NTNU
Innovation and Creativity

Reusing External Library Components in the Creek CBR System

Erik Stiklestad

Master of Science in Computer Science
Submission date: June 2007
Supervisor: Agnar Aamodt, IDI
Co-supervisor: Frode Sørmo, Volve AS

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The Creek system has an architecture that facilitates combined case-based and model-based reasoning. jColibri, developed in the CBR group of Universidad Complutense in Madrid, contains a library of CBR system components intended for sharing and reuse, and an ontology (CBROnto) of CBR methods for explicit modelling of a CBR system's operation. In this master degree project, the Creek framework and the jColibri structure shall be compared with the aim of developing a mechanism for importing jColibri components into Creek, so that they can be integrated into a running Creek system. The mechanism shall be exemplified through selection of a few (two or more) specific components, and integration of these components into an implemented demonstrator system.

Assignment given: 20. January 2007
Supervisor: Agnar Aamodt, IDI

Abstract

The Creek system has an architecture that facilitates combined case-based and model-based reasoning. The jColibri system, developed by the CBR group of Universidad Complutense in Madrid, contains a library of CBR system components intended for sharing and reuse. The system also contains an ontology (CBROnto) of CBR tasks and methods for explicit modelling of a CBR systems, in addition to general CBR terminology. In this master degree project, Creek and jColibri are compared with the aim of developing a mechanism for importing jColibri components to Creek, so that they can be integrated into a running Creek system. The mechanism is exemplified through selection of a few specific components, and integration of these components into an implemented demonstrator system. In addition, efforts needed to bring Creek into the jColibri framework are identified.

Preface

This document presents the work by Erik Stiklestad in TDT4900, which is a Master's thesis in Computer Science (Datateknikk). It is written for the Artificial Intelligence and Learning Group (AIL) at the Norwegian University of Science and Technology (NTNU), and the software company Volve AS.

The goal is to analyze and compare Creek and jColibri with the aim of developing a mechansim for importing jColibri components into Creek, so that they can be integrated into a running Creek system. In addition, efforts needed to bring Creek into the jColibri framework shall be identified.

I would like to thank my supervisor Agnar Aamodt from NTNU and co-advisor Frode Sørmo from Volve AS for their good and patient guidance. Thanks also to the employees of Volve AS, for letting me work in their offices during the most technical period.

Trondheim, 2007-06-17.

Erik Stiklestad

Contents

List of Figures

Chapter 1

Introduction

The introduction chapter describes the background and motivation for the project, before defining its goals. The methodology is also described, and a structural overview of the project report is provided.

1.1 Background and Motivation

One of the main research areas for the Artificial Intelligence and Learning Group (AIL) at NTNU is Case-Based Reasoning (CBR). The research has a focus on knowledge-intensive approaches, and the Creek system has been developed. Creek facilitates combined CBR and Model-Based Reasoning (MBR). The model contains knowledge about a domain in general, while a collection of cases describe specific problem situations.

Recent development of Creek by Volve AS[1] customizes the system to be used while drilling for oil. The system will help avoid unwanted events by giving a warning when real-time data is getting dangerously similar to problem situations. The problem situations are based on both historical data and the experience of domain experts. To reason within this domain there is built a general domain model, while the problem situations are represented as cases. Combining the two, we have CBR and MBR in one system.

jColibri, developed by the Group for Artificial Intelligence Applications (GAIA[2]) at the Universidad Complutense de Madrid, contains a library of CBR system components intended for sharing and reuse. In addition, it has

[1] http://www.volve.no/
[2] http://gaia.fdi.ucm.es/

an ontology (CBROnto) containing general CBR terminology and knowledge about tasks and Problem Solving Methods (PSMs). jColibri attempts to formalize CBR and become the standard for CBR system development.

The Creek developers became interested in jColibri after learning about its goals. They would like to see if the two systems are able to cooperate on some level, since that would be positive for both systems. In addition, because of jColibri's goal to formalize CBR, it is also interesting to see if Creek can adapt to jColibri's framework.

1.2 Goals

In this project, Creek and jColibri will be analyzed and compared with the goal of developing a mechanism for importing jColibri components to Creek. The mechanism will be exemplified through the selection of a few specific components, and integration of these components into a demonstrator system. In addition, efforts needed to bring Creek into the jColibri framework will be identified.

