

<owl:ObjectProperty rdf:ID="hasMethodPrecondition">
<owl:ObjectProperty rdf:ID="hasMethodPostCondition"/>
<owl:ObjectProperty rdf:ID="hasApplicationRequirements"/>
<owl:ObjectProperty rdf:ID="hasMethodType"/>

<owl:ObjectProperty rdf:ID="#hasCompetences">
  <rdfs:range rdf:resource="#Task"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="#hasSubTasks">
  <rdfs:range rdf:resource="#Task"/>
</owl:ObjectProperty>
```

These describe the methods in terms of the competence, specifications and requirements properties of the PSM. The CBROnto itself does not specify any PSMs, but defines the vocabulary to do so.

When PSMs are used in the building of a CBR system, the PSMs must be connected with the domain knowledge. Because each PSM is a reusable generic

component, some gaps may be needed to be filled in order to apply the methods to the domain knowledge. This is done by mapping terms in the domain knowledge to terms in CBROnto, using the classification offered by Description Logics reasoners.

4.3.2.2 A knowledge model ontology

The CBR tasks are applied to the knowledge in the knowledge base. The general domain knowledge may be reused in the form of imported ontologies. Description Logics (DL) classification method is used to integrate the domain knowledge with the CBROnto terms.

The specific knowledge takes the form of cases, and the concepts and relations used to describe and structure a case base are generic in respect to many CBR systems. CBROnto defines a vocabulary aimed at being a standard way to describe CBR cases. CBROnto defines the class `cbronto:Case` as

```
<owl:Class rdf:ID="Case">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty  rdf:resource="#hasDescription"/>
      <owl:someValuesFrom rdf:resource="#CaseDescription"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="CaseDescription"/>

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

In other words, the class of instances having at least one property `cbronto:hasDescription` having value from the class `cbronto:CaseDescription`. CBROnto also defines properties to relate and describe cases:

```
<owl:ObjectProperty rdf:ID="hasSolution">
    <rdfs:domain rdf:resource="#Case"/>
    <rdfs:range rdf:resource="#CaseSolution"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasDescription">
  <rdfs:domain rdf:resource="#Case"/>
  <rdfs:range rdf:resource="#CaseDescription"/>
</owl:ObjectProperty>
```

A case is an instance of the `cbronto:Case` class, or instances of various subconcepts of this class. An instance of `cbronto:CaseDescription` is used to describe a case, and the relation `cbronto:hasDescription` is used to link a

case to its description. Further, the CBROnto contains terms to describe the description. Likewise, an instance of `cbronto:CaseSolution` is used to represent the solution of a case.

The `owl:Restriction` part of the `cbronto:Case` declaration above is what makes the DL classification possible. It says that any instance that have a `hasDescription` property, or a sub-property of this, with a value from the class `cbronto:CaseDescription`, is also an instance of `cbronto:Case`. This is an inference one can make without this being explicitly stated, and thus makes it possible to classify instances in a knowledge model as a `cbronto:Case`.

4.3.3 Software Reuse

jColibri is framework for developing CBR systems. By describing components of reasoning in CBR as PSMs, these components can be reused when new systems are made. jColibri contains a library of code that supports these methods. The configuration of a new system is an interactive process between the user and the jColibri system. One selects a task, and then select some methods for this task. One might also need to develop new tasks or methods. This is possible in jColibri. It is an extensible, distributed framework, and the design envisions new methods being published and put in a central registry. Web Services may then be used to detect and find PSMs developed by others and that are applicable for useage.

Chapter 5

Creek and the Semantic Web

> *Headlong motion through walls of emerald green, milky jade, the sensation of speed beyond anything he'd known before in cyberspace ...*
> William Gibson, Neuromancer

This chapter will discuss the approach and the research done to reach the goals of this master thesis. We propose a framework for knowledge sharing and reuse in the Creek framework. Further, an OWL vocabulary of the Creek ISOPOD model is defined.

5.1 A Design for Distributed Knowledge

This section will present a design for sharing and reuse of knowledge from and to a Creek Case-Based Reasoning (CBR) system.

A CBR system based on Creek has a *knowledge model* on which the reasoning is based. This knowledge is dynamic — it changes as the system solves new problems. Assuming that this knowledge is of interest to others, the *sharing* of this knowledge on the Semantic Web should be useful. Likewise, one might find domain models defined by others and reuse this in order to create or extend the knowledge base in an effective manner. The knowledge will in both these cases be in the form of OWL ontologies, as described in chapter 3.

We will assume that we have a Creek CBR system running and accessible on the Web. We call this system CreekServer. Such a system will have knowledge about some problem domain. For illustrative purposes we assume that the knowledge of the CreekServer is the domain knowledge from the `car-model.owl` ontology from appendix A.2.

From chapter 2 and figure 2.2 we have that a Creek knowledge model can be

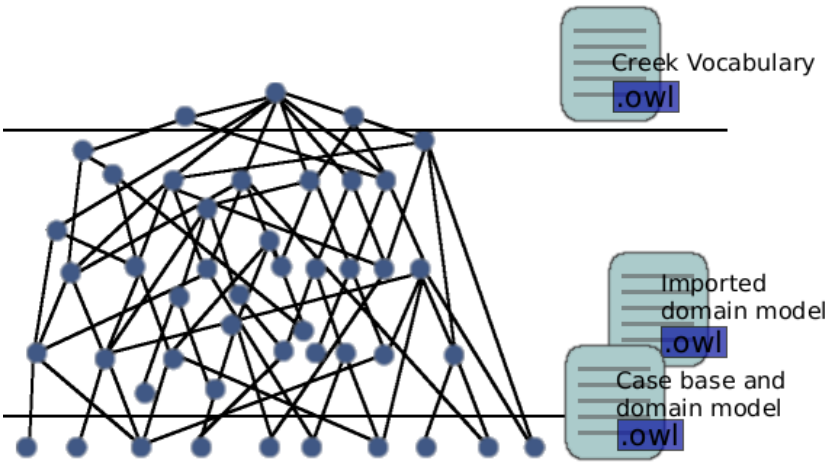

Figure 5.1: The different parts of a Creek knowledge model and how they are split into different OWL ontologies

divided in three distinct parts:

Generic concepts The top-level general concepts used internally in the reasoning and to describe and structure the knowledge model

Case base The specific knowledge of past problem situations and the corresponding solutions

General domain model The general domain knowledge used in the reasoning in Knowledge-Intensive CBR in addition to the case-base

The generic concepts form what is called the ISOPOD model. This is independent of the problem domain of the case-base and general domain knowledge, and is the same for all Creek systems. This part of the knowledge model will be *static*. The ISOPOD model also contains concepts and relations to describe and structure the case-base and general domain knowledge model.

In order to share and reuse knowledge over the Semantic Web, one need to agree upon what terms to use, what they mean and how they relate to each other. We propose an ontology based on the ISOPOD model to define a vocabulary. We call this the **Creek OWL Vocabulary**. We use the term *vocabulary* rather than *ontology* on the background that it defines terms to describe and model a knowledge base. Such a knowledge base may contain both general domain knowledge and specific knowledge in the form of cases. Using this vocabulary one can share and reuse knowledge over the Semantic Web such that any system can interpret the knowledge in a meaningful way. The Creek OWL Vocabulary is defined in section 5.2.

The **case-base and general domain model** is the knowledge that the CreekServer will share to the Semantic Web. This knowledge is shared in the form of

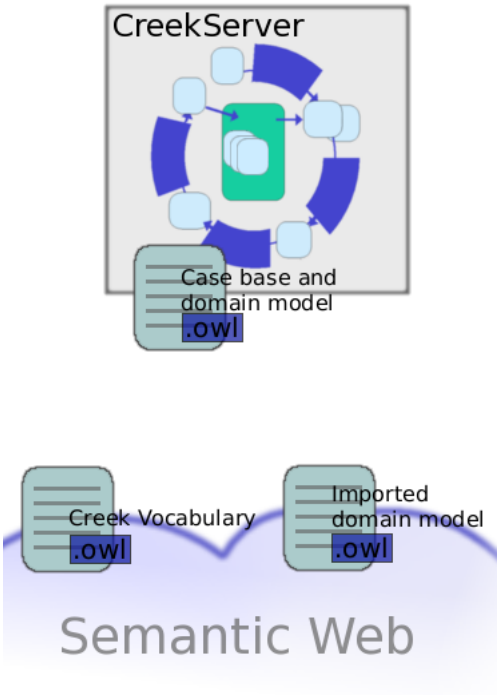

Figure 5.2: The different parts of a Creek knowledge model and where they are placed in the Semantic Web

an OWL ontology, which means that the knowledge model of the Creek system needs to be exported to OWL. This is further described in section 5.3.

Finally, the CreekServer system may **import domain models** in the form of OWL ontologies into its knowledge base. Section 5.4 goes into detail about the importing of knowledge.

Pulling the strings together, in our design we have the knowledge distributed among several OWL ontologies;

1. The Creek vocabulary
2. The case-base and domain model
3. The imported domain models

Figure 5.1 shows how these three ontologies correspond to the Creek knowledge model as described in figure 2.2. How these ontologies are organized between the Semantic Web and the CreekServer is shown in figure 5.2. All three ontologies shown are part of the knowledge used internally in the CBR process, but the Creek vocabulary and the imported domain models are defined outside of the CreekServer.

In the following three sections we go into detail of each of these parts of the proposed design.

5.2 Creek OWL Vocabulary

This section defines the Creek OWL Vocabulary. The vocabulary is available as a RDF/XML document in appendix A.1, as well as online at the location `http://creek.idi.ntnu.no/owl/creek-vocabulary`. The terms *vocabulary*, *namespace* and *document* are used in this chapter as described in section 3.2.2.

The vocabulary defined here is based on the ISOPOD model in the Creek architecture. The goal and approach for this vocabulary is very similar to the knowledge model ontology part of CBROnto described in section 4.3.2.2. As we shall see, we define some terms in this vocabulary to be equal to terms in CBROnto. This aspect is further discussed in sections 7.1.3 and 7.3.

5.2.1 Namespace

The Creek OWL Vocabulary is defined in the namespace identified by the URIref

```
http://creek.idi.ntnu.no/owl/creek-vocabulary#
```

This namespace is assigned the prefix `creek:`, which is used for the remaining of this report. For example is `creek:Case` used in place of the full URIref `http://creek.idi.ntnu.no/owl/creek-vocabulary#Case`.

The `creek:` namespace should only be used together with terms that belong to the Creek ISOPOD model.

Also, we use the prefixes `rdf:`, `rdfs:` and `owl:` as they have been used and defined in the previous chapters.

5.2.2 ISOPOD Version

The vocabulary is based on the ISOPOD model from jCreek version 0.96 build 56 from September 2005. This is stated in the ontology header using the `owl:versionInfo` property:

```
<owl:Ontology rdf:about="">
  <owl:versionInfo>0.96-56</owl:versionInfo>
</owl:Ontology>
```

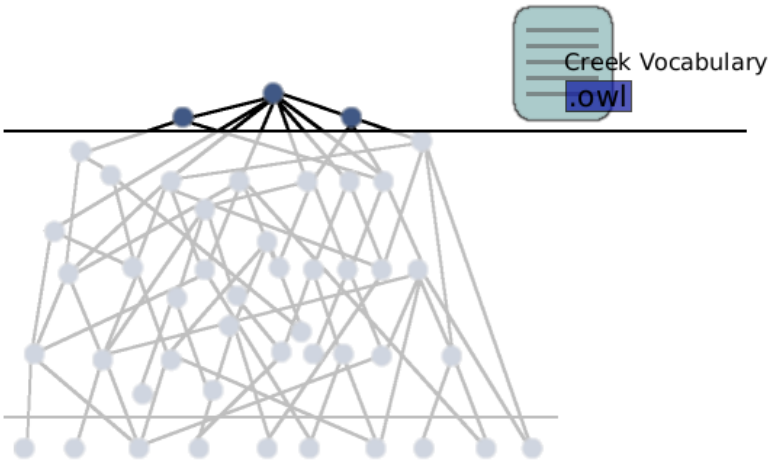

Figure 5.3: The part of a Creek knowledge model defined by the Creek OWL Vocabulary

5.2.3 An OWL Representation

The Creek OWL Vocabulary defines a set of terms — classes, individuals and properties — that are used to describe knowledge models for Creek systems. The vocabulary contains the terms from the ISOPOD model. More precise, it is an OWL *representation* of the Creek ISOPOD model. It is defined using the OWL Full variant. Reasons for are discussed in section 7.2.4. Figure 5.3 shows what portion of a Creek knowledge model that is defined by the vocabulary.

One of the key new aspects being introduced by making an OWL representation of the Creek ISOPOD model, is the URIref naming scheme. When someone referring to a `creek:Case`, there is no doubt that it is a 'case' as understood in a Creek context — intended for use in Case-Based Reasoning. The semantics of the OWL constructs, such as class hierarchies and individuals, are also well defined and should be well known. Some difficulties arise with regard to semantics in the Creek framework versus semantics in the Semantic Web. This is discussed further in section 7.2.1.

Another way to look at the vocabulary is as an informal mapping of the ISOPOD model to an OWL model. As we shall see in the next section, some of the central concepts in the ISOPOD model corresponds to OWL concepts, such as `creek:Thing` / `owl:Thing`, `creek:Concept` / `owl:Class` and `creek:Relation` / `owl:ObjectProperty`.

As stated above, this vocabulary defines the set of terms used to describe Creek knowledge models. However, these terms are general enough and the vocabulary wide enough for it to be used as a starting point to describe knowledge models for other Artificial Intelligence systems in general and other Case-Based Reasoning systems in particular. Such reuse of knowledge and vocabularies are indeed part of the motivation behind the development of the Semantic Web.

5.2.4 The Vocabulary

Here we give an informal and textual description of the more important and central parts of the vocabulary. The full formal definition is available in appendix A.1 and online. In case there are differences between what is described here, the document in appendix A.1 and/or the document available online, the online version should be considered to be correct.

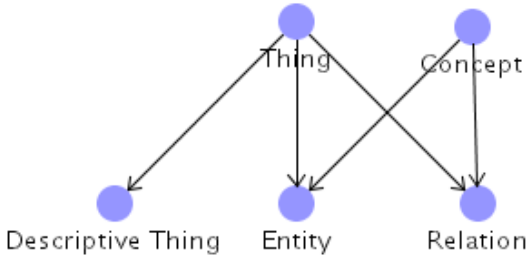

Figure 5.4: The Creek ISOPOD model top-level classes

5.2.4.1 owl:Thing and owl:Nothing

A Creek knowledge model has two top-class concepts, `creek:Thing` and `creek:Concept`. Everything in the world one wants to talk about is a subclass or an instance of `creek:Thing`. In OWL, every instance and user defined class is a member of the class `owl:Thing`. So, from a Creek system standpoint, `creek:Thing` and `owl:Thing` are the same.

Similarly, `creek:NoValue` is same in meaning as `owl:Nothing`, the empty class.

It is recommended to use the OWL classes `owl:Thing` and `owl:Nothing` rather than the two corresponding `creek:` classes.

5.2.4.2 owl:Class, rdfs:subClassOf, rdf:type

The other top-level class is `creek:Concept`. This class contains all things that can be described with a relation. `creek:Concept` is similar in meaning to `owl:Class`.

The Creek framework operates with subclasses and instances. The ISOPOD model uses the relations `creek:subclassOf` and `creek:instanceOf` to organize class hierarchies and declare instants. We recommend using the OWL constructs `rdfs:subClassOf` and `rdf:type` instead. Remember that `rdf:type` is used to declare class membership, that is to declare instances.

Further, we use the RDF/XML abbreviation for the `rdf:type` property, such that

```
<creek:Case rdf:ID="CarCase1"/>
```

states that `CarCase1` is an instance of the class `creek:Case`, and is equal in meaning to

```
<owl:Thing rdf:ID="CarCase1">
   <rdf:type
     rdf:resource="http://creek.idi.ntnu.no/owl/creek-vocabulary#Case"/>
</owl:Thing>
```

which again is similar in meaning to

```
<creek:Thing rdf:ID="CarCase1">
   <creek:instanceOf
     rdf:resource="http://creek.idi.ntnu.no/owl/creek-vocabulary#Case"/>
</creek:Thing>
```

5.2.4.3 creek:Entity

`creek:Entity` is a subclass of `creek:Thing` and `creek:Concept`. An entity corresponds to physical or abstract entities in the real world. An entity is something having relations to it and from it describing it. The current implementation of Creek allows for procedures and attachments in the form of Java classes represented as entities. This feature is not supported in the OWL representation.

`creek:Entity` is the superclass of the following classes (which again have various subclasses and instances. See the full ontology for details): `creek:Context`, `creek:Environment`, `creek:Role`, `creek:Situation`, `creek:PhysicalObject`, `creek:MentalObject`, `creek:State`, `creek:Task` and `creek:Case`.

5.2.4.4 creek:Case

`creek:Case` is the class of all cases. In a Creek reasoning system, a case describes a solution to a previously solved problem. In a knowledge model the cases are the specific knowledge, as opposed to the general domain knowledge, and is used in the problem-solving process of a Case-Based Reasoning system.

We define the class `creek:Case` to be the same class as `cbronto:Case`:

```
<owl:Class rdf:ID="Case">
  <owl:equivalentClass
      rdf:resource="http://gaia.sip.ucm.es/cbronto.owl#Case"/>
</owl:Class>
```

Cases that are modelled or represented in a knowledge model are instances of `creek:Case`. Typically, a case is described by a set of relations to concepts from

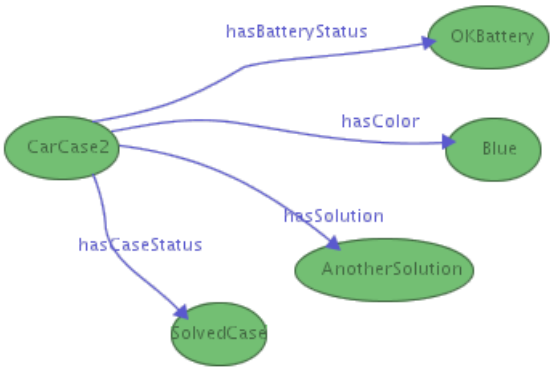

Figure 5.5: Graph of example case

Listing 5.1: RDF/XML notation of the graph in figure 5.5

```
<creek:Case rdf:ID="CarCase2">
  <rdfs:label>This is a description of car problem</rdfs:label>
  <hasColor rdf:resource="#Blue"/>
  <hasEngineStatus rdf:resource="#EngineNotStarting"/>
  <hasBatteryStatus rdf:resource="#OKBattery"/>
  <creek:hasSolution rdf:resource="#AnotherSolution"/>
  <creek:hasCaseStatus rdf:resource="http://creek.idi.ntnu.no\\
/owl/creek-vocabulary#SolvedCase"/>
</creek:Case>
```

the general domain knowledge. These relations can themselves be defined in the general domain knowledge, or one might use the generic `creek:hasFinding` relation

```
<owl:ObjectProperty rdf:ID="hasFinding">
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
</owl:ObjectProperty>
```

The concept, class or instance that are the solution of the case is described by the `creek:hasSolution` relation

```
<owl:ObjectProperty rdf:ID="hasSolution">
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
  <owl:equivalentProperty
      rdf:resource="http://gaia.sip.ucm.es/cbronto.owl#hasSolution"/>
</owl:ObjectProperty>
```

This relation is defined to be the same as `cbronto:hasSolution` defined in CBROnto.