- Analyze Creek and jColibri, and identify their similarities and differences;
- Construct a mechanism able to import jColibri components into the existing Creek implementation;
- Create and test a demonstrator system featuring a few selected components;
- Discuss efforts needed to bring the Creek system into jColibri.

1.3 Methodology

This project will be based on analytical and experimental methods, with some general background theory covered at the start. A lot of time will be used to analyze Creek and jColibri. The analysis will be both conceptual and close to the implementation, and will result in a comparison essential to the project. Based on the comparison, the rest of the project will consist of looking at various solutions to accomplish the project goals.

1.4 Structure of the Report

The first chapter of the project report is this introduction, which defines the project goals and describes how and why we want to accomplish them. Chapter two describes and introduces the main research areas. This includes the two systems that are thoroughly analyzed in the third chapter, in addition to a short introduction to CBR and ontologies. Chapter three is closed with a system comparison, which is used by the fourth chapter to construct a possible solution and a demonstrator system. This solution is implemented in chapter five. The demonstrator system is used to test the solution in the sixth chapter. The report continues with a discussion in chapter seven, before being closed with further work and a conclusion in the eighth and final chapter.

1.5 Summary

This project is motivated by NTNU's focus on CBR, and the recent interest in the jColibri system. The goals are to import components from jColibri to Creek, and to identify efforts needed to bring Creek into jColibri. To accomplish these goals, the two systems will be analyzed and compared. A demonstrator system will be implemented to test a possible solution.

Chapter 2

Research Focus

This chapter presents the research focus for the project. Although readers are assumed familiar with CBR and ontologies, we will start by introducing them. The rest of the chapter is devoted to the background, history and motivations of the two systems COLIBRI and Creek.

2.1 Case-Based Reasoning

Case-Based Reasoning (CBR) is an approach to problem solving and learning. When solving a new problem, this approach makes use of previously solved problems when reasoning. The previously solved problems are also referred to as *experiences*. After the new problem has been solved, it is retained in the system as additional experience. The latter step represents *learning*.

A problem is what we refer to as a *case* in CBR, and both new problems and old experiences are cases. A case is described by a set of *features* which in sum defines the problem. A feature can be anything giving relevant information about the case.

CBR is typically done in a four-step cycle as shown in figure 2.1 taken from [AP94]. First, we *retrieve* all learned cases (experiences) that are relevant for solving a new problem that entered the system. To find out which cases are relevant, we compare the new problem to the learned cases by comparing their features. The best matching case or cases are chosen, which finalizes the retrieval step. Second, we *reuse* the chosen case's solution by copying it or adapting it to fit our needs. If several cases were retrieved then we adapt a solution by combining parts of the retrieved cases. Third, we *revise* how

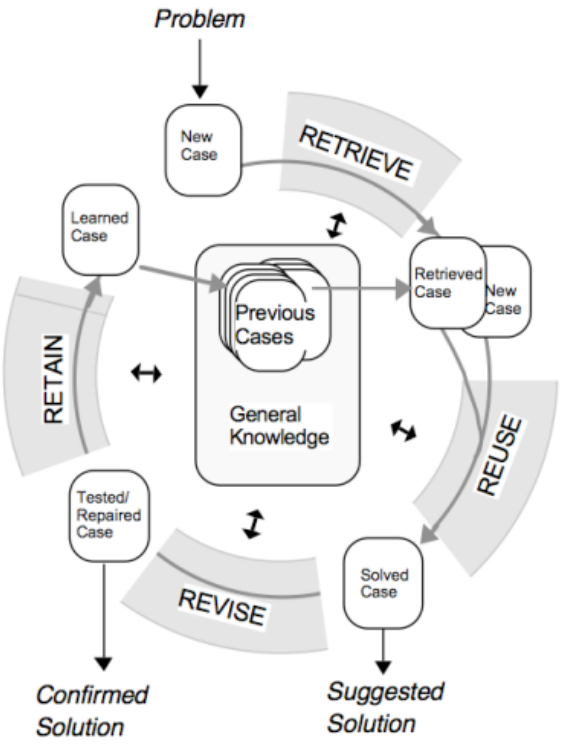

Figure 2.1: The four-step CBR cycle

well the new problem was solved with the solution we just reused. This is done by testing it in the real world or some kind of test scenario. Finally, we *retain* this new experience as a case in our system for future problem solving.

This approach is very similar to how humans reason when solving problems.

2.2 Ontologies

A common definition of an ontology states that it is *"a specification of a conceptualization"* [Gru93]. Related to computer science, an ontology can be seen as a data model representing a set of concepts within a domain, and the relationships between them. We can use ontologies as a form of knowledge representation for our domain, and use its components when reasoning.

An ontology generally describes individuals, classes, attributes and relations. The individuals are the basic components of the ontology and may include concrete objects such as specific cars, or abstract things like words.

An ontology does not necessary have any individuals, since an ontology may be created to provide a way to classify individuals in several systems sharing the same model (a general purpose ontology). The classes collect individuals and other classes. E.g., a class `car` can collect individuals `car#1` and `car#2`, while the class `vehicle` can collect classes `car` and `truck`. This would make `car` and `truck` subclasses of `vehicle`. Attributes are characteristics of an individual. E.g., `car#1` can have the attribute `red` to describe its color. Relationships describe how things relate to each other. A possible relationship between `car#1` and `car#2` could be that one is the successor of the other.

If we create general ontologies about a domain, we can reuse the ontologies in all systems reasoning within that domain. Application specific ontologies can later be mapped to the more general ontologies, creating several layers of specificity. Reasoning mechanisms defined for a general concept or relation of an ontology can then automatically be used for more specific ones because of inheritance.

If we want to integrate several systems, or simply make them work with each other at some level, it is a huge benefit if they are based on the same ontology. This is the essential motivation behind CBROnto in the COLIBRI system which will be presented in the next section.

2.3 COLIBRI

In 2002, Belén Díaz-Agudo proposed a domain independent architecture called COLIBRI[1] in her PhD thesis directed by Pedro González-Calero. COLIBRI tries to formalize CBR, and provide design assistance when creating KI-CBR systems. A system may combine domain specific knowledge with various knowledge types and reasoning methods common to all domains.

Very important to the COLIBRI system is CBROnto, which is an ontology containing general CBR terminology. It is also a task and method ontology. The root of CBROnto is `CBRTerm`, which all concepts of the ontology are specialized from.

The idea is that COLIBRI should be based on Knowledge Acquisition (KA) from a library of application independent ontologies, and that these ontologies should be mapped to CBROnto by the system designer. More specifically, the designer should classify the domain knowledge's concepts and

[1] Cases and Ontology Libraries Integration for Building Reasoning Infrastructures

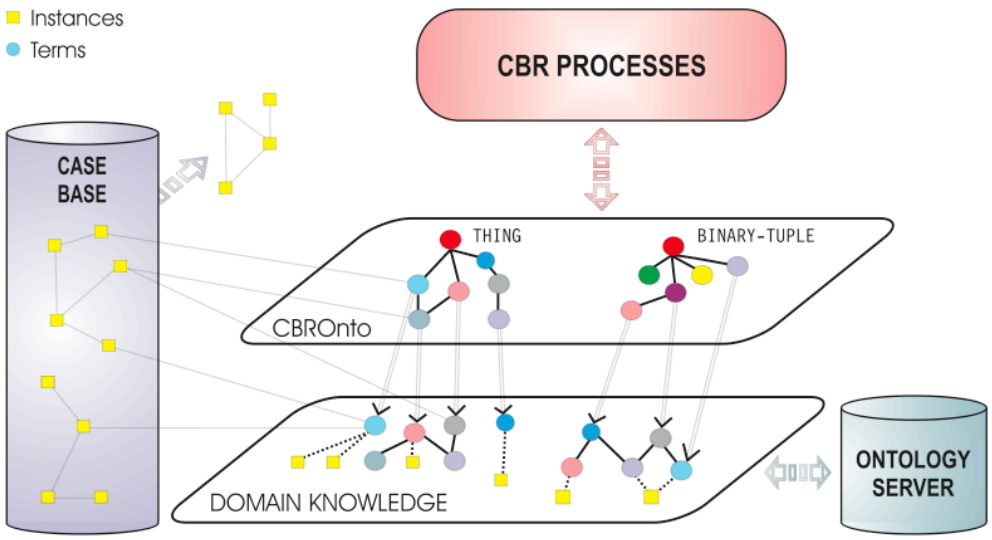

Figure 2.2: Integrating domain ontologies and CBR processes

relations to CBROnto. Since we are dealing with a hierarchical structure, only the top level concepts and relations of the domain knowledge need to be classified. The rest is solved automatically through inheritance.