Further, a case has a status described by the `hasCaseStatus` relation. It is

either creek:SolvedCase, creek:ProcessedCase or creek:UnsolvedCase.

```
<owl:ObjectProperty rdf:ID="hasCaseStatus">
  <rdf:type
      rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
  <rdfs:range rdf:resource="#CaseStatus"/>
  <rdfs:domain rdf:resource="#Case"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="CaseStatus">
  <rdfs:subClassOf rdf:resource="#Symbol"/>
</owl:Class>

<CaseStatus rdf:ID="SolvedCase"/>
<CaseStatus rdf:ID="ProcessedCase"/>
<CaseStatus rdf:ID="UnsolvedCase"/>
```

If no status is specified and there is no hasSolution relation, one should consider the case unsolved.

Cases can be organized using the creek:hasSubcase relation:

```
<owl:ObjectProperty rdf:ID="hasSubcase">
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
</owl:ObjectProperty>
```

Figure 5.5 and listing 5.1 shows the RDF graph and RDF/XML notation of an example creek:Case. The graph is shown without using full URIrefs. Here hasColor, hasEngineStatus and hasBatterystatus are relations from some given general domain model, and Blue, EngineNotStarting, OKBattery and AnotherSolution are concepts (classes or individuals) from the same general domain model.

5.2.4.5 creek:DescriptiveThing

creek:DescriptiveThing is a subclass of creek:Thing. It is the class of things that represents how other things are described. Included in this in the Creek ISOPOD model is classes for datatypes such as text and numbers. We recommend to use the datatypes defined by XML Schema Datatypes (XSD) the same way OWL uses these atomic datatypes. More complex datatypes may be better represented using terms from the creek: namespace or other vocabularies.

5.2.4.6 creek:Relation, creek:RelationInverse and owl:inverseOf

The class creek:Relation is the superclass of all relations in a Creek knowledge model. As seen in figure 5.4, in the ISOPOD model creek:Relation is a

subclass of the `creek:Thing` and `creek:Concept` classes. In this vocabulary we define `creek:Relation` to be both a subclass of the two top-level classes and an instance of `owl:ObjectProperty`:

```
<owl:Class rdf:ID="Relation">
   <rdfs:subClassOf rdf:resource="#Thing"/>
   <rdfs:subClassOf rdf:resource="#Concept"/>
   <rdf:type
       rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
</owl:Class>
```

All relations in Creek have an inverse relation. The super-property of these are the property `creek:RelationInverse`;

```
<owl:ObjectProperty rdf:ID="RelationInverse">
  <owl:inverseOf rdf:resource="#relation"/>
</owl:ObjectProperty>
```

The property `creek:hasInverse` is used to declare a property the inverse of another. We recommend using the OWL built-in `owl:inverseOf` property.

```
<owl:ObjectProperty rdf:ID="hasInverse">
  <rdf:type
      rdf:resource="http://www.w3.org/2002/07/owl#SymmetricProperty"/>
</owl:ObjectProperty>
```

The plausible inheritance reasoning described in section 2.3 can be modelled using the `creek:transfers` and `creek:inheritsOver` properties;

```
<owl:ObjectProperty rdf:ID="transfers">
  <rdfs:subPropertyOf rdf:resource="#systemRelationInverse"/>
  <rdfs:subPropertyOf rdf:resource="#isopodRelation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="inheritsOver">
  <rdfs:subPropertyOf rdf:resource="#systemRelationInverse"/>
  <rdfs:subPropertyOf rdf:resource="#isopodRelationInverse"/>
  <owl:inverseOf rdf:resource="#transfers"/>
</owl:ObjectProperty>
```

`owl:inverseOf`, `creek:transfers` and `creek:inhertitsOver` are all properties that apply to other properties. Properties in RDF and OWL are binary, in that they describe a relationship between two resources. Relations or properties in Creek may have other information attached to them, such as their weight.

N-ary relations may be modelled using an anonymous `creek:Relation` node. This way several values may be tied together. When using such structured values, the `rdf:value` property is used to indicate the 'main' value for the relation.

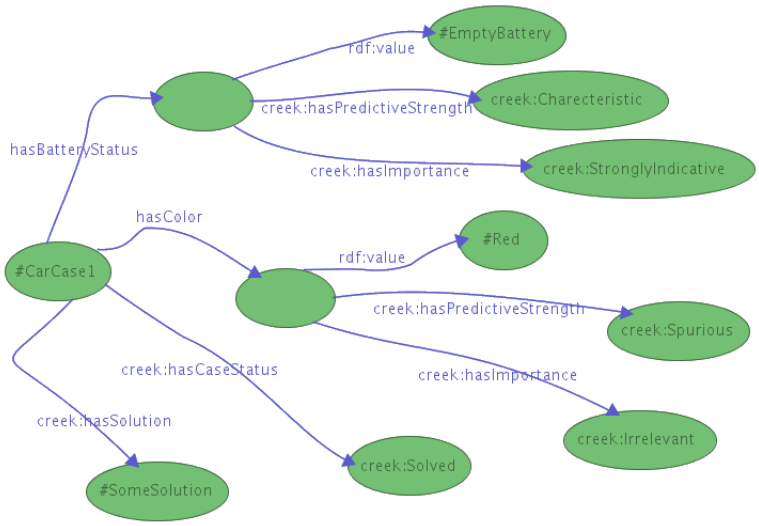

Figure 5.6: Graph of example case using anonymous `Relation` node

Listing 5.2: RDF/XML notation of the graph in figure 5.6

```
<creek:Case rdf:ID="CarCase1">
  <hasColor>
    <creek:Relation>
      <creek:hasImportance rdf:resource="http://creek.idi.ntnu.no\\
/owl/creek-vocabulary#Irrelevant"/>
      <creek:hasPredictiveStrength rdf:resource="http://creek.idi\\
.ntnu.no/owl/creek-vocabulary#Spurious"/>
      <rdf:value rdf:resource="#Red"/>
    </creek:Relation>
  </hasColor>
  <hasBatteryStatus>
    <creek:Relation>
      <creek:hasImportance rdf:resource="http://creek.idi.ntnu.no\\
/owl/creek-vocabulary#StronglyIndicative"/>
      <creek:hasPredictiveStrength rdf:resource="http://creek.idi\\
.ntnu.no/owl/creek-vocabulary#Charecteristic"/>
      <rdf:value rdf:resource="#EmptyBattery"/>
    </creek:Relation>
  </hasBatteryStatus>
  <creek:hasSolution rdf:resource="#SomeSolution"/>
  <creek:hasCaseStatus  rdf:resource="http://creek.idi.ntnu.no/owl\\
/creek-vocabulary#SolvedCase"/>
</creek:Case>
```

Figure 5.6 shows an RDF graph of a `creek:Case` adding more information to the findings relations. Listing 5.2 shows how this is modelled in RDF/XML notation. This way to describe n-ary relations is the method used for 'Use Case 1' described in (Noy and Rector, 2004).

A relation can be weighted by giving them a `creek:Importance` and/or a `creek:PredictiveStrength` strengths. These are defined as

```
<owl:Class rdf:ID="Importance">
  <rdfs:subClassOf rdf:resource="#Symbol"/>
</owl:Class>
<Importance rdf:ID="Necessary"/>
<Importance rdf:ID="Characteristic"/>
<Importance rdf:ID="Informative"/>
<Importance rdf:ID="DefaultImportance"/>
<Importance rdf:ID="Irrelevant"/>

<owl:ObjectProperty rdf:ID="hasImportance">
  <rdfs:range rdf:resource="#Importance"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="PredictiveStrength">
  <rdfs:subClassOf rdf:resource="#Symbol"/>
</owl:Class>
<PredictiveStrength rdf:ID="Sufficient"/>
<PredictiveStrength rdf:ID="StronglyIndicative"/>
<PredictiveStrength rdf:ID="DefaultPredictiveStrength"/>
<PredictiveStrength rdf:ID="Indicative"/>
<PredictiveStrength rdf:ID="Spurious"/>

<owl:ObjectProperty rdf:ID="hasPredictiveStrength">
  <rdfs:range rdf:resource="#PredictiveStrength"/>
</owl:ObjectProperty>
```

These textual weights are translated internally to a numerical weight between 0 and 1 in a Creek knowledge model.

The Creek OWL vocabulary defines other classes of properties such as;
`creek:CausalRelation`, `creek:associationalRelation`,
`creek:structuralRelation`, `creek:functionalRelation`,
`creek:temporalRelation` and `creek:spatialRelation`.
See the full ontology for further details.

5.2.4.7 Restrictions

The OWL constructs for declaring restrictions and defining classes using restrictions are not currently supported by the Creek framework. More on this in section 7.2.3.

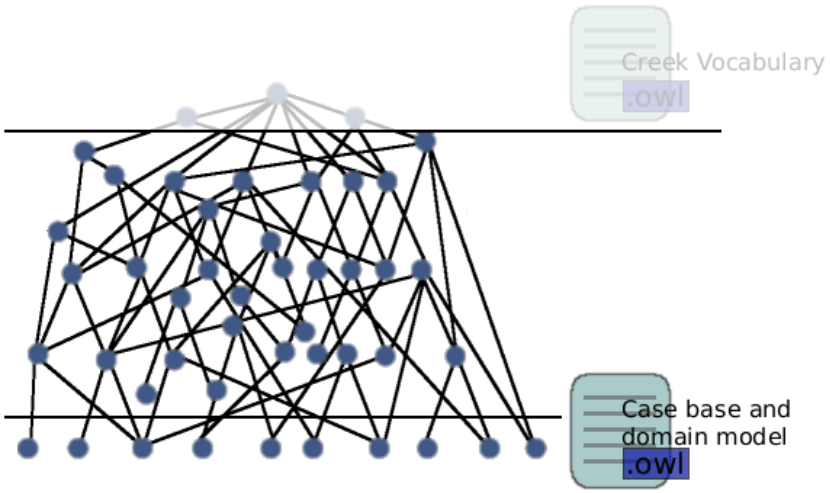

Figure 5.7: The part of a Creek knowledge model being exported

5.3 Sharing Knowledge

In this section we go into the details of our design concerning sharing of knowledge.

As described in section 5.1 above, a CreekServer system has an internal knowledge model on which the reasoning is based. This is the knowledge we want to share — using the Semantic Web as the medium. In order to share the knowledge we export the the internal knowledge model of the CreekServer to an OWL ontology.

5.3.1 Relation with the ISOPOD Model

The ISOPOD part of the knowledge model is already defined in the Creek OWL Vocabulary. The concepts and relations that is defined here will not be exported. Figure 5.7 shows what parts of the knowledge model that is being exported to OWL. Still, it is important to remember that the ISOPOD model is part of the *internal* knowledge model of the CreekServer. In our design we make this clear by using the `owl:imports` construct in the ontology header of the ontology we export;

```
<owl:Ontology rdf:about="">
    <owl:imports
       rdf:resource="http://creek.idi.ntnu.no/owl/creek-vocabulary#"/>
</owl:Ontology>
```

Note that it is not necessary to import an ontology in order to use and/or refer to resources defined in it. It is perfectly legal to define a case in some ontology

like

```
<creek:Case rdf:ID="Case1">
  <creek:hasFinding rdf:resource="#EngineRunning"/>
  <creek:hasFinding rdf:resource="#BatteryLow"/>
  <creek:hasCaseStatus
      rdf:resource="http://creek.idi.ntnu.no/owl/creek-vocabulary#Unsolved"/>
</creek:Case>
```

without the `creek:` vocabulary being imported. It is sufficient to define the `creek:` prefix and namespace. By using `owl:imports`, all the concepts and relations in the importee ontology will also be part of the importer ontology. Given the tight integration in Creek between the generic concepts, the general domain knowledge and the cases, it is natural to explicitly make the Creek OWL ontology part of the knowledge we share.

5.3.2 An OWL Representation

We use the Semantic Web as the medium for sharing the knowledge from the CreekServer system — in the form of an OWL ontology.

Like other OWL ontologies, the knowledge shared by the CreekServer system is defined in a **namespace**. The namespace is `http://creek.idi.ntnu.no/owl/example/car-model.owl#`. Remember that we use the ontology from appendix A.2 as our knowledge model. We assign the prefix `km:` to this namespace. Other similar Creek systems will of course have different namespaces.

We have to translate our knowledge in the CreekServer from the semantic net/frame representation used internally in Creek to an OWL representation used within the Semantic Web. We need to make some choices for how the translation takes place. When we represent our knowledge model in OWL, we follow these rules:

- Represent all Creek concepts as `owl:Class`'es or individuals
- Represent all Creek relations as `owl:ObjectProperty`'s
  - Represent `creek:subclassOf` relations as `rdfs:subClassOf` or `rdfs:subPropertyOf` properties
  - Represent `creek:instanceOf` relations as `rdf:type` properties
  - Represent `creek:hasInverse` relation as `owl:inverseOf` property
- Represent Creek entities that represents literals as XML literals

The reason for using the `owl:` constructs is because when we share our knowledge model, we want to make it as accessible and general as possible. As these constructs are integral parts of OWL they are in general more familiar and

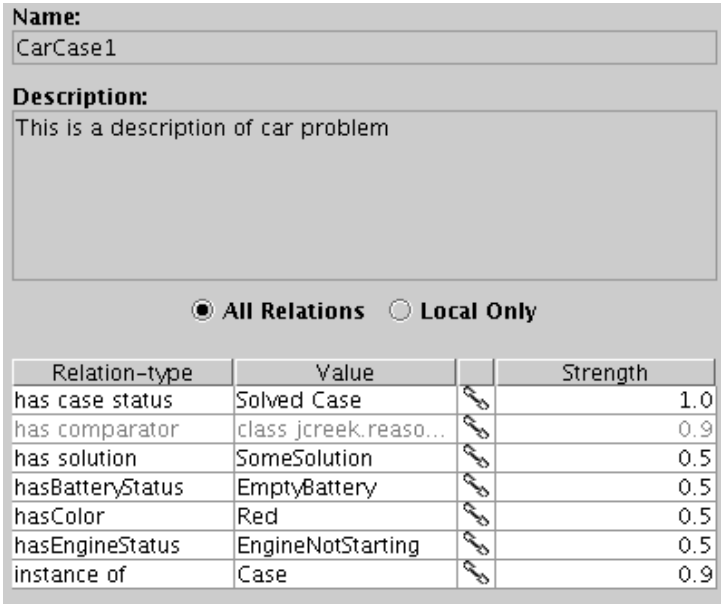

Figure 5.8: Frame view from TrollCreek of CarCase1

Listing 5.3: RDF/XML notation of CarCase1 from figure 5.8

```
<creek:Case rdf:ID="CarCase1">
  <rdfs:label>This is a description of car problem</rdfs:label>
  <km:hasColor rdf:resource="#Red"/>
  <km:hasEngineStatus rdf:resource="#EngineNotStarting"/>
  <km:hasBatteryStatus  rdf:resource="#EmptyBattery"/>
  <creek:hasSolution rdf:resource="#SomeSolution"/>
  <creek:hasCaseStatus rdf:resource="http://creek.idi.ntnu.no/owl/
creek-vocabulary#SolvedCase"/>
</creek:Case>
```

more widely used than the corresponding Creek ones. The semantics of, say `rdfs:subClassOf`, should be well known as opposed to `creek:subclassOf`. These Creek constructs makes most sense within a Knowledge-Intensive CBR context, whereas the OWL ones is used throughout the Semantic Web.

This seems simple enough. But given the different backgrounds of Creek and OWL, some of the OWL primitives we use have somewhat different semantics than the Creek primitives we substitute them with. This is further discussed in section 7.2.1.

Figure 5.8 shows a frame view of a Creek case from the `km:` ontology. The RDF/OWL representation of this is shown in listing 5.3. Note that here all properties for the `km:CarCase1` individual is clustered together, but that an ontology model can be serialized to different RDF/OWL representations.

The last point we make here is that in our design, every entity and relation in the knowledge model of the CreekServer is identified by an URIref. The case from

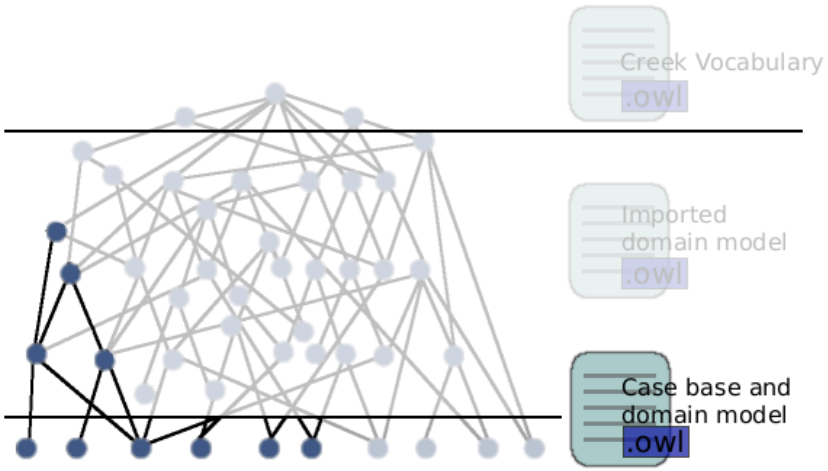

Figure 5.9: The different parts of a Creek knowledge model after importing another OWL domain model

listing 5.3 is universally identified by the URIref `km:CarCase1`, as the relations `km:hasColor` and `km:hasEngineStatus` and the entities `km:EmptyBattery` and `km:Somesolution` and so on are identified by their URIrefs.

5.4 Reusing Knowledge

The last part of our proposed design is about reuse of knowledge. Say for example we want to use the knowledge about car engines from an ontology `http://someURI/engine.owl#`. We would want to import this into the knowledge base of our CreekServer.

Conceptually this ontology is imported into CreekServer the same way as the `creek:` vocabulary:

```
<owl:Ontology rdf:about="">
    <owl:imports
       rdf:resource="http://creek.idi.ntnu.no/owl/creek-vocabulary#"/>
    <owl:imports
       rdf:resource="http://someURI/engine.owl#"/>
</owl:Ontology>
```

We now have knowledge from three different sources in CreekServer, each identified by a namespace; the Creek Vocabulary from `http://creek.idi.ntnu.no/owl/creek-vocabulary#`, the domain knowledge and case-base from `http://creek.idi.ntnu.no/owl/example/car-model.owl#`, and the engine ontology from `http://someURI/engine.owl#`.

Listing 5.4: Concepts from the `km:` ontology not attached to the Creek ISOPOD model

```
<owl:Class rdf:ID="Car">
  <rdfs:label>a car</rdfs:label>
</owl:Class>

<owl:ObjectProperty rdf:ID="requires">
  <creek:inheritsOver rdf:resource="#hasPart"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="requiredBy">
  <owl:inverseOf rdf:resource="#requires"/>
</owl:ObjectProperty>
```

The case-base and domain model is a combination of the to last parts. Figure 5.9 shows how these three ontologies are organised in the knowledge model of the CreekServer.

As when exporting, we need to make some choices for how to actually do the translation from the ontology we import to the representation used internally in Creek:

- Represent all classes and individuals as Creek Entities
  - Use the Creek 'Symbol' Entity as superclass if none is defined
- Represent all properties as new Creek RelationTypes
  - Use the Creek 'has finding' RelationType as superclass if none is defined
  - Use the Creek 'instance of', 'subclass of' and 'has inverse' for the corresponding OWL constructs

In Creek, all the entities and the relation types in the knowledge model must be connected to the ISOPOD model. That is, they must all be subclasses, members or sub-properties of entities in the ISOPOD model. Classes and properties in a given ontology may explicitly be defined to be so (by the use of the Creek OWL Vocabulary), but in general they will have no relation to the ISOPOD model. In that case, we represent a class as a subclass of the Creek *'Symbol'* Entity, and a property as a subclass of the Creek *'has finding'* RelationType. This is not a random choice. The Symbol entity has special comparators attached to it. These are used in the reasoning during the CBR process in Creek. Thus in order for Creek to use such resource in for example case matching, we need to define them as subclasses of 'Symbol'. As for the 'has finding', all relation types used for describing a case must be a sub-property of this. It is likely that we want to use some of the properties defined in an ontology we import to describe cases.

Listing 5.4 shows part of the `km:` ontology with concepts that are not related to the ISOPOD model. Figure 5.10 shows how these are integrated into the ISOPOD model after importing this ontology.

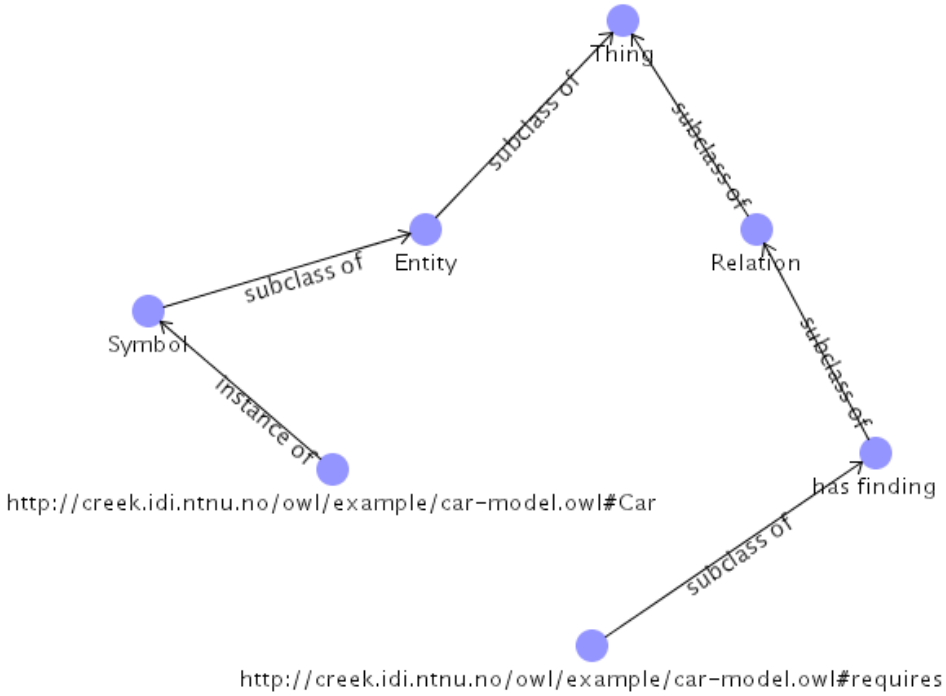

Figure 5.10: The `km:Car` class and `km:requires` property from listing 5.4 attached to the ISOPOD model

Chapter 6

Implementation

> *The Kuang program spurted from tarnished cloud. Case's consciousness divided like the beads of mercury, arcing above an endless beach the color of the dark silver clouds. His vision was spherical, as though a single retina lined the inner surface of a globe that contained all things, if all things could be counted.* William Gibson, Neuromancer

This chapter describes the proof-of-concept implementation of the design proposed in the previous chapter. The three sections 5.2 - 5.4 describing the design each have an according section in this chapter describing the implementation done.

The implementation uses APIs from the jCreek and Jena packages. jCreek is the Java implementation of the Creek framework. Jena [1] (McBride, 2001) is an open source Java framework for Semantic Web functionality developed by HP Labs [2]. It includes among other things a RDF API and an OWL API. These again consist of classes such as *OntModel*, *Ontology*, *OntResource*, *OntClass*, *OntProperty* and *Individual* to handle the according OWL concepts. *OntModel* is a class for representing a knowledge model in the form of an OWL ontology. *OntResource* is a representation of a Semantic Web resource, that is something identified by an URIref, and is the superclass of *OntClass*, *OntProperty* and *Individual*.

All the three Java classes described in the following subsections belong to the `jcreek.representation.sw` package in jCreek. In addition, two other Java classes are created to handle the GUI and interaction with TrollCreek [3]:

- `jcreek.cke.importexport.owl.OWLexport`
- `jcreek.cke.importexport.owl.OWLimport`

[1] Jena, `http://jena.sourceforge.net/`
[2] HP Labs, `http://www.hpl.hp.com/semweb/`
[3] TrollCreek is a GUI front-end for knowledge modelling in jCreek

The code for these last two classes is mostly copied from the corresponding classes handling the import and export of the JCXML format (Tomassen, 2002). These classes are not described further in this chapter, but their JavaDocs are included in appendix B.2.

Minor additions are done to the `jcreek.cke.importexport.ImportExportsWizards` class to handle the OWL functionality. This is not included in this report.

6.1 Vocabulary

Jena includes `schemagen`, an utility that makes a Java class out of a vocabulary. This is used to convert the `creek:` vocabulary defined in section 5.2 into the *CREEK* class. This contains static constants for all the OWL resources in the vocabulary. For example is the `creek:Case` resource defined as

```
   public static final OntClass Case =  m_model.createClass(
"http://creek.idi.ntnu.no/owl/creek-vocabulary#Case" );
```

These constants can be used like for example

```
if (swProperty.equals(CREEK.instanceOf))
      ...
```

to check if the object `swProperty` (an instance of *OntResource*) represents the same OWL or Semantic Web resource as `creek:instanceOf`.

Jena includes similar classes for the `rdf:`, `rdfs:` and `owl:` vocabularies.

The JavaDoc for the `jcreek.representation.sw.CREEK` class can be found in appendix B.1.1.

6.2 Export

The sharing of knowledge is implemented as an extension to TrollCreek. The internal knowledge model is exported to an OWL ontology, which is saved to a file.

The algorithm for exporting a Creek knowledge model to an OWL ontology is as follows:

1. Create a new (Jena) OntModel
2. Add an (Jena) Ontology
3. Set metadata (such as author, name, description) for the (Jena) Ontology

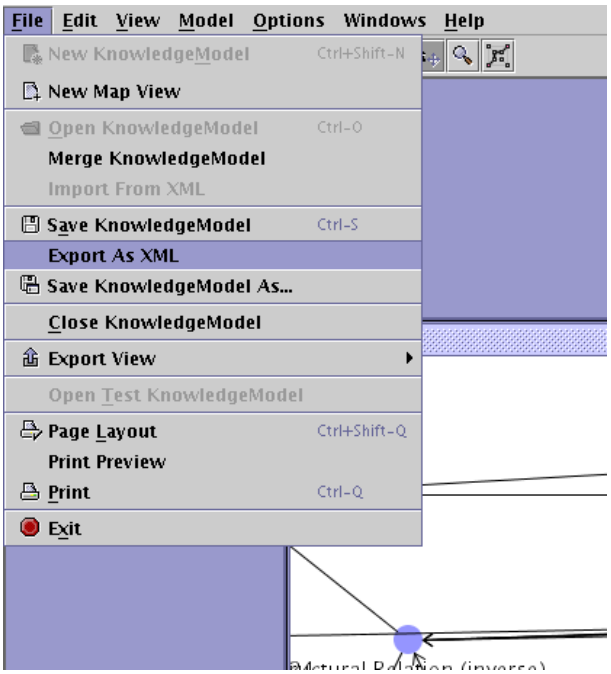

Figure 6.1: Exporting to OWL: Select 'Export as XML'

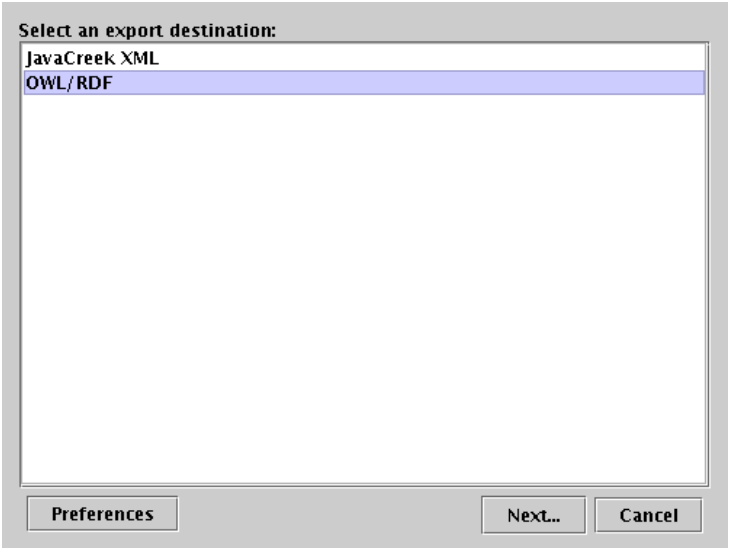

Figure 6.2: Exporting to OWL: Select 'OWL/RDF'

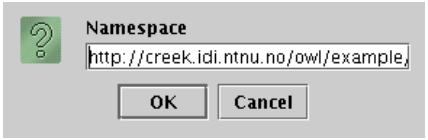

Figure 6.3: Exporting to OWL: Namespace input

4. FOR ALL (jCreek) Entities in the (jCreek) KnowledgeModel not defined in the `creek:` vocabulary DO
   (1) IF (jCreek) Entity is a subclass/instance of (jCreek) Relation THEN
      1. add as (Jena) ObjectProperty to the (Jena) Ontology
      2. set the `owl:inverseOf`, `owl:subPropertyOf`, `creek:hasDefaultExplanationStrength`, `rdfs:label` and `rdfs:comment` properties.
   (2) ELSE
      1. add as (Jena) OntClass to the (Jena) Ontology
      2. add `rdfs:label` and `rdfs:comment` properties to the (Jena) OntClass
      3. FOR ALL (jCreek) localRelations from this (jCreek) Entity DO
         1. IF Relation == `creek:instanceOf` THEN
            - add `rdf:type` property to the (Jena) OntClass
            - change the (Jena) OntClass to (Jena) Individual
         2. ELSE IF (jCreek) Relation == `creek:subClassOf` THEN
            - add `rdfs:subClassOf` property to the (Jena) OntClass/Individual
         3. ELSE IF the inverse (jCreek) Relation is not added
            - add the (jCreek) Relation as (Jena) ObjectProperty along with its value to the (Jena) OntClass/Individual
5. Write the (Jena) OntModel to file

The export functionality is made available through the TrollCreek Knowledge Editor. The procedure for exporting to OWL is the same as when exporting to the JCXML format.

To create an OWL ontology from a knowledge model, select *'Export as XML'* from the *'File'* menu, then select the *'OWL/RDF'* format and the file to write the exported OWL ontology to. Finally, give an URIref that will be the namespace for the OWL ontology. Figures 6.1 and 6.2 show the first two steps. Figure 6.3 shows the namespace input dialog.

The export functionality is implemented in the `jcreek.representation.sw.OWLexportParser` class, whose JavaDoc can be found in appendix B.1.2.

6.3 Import

The reuse of knowledge is, as the sharing of knowledge described above, implemented as an extension to TrollCreek. This takes place as an import of an OWL ontology into a Creek knowledge model. The ontology to import is identified by an URI.

Below is the algorithm for importing an OWL ontology:

1. Create a new (jCreek) CBRModel
2. Load the importing ontology URI into a (Jena) OntModel
3. Gather the metadata from the (Jena) Ontology
4. FOR ALL (Jena) OntResources (classes and individuals) in the (Jena) OntModel DO
   (1) IF (Jena) OntResource is not part of the ISOPOD model THEN
      1. IF (Jena) OntResouce is instance of `creek:Case`
         - Add (Jena) OntResource as (jCreek) Case to the (jCreek) CBRModel
      2. ELSE
         - Add (Jena) OntResource as (jCreek) Entity to the (jCreek) CBRModel
      3. Add `rdfs:label` and `rdfs:comment` as (jCreek) Description to the (jCreek) Entity
      4. FOR ALL (Jena) OntProperties from this (Jena) OntResource DO
         1. IF (Jena) OntProperty == `rdfs:subclassOf` THEN
            - add as (jCreek) 'subclass of' RelationType
         2. ELSE IF (Jena) OntProperty == `rdf:type` THEN
            - add as (jCreek) 'instance of' RelationType (if it is an instance of `owl:Class`, add as instance of (jCreek) Symbol Entity)
         3. ELSE
            - add OntProperty as new (jCreek) RelationType, along with the (jCreek) inverse RelationType and (jCreek) RelationType superclass (if no superclass, set `creek:hasFinding` as superclass)
            - add as this (jCreek) RelationType

Like the export functionality, the procedure for importing an OWL ontology is the same as for importing from JCXML. In the TrollCreek Knowledge Editor, select *'Import from XML'* from the *'File'* menu, then select the *'OWL/RDF'* format The OWL ontology to import is decided by giving its URI. This can be an online resource such as `http://creek.idi.ntnu.no/owl/example/car-model.owl`, or some local file like for example `file:///somefolder/car-model.owl`.

Figures 6.4, 6.5 and 6.6 show the GUI of these steps.

As this is a proof-of-concept implementation, not all OWL constructs are supported. Anonymous nodes and certain OWL constructs such as restrictions and set operators are not handled.

The class `jcreek.representation.sw.OWLimportParser` handles the importing. Appendix B.1.3 contains the JavaDoc for this class.

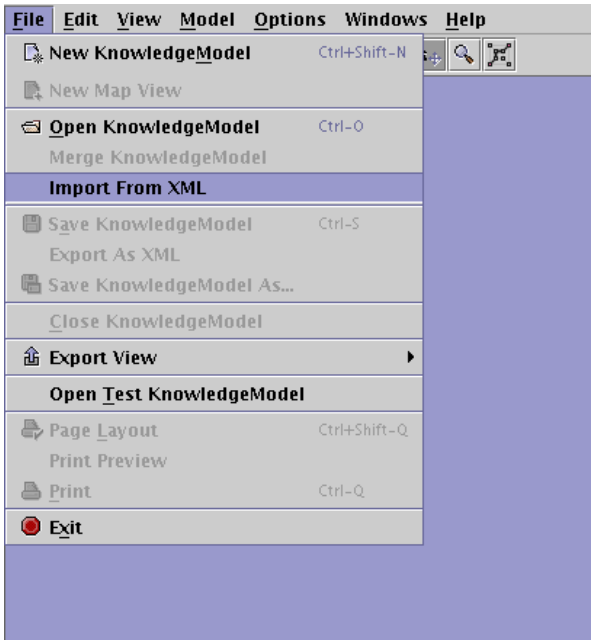

Figure 6.4: Importing from OWL: Select 'Import from XML'

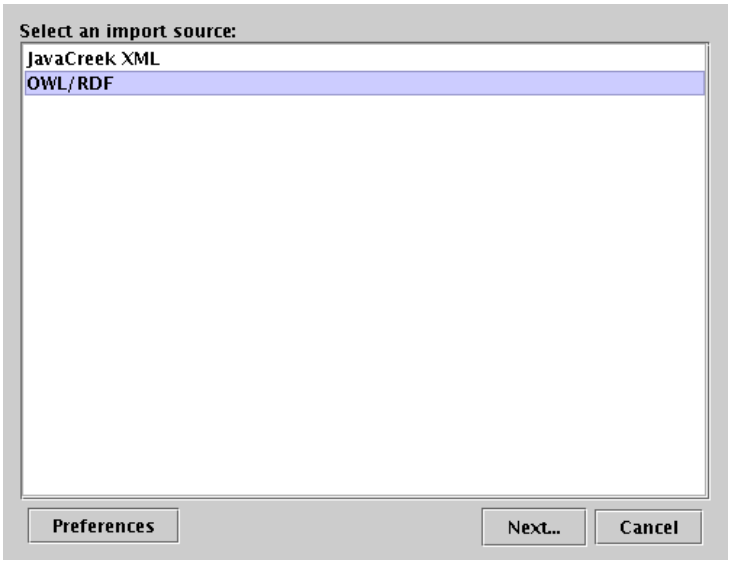

Figure 6.5: Importing from OWL: Select 'OWL/RDF'

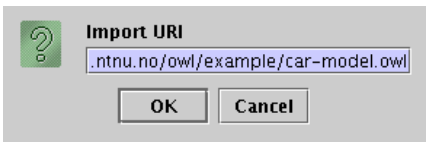

Figure 6.6: Importing from OWL: URI for ontology

Chapter 7

Evaluation and Discussion

> *'An artificial intelligence, but you know that. Your mistake, and it's quite a logical one, is in the confusing the Wintermute mainframe, Berne, with the Wintermute entity.'* William Gibson, Neuromancer

In this final chapter we start by evaluating the result from in chapter 5 with regard to distributed knowledge and Case-Based Reasoning. The result is compared to the other related work presented in chapter 4. Further, we will discuss some limitations to the approach and then some of the difficulties that arise. We finish this report by summarizing and concluding the work done and suggest some interesting areas for further research.

7.1 Evaluation

7.1.1 Distributed Knowledge

From a Creek perspective, the work done in chapter 5 introduces some new ideas and concepts:

- The ISOPOD model is defined as a **vocabulary** identified by a namespace
- The knowledge base itself is identified by a **namespace**
- All concepts and relation types in the knowledge model will be universally identified by an **URIref**
- By consequence, all **cases** in the case-base is also identified by an unique URIref
- Ontologies from other namespaces might be **imported** into the Creek knowledge model

All these aspects play a part when it comes to the theme of this thesis — *distributed knowledge*. The knowledge model of the CreekServer system from section 5.1 is made up of knowledge from at least two sources; from the `creek:` and `km:` namespaces (we use the same prefixes as in the previous chapters). The possible imported ontologies come in addition.

The ISOPOD part of a Creek knowledge model is defined in an ontology. This knowledge will be reused by all Creek knowledge bases that are exported to OWL. The vocabulary is identified by the `creek:` namespace. All concepts and relations here are universally identifiable. For example is the concept of a *case* identified by the URIref `creek:Case`. This, and all the other terms defined in the vocabulary, may be referred to and used by other ontologies or vocabularies. More importantly, there is a common understanding and agreement of the terms, and how they are used in modelling the knowledge and case base. It is this agreement that ensure that knowledge can be shared between systems using the `creek:` vocabulary.