Since CBROnto describes both tasks and methods, there are no gaps between the system's goals (tasks) and the Problem Solving Methods (PSMs) used to accomplish them. This has been an issue with older systems. The tasks define the structure of an application, and how it will be executed. A PSM can either decompose a task into subtasks, or solve it directly. Subtasks are in turn solved by other PSMs, and this process continues until all tasks are solved. This means that we need a resolution PSM for all tasks that are not decomposed, or the system will not be able to execute successfully. The approach is inspired by the *Components of Expertise* methodology [Ste90].

Further, the gap between the PSMs described by CBROnto and the domain knowledge is removed by the classification done by the system designer. See Figure 2.2 taken from [Dia00]. This solves another important issue with earlier CBR systems.

It is also important to address a PSM's dependency on knowledge. If a resolution PSM with the competence to accomplish a certain task does not have the knowledge necessary to do so, we have a problem. This is solved using the classification mechanism, checking if the necessary knowledge is available when a certain PSM will be executed. We know what knowledge

is needed since each PSM is defined with a set of conditions.

The idea behind CBROnto is to create a common data model which is able to represent all CBR systems. This way, CBROnto is an attempt to generalize and formalize CBR by making a domain independent ontology. CBROnto tries to contain all general CBR terminology, and support the semantic needs for all CBR systems. When creating a CBR system, CBROnto will guide the case representation, and help describe flexible, generic and reusable PSMs. At the top level are the four well known CBR steps which can be seen in Figure 2.1: Retrieve, Reuse, Revise and Retain. Each task can be decomposed or solved directly like mentioned earlier.

CBR concepts from CBROnto are implemented as abstract classes or interfaces in the COLIBRI framework. Typically, the `is-a` relations from the ontology are implemented using inheritance between classes, and the `part-of` relations are implemented as a composition of classes. By doing this, we have an implementation which represents concepts from the ontology, providing two main things. First off they give us an abstract interface for CBR methods and tasks. They can be developed independently from the actual CBR components such as case structure, similarity functions and so forth. Second, they serve as hooks where new types of building blocks can be added.

COLIBRI's implementation is based on the reasoning capabilities of Description Logics (DL). Originally, it was implemented with LOOM[2] which is a knowledge representation language developed specifically for artificial intelligence. It contains a set of advanced tools for knowledge representation and reasoning. The LOOM implementation assumed rather advanced users, and that is why a new implementation was started. The new implementation was given the name jColibri[3], and it has a distributed architecture, a DL engine, a GUI for non-technical users and an object-oriented framework in Java. The ontologies are represented using the Web Ontology Language (OWL). This implementation will be analyzed in the next chapter.

2.4 Creek

In 1991, Agnar Aamodt developed the CREEK[4] architecture in his PhD thesis, normally written "Creek". It is an architecture for knowledge inten-

[2]http://www.isi.edu/isd/LOOM/

[3]Should be written *jCOLIBRI*, but for improved readability we will write *jColibri* in this project report

[4]Case-based Reasoning through Extensive Explicit Knowledge

sive problem solving and sustained learning [Creek]. Since the Creek system is well known to the readers of this project report, it will be given less focus than the COLIBRI system.

The basis of the Creek knowledge representation is a graph. A graph is a pure mathematical model with a set of objects (called *nodes*, *points* or *vertices*) connected by links (called *edges* or *lines*). If we give meaning to the objects and links, we get a semantic network which is what Creek is using. Henceforth, objects will be referred to as *entities* and the links as *relations*. If we collect all relations connected to a certain entity, we have a *frame*. In short, Creek's frame based knowledge model is a semantic net of entities interconnected by relations.

Creek is also using ontologies in an attempt to generalize certain domains. At the very top level, we have the `Thing` concept. Everything in the world is a `Thing`, meaning that it is the most general term in the model. The knowledge located in the ontologies can be about CBR in general, or well established knowledge about a specific domain. This knowledge may be application independent, and can potentially be reused.