Similarly, the knowledge base for some specific Creek system is identifiable by a namespace, that is an URIref. The knowledge base for the CreekServer is identified by the `km:` namespace. Also, all concepts and relations defined in this knowledge base have their own URIref. The URIref `km:CarCase1` conceptually refers to the following case and its findings:

```
<creek:Case rdf:ID="CarCase1">
  <hasColor rdf:resource="#Red"/>
  <hasEngineStatus rdf:resource="#EngineNotStarting"/>
  <hasBatteryStatus  rdf:resource="#EmptyBattery"/>
  <creek:hasSolution rdf:resource="#SomeSolution"/>
  <creek:hasCaseStatus rdf:resource="http://creek.idi.ntnu.no/owl/
creek-vocabulary#SolvedCase"/>
</creek:Case>
```

Another Creek system, with a knowledge model defined in some `cb:` namespace, can refer to and use this specific case. Given a concept `cb:SomeConcpet`

```
<owl:Class rdf:ID="SomeConcept">
   <creek:reminding rdf:resource="http://creek.idi.ntnu.no/owl/example/
car-model.owl#CarCase1"/>
</owl:Class>
```

`km:CarCase1` is a *reminder* for `cb:SomeConcept`. The system using the `cb:` knowledge model might choose to import the whole or parts of the `km:` ontology.

The export and import of OWL ontologies further integrate our approach with the distributed nature of the Semantic Web.

Figure 7.1: `km:CarCase3` matched against `km:CarCase2` in TrollCreek

Listing 7.1: km:CarCase3 and km:CarCase2

```
<creek:Case rdf:ID="CarCase2">
  <hasColor rdf:resource="#Blue"/>
  <hasEngineStatus rdf:resource="#EngineNotStarting"/>
  <hasBatteryStatus rdf:resource="#BatteryOK"/>
  <creek:hasSolution rdf:resource="#AnotherSolution"/>
  <creek:hasCaseStatus rdf:resource="http://creek.idi.ntnu.no/owl/
creek-vocabulary#SolvedCase"/>
</creek:Case>

<creek:Case rdf:ID="CarCase3">
  <hasColor rdf:resource="#Green"/>
  <hasEngineStatus rdf:resource="#EngineNotStarting"/>
  <hasBatteryStatus rdf:resource="#LowBattery"/>
  <creek:hasCaseStatus rdf:resource="http://creek.idi.ntnu.no/owl/
creek-vocabulary#UnsolvedCase"/>
</creek:Case>
```

7.1.2 CBR with Semantic Web Resources

In this section we will show that Creek reasoning can be done on knowledge based on Semantic Web resources. We do a case matching, and show an example of the plausible inheritance reasoning method.

Again we use the km: ontology from appendix A.2. The reasoning is done after this ontology is imported into a knowledge model in TrollCreek as described in section 6.3.

Two cases from this ontology and their findings are shown in listing 7.1. Figure 7.1 shows km:CarCase3 matched against km:CarCase2 done in TrollCreek. The result is a 33% match, which is due to the common km:hasEngineStatus km:EngineNotStarting property.

Default inheritance is a reasoning mechanism in Creek where properties can be inherited over other relations than the usual subclass and instance relations. The creek:inheritsOver and creek:transfers properties are used in Creek to realize this. Listing 7.2 shows a part of our km: ontology. km:hasPart is defined to be inherited over the km:requires relation. Given the structure of the km:Car, km:Engine and km:Oil classes, Creek concludes that there is a km:Car km:requires km:Oil relation. Figure 7.2 shows the dialog in TrollCreek explaining this inherited relation.

These are two very simple examples of the reasoning possible within Creek. It is presented here to show that the reasoning from Semantic Web resources works out as one would expect from a Creek point of view.

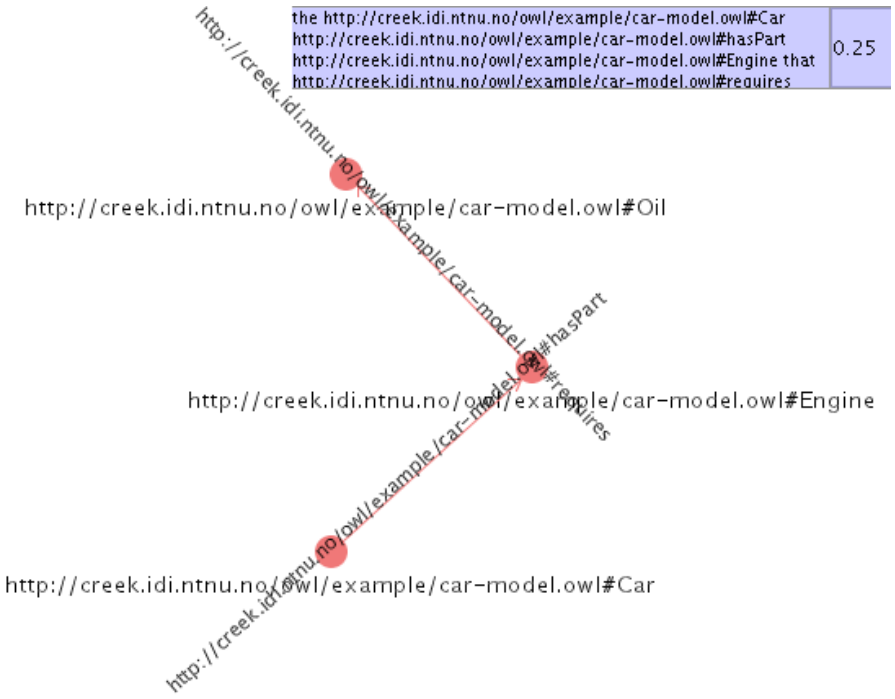

Figure 7.2: The `km:Car` `km:requires` `km:Oil` relation explained

Listing 7.2: Plausible inheritance in the `km:` ontology

```
<owl:Class rdf:ID="Car">
   <hasPart>
      <owl:Class rdf:ID="Engine">
         <requires>
            <owl:Class rdf:ID="Oil"/>
         </requires>
      </owl:Class>
   </hasPart>
</owl:Class>

<owl:ObjectProperty rdf:ID="requires">
   <creek:inheritsOver>
      <owl:ObjectProperty rdf:ID="hasPart">
   </creek:inheritsOver>
</owl:ObjectProperty>
```

7.1.3 Comparison with Other Research

In chapter 4 we presented three related areas of research, namely CaseML, DzCBR and Colibri and its CBROnto. Compared to each other and to the work done in the previous chapter, they have some similarities but also some distinct differences.

All these three have a similar way to represent a case. Given the problem space P and the solution space S, a case is a pair $(p, s) \in P \times S$. CBROnto does this by defining two classes `cbronto:CaseDescription` and `cbronto:CaseSolution` and relate each case to an instance of subclass of each of these classes. In Creek, the cases are more tightly integrated into the general domain knowledge. The representation of a case is not limited to only its solution and an object representing its description. A case may be described by several relations from the domain model as for example the case shown in listings 5.1.

Both CaseML and CBROnto define vocabularies with terms used to describe case bases and domain models. The CaseML is very basic when compared to CBROnto and the Creek vocabulary. It has a limited set of terms, and it can only be used to describe cases. In other words, there is no direct integration with the domain model as is possible with jColibri and Creek. The CaseML vocabulary is defined in RDF rather than OWL, which further restricts what might expressed using this vocabulary.

CaseML, CBROnto and the Creek Vocabulary are aimed at distributed knowledge in that they facilitates reuse and sharing of knowledge. DzCBR is different in that the CBR process itself is a distributed process. This is due to the internal decentralization of the ontologies resulting from using C-OWL. Another unique aspect of DzCBR compared to jColibri and Creek is the inclusion of adaption knowledge. This type of knowledge is not expressed in either of the other two systems.

The one of the three other system most similar to Creek is Colibri. They are both approaches within a Knowledge-Intensive CBR framework. They both use a large amount of general domain knowledge in addition to the cases in the reasoning process. One thing that separate them is jColibri's focus on Problem-Solving Methods and software reuse. The work done with the Creek framework in this report is about reuse and sharing of knowledge, and knowledge alone.

As the goals of the Creek vocabulary and CBROnto are very much similar, further work should be done to define one single vocabulary for KI-CBR. This is discussed further in section 7.3 below.

7.2 Discussion

In this section we discuss some of the limitations of our approach and some of the difficulties that we have come across.

7.2.1 Semantics

One of the major difficulties that arise from the work done in chapter 5 is the question of semantics. When we share or reuse knowledge between a Creek system and the Semantic Web, does a piece of knowledge mean the same thing?

Does a frame or an entity in Creek give the same meaning as a concept in Description Logics? Generally speaking, an entity in Creek represent a *prototype* of the underlying concept, described by various findings or relations. The terminological knowledge in Description Logics defines concepts in terms of other concepts. As shown in the work done in the previous chapters, it is possible to go from one of these representation to the other and still maintain the knowledge in a meaningful way. The next sections below will discuss other things that might get lost in this process.

RDF, OWL and the Semantic Web in general all have their semantics firmly rooted in the logistic branch of Knowledge Representation (KR). Reasoning is deductive and monotonic. Creek on the other hand uses frames and semantic nets for the KR, and the reasoning is abductive and non-monotonic.

The debate between these two approaches have been long (see for example (Nilsson, 1991) versus (Birnbaum, 1991)), and is still ongoing — even in the context of the Semantic Web (Russell, 2003). It is beyond the scope of the work in this thesis to conclude what is the correct or best way of reasoning. However, we will point out some difficulties arising when going from knowledge representation formalism to the other.

One of the basic reasoning methods in Creek is default inheritance. This allows subclasses and instances inherit relations from a superclass, unless it is overridden at the local level. Default inheritance is therefor non-monotonic in nature. Given the knowledge base from the CreekServer system, a new concept `km:TwoStrokeEngine` is added. Here, the value of the relation `km:requires` is overridden from `km:Oil` to the value `km:OilInGasTank`. Figure 7.3 shows part of the knowledge model, and a frame-view of the `km:TwoStrokeEngine` entity.

There is no way to express default inheritance in OWL. The result of representing the knowledge from figure 7.3 would be something like the RDF/XML in listing 7.3. A Description Logics reasoner will based on this conclude that an instance of `km:TwoStrokeEngine` both has a relation `km:requires km:Oil` **and** a relation `km:requires km:OilInGasTank`. As we see, we may end up with inconsistencies in the OWL representation of our knowledge model — or inconsistencies between our original Creek knowledge model and the OWL ontology exported from it.

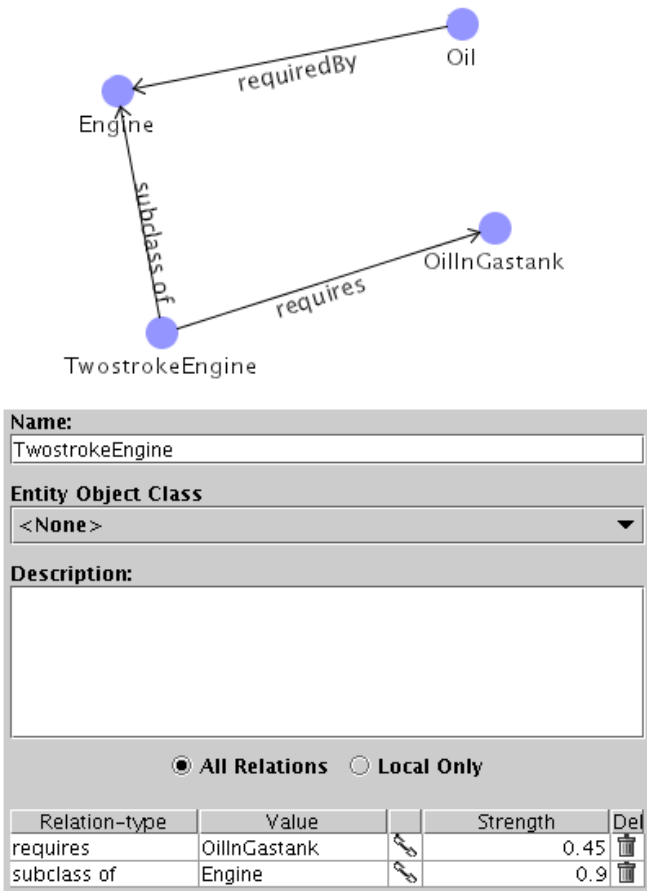

Figure 7.3: `km:requires` overridden in the `TwoStrokeEngine` entity

No solution to this problem is presented here. The problem is a fundamental one, so one need to have this is in mind when knowledge models are to be exported from a Creek representation to an OWL ontology.

7.2.2 Simplification

Generally speaking, when exporting a Creek knowledge model to OWL the resulting ontology will be a more simple model than the original Creek model. In the Semantic Web a property is binary — it is a relation from one subject to one object. Relations in Creek each have a weight assigned. This is used in the reasoning, for example in plausible inheritance to decide which relations to inherit. These weights are lost when a knowledge model is exported.

Figure 7.4 shows a frame view of an example relation in Creek. Listing 7.4 shows how this will be represented after being exported to OWL.

We have shown in our design that it is possible to use anonymous nodes to

Listing 7.3: OWL representation of `km:TwoStrokeEngine` entity from figure 7.3

```
<owl:Class rdf:ID="TwoStrokeEngine">
   <rdfs:subClassOf>
      <owl:Class rdf:ID="Engine">
          <requires>
             <owl:Class rdf:ID="Oil">
          </requires>
      </owl:Class>
   </rdfs:subClassOf>
   <requires>
      <owl:Class rdf:ID="OilInGasTank"/>
   </requires>
</owl:Class>

<owl:ObjectProperty rdf:ID="requires">
   <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="requiredBy"/>
   </owl:inverseOf>
</owl:ObjectProperty>
```

Listing 7.4: OWL representation of the `has_colour` relation from figure 7.4

```
<owl:ObjectProperty rdf:ID="has_colour">
   <owl:inverseOf rdf:resource="#colour_of"/>
   <creek:hasDefaultExplanationStrength>
      0.1
   </creek:hasDefaultExplanationStrength>
   <rdfs:subPropertyOf rdf:resource="http://creek.idi.ntnu.no/owl/
creek-vocabulary#hasFinding"/>
</owl:ObjectProperty>
```

attach additional information to a property in OWL. Figure 5.6 and listing 5.2 in chapter 5 show how importance and predictive strength properties can be expressed for the findings of a given Creek case. However, this adds a level of complexity to the ontology that is being exported. Also, it will only be meaningful to other Creek systems or other systems weighting the relations. A reasoning system based on Description Logics will for example not be able to utilize these weights.

Other aspects of a Creek knowledge model are also left out. For example, it is possible to define what Java class to use when cases and concepts are matched in the CBR process. This will not be included in the OWL ontology when exporting a knowledge model.

7.2.3 Restrictions

One major limitation to how Creek imports OWL ontologies are the lack of support for the Description Logics style restrictions that can be expressed in OWL. When importing an ontology, such restriction declarations are ignored. As these may define important aspects of the problem domain, it is possible that random part of the knowledge is actually removed. As in the section about

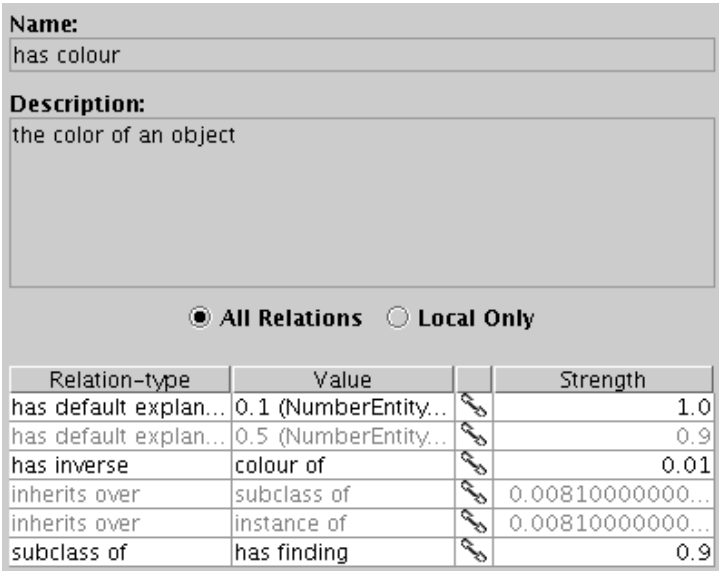

Figure 7.4: Internal Creek representation of the `has_colour` relation

semantics above, we may end up with inconsistencies between the ontology we import from, and the resulting Creek knowledge model. The ontologies used as examples in this work are chosen as not to contain OWL constructs such as `owl:Restriction`.

The original Creek framework was implemented in Lisp. The frame-based language CreekL was used to describe the semantic net/frame based knowledge in this system. CreekL allowed for some constraints to be expressed. The current Java implementation of Creek does not support this however. There is ongoing research (Stige, 2006) to express constraints in Creek models again. The work focuses on constraints in general such as cardinality and value constraints, but also include domain and range restrictions as the ones found in RDF and OWL.

Once this is in place, it could be possible to use constraints in Creek to represent imported OWL classes defined using restrictions. One might need to discuss the semantic differences of the Description Logics restrictions, and the constraints that one choose to add to the Creek system.

7.2.4 Description Logics

Beside restrictions as described above, other aspect of Description Logics are shown useful in CBR. (Gómez-Albarrán et al., 1999) describes how for example the retrieval task in CBR can be performed using Description Logics (DL) reasoning such as classification and instance checking. Also, jColibri uses DL classification for organizing the cases in the case base.

The Creek OWL vocabulary is defined using the OWL Full variant. In OWL DL, the sets of classes and the set of properties must be disjoint. In the

Creek vocabulary `creek:Relation` is defined both as an `owl:Class` and as an `owl:ObjectProperty`:

```
<owl:Class rdf:ID="Relation">
   <rdfs:subClassOf rdf:resource="#Thing"/>
   <rdfs:subClassOf rdf:resource="#Concept"/>
   <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
</owl:Class>
```

This is legal in OWL Full. Also, OWL DL requires separation between the set of classes and the set of individuals. For example `creek:Process` is used both as an instance of `creek:Symbol`, and a superclass of `creek:Method`:

```
<Symbol rdf:ID="Process">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Class"/>
</Symbol>
<owl:Class rdf:ID="Method">
  <rdfs:subClassOf rdf:resource="#Process"/>
</owl:Class>
```

The ontology resulting from the export of an Creek knowledge model will in general be in OWL Full. Description Logics reasoning — such as with OWL DL resources — is computational tractable. OWL Full sacrifices effectiveness to expressiveness. As long as the OWL resources offered by Creek systems are of the Full variant, this may have a negative influence on the attractiveness and usability of them for others.

7.3 Further Work

Both Creek and jColibri are frameworks for Knowledge-Intensive Case-Based Reasoning (KI-CBR). As discussed in section 7.1.3 above, the Creek OWL Vocabulary and CBROnto are in many respects similar. The goal of both of them is to define a standard vocabulary so that knowledge can be shared and reused. It would be fruitful to agree upon one single vocabulary for use in KI-CBR. This should be possible given the common background and approach by these two systems. One might giving it a `cbronto:`, `cbr:` or similar prefix and some namespace, and defining an ontology that can be used by both the Creek and jColibri systems. Obvious terms are `cbr:Case` and `cbr:hasSolution` and so on. One need to agree upon a common subset or a common superset of the two existing vocabularies.

The ISOPOD vocabulary presented in this report uses OWL constructs such as `owl:equivalentClass` and `owl:equivalentProperty` to map a few terms in the Creek ISOPOD ontology and CBROnto to each other. This should be seen as a first step towards a KI-CBR vocabulary that can be used by both systems.

To further integrate Creek with jColibri, some of the reasoning processes in Creek, such as the explanation and plausible inheritance, could be described as Problem-Solving Methods.

7.4 Conclusion

In this thesis we have proposed a design for knowledge sharing and reuse for the Creek architecture and the Semantic Web. A vocabulary for Knowledge-Intensive Case-Based Reasoning based on the ISOPOD model has been defined. This together with the import and export parts of the design integrate the Creek framework with the distributed nature of the Semantic Web.

Knowledge is represented and shared in the form of web ontologies using the OWL language. OWL and the Semantic Web are World Wide Web Consortium (W3C) activities. As the Semantic Web and OWL are both new technologies, it is somewhat difficult to predict their success and eventual breakthrough. If, however, the Semantic Web evolves as envisioned by Tim Berners-Lee and the other people behind it, vast amount of knowledge in the form of structured ontologies could be available on the Web. The ability to handle and understand OWL ontologies in Creek and Case-Based Reasoning will no doubt be an important feature enhancing knowledge sharing and reuse.

Bibliography

Aamodt, A. (1991). *A Knowledge-Intensive, Integrated Approach to Problem Solving and Sustained Learning*. PhD thesis, Norwegian University of Science and Technology, Trondheim.

Aamodt, A. (1994). Explanation-driven case-based reasoning. In *Topics in case-based reasoning*, pages 274–288. Springer-Verlag. http://www.idi.ntnu.no/emner/it3704/lectures/papers/Aamodt_1994_Explanation.pdf.

Aamodt, A. (2004). Knowledge-intensive case-based reasoning in creek. In *Advances in case-based reasoning, 7th European Conference, ECCBR 2004*. Springer. http://www.idi.ntnu.no/~agnar/publications/eccbr04-aamodt.pdf.

Aamodt, A. and Plaza, E. (1994). Case-based reasoning: Foundationla issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59. http://www.iiia.csic.es/People/enric/AICom.html.

Beckett, D. (2004). Rdf/xml syntax specification. http://www.w3.org/TR/rdf-syntax-grammar/.

Beckett, D. and Barstow, A. (2001). N-triples. http://www.w3.org/2001/sw/RDFCore/ntriples/.

Benjamins, V. R., Enric Plaza, E. M., Fensel, D., Studer, R., Wielinga, B., Schreiber, G., and Zdrahal, Z. (1998). Ibrow3: An intelligent brokering service for knowledge-component reuse on the world-wide web. http://hcs.science.uva.nl/projects/IBROW3/docs/papers/kaw98/ibrow3-kaw98.html.

Berners-Lee, T. (1998a). What the semantic web can represent. http://www.w3.org/DesignIssues/RDFnot.html.

Berners-Lee, T. (1998b). The world wide web: A very short personal history. http://www.w3.org/People/Berners-Lee/ShortHistory.

Berners-Lee, T., Connolly, D., and Swick, R. R. (1999). Web architecture: Describing and exchanging data. http://www.w3.org/1999/04/WebData.

Berners-Lee, T. and Miller, E. (2002). The semantic web lifts off. *ERCIM News*. http://www.ercim.org/publication/Ercim_News/enw51/berners-lee.html.

Birnbaum, L. (1991). Rigor mortis: a response to nilsson's "logic and artificial intelligence". *Artif. Intell.*, 47(1-3):57–77.

Biron, P. V. and Malhotra, A. (2004). Xml schema part 2: Datatypes second edition. http://www.w3.org/TR/xmlschema-2/.

Bouquet, P., Giunchiglia, F., van Harmelen, F., Serafini, L., and Stuckenschmidt, H. (2003). C-owl: Contextualizing ontologies. http://www.cs.vu.nl/~frankh/postscript/ISWC03.pdf.

Brachman, R. J. and Schmolze, J. G. (1985). An overview of the kl-one knowledge representation system. *Cognitive Science*. http://www.cogsci.rpi.edu/CSJarchive/1985v09/i02/p0171p0216/MAIN.PDF.

Bray, T., Hollander, D., and Layman, A. (1999). Namespaces in xml. http://www.w3.org/TR/1999/REC-xml-names-19990114/.

Brickley, D. and R.V.Guha (2004). Rdf vocabulary description language 1.0: Rdf schema. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

Chen, H. and Wu, Z. (2003). Caseml: a rdf-based case markup language for case-based reasoning in semantic web. http://bat710.univ-lyon1.fr/~bfuchs/ws-iccbr03/papers/06_Wu_Chen.pdf.

d'Aquin, M., Lieber, J., and Napoli, A. (2005). Decentralized case-based reasoning for the semantic web. In *International Semantic Web Conference*, pages 142–155. http://www.loria.fr/equipes/orpailleur/Documents/daquin05b.pdf.

Dean, M. and Schreiber, G. (2004). Owl web ontology reference. http://www.w3.org/TR/owl-ref/.

Díaz-Agudo, B. and González-Calero, P. A. (2000). An architecture for knowledge intensive cbr systems. In *Advances in Case-Based Reasoning, 5th European Workshop, EWCBR 2000, Trento, Italy, September 6-9, 2000, Proceedings*, pages 37–48. http://www.idi.ntnu.no/~agnar/it8000/2000_ewcbr_belen-2.pdf.

Díaz-Agudo, B. and González-Calero, P. A. (2001). Knowledge intensive cbr made affordable. http://www.aic.nrl.navy.mil/papers/2001/AIC-01-003/ws2/ws2toc4.pdf.

Díaz-Agudo, B. and González-Calero, P. A. (2002). Cbronto: A task/method ontology for cbr. In *FLAIRS Conference*, pages 101–105. http://gaia.fdi.ucm.es/people/pedro/papers/2002_flairs_belen.pdf.

Díaz-Agudo, B., González-Calero, P. A., Gómez-Martín, P. P., and Gómez-Martín, M. A. (2005). On developing a distributed CBR framework through semantic web services. In *Workshop OWL: Experiences and Directions, at International Conference on Rules and Rule Markup Languages for the Semantic Web*. http://www.mindswap.org/2005/OWLWorkshop/sub30.pdf.

Donini, F. M., lenzerini, M., Nardi, D., and Schaerf, A. (1996). Reasoning in description logics. pages 191–236. http://www8.informatik.uni-erlangen.de/IMMD8/Lectures/KRR/reasoning-in-DL.ps.gz.

Fensel, D., Benjamins, V. R., Motta, E., and Wielinga, B. J. (1999). UPML: A framework for knowledge system reuse. In *IJCAI*, pages 16–23. http://kmi.open.ac.uk/people/motta/papers/upml.ijcai.pdf.

Fensel, D. and Motta, E. (2001). Structured development of problem solving methods. *Knowledge and Data Engineering*, 13(6):913–932. http://ksi.cpsc.ucalgary.ca/KAW/KAW98/fensel2/.

Gibson, W. (1984). *Neuromancer*. Gollancz, London. ISBN: 057503470X.

Gómez-Albarrán, M., González-Calero, P. A., Daz-Agudo, B., and Fernández-Conde, C. (1999). Modelling the cbr life cycle using description logics. http://gaia.fdi.ucm.es/people/pedro/papers/1999_iccbr_mercedes.pdf.

Gruber, T. (1993). A translation approach to portable ontology specifications. http://tomgruber.org/writing/ontolingua-kaj-1993.pdf.

Hayes, P. (2004). Rdf semantics. http://www.w3.org/TR/rdf-mt/.

Heflin, J. and Hendler, J. (2000). Semantic interoperability on the web. http://www.cs.umd.edu/projects/plus/SHOE/pubs/extreme2000.pdf.

Heflin, J., Hendler, J., and Luke, S. (1999). SHOE: A knowledge representation language for internet applications. Technical Report CS-TR-4078. http://www.cs.umd.edu/projects/plus/SHOE/pubs/techrpt99.pdf.

Hendler, J., Berners-Lee, T., and Miller, E. (2002). Integrating applications on the semantic web. *Journal of the Institute of Electrical Engineers of Japan*, 122(10):676–680. http://www.w3.org/2002/07/swint.

Horrocks, I. (2000). A denotational semantics for standard oil and instance oil. http://www.ontoknowledge.org/oil/downl/semantics.pdf.

Horrocks, I., Patel-Schneider, P., and van Harmelen, F. (2003). From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26. http://www.cs.man.ac.uk/~horrocks/Publications/download/2003/HoPH03a.pdf.

Jacobs, I. and Walsh, N. (2004). Architecture of the world wide web, volume one. http://www.w3.org/TR/webarch/.

Klyne, G. and Carroll, J. J. (2004). Resource description framework (rdf): Concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/.

Lassila, O. (2005). Xml considered harmful, and other things. http://www.lassila.org/blog/archive/2005/05/xml_considered.html.

Manola, F. and Miller, E. (2004). Rdf primer. http://www.w3.org/TR/rdf-primer/.

Masinter, L., Fielding, R., and Berners-Lee, T. (2005). Uniform resource identifier (uri): Generic syntax. http://www.gbiv.com/protocols/uri/rfc/rfc3986.html.

McBride, B. (2001). Jena: Implementing the rdf model and syntax specification. http://www.hpl.hp.com/personal/bwm/papers/20001221-paper/.

McGuinness, D. L. (2002). Ontologies come of age. http://www.ksl.stanford.edu/people/dlm/papers/ontologies-come-of-age-mit-press-(with-citation).htm.

McGuinness, D. L., Smith, M. K., and Welty, C. (2004). Owl web ontology language guide. http://www.w3.org/TR/owl-guide/.

McGuinness, D. L. and van Harmelen, F. (2004). Owl web ontology language overview. http://www.w3.org/TR/owl-features/.

Miller, E. (1998). An introduction to the resource description framework. *D-Lib Magazine*. http://www.dlib.org/dlib/may98/miller/05miller.html.

Nardi, D. and Brachman, R. J. (2003). An introduction to description logics. http://www.inf.unibz.it/~franconi/dl/course/dlhb/dlhb-01.pdf.

Nilsson, N. J. (1991). Logic and artificial intelligence. *Artif. Intell.*, 47(1-3):31–56.

Noy, N. and Rector, A. (2004). Defining n-ary relations on the semantic web. http://smi-web.stanford.edu/people/noy/nAryRelations/n-aryRelations-2nd-WD.html.

O'Donnell, M. J. (1993). Against nonmonotonic logic (draft). http://people.cs.uchicago.edu/~odonnell/Scholar/Work_in_progress/Against_Nonmonotonic_Logic/attack.pdf.

Ogbuji, U. and Ouellet, R. (2002). Daml reference. http://www.xml.com/pub/a/2002/05/01/damlref.html.

Patel-Schneider, P. F., Hayes, P., and Horrocks, I. (2004). Owl web ontology language semantics and abstract syntax. http://www.w3.org/TR/owl-semantics/.

Russell, S. (2003). Readings about the question: It is said that resoning on the semantic web must be monotonic. why is this so, when human reasoning, which seems to have served us well, is nonmonotonic? http://robustai.net/papers/Monotonic_Reasoning_on_the_Semantic_Web.html.

Sørmo, F. (2000). Plausible inheritance, semantic network infernce for case-based reasoning. Master's thesis, Norwegian University of Science and Technology, Department of Computer and Information Science.

Stige, M. (2006). Representing and reasoning with constraints in creek (evolving document). Master's thesis, Norwegian University of Science and Technology, Department of Computer and Information Science. http://www.stud.ntnu.no/~martisti/Thesis.pdf.

Tomassen, S. L. (2002). Semi-automatic generation of ontologies for knowledge-intensitive cbr. Master's thesis, Norwegian University of Science and Technology, Department of Computer and Information Science. http://www.idi.ntnu.no/~steint/Master_thesis_of_Stein_L_Tomassen.pdf.

Updegrove, A. (2005). The semantic web: An interview with Tim Berners-Lee. Interview from ConsortiumInfo.org. http://www.consortiuminfo.org/bulletins/semanticweb.php.

van Harmelen, F. and Fensel, D. (1999). Practical knowledge representation for the web. http://www.cs.vu.nl/~frankh/postscript/IJCAI99-III.html.

van Harmelen, F., Patel-Schneider, P. F., and Horrocks, I. (2001). Reference description of the daml+oil (march 2001) ontology markup language. http://www.daml.org/2001/03/reference.

W3C (2001). Uris, urls, and urns: Clarifications and recommendations. http://www.w3.org/TR/uri-clarification/.

wikipedia.org (OWL). Web ontology language. http://en.wikipedia.org/wiki/Web_Ontology_Language.

Appendix A

Creek OWL Representation

A.1 Creek OWL Vocabulary

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY base "http://creek.idi.ntnu.no/owl/creek-vocabulary">
    <!ENTITY creek "&base;#">
    <!ENTITY dc "http://purl.org/dc/elements/1.1/">
    <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
    <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns#">
    <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema#">
    <!ENTITY owl "http://www.w3.org/2002/07/owl#">
    <!ENTITY cbronto "http://gaia.sip.ucm.es/cbronto.owl#">
  ]>
<rdf:RDF
  xmlns         = "&creek;"
  xml:base      = "&base;"
  xmlns:creek   = "&creek;"
  xmlns:owl     = "&owl;"
  xmlns:rdf     = "&rdf;"
  xmlns:rdfs    = "&rdfs;"
  xmlns:xsd     = "&xsd;"
  xmlns:dc      = "&dc;"
  xmlns:cbronto = "&cbronto;">


<owl:Ontology rdf:about="">
  <owl:versionInfo>0.96-56</owl:versionInfo>
  <rdfs:label>Creek OWL Vocabulary of the ISOPOD model</rdfs:label>
  <rdfs:comment>
     Based on the ISOPOD knowledge model from
     jCreek devbuild version 0.96 build 56,
     built 22/09/2005
   </rdfs:comment>
  <dc:creator>Fidjeland, Mikael Kirkeby</dc:creator>
```

```
  <dc:title>Creek OWL Vocabulary</dc:title>
  <dc:date>2006-02</dc:date>
</owl:Ontology>


<owl:Class rdf:ID="Thing">
  <rdfs:label>creek thing superclass</rdfs:label>
  <rdfs:comment>
    Any thing in the world worth naming or characterising.
    Everything we want to talk about IN THE WORLD is a (subclass of) Thing.
    Differs from Concept in that everything in a CONCEPTUAL MODEL
    is a (subclass of) concept, including Thing itself.
  </rdfs:comment>
</owl:Class>

<owl:Class rdf:ID="Concept">
  <rdfs:label>concept</rdfs:label>
</owl:Class>


<!-- ENTITY -->

<owl:Class rdf:ID="Entity">
  <rdfs:label>entity</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Thing"/>
  <rdfs:subClassOf rdf:resource="#Concept"/>
</owl:Class>

<!-- no value -->

<Symbol rdf:ID="NoValue">
  <rdfs:label>no value</rdfs:label>
  <rdfs:comment>Note; same as owl:Nothing</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#Entity"/>
  <rdf:type rdf:resource="&owl;Class"/>
</Symbol>

<!-- case -->

<owl:Class rdf:ID="Case">
  <rdfs:label>a creek case</rdfs:label>
  <rdfs:comment>
a case is used in case-based reasoning
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="#Entity"/>
  <owl:equivalentClass rdf:resource="&cbronto;Case"/>
</owl:Class>

<!-- process -->

<Symbol rdf:ID="Process">
  <rdfs:label>process</rdfs:label>
  <rdf:type rdf:resource="&owl;Class"/>
</Symbol>
```

```
<owl:Class rdf:ID="Method">
  <rdfs:label>method</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Process"/>
</owl:Class>
<owl:Class rdf:ID="Activity">
  <rdfs:label>activity</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Process"/>
</owl:Class>
<owl:Class rdf:ID="Service">
  <rdfs:label>service</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Activity"/>
</owl:Class>

<!-- context -->

<Symbol rdf:ID="Context">
  <rdfs:label>context</rdfs:label>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:subClassOf rdf:resource="#Entity"/>
</Symbol>

<owl:Class rdf:ID="EnvironmentalContext">
  <rdfs:label>environmental context</rdfs:label>
  <partOf rdf:resource="#Context"/>
</owl:Class>
<owl:Class rdf:ID="PersonalContext">
  <rdfs:label>personal context</rdfs:label>
  <partOf rdf:resource="#Context"/>
</owl:Class>
<owl:Class rdf:ID="MentalContext">
  <rdfs:label>mental context</rdfs:label>
  <partOf rdf:resource="#PersonalContext"/>
</owl:Class>

<owl:Class rdf:ID="SpatioTemporalContext">
  <rdfs:label>spatio-temporal context</rdfs:label>
  <partOf rdf:resource="#Context"/>
</owl:Class>
<owl:Class rdf:ID="SpatialContext">
  <rdfs:label>spatial context</rdfs:label>
  <rdfs:subClassOf rdf:resource="#SpatioTemporalContext"/>
</owl:Class>
<owl:Class rdf:ID="TemporalContext">
  <rdfs:label>temporal context</rdfs:label>
  <rdfs:subClassOf rdf:resource="#SpatioTemporalContext"/>
</owl:Class>

<owl:Class rdf:ID="SocialContext">
  <rdfs:label>social context</rdfs:label>
  <partOf rdf:resource="#Context"/>
</owl:Class>
<owl:Class rdf:ID="TaskContext">
  <rdfs:label>task context</rdfs:label>
  <partOf rdf:resource="#Context"/>
```

```
</owl:Class>

<owl:Class rdf:ID="PhysicalContext">
  <rdfs:label>physical context</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Context"/>
</owl:Class>
<owl:Class rdf:ID="PhysiologicalContext">
  <rdfs:label>physiological context</rdfs:label>
  <partOf rdf:resource="#PersonalContext"/>
  <rdfs:subClassOf rdf:resource="#PhysicalContext"/>
</owl:Class>

<owl:Class rdf:ID="AgentContext">
  <rdfs:label>agent context</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Context"/>
</owl:Class>

<owl:Class rdf:ID="TagContext">
  <rdfs:label>tag context</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Context"/>
</owl:Class>

<owl:Class rdf:ID="UserContext">
  <rdfs:label>user context</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Context"/>
</owl:Class>

<!-- environment -->

<Symbol rdf:ID="Environment">
  <rdfs:label>environment</rdfs:label>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:subClassOf rdf:resource="#Entity"/>
  <hasContext rdf:resource="#EnvironmentalContext"/>
</Symbol>

<!-- situation -->

<Symbol rdf:ID="Situation">
  <rdfs:label>situation</rdfs:label>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:subClassOf rdf:resource="#Entity"/>
  <hasContext rdf:resource="#Context"/>
</Symbol>