Recently, it has been created a case model for Creek. This model contains general CBR terminology. The model is important to this project, since it contains knowledge needed to use recently developed packages for Creek. This involves general CBR things like cases, attributes and so forth.

The version of Creek which will be used in this project, is a development snapshot from Volve AS and will be called VolveCreek in this report. Some components of the system are not fully developed. VolveCreek will be analyzed in the next chapter.

2.5 Summary

CBR and ontologies are important research areas to this project. CBR is an approach to problem solving and learning, while ontologies are used to create knowledge models. The two systems which will be analyzed both conceptually and close to their implementation in the next chapter are COLIBRI and Creek. COLIBRI uses the reasoning capabilities of DL, and is based on the use of CBROnto which is an ontology with general CBR terminology, in addition to being a task and method ontology. OWL is used to represent the ontologies. Creek uses a frame based knowledge representation which is a semantic net of entities interconnected by relations.

Chapter 3

Software Analysis

The software analysis chapter analyzes jColibri and VolveCreek, which are Java implementations of the COLIBRI and Creek systems, respectively. The analysis is rather close to their implementation, while their background was presented in chapter 2. A more practical introduction of the systems can be found in [RSDG05], [BSAB04] and [Bra04]. These were used to learn how to use the systems.

3.1 jColibri

jColibri is a Java implementation of the COLIBRI system introduced in section 2.3, and is intended for a large audience. Anyone from new students to technically advanced users should be able to use it on some level. It is possible to prototype and test CBR systems very quickly using jColibri's GUI, which makes it significantly more user friendly than its predecessor implemented in LOOM. It is also designed to support anything from the simplest CBR systems to the large and complex ones. With its focus on reuse, jColibri makes it easy to take advantage of past designs when creating a new system. It is semi-complete and ready to be extended for custom applications.

This chapter will analyze all major components of the jColibri framework, and describe how they work together when creating and running a CBR application. Components that are particularly important to this project will be analyzed in greater detail. These are tasks (`CBRTask`), PSMs (`CBRMethod`) and similarity functions (`CBRSimilarity`). They all implement the `CBRTerm` interface which represents the most general concept of CBROnto. Data types are also important.

We will start by explaining how the representation is supported, before moving on to the specific components.

3.1.1 Representation

The jColibri system is based on Description Logics (DL) [NB03], which can be translated to first-order predicate logic. It is, in other words, a representation with logic-based semantics. This type of representation works best in domains with a strong domain theory. This is typically domains that can be modelled in a formally well-understood way [DINS96] [GGDF99].

To support the representation (see section 2.3), and as a parallel to the use of ontologies, jColibri has an interface called `Individual`. An individual has a collection of relationships to other individuals in addition to parents and a value.

A class `SimpleIndividual` has been implemented and is currently being used by the example applications, but new individuals can of course be implemented.

The class `IndividualRelation` implements the relation concept between individuals. A relation has a description, target and a weight. When, e.g., giving a case a set of attributes, we create a relation from the case to its attributes. Both the case and the attributes are `Individual` objects, and their relationships between them are `IndividualRelation` objects.

This is a very general way to support the representation. How the individuals are used will be described further in later sections, and in particular section 3.1.4 about cases and section 3.1.6 where their comparison functions are described.

The OWL DL reasoner used by jColibri is called PELLET [SPGKY07].

3.1.2 The Core

The core of jColibri is called `CBRCore`, and can be seen in figure 3.1 taken from [jColibri]. It is the most important component of the framework. The core is in charge of the application, and must always be present for an application to run. It handles the configuration, and also executes the application. To do all this, the core is divided into three main components which will be described in turn: state, context and packages.