<owl:Class rdf:ID="StaticSituation">
  <rdfs:label>static situation</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Situation"/>
</owl:Class>

<owl:Class rdf:ID="Event">
  <rdfs:label>event</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Situation"/>
</owl:Class>
```

```
<owl:Class rdf:ID="Action">
  <rdfs:label>event</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Event"/>
  <hasResult rdf:resource="#Goal"/>
</owl:Class>

<!-- role -->

<Symbol rdf:ID="Role">
  <rdfs:label>role</rdfs:label>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:subClassOf rdf:resource="#Entity"/>
  <hasContext rdf:resource="#SocialContext"/>
</Symbol>
<owl:Class rdf:ID="ServiceProvider">
  <rdfs:label>service provider</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Role"/>
  <rdfs:subClassOf rdf:resource="#PhysicalObject"/>
  <performs rdf:resource="#Service"/>
</owl:Class>
<owl:Class rdf:ID="User">
  <rdfs:label>user</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Role"/>
  <rdfs:subClassOf rdf:resource="#Agent"/>
  <performs rdf:resource="#Service"/>
  <hasContext rdf:resource="#UserContext"/>
  <hasState rdf:resource="#UserState"/>
</owl:Class>

<!-- physical object -->

<Symbol rdf:ID="PhysicalObject">
  <rdfs:label>physical object</rdfs:label>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:subClassOf rdf:resource="#Entity"/>
  <hasContext rdf:resource="#PhysicalContext"/>
</Symbol>

<owl:Class rdf:ID="Agent">
  <rdfs:label>agent</rdfs:label>
  <rdfs:subClassOf rdf:resource="#PhysicalObject"/>
  <hasContext rdf:resource="#AgentContext"/>
</owl:Class>
<owl:Class rdf:ID="ComputerAgent">
  <rdfs:label>computer agent</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Agent"/>
</owl:Class>

<owl:Class rdf:ID="TangibleObject">
  <rdfs:label>tangible object</rdfs:label>
  <rdfs:subClassOf rdf:resource="#PhysicalObject"/>
</owl:Class>

<owl:Class rdf:ID="IntangibleObject">
```

```
  <rdfs:label>intangible object</rdfs:label>
  <rdfs:subClassOf rdf:resource="#PhysicalObject"/>
</owl:Class>

<owl:Class rdf:ID="ContextTag">
  <rdfs:label>tangible object</rdfs:label>
  <rdfs:subClassOf rdf:resource="#PhysicalObject"/>
  <hasContext rdf:resource="#TagContext"/>
</owl:Class>

<!-- mental object -->

<Symbol rdf:ID="MentalObject">
  <rdfs:label>mental object</rdfs:label>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:subClassOf rdf:resource="#Entity"/>
  <hasContext rdf:resource="#MentalContext"/>
</Symbol>

<!-- state -->

<Symbol rdf:ID="State">
  <rdfs:label>state</rdfs:label>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:subClassOf rdf:resource="#Entity"/>
</Symbol>
<owl:Class rdf:ID="NormalState">
  <rdfs:label>normal state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#State"/>
</owl:Class>
<owl:Class rdf:ID="HumanState">
  <rdfs:label>human state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#State"/>
</owl:Class>
<owl:Class rdf:ID="EnvironmentalState">
  <rdfs:label>environmental state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#State"/>
</owl:Class>
<owl:Class rdf:ID="UserState">
  <rdfs:label>user state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#State"/>
</owl:Class>
<owl:Class rdf:ID="AlarmState">
  <rdfs:label>alarm state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#State"/>
</owl:Class>
<owl:Class rdf:ID="ManagementState">
  <rdfs:label>management state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#State"/>
</owl:Class>
<owl:Class rdf:ID="EquipmentState">
  <rdfs:label>equipment state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#State"/>
</owl:Class>
```

```
<owl:Class rdf:ID="FailureState">
  <rdfs:label>failure state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#State"/>
</owl:Class>

<!-- task -->

<Symbol rdf:ID="Task">
  <rdfs:label>task</rdfs:label>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:subClassOf rdf:resource="#Entity"/>
  <hasContext rdf:resource="#TaskContext"/>
  <triggers rdf:resource="#Action"/>
  <achieves rdf:resource="#Goal"/>
</Symbol>

<owl:Class rdf:ID="DataManagementTask">
  <rdfs:label>data management task</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Task"/>
</owl:Class>
<owl:Class rdf:ID="FormulateNewCase">
  <rdfs:label>formulate new case</rdfs:label>
  <rdfs:subClassOf rdf:resource="#DataManagementTask"/>
</owl:Class>
<owl:Class rdf:ID="UpdateKnowledgeModel">
  <rdfs:label>datamanagement task</rdfs:label>
  <rdfs:subClassOf rdf:resource="#DataManagementTask"/>
</owl:Class>

<owl:Class rdf:ID="OperationalTask">
  <rdfs:label>operational task</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Task"/>
</owl:Class>
<owl:Class rdf:ID="ObserveParameters">
  <rdfs:label>observe paramters</rdfs:label>
  <rdfs:subClassOf rdf:resource="#OperationalTask"/>
</owl:Class>
<owl:Class rdf:ID="RepairFailure">
  <rdfs:label>repair failure</rdfs:label>
  <rdfs:subClassOf rdf:resource="#OperationalTask"/>
</owl:Class>
<owl:Class rdf:ID="PredictProblem">
  <rdfs:label>predict problem</rdfs:label>
  <rdfs:subClassOf rdf:resource="#OperationalTask"/>
</owl:Class>
<owl:Class rdf:ID="ClassifySituation">
  <rdfs:label>repair failure</rdfs:label>
  <rdfs:subClassOf rdf:resource="#OperationalTask"/>
</owl:Class>
<owl:Class rdf:ID="DetermineMissingInformation">
  <rdfs:label>determine missing information</rdfs:label>
  <rdfs:subClassOf rdf:resource="#OperationalTask"/>
  <rdfs:subClassOf rdf:resource="#ClassifySituation"/>
</owl:Class>
```

```
<owl:Class rdf:ID="ClassifyFailureState">
  <rdfs:label>repair failure</rdfs:label>
  <rdfs:subClassOf rdf:resource="#ClassifySituation"/>
</owl:Class>
<owl:Class rdf:ID="IdentifyNormalState">
  <rdfs:label>identify normal state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#ClassifySituation"/>
  <rdfs:subClassOf rdf:resource="#PredictProblem"/>
</owl:Class>
<owl:Class rdf:ID="IdentifyAlarmState">
  <rdfs:label>identify alarm state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#ClassifySituation"/>
  <rdfs:subClassOf rdf:resource="#PredictProblem"/>
</owl:Class>
<owl:Class rdf:ID="IdentifyWarningState">
  <rdfs:label>identify warning state</rdfs:label>
  <rdfs:subClassOf rdf:resource="#ClassifySituation"/>
  <rdfs:subClassOf rdf:resource="#PredictProblem"/>
</owl:Class>

<!-- goal -->

<owl:Class rdf:ID="Goal">
  <rdfs:label>goal</rdfs:label>
  <rdfs:comment>
a particular end state associated with a Task
</rdfs:comment>
</owl:Class>


<!-- java class-->

<owl:Class rdf:ID="JavaClass">
  <rdfs:label>Java Class</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Entity"/>
</owl:Class>

<owl:Class rdf:ID="Comparator">
  <rdfs:label>Comparator</rdfs:label>
  <rdfs:subClassOf rdf:resource="#JavaClass"/>
</owl:Class>

<Comparator rdf:ID="URLComparison">
  <rdfs:label>URLComparison</rdfs:label>
  <comparatorOf rdf:resource="#URL"/>
</Comparator>
<Comparator rdf:ID="SymbolComparison">
  <rdfs:label>SymbolComparison</rdfs:label>
</Comparator>
<Comparator rdf:ID="CaseComparison">
  <rdfs:label>CaseComparison</rdfs:label>
</Comparator>
<Comparator rdf:ID="StringComparison">
  <rdfs:label>StringComparison</rdfs:label>
```

```
  <comparatorOf rdf:resource="#String"/>
</Comparator>
<Comparator rdf:ID="NumberComparison">
  <rdfs:label>NumberComparison</rdfs:label>
</Comparator>


<!-- string & URL -->

<owl:Class rdf:ID="String">
  <rdfs:label>String</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Entity"/>
</owl:Class>

<owl:Class rdf:ID="URL">
  <rdfs:label>URL</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Entity"/>
</owl:Class>


<!-- DESCRIPTIVE THING -->

<owl:Class rdf:ID="DescriptiveThing">
  <rdfs:label>descriptive thing</rdfs:label>
  <rdfs:comment>
    A meta-level thing that represents how other things
    (entities and relations) are described and represented.
  </rdfs:comment>
  <rdfs:subClassOf rdf:resource="#Thing"/>
</owl:Class>

<!-- parameter -->

<Symbol rdf:ID="Parameter">
  <rdfs:label>parameter</rdfs:label>
  <rdfs:comment>
Descibes the name and structure of value assignments
  </rdfs:comment>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:subClassOf rdf:resource="#DescriptiveThing"/>
</Symbol>

<!-- basic representional type -->

<owl:Class rdf:ID="BasicRepresentionalType">
  <rdfs:label>basic representional type</rdfs:label>
  <rdfs:subClassOf rdf:resource="#DescriptiveThing"/>
</owl:Class>

<owl:Class rdf:ID="Symbol">
  <rdfs:label>symbol</rdfs:label>
  <rdf:type rdf:resource="&owl;Class"/>
  <rdfs:subClassOf rdf:resource="#BasicRepresentionalType"/>
</owl:Class>
```

```
<owl:Class rdf:ID="CaseStatus">
  <rdfs:label>case status</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Symbol"/>
</owl:Class>
<CaseStatus rdf:ID="SolvedCase">
  <rdfs:label>solved</rdfs:label>
</CaseStatus>
<CaseStatus rdf:ID="ProcessedCase">
  <rdfs:label>processed</rdfs:label>
</CaseStatus>
<CaseStatus rdf:ID="UnsolvedCase">
  <rdfs:label>unsolved</rdfs:label>
</CaseStatus>
<owl:Class rdf:ID="Importance">
  <rdfs:label>importance</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Symbol"/>
</owl:Class>
<Importance rdf:ID="Necessary">
  <rdfs:label>neccessary</rdfs:label>
</Importance>
<Importance rdf:ID="Characteristic">
  <rdfs:label>characteristic</rdfs:label>
</Importance>
<Importance rdf:ID="Informative">
  <rdfs:label>informative</rdfs:label>
</Importance>
<Importance rdf:ID="DefaultImportance">
  <rdfs:label>default importance</rdfs:label>
</Importance>
<Importance rdf:ID="Irrelevant">
  <rdfs:label>irrelevant</rdfs:label>
</Importance>
<owl:Class rdf:ID="PredictiveStrength">
  <rdfs:label>predictive strength</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Symbol"/>
</owl:Class>
<PredictiveStrength rdf:ID="Sufficient">
  <rdfs:label>sufficient</rdfs:label>
</PredictiveStrength>
<PredictiveStrength rdf:ID="StronglyIndicative">
  <rdfs:label>strongly indicative</rdfs:label>
</PredictiveStrength>
<PredictiveStrength rdf:ID="DefaultPredictiveStrength">
  <rdfs:label>default predictive strength</rdfs:label>
</PredictiveStrength>
<PredictiveStrength rdf:ID="Indicative">
  <rdfs:label>indicative</rdfs:label>
</PredictiveStrength>
<PredictiveStrength rdf:ID="Spurious">
  <rdfs:label>spurious</rdfs:label>
</PredictiveStrength>

<owl:Class rdf:ID="Set">
  <rdfs:label>set</rdfs:label>
```

```
  <rdfs:subClassOf rdf:resource="#BasicRepresentionalType"/>
</owl:Class>

<owl:Class rdf:ID="Table">
  <rdfs:label>table</rdfs:label>
  <rdfs:subClassOf rdf:resource="#BasicRepresentionalType"/>
</owl:Class>

<owl:Class rdf:ID="MathematicalExpression">
  <rdfs:label>mathematical expression</rdfs:label>
  <rdfs:subClassOf rdf:resource="#BasicRepresentionalType"/>
</owl:Class>

<owl:Class rdf:ID="Text">
  <rdfs:label>text</rdfs:label>
  <rdfs:comment>text realized directly in the owl structure</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#BasicRepresentionalType"/>
</owl:Class>

<owl:Class rdf:ID="Number">
  <rdfs:label>number</rdfs:label>
  <rdfs:comment>number realized through xsd:decimal type</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#BasicRepresentionalType"/>
</owl:Class>


<!-- RELATION -->

<owl:Class rdf:ID="Relation">
   <rdfs:label>relation</rdfs:label>
  <rdfs:comment>
all relations defined as owl:ObjectProperty
  also used as anonymous node to make n-ary relations</rdfs:comment>
   <rdfs:subClassOf rdf:resource="#Thing"/>
   <rdfs:subClassOf rdf:resource="#Concept"/>
   <rdf:type rdf:resource="&owl;ObjectProperty"/>
   <inheritsOver rdf:resource="#instanceOf"/>
   <inheritsOver rdf:resource="#subclassOf"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="RelationInverse">
  <rdfs:label>inverse relation</rdfs:label>
  <owl:inverseOf rdf:resource="#Relation"/>
</owl:ObjectProperty>

<!-- isopod relation -->

<owl:ObjectProperty rdf:ID="isopodRelation">
  <rdfs:comment>
Special relation used only internally in the ISOPOD structure
  </rdfs:comment>
  <rdfs:label>creek isopod relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
```

```
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="isopodRelationInverse">
  <rdfs:label>inverse creek isopod relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <owl:inverseOf rdf:resource="#isopodRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.9
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasInverse">
  <rdfs:label>has inverse</rdfs:label>
  <rdfs:comment>realized through owl:inverseOf constructs</rdfs:comment>
  <rdf:type rdf:resource="&owl;SymmetricProperty"/>
  <rdfs:subPropertyOf rdf:resource="#systemRelation"/>
  <rdfs:subPropertyOf rdf:resource="#isopodRelation"/>
  <rdfs:subPropertyOf rdf:resource="#systemRelationInverse"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="transfers">
  <rdfs:label>transfers</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#systemRelation"/>
  <rdfs:subPropertyOf rdf:resource="#isopodRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="inheritsOver">
  <rdfs:label>inherits over</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#systemRelationInverse"/>
  <rdfs:subPropertyOf rdf:resource="#isopodRelationInverse"/>
  <owl:inverseOf rdf:resource="#transfers"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<!-- system relation -->

<owl:ObjectProperty rdf:ID="systemRelation">
  <rdfs:comment>Relation used by the core Creek system</rdfs:comment>
  <rdfs:label>creek system relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="systemRelationInverse">
  <rdfs:label>inverse creek system relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <owl:inverseOf rdf:resource="#systemRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.9
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:ID="hasDefaultExplanationStrength">
  <rdfs:label>has default explanation strength</rdfs:label>
```

```
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.5
</hasDefaultExplanationStrength>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="defaultExplanationStrengthOf"/>

<owl:ObjectProperty rdf:ID="hasImportance">
  <rdfs:label>has importance</rdfs:label>
  <rdfs:comment>importance of a relation</rdfs:comment>
  <rdfs:subPropertyOf rdf:resource="#systemRelation"/>
  <rdfs:range rdf:resource="#Importance"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasPredictiveStrength">
  <rdfs:comment>predictive strength of a relation</rdfs:comment>
  <rdfs:label>has predictive strength</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#systemRelation"/>
  <rdfs:range rdf:resource="#PredictiveStrength"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasCaseStatus">
  <rdfs:label>has case status</rdfs:label>
  <rdfs:comment>status of a case's solution</rdfs:comment>
  <rdf:type rdf:resource="&owl;FunctionalProperty"/>
  <rdfs:subPropertyOf rdf:resource="#systemRelation"/>
  <rdfs:range rdf:resource="#CaseStatus"/>
  <rdfs:domain rdf:resource="#Case"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="caseStatusOf">
  <rdfs:label>case status of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasCaseStatus"/>
  <rdfs:subPropertyOf rdf:resource="#systemRelationInverse"/>
  <rdfs:domain rdf:resource="#CaseStatus"/>
  <rdfs:range rdf:resource="#Case"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="makesSenseForDomain">
  <rdfs:comment>realized through rdfs:domain contstruct</rdfs:comment>
  <rdfs:subPropertyOf rdf:resource="#systemRelation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasValueType">
  <rdfs:comment>realized through rdfs:domain contstruct</rdfs:comment>
  <rdfs:subPropertyOf rdf:resource="#systemRelation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="valueTypeOf">
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <owl:inverseOf rdf:resource="#hasValueType"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="makesSenseForRange">
  <rdfs:comment>realized through rdfs:range contstruct</rdfs:comment>
```

```
  <rdfs:subPropertyOf rdf:resource="#systemRelation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasFromType">
  <rdfs:comment>realized through rdfs:range contstruct</rdfs:comment>
  <rdfs:subPropertyOf rdf:resource="#systemRelation"/>
</owl:ObjectProperty>

<!-- structural relation -->

<owl:ObjectProperty rdf:ID="structuralRelation">
  <rdfs:label>structural relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="structuralRelationInverse">
  <rdfs:label>inverse structural relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <owl:inverseOf rdf:resource="#structuralRelation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasSubclass">
  <rdfs:label>has subclass</rdfs:label>
  <rdfs:comment>realized through owl:subClassOf constructs</rdfs:comment>
  <owl:inverseOf rdf:resource="&rdfs;subClassOf"/>
  <rdfs:subPropertyOf rdf:resource="#structuralRelation"/>
  <rdfs:subPropertyOf rdf:resource="#isopodRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="subclassOf">
  <rdfs:label>subclass of</rdfs:label>
  <rdfs:comment>realized through owl:subClassOf constructs</rdfs:comment>
  <owl:inverseOf rdf:resource="#hasSubclass"/>
  <rdfs:subPropertyOf rdf:resource="#structuralRelationInverse"/>
  <rdfs:subPropertyOf rdf:resource="#isopodRelationInverse"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="instanceOf">
  <rdfs:label>instance of</rdfs:label>
  <rdfs:comment>
realized through owl instanziation constructs (rdf:type)
  </rdfs:comment>
  <rdfs:subPropertyOf rdf:resource="#structuralRelationInverse"/>
  <rdfs:subPropertyOf rdf:resource="#isopodRelationInverse"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasInstance">
  <rdfs:label>has instance</rdfs:label>
  <rdfs:comment>
realized through owl instanziation constructs (rdf:type)
</rdfs:comment>
  <owl:inverseOf rdf:resource="#instanceOf"/>
  <rdfs:subPropertyOf rdf:resource="#structuralRelation"/>
  <rdfs:subPropertyOf rdf:resource="#isopodRelation"/>
```

```
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasPart">
  <rdfs:label>has part</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#structuralRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="partOf">
  <rdfs:label>part of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasPart"/>
  <rdfs:subPropertyOf rdf:resource="#structuralRelationInverse"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasValue">
  <rdfs:label>has value</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#structuralRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="valueOf">
  <rdfs:label>value of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasValue"/>
  <rdfs:subPropertyOf rdf:resource="#structuralRelationInverse"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasMember">
  <rdfs:label>has member</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#structuralRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="memberOf">
  <rdfs:label>member of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasMember"/>
  <rdfs:subPropertyOf rdf:resource="#structuralRelationInverse"/>
</owl:ObjectProperty>

<!-- functional relation -->

<owl:ObjectProperty rdf:ID="functionalRelation">
  <rdfs:label>functional relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="functionalRelationInverse">
  <rdfs:label>inverse functional relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <owl:inverseOf rdf:resource="#functionalRelation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasFunction">
  <rdfs:label>has function</rdfs:label>
```

```
  <rdfs:subPropertyOf rdf:resource="#functionalRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.8
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="functionOf">
  <rdfs:label>function of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasFunction"/>
  <rdfs:subPropertyOf rdf:resource="#functionalRelationInverse"/>
</owl:ObjectProperty>

<!-- associational relation -->

<owl:ObjectProperty rdf:ID="associationalRelation">
  <rdfs:label>associational relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.5
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="associationalRelationInverse">
  <rdfs:label>inverse associtational relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <owl:inverseOf rdf:resource="#associationalRelation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="reminding">
  <rdfs:label>finding</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelationInverse"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="occursIn">
  <rdfs:label>occurs in</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="siteOfOccurance">
  <rdfs:label>site of occurance</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelationInverse"/>
  <owl:inverseOf rdf:resource="#occursIn"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasAssociation">
  <rdfs:label>has association</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelation"/>
  <inheritsOver rdf:resource="#hasAssociation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="associatedWith">
  <rdfs:label>associated with</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelationInverse"/>
  <inheritsOver rdf:resource="#associatedWith"/>
  <owl:inverseOf rdf:resource="#hasAssociation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="involves">
  <rdfs:label>involves</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelation"/>
```

```
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="involvedBy">
  <rdfs:label>involved by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelationInverse"/>
  <owl:inverseOf rdf:resource="#involves"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasResult">
  <rdfs:label>has result</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="resultOf">
  <rdfs:label>result of</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelationInverse"/>
  <owl:inverseOf rdf:resource="#hasResult"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="describes">
  <rdfs:label>describes</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="describedBy">
  <rdfs:label>involved by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelationInverse"/>
  <owl:inverseOf rdf:resource="#describes"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="achieves">
  <rdfs:label>achieves</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="achievedBy">
  <rdfs:label>achieved by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#associationalRelationInverse"/>
  <owl:inverseOf rdf:resource="#achieves"/>
</owl:ObjectProperty>

<!-- temporal relation -->

<owl:ObjectProperty rdf:ID="temporalRelation">
  <rdfs:label>temporal relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="temporalRelationInverse">
  <rdfs:label>inverse temporal relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <owl:inverseOf rdf:resource="#temporalRelation"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="before">
  <rdfs:label>before</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="after">
  <rdfs:label>achieved by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelation"/>
  <owl:inverseOf rdf:resource="#before"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="temporallyEqual">
  <rdfs:label>temporally equal</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="during">
  <rdfs:label>during</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="temporallyContains">
  <rdfs:label>temporally contains</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelationInverse"/>
  <owl:inverseOf rdf:resource="#during"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="temporallyStarts">
  <rdfs:label>temporally starts</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="temporallyStartedBy">
  <rdfs:label>temporally started by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelationInverse"/>
  <owl:inverseOf rdf:resource="#temporallyStarts"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="temporallyOverlaps">
  <rdfs:label>temporally overlaps</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="temporallyOverlappedBy">
  <rdfs:label>temporally overlapped by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelationInverse"/>
  <owl:inverseOf rdf:resource="#temporallyOverlaps"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="temporallyMeets">
  <rdfs:label>temporally meets</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="temporallyMetBy">
  <rdfs:label>temporally started by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelationInverse"/>
```

```
  <owl:inverseOf rdf:resource="#temporallyMeets"/>
</owl:ObjectProperty>

<!-- state-effect relation -->

<owl:ObjectProperty rdf:ID="stateEffectRelation">
  <rdfs:label>state-effect relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
  <hasActivity rdf:resource="#hasState"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="stateEffectRelationInverse">
  <rdfs:label>inverse state-effect relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <owl:inverseOf rdf:resource="#stateEffectRelation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasState">
  <rdfs:label>has state</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#stateEffectRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.1
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="stateOf">
  <rdfs:label>state of</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#temporalRelationInverse"/>
  <owl:inverseOf rdf:resource="#hasState"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="CausalRelation">
  <rdfs:label>causal relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#stateEffectRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.5
</hasDefaultExplanationStrength>
  <inheritsOver rdf:resource="#CausalRelation"/>
  <transfers rdf:resource="#CausalRelation"/>
  <inheritsOver rdf:resource="#partOf"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="CausalRelationInverse">
  <rdfs:label>inverse causal relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#stateEffectRelationInverse"/>
  <owl:inverseOf rdf:resource="#CausalRelation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="causes">
  <rdfs:label>causes</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.7
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="causedBy">
  <rdfs:label>caused by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
```

```
  <owl:inverseOf rdf:resource="#causes"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="causesAlways">
  <rdfs:label>causes always</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="causedByAlways">
  <rdfs:label>caused by always</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
  <owl:inverseOf rdf:resource="#causesAlways"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="causesSometimes">
  <rdfs:label>causes sometimes</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.5
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="causedBySometimes">
  <rdfs:label>caused by sometimes</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
  <owl:inverseOf rdf:resource="#causesSometimes"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="causesOccasionally">
  <rdfs:label>causes occasionally</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.3
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="causedByOccasionally">
  <rdfs:label>caused by occasionally</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
  <owl:inverseOf rdf:resource="#causesOccasionally"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="implies">
  <rdfs:label>implies</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.8
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="impliedBy">
  <rdfs:label>implied by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
  <owl:inverseOf rdf:resource="#implies"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="leadsTo">
  <rdfs:label>leads to</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
```

```
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.1
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="leadToBy">
  <rdfs:label>lead to by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#CausalRelation"/>
  <owl:inverseOf rdf:resource="#leadsTo"/>
</owl:ObjectProperty>

<!-- case relation -->

<owl:ObjectProperty rdf:ID="caseRelation">
  <rdfs:label>case relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="caseRelationInverse">
  <rdfs:label>inverse case relation</rdfs:label>
  <owl:inverseOf rdf:resource="#caseRelation"/>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.9
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasFinding">
  <rdfs:label>has finding</rdfs:label>
  <owl:inverseOf rdf:resource="#reminding"/>
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="findingOf">
  <rdfs:label>finding of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasFinding"/>
  <rdfs:subPropertyOf rdf:resource="#caseRelationInverse"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.5
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasSubcase">
  <rdfs:label>has subcase</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="subcaseOf">
  <rdfs:label>subcase of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasSubcase"/>
  <rdfs:subPropertyOf rdf:resource="#caseRelationInverse"/>
  <rdfs:subPropertyOf rdf:resource="#findingOf"/>
  <rdfs:subPropertyOf rdf:resource="#reminding"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.5
</hasDefaultExplanationStrength>
```

```
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasSolution">
  <rdfs:label>has solution</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
  <inheritsOver rdf:resource="#causedBy"/>
  <owl:equivalentProperty rdf:resource="&cbronto;hasSolution"/>
  <rdfs:range rdf:resource="&owl;Class"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="solutionOf">
  <rdfs:label>solution of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasSolution"/>
  <rdfs:subPropertyOf rdf:resource="#caseRelationInverse"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasTask">
  <rdfs:label>has task</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="taskOf">
  <rdfs:label>task of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasTask"/>
  <rdfs:subPropertyOf rdf:resource="#caseRelationInverse"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasContext">
  <rdfs:label>has context</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="contextOf">
  <rdfs:label>context of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasContext"/>
  <rdfs:subPropertyOf rdf:resource="#caseRelationInverse"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.1
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOutcome">
  <rdfs:label>has outcome</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="outcomeOf">
  <rdfs:label>outcome of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasOutcome"/>
  <rdfs:subPropertyOf rdf:resource="#caseRelationInverse"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.5
```

```
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="textualRelation">
  <rdfs:label>textual relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="textualRelationInverse">
  <rdfs:label>inverse textual relation</rdfs:label>
  <owl:inverseOf rdf:resource="#textualRelation"/>
  <rdfs:subPropertyOf rdf:resource="#caseRelationInverse"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasLessonsLearned">
  <rdfs:label>has lessons learned</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#textualRelation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="lessonsLearnedOf">
  <rdfs:label>lessons learned of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasLessonsLearned"/>
  <rdfs:subPropertyOf rdf:resource="#textualRelationInverse"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="Pointer">
  <rdfs:label>pointer</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#caseRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasURL">
  <rdfs:label>has URL</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Pointer"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="pointsAtDatabase">
  <rdfs:label>points at database</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Pointer"/>
</owl:ObjectProperty>

<!-- spatial relation -->

<owl:ObjectProperty rdf:ID="spatialRelation">
  <rdfs:label>spatial relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.9
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="spatialRelationInverse">
  <rdfs:label>inverse spatial relation</rdfs:label>
  <owl:inverseOf rdf:resource="#spatialRelation"/>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="contains">
  <rdfs:label>contains</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#spatialRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="containedIn">
  <rdfs:label>contained in</rdfs:label>
  <owl:inverseOf rdf:resource="#contains"/>
  <rdfs:subPropertyOf rdf:resource="#spatialRelationInverse"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="movesBy">
  <rdfs:label>moves by</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#spatialRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.5
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="modeOfMovementFor">
  <rdfs:label>mode of movement for</rdfs:label>
  <owl:inverseOf rdf:resource="#movesBy"/>
  <rdfs:subPropertyOf rdf:resource="#associationalRelationInverse"/>
</owl:ObjectProperty>

<!-- enabling relation -->

<owl:ObjectProperty rdf:ID="enablingRelation">
  <rdfs:label>enabling relation</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.9
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="enablingRelationInverse">
  <rdfs:label>inverse enabling relation</rdfs:label>
  <owl:inverseOf rdf:resource="#enablingRelation"/>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="triggers">
  <rdfs:label>triggers</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#enablingRelation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.8
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="triggeredBy">
  <rdfs:label>triggered by</rdfs:label>
  <owl:inverseOf rdf:resource="#triggers"/>
  <rdfs:subPropertyOf rdf:resource="#enablingRelationInverse"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="enables">
  <rdfs:label>enables</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#enablingRelation"/>
```

```
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.1
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="enabledBy">
  <rdfs:label>enabled by</rdfs:label>
  <owl:inverseOf rdf:resource="#enables"/>
  <rdfs:subPropertyOf rdf:resource="#enablingRelationInverse"/>
</owl:ObjectProperty>

<!-- performs -->

<owl:ObjectProperty rdf:ID="performs">
  <rdfs:label>performs</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="performedBy">
  <rdfs:label>performed by</rdfs:label>
  <owl:inverseOf rdf:resource="#performs"/>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.1
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<!-- has role -->

<owl:ObjectProperty rdf:ID="hasRole">
  <rdfs:label>has role</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">1.0
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="roleOf">
  <rdfs:label>performed by</rdfs:label>
  <owl:inverseOf rdf:resource="#hasRole"/>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.9
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<!-- has activity -->

<owl:ObjectProperty rdf:ID="hasActivity">
  <rdfs:label>has activity</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.5
</hasDefaultExplanationStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="activityOf">
  <rdfs:label>activity of</rdfs:label>
  <owl:inverseOf rdf:resource="#hasActivity"/>
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
  <hasDefaultExplanationStrength rdf:datatype="&xsd;decimal">0.5
```

```
</hasDefaultExplanationStrength>
</owl:ObjectProperty>

<!-- comparator -->

<owl:ObjectProperty rdf:ID="comparatorOf">
  <rdfs:subPropertyOf rdf:resource="#RelationInverse"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasComparator">
  <rdfs:subPropertyOf rdf:resource="#Relation"/>
  <owl:inverseOf rdf:resource="#comparatorOf"/>
</owl:ObjectProperty>


<!--   extend with more .....  -->



</rdf:RDF>
```

A.2 Example OWL Domain Model

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE rdf:RDF [
    <!ENTITY base "http://creek.idi.ntnu.no/owl/example/car-model.owl" >
    <!ENTITY km "&base;#" >
    <!ENTITY car "http://mknows.etri.re.kr/moa/ontology/SamsungCar.owl#" >
    <!ENTITY creek "http://creek.idi.ntnu.no/owl/creek-vocabulary#" >
]>
<rdf:RDF
    xmlns = "&km;"
    xmlns:km = "&km;"
    xml:base = "&base;"
    xmlns:creek = "&creek;"
    xmlns:owl = "http://www.w3.org/2002/07/owl#"
    xmlns:rdf = "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs = "http://www.w3.org/2000/01/rdf-schema#"
    xmlns:xsd = "http://www.w3.org/2001/XMLSchema#">

<owl:Ontology rdf:about="">
    <rdfs:label>Car example domain model and case base</rdfs:label>
    <rdfs:comment>
by Stein L. Tomassen, 2002-06-01.
Modified by Mikael Kirkeby Fidjeland for OWL modelling demonstration, 2005
    </rdfs:comment>
    <owl:imports rdf:resource="&creek;"/>
</owl:Ontology>


<owl:Class rdf:ID="Car">
  <rdfs:label>a car</rdfs:label>
</owl:Class>


<owl:ObjectProperty rdf:ID="requires">
  <creek:inheritsOver rdf:resource="#hasPart"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="requiredBy">
  <owl:inverseOf rdf:resource="#requires"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasPart"/>
<owl:ObjectProperty rdf:ID="partOf">
  <owl:inverseOf rdf:resource="#hasPart"/>
  <creek:hasDefaultExplanationStrength>0.9</creek:hasDefaultExplanationStrength>
</owl:ObjectProperty>

<owl:Class rdf:ID="Engine">
  <partOf rdf:resource="#Car"/>
  <requires>
      <owl:Class rdf:ID="Oil"/>
  </requires>
  <rdfs:subClassOf rdf:resource="&creek;Symbol"/>
```

```
</owl:Class>

<owl:Class rdf:ID="Battery">
  <partOf rdf:resource="#Car"/>
  <rdfs:subClassOf rdf:resource="&creek;Symbol"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasStatus">
  <rdfs:range rdf:resource="#Status"/>
  <rdfs:subPropertyOf rdf:resource="&creek;hasFinding"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Status">
  <rdfs:subClassOf rdf:resource="&creek;Symbol"/>
</owl:Class>
<owl:Class rdf:ID="EngineStatus">
   <rdfs:subClassOf rdf:resource="#Status"/>
</owl:Class>
<EngineStatus rdf:ID="EngineRunning">
  <rdfs:label>engine is  running</rdfs:label>
</EngineStatus>
<EngineStatus rdf:ID="EngineNotRunning">
  <rdfs:label>engine is not running</rdfs:label>
</EngineStatus>
<EngineStatus rdf:ID="EngineNotStarting">
  <rdfs:label>engine is not starting</rdfs:label>
</EngineStatus>

<owl:Class rdf:ID="BatteryStatus">
   <rdfs:subClassOf rdf:resource="#Status"/>
</owl:Class>
<BatteryStatus rdf:ID="BatteryOK">
  <rdfs:label>the battery is ok</rdfs:label>
  <creek:causes rdf:resource="#EngineRunning"/>
</BatteryStatus>
<BatteryStatus rdf:ID="BatteryEmpty">
  <rdfs:label>the battery is empty</rdfs:label>
  <creek:causes rdf:resource="#EngineNotRunning"/>
</BatteryStatus>
<BatteryStatus rdf:ID="BatteryLow">
  <rdfs:label>the battery is low</rdfs:label>
  <creek:causes rdf:resource="#EngineNotRunning"/>
</BatteryStatus>

<owl:Class rdf:ID="Color">
  <rdfs:subClassOf rdf:resource="&creek;Symbol"/>
</owl:Class>
<Color rdf:ID="Red"/>
<Color rdf:ID="Green"/>
<Color rdf:ID="Blue"/>

<owl:Class rdf:ID="SomeSolution">
  <rdfs:subClassOf rdf:resource="&creek;Symbol"/>
</owl:Class>
```

```
<owl:Class rdf:ID="AnotherSolution">
  <rdfs:subClassOf rdf:resource="&creek;Symbol"/>
</owl:Class>


<owl:ObjectProperty rdf:ID="hasColor">
  <rdfs:label>describes the color of an entity</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="&creek;hasFinding"/>
  <rdfs:range rdf:resource="#Color"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="colorOf">
  <owl:inverseOf rdf:resource="#hasColor"/>
  <rdfs:domain rdf:resource="#Color"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasEngineStatus">
  <rdfs:label>points to the status of a car engine</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="&creek;hasFinding"/>
  <rdfs:range rdf:resource="#EngineStatus"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="engineStatusOf">
  <owl:inverseOf rdf:resource="#hasEngineStatus"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasBatteryStatus">
  <rdfs:label>points to the status of a car battery</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="&creek;hasFinding"/>
  <rdfs:range rdf:resource="#BatteryStatus"/>
  <creek:defaultStrength>0.5</creek:defaultStrength>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="batteryStatusOf">
  <owl:inverseOf rdf:resource="#hasBatteryStatus"/>
</owl:ObjectProperty>


<creek:Case rdf:ID="CarCase1">
  <rdfs:label>This is a description of car problem</rdfs:label>
  <hasColor rdf:resource="#Red"/>
  <hasEngineStatus rdf:resource="#EngineNotStarting"/>
  <hasBatteryStatus  rdf:resource="#EmptyBattery"/>
  <creek:hasSolution rdf:resource="#SomeSolution"/>
  <creek:hasCaseStatus rdf:resource="&creek;SolvedCase"/>
</creek:Case>

<creek:Case rdf:ID="CarCase2">
  <rdfs:label>This is a description of car problem</rdfs:label>
  <hasColor rdf:resource="#Blue"/>
  <hasEngineStatus rdf:resource="#EngineNotStarting"/>
  <hasBatteryStatus rdf:resource="#BatteryOK"/>
  <creek:hasSolution rdf:resource="#AnotherSolution"/>
  <creek:hasCaseStatus rdf:resource="&creek;SolvedCase"/>
</creek:Case>

<creek:Case rdf:ID="CarCase3">
```

```
  <rdfs:label>This is a description of car problem</rdfs:label>
  <hasColor rdf:resource="#Green"/>
  <hasEngineStatus rdf:resource="#EngineNotStarting"/>
  <hasBatteryStatus rdf:resource="#LowBattery"/>
  <creek:hasCaseStatus rdf:resource="&creek;UnsolvedCase"/>
</creek:Case>

</rdf:RDF>
```

Appendix B

jCreek API

B.1 Package jcreek.representation.sw

B.1.1 Class **CREEK**

Vocabulary definitions from http://creek.idi.ntnu.no/owl/creek-vocabulary

B.1.1.1 Declaration

public class CREEK **extends** java.lang.Object

B.1.1.2 Fields

- public static OntModel m_model
  - –

    The ontology model that holds the vocabulary terms
- public static final String NS
  - –

    The namespace of the vocabulary as a string
- public static final Resource NAMESPACE

–

The namespace of the vocabulary as a resource

- public static final ObjectProperty structuralRelationInverse
  –
- public static final ObjectProperty Relation
  –

  all relations defined as owl:ObjectProperty also used as anonymous node to make n-ary relations
- public static final ObjectProperty subclassOf
  –

  realized through owl:subClassOf constructs
- public static final ObjectProperty hasSubcase
  –
- public static final ObjectProperty hasTask
  –
- public static final ObjectProperty structuralRelation
  –
- public static final ObjectProperty hasFunction
  –
- public static final ObjectProperty textualRelation
  –
- public static final ObjectProperty memberOf
  –
- public static final ObjectProperty hasOutcome
  –
- public static final ObjectProperty impliedBy
  –
- public static final ObjectProperty activityOf
  –
- public static final ObjectProperty instanceOf
  –

  realized through owl instanziation constructs
- public static final ObjectProperty temporalRelationInverse
  –
- public static final ObjectProperty hasValue
  –
- public static final ObjectProperty siteOfOccurance
  –

- public static final ObjectProperty hasMember
  –
- public static final ObjectProperty caseRelationInverse
  –
- public static final ObjectProperty involves
  –
- public static final ObjectProperty functionalRelationInverse
  –
- public static final ObjectProperty triggeredBy
  –
- public static final ObjectProperty functionalRelation
  –
- public static final ObjectProperty enablingRelation
  –
- public static final ObjectProperty solutionOf
  –
- public static final ObjectProperty achievedBy
  –
- public static final ObjectProperty causes
  –
- public static final ObjectProperty containedIn
  –
- public static final ObjectProperty movesBy
  –
- public static final ObjectProperty causesOccasionally
  –
- public static final ObjectProperty associationalRelationInverse
  –
- public static final ObjectProperty describedBy
  –
- public static final ObjectProperty caseStatusOf
  –
- public static final ObjectProperty spatialRelationInverse
  –
- public static final ObjectProperty associationalRelation
  –
- public static final ObjectProperty hasResult
  –
- public static final ObjectProperty spatialRelation
  –

- public static final ObjectProperty caseRelation
  –
- public static final ObjectProperty hasFinding
  –
- public static final ObjectProperty contains
  –
- public static final ObjectProperty outcomeOf
  –
- public static final ObjectProperty temporallyMeets
  –
- public static final ObjectProperty occursIn
  –
- public static final ObjectProperty reminding
  –
- public static final ObjectProperty transfers
  –
- public static final ObjectProperty findingOf
  –
- public static final ObjectProperty causedByOccasionally
  –
- public static final ObjectProperty RelationInverse
  –
- public static final ObjectProperty systemRelation
  –

  Relation used by the core Creek system
- public static final ObjectProperty hasURL
  –
- public static final ObjectProperty valueOf
  –
- public static final ObjectProperty makesSenseForDomain
  –

  realized through rdfs:domain contstruct
- public static final ObjectProperty temporallyEqual
  –
- public static final ObjectProperty hasInverse
  –

  realized through owl:inverseOf constructs
- public static final ObjectProperty temporallyContains
  –

- public static final ObjectProperty hasSubclass
  –

    realized through owl:subClassOf constructs
- public static final ObjectProperty temporalRelation
  –
- public static final ObjectProperty describes
  –
- public static final ObjectProperty enabledBy
  –
- public static final ObjectProperty during
  –
- public static final ObjectProperty systemRelationInverse
  –
- public static final ObjectProperty inheritsOver
  –
- public static final ObjectProperty before
  –
- public static final ObjectProperty temporallyMetBy
  –
- public static final ObjectProperty roleOf
  –
- public static final ObjectProperty Pointer
  –
- public static final ObjectProperty taskOf
  –
- public static final ObjectProperty hasContext
  –
- public static final ObjectProperty implies
  –
- public static final ObjectProperty subcaseOf
  –
- public static final ObjectProperty involvedBy
  –
- public static final ObjectProperty stateEffectRelation
  –
- public static final ObjectProperty causesAlways
  –
- public static final ObjectProperty performedBy
  –

- public static final ObjectProperty leadToBy
  –
- public static final ObjectProperty hasLessonsLearned
  –
- public static final ObjectProperty leadsTo
  –
- public static final ObjectProperty CausalRelation
  –
- public static final ObjectProperty isopodRelationInverse
  –
- public static final ObjectProperty hasRole
  –
- public static final ObjectProperty modeOfMovementFor
  –
- public static final ObjectProperty hasPredictiveStrength
  –

  predictive strength of a relation
- public static final ObjectProperty partOf
  –
- public static final ObjectProperty hasImportance
  –

  importance of a relation
- public static final ObjectProperty hasValueType
  –

  realized through rdfs:domain contstruct
- public static final ObjectProperty valueTypeOf
  –
- public static final ObjectProperty valueTypeOf2
  –
- public static final ObjectProperty hasComparator
  –
- public static final ObjectProperty comparatorOf
  –
- public static final ObjectProperty causesSometimes
  –
- public static final ObjectProperty hasPart
  –
- public static final ObjectProperty triggers
  –

- public static final ObjectProperty performs
  –
- public static final ObjectProperty stateOf
  –
- public static final ObjectProperty hasActivity
  –
- public static final ObjectProperty makesSenseForRange
  –

    realized through rdfs:range contstruct
- public static final ObjectProperty hasInstance
  –

    realized through owl instanziation constructs
- public static final ObjectProperty after
  –
- public static final ObjectProperty enablingRelationInverse
  –
- public static final ObjectProperty hasCaseStatus
  –

    status of a case's solution
- public static final ObjectProperty CausalRelationInverse
  –
- public static final ObjectProperty textualRelationInverse
  –
- public static final ObjectProperty associatedWith
  –
- public static final ObjectProperty achieves
  –
- public static final ObjectProperty temporallyOverlappedBy
  –
- public static final ObjectProperty temporallyStartedBy
  –
- public static final ObjectProperty causedBy
  –
- public static final ObjectProperty isopodRelation
  –

    Special relation used only internally in the ISOPOD structure
- public static final ObjectProperty temporallyStarts
  –

- public static final ObjectProperty hasSolution
  –
- public static final ObjectProperty contextOf
  –
- public static final ObjectProperty causedBySometimes
  –
- public static final ObjectProperty pointsAtDatabase
  –
- public static final ObjectProperty stateEffectRelationInverse
  –
- public static final ObjectProperty hasState
  –
- public static final ObjectProperty resultOf
  –
- public static final ObjectProperty enables
  –
- public static final ObjectProperty temporallyOverlaps
  –
- public static final ObjectProperty lessonsLearnedOf
  –
- public static final ObjectProperty hasFromType
  –

  realized through rdfs:range contstruct
- public static final ObjectProperty functionOf
  –
- public static final ObjectProperty hasAssociation
  –
- public static final ObjectProperty causedByAlways
  –
- public static final DatatypeProperty hasDefaultExplanationStrength
  –
- public static final ObjectProperty defaultExplanationStrengthOf
  –
- public static final OntClass UpdateKnowledgeModel
  –
- public static final OntClass CaseStatus
  –
- public static final OntClass Symbol
  –

- public static final OntClass StaticSituation
  –
- public static final OntClass State
  –
- public static final OntClass Text
  –

  text realized directly in the owl structure
- public static final OntClass SpatioTemporalContext
  –
- public static final OntClass Process
  –
- public static final OntClass PredictiveStrength
  –
- public static final OntClass PhysiologicalContext
  –
- public static final OntClass TemporalContext
  –
- public static final OntClass IdentifyNormalState
  –
- public static final OntClass Thing
  –

  Any thing in the world worth naming or characterising. Everything we want to talk about IN THE WORLD is a (subclass of) Thing. Differs from Concept in that everything in a CONCEPTUAL MODEL is a (subclass of) concept, including Thing itself.
- public static final OntClass TangibleObject
  –
- public static final OntClass Table
  –
- public static final OntClass ClassifyFailureState
  –
- public static final OntClass SpatialContext
  –
- public static final OntClass User
  –
- public static final OntClass ContextTag
  –
- public static final OntClass Task
  –
- public static final OntClass MentalContext

–

- public static final OntClass Entity
  –
- public static final OntClass PersonalContext
  –
- public static final OntClass BasicRepresentionalType
  –
- public static final OntClass ServiceProvider
  –
- public static final OntClass Case
  –

  a case is used in case-based reasoning
- public static final OntClass MentalObject
  –
- public static final OntClass RepairFailure
  –
- public static final OntClass Agent
  –
- public static final OntClass Context
  –
- public static final OntClass IntangibleObject
  –
- public static final OntClass DataManagementTask
  –
- public static final OntClass EnvironmentalContext
  –
- public static final OntClass SocialContext
  –
- public static final OntClass UserState
  –
- public static final OntClass FailureState
  –
- public static final OntClass TagContext
  –
- public static final OntClass NormalState
  –
- public static final OntClass NoValue
  –

  Note; same as owl:Nothing

- public static final OntClass DetermineMissingInformation
  –
- public static final OntClass Concept
  –
- public static final OntClass Role
  –
- public static final OntClass Parameter
  –

  Descibes the name and structure of value assignments
- public static final OntClass OperationalTask
  –
- public static final OntClass Importance
  –
- public static final OntClass IdentifyWarningState
  –
- public static final OntClass FormulateNewCase
  –
- public static final OntClass PhysicalObject
  –
- public static final OntClass EnvironmentalState
  –
- public static final OntClass Method
  –
- public static final OntClass PhysicalContext
  –
- public static final OntClass Event
  –
- public static final OntClass ManagementState
  –
- public static final OntClass ObserveParameters
  –
- public static final OntClass Activity
  –
- public static final OntClass ComputerAgent
  –
- public static final OntClass Goal
  –

  a particular end state associated with a Task
- public static final OntClass EquipmentState

–

- public static final OntClass UserContext
  –
- public static final OntClass AlarmState
  –
- public static final OntClass Environment
  –
- public static final OntClass AgentContext
  –
- public static final OntClass Action
  –
- public static final OntClass MathematicalExpression
  –
- public static final OntClass Number
  –

  number realized through xsd:decimal type
- public static final OntClass IdentifyAlarmState
  –
- public static final OntClass Set
  –
- public static final OntClass ClassifySituation
  –
- public static final OntClass Service
  –
- public static final OntClass DescriptiveThing
  –

  A meta-level thing that represents how other things (entities and relations) are described and represented.
- public static final OntClass PredictProblem
  –
- public static final OntClass Situation
  –
- public static final OntClass HumanState
  –
- public static final OntClass TaskContext
  –
- public static final OntClass String
  –
- public static final OntClass URL

–

- public static final OntClass JavaClass
  –
- public static final OntClass Comparator
  –
- public static final Individual URLComparison
  –
- public static final Individual SymbolComparison
  –
- public static final Individual CaseComparison
  –
- public static final Individual StringComparison
  –
- public static final Individual NumberComparison
  –
- public static final Individual Indicative
  –
- public static final Individual DefaultPredictiveStrength
  –
- public static final Individual Characteristic
  –
- public static final Individual Informative
  –
- public static final Individual Spurious
  –
- public static final Individual Sufficient
  –
- public static final Individual StronglyIndicative
  –
- public static final Individual DefaultImportance
  –
- public static final Individual Irrelevant
  –
- public static final Individual ProcessedCase
  –
- public static final Individual UnsolvedCase
  –
- public static final Individual Neccessary
  –
- public static final Individual SolvedCase
  –

B.1.1.3 Constructors

- *CREEK*
  `public` **CREEK**`( )`

B.1.1.4 Methods

- *getURI*
  `public static String` **getURI**`( )`
  - **Usage**
    *

      The namespace of the vocabulary as a string

B.1.2 Class **OWLexportParser**

Does the parsing of the Creek Knowledge Model and extracts the complete model to OWL.

B.1.2.1 Declaration

public class OWLexportParser **extends** java.lang.Object

B.1.2.2 Constructors

- *OWLexportParser*
  `public` **OWLexportParser**`( KnowledgeModel` **km**`, java.lang.String` **namespace** `)`
  - **Usage**
    - Does set some basic settings before any parsing may begin.
  - **Parameters**
    - `km` - the Knowledge Model in Creek to export
    - `namespace` - the namespace for the ontology

B.1.2.3 Methods

- *addConcept*
  `public OntResource` **addConcept**`( Entity` **kmConcept** `)`
  - **Usage**
    - Adds a Creek concept as a Class to OntModel

      Adds all relations from this concept

      Recursive call to all relations from and concepts related to this concept
  - **Parameters**
    - `kmConcept` - Creek Entity
  - **Returns** - The OntClass that were added to the OntModel
- *addProperty*
  `public Property` **addProperty**`( Entity` **kmRelation** `)`

- **Usage**
  - Adds a Creek relation as an ObjectProperty to OntModel if it is not added already

    Adds owl:inverseOf, rdfs:subPropertyOf, creek:hasDefaultExplanationStrength, rdfs:label and rdfs:comment properties of the relation

    Recursive call to parent property and inverse property

- **Parameters**
  - `kmRelation` - Creek Relation
- **Returns** - The OntModel Property that were added, or the corresponding property if it is allready added.

- *changeName*
`public String` **changeName**`( java.lang.String` **name** `)`
  - **Usage**
    - Translates a name from Creek form to OWL/Java etc. form, e.g. 'has finding' ->'hasFinding'
  - **Parameters**
    - `name` - Name in Creek form
  - **Returns** - name in OWL/Java etc form

- *createOntology*
`public void` **createOntology**`( )`
  - **Usage**
    - Starts the creation of the OWL document.

- *nameToURI*
`public String` **nameToURI**`( java.lang.String` **name** `)`
  - **Usage**
    - Translate Creek Entity name to URI.

      Removes spaces and capitalize character after space in names

      Adds namespace to name, using default namespace

      If name is already a valid URIref, just return it
  - **Parameters**
    - `name` - Entity name in Creek knowledgemodel
  - **Returns** - URIref for Entity

- *nameToURI*
`public String` **nameToURI**`( java.lang.String` **name**`, java.lang.String` **namespace** `)`
  - **Usage**

* Translate Creek Entity name to URI.

  Removes spaces and capitalize character after space in names

  Adds namespace to name

  - **Parameters**
    * `name` - Entity name in Creek knowledgemodel
    * `namespace` - Namespace URI
  - **Returns** - URIref for Entity

- *write*
  `public void` **write**`( )`
  - **Usage**
    * Writes the Jena OntModel to stdout

- *write*
  `public void` **write**`( java.lang.String` **fileName** `)`
  - **Usage**
    * Writes the Jena OntModel to disk
  - **Parameters**
    * `fileName` - Path to file

B.1.3 Class **OWLimportParser**

Does the parsing of an OWL ontology and imports it into a Creek km.

Creates a new empty ISOPOD model before importing

B.1.3.1 Declaration

public class OWLimportParser **extends** java.lang.Object

B.1.3.2 Constructors

- *OWLimportParser*
  `public` **OWLimportParser**`( java.lang.String` **importURI**`, KnowledgeModel` **km** `)`
  - **Usage**
    - Does the parsing of an ontology and imports it into a Creek knowledge model
  - **Parameters**
    - `importURI` - the URI of the ontology
    - `km` - the knowledge model to import into

B.1.3.3 Methods

- *addConcept*
  `public String` **addConcept**`( OntResource` **concept** `)`
  - **Usage**
    - Adds a OntResource (Class or Individual) from the ontology as an Enitiy in the knowledge model
  - **Parameters**
    - `concept` - the Class or Individual from the Ontology to add
  - **Returns** - the name of the entity added
- *addRelationType*
  `public String` **addRelationType**`( OntProperty` **prop** `)`
  - **Usage**
    - Adds the properties of the ontology as new relation types in Creek Adds inverse, super property, default explanationstrength, transfers and inherits over relations

  - **Parameters**
    * `prop` - the Ontology property to add
  - **Returns** - the name of the property added

- *importOntology*
`public void` **importOntology**`( )`
  - **Usage**
    * Starts the importing. Goes through all ObjectProperty's, DatatypeProperty's, Class'es and Individual's in the model and adds them to the knowledge model.

- *URItoName*
`public String` **URItoName**`( java.lang.String` **URI** `)`
  - **Usage**
    * Translates a URI to the format used in jCreek
  - **Parameters**
    * `URI` - Name of resource
  - **Returns** - Name in the form used internally in jCreek

B.2 Package jcreek.cke.importexport.owl

B.2.1 Class **OWLexport**

This class is the one having the overall control of the export process of the Knowledge Model of Creek to OWL ontology representation

Copied from JCXMLexport by Tomassen, 2002

B.2.1.1 Declaration

public class OWLexport **extends** java.lang.Thread **implements** jcreek.cke.importexport.ProcessControlInterface

B.2.1.2 Constructors

- *OWLexport*
  `public` **OWLexport**`( java.awt.Frame` **parent**`, jcreek.representation.KnowledgeModel` **km** `)`
  - **Usage**
    - Constructor for OWLexport, which does set some basic settings before the process of exporting the Knowledge Model may start.
  - **Parameters**
    - `parent` - the parent frame window.
    - `km` - the JavaCreek knowledge model to export the model to.

B.2.1.3 Methods

- *abortProcess*
  `public void` **abortProcess**`( )`
  - **Usage**
    - Aborts the export process.
- *isAborted*
  `public boolean` **isAborted**`( )`
  - **Usage**
    - Checks if the process was aborted;
  - **Returns** - true if the porcess was aborted, otherwise false.

- *startExport*
  `public void` **startExport**`( )`
  - **Usage**
    - Starts the process by first showing the FileChooser dialog then starts the export process, by first starting the thread and gives feedback of the process to the OWL status dialog.

B.2.2 Class **OWLimport**

This class is the one having the overall control of the import process of OWL files to the Knowledge Model of Creek.

Copied from JCXMLimport, Stein L. Tomassen, NTNU 2002

B.2.2.1 Declaration

public class OWLimport **extends** java.lang.Thread **implements** jcreek.cke.importexport.ProcessControlInterface

B.2.2.2 Constructors

- *OWLimport*
  `public` **OWLimport**`( java.awt.Frame` **parent**`, jcreek.representation.KnowledgeModel` **km** `)`
  - **Usage**
    - Constructor for OWLimport, which sets some basic settings.
  - **Parameters**
    - `parent` - the parent frame window.
    - `km` - the JavaCreek knowledge model to import the model to.

B.2.2.3 Methods

- *abortProcess*
  `public void` **abortProcess**`( )`
  - **Usage**
    - Aborts the import process. This is the implementation of the method from 'ProcessControlInterface'. This method could be used by other classes to abort a process.
- *isAborted*
  `public boolean` **isAborted**`( )`
  - **Usage**
    - Checks if the process was aborted;
  - **Returns** - true if the porcess was aborted, otherwise false.
- *startImport*
  `public void` **startImport**`( )`

- **Usage**
  - Starts the process by first showing the URI input dialog then starts the import process, by first starting the thread and gives feedback of the process to the OWL status dialog.