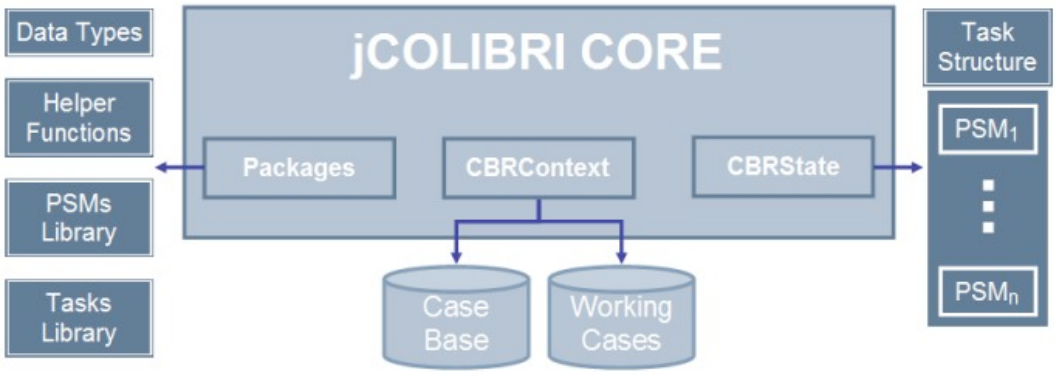

Figure 3.1: The jColibri Core

State

The state, which is called `CBRState`, handles the configuration of tasks and methods. It will always have the current configuration status of the CBR application.

Context

The context, called `CBRContext`, acts as a communication blackboard where methods can share data during the execution of an application. The context will have the case base and the working cases. What the working cases are depends on the execution step, but they can, e.g., be newly retrieved cases, adapted cases, and so forth.

jColibri also features a context checker, which ensures that the components configured for an application are compatible with the context at every moment during development. The final application configuration is sure to satisfy each component's conditions because of the context checker. E.g., a PSM may be given preconditions and postconditions, defining which conditions the PSM is dependent on before and after its execution. This is part of the solution for an issues that was introduced in section 2.3: a PSM's dependency on knowledge to accomplish a task.

Packages

The remaining components of the system are located in packages. Examples of components located in these packages are data types, similarity functions, case structures, PSMs and so forth. Since these components are rather

complex and important to this project, they will be described in further detail in the following sections.

Each package may contain a set of one or more components. jColibri comes with a few rather stable packages at this time: `core`, `textual`, `description logics` and `web`. The core package should always be enabled for an application, and the GUI does this by default and in addition lets the system designer enabled others as the first step.

An essential goal of this project is to import parts of these packages to the VolveCreek system.

3.1.3 Data Types

Data types are important in any computer system. Knowing the data type of something enables us to make assumptions, and this is important also in CBR. One obvious example is when we want to compare two cases by applying a similarity function to two values. Which function to use is highly dependent on the data type. To define which data type each attribute is, we specify them already when we create the case structure.

jColibri comes with a set of data types in the core package which covers all common data types, and in addition the DL extensions provides a `ConceptType` data type. The system designer is able to configure new data types using the GUI, or by writing the XML configuration file manually. A data type is configured with a name, Java object and the identity of a GUI editor. The name can be anything, the Java object must exist and hold this specific type of data, and a GUI editor should be chosen to let users enter values when this data type is asked for. Example XML configuration format for two data types follows. The data types are `Boolean` and `String`, provided by the `core` package.

```
<DataTypes>
  <DataType>
    <Name>Boolean</Name>
    <Class>java.lang.Boolean</Class>
    <GUIEditor>jcolibri.gui.editor.BooleanEditor</GUIEditor>
   </DataType>
   <DataType>
     <Name>String</Name>
     <Class>java.lang.String</Class>
     <GUIEditor>jcolibri.gui.editor.StringEditor</GUIEditor>
   </DataType>
</DataTypes>
```

3.1.4 Cases

The most important thing in any CBR system is the cases, and hence it is also important how they are represented. With jColibri it is possible to create anything from simple plain cases to the most complex hierarchical structures with attributes connected. This case structure is important. It is for example used when loading cases into the system from a case persistency (see section 3.1.5), and when obtaining a query[1] from the user before the retrieve CBR step.

Following the definition provided by CBROnto, a case has a `Description`, a `Solution` and a `Result`. `Description` describes the problem by using a set of attributes. `Solution` is also a set of attributes, but describes the solution of the problem. `Result` stores the consequence of applying the solution in the real world or a test scenario. The result may be good or bad, depending on if the solution actually solved the problem, or if it did not. We can see how this definition of a case makes sense if comparing it to the description of CBR in section 2.1.

An attribute can be either simple or compound. The case structure can be compared to a tree structure, where leaf-nodes are simple attributes, internal nodes are compound attributes and the `Description` and the `Solution` are the root nodes.

Simple attributes have a name, type, weight and local similarity function. The name can be anything, type is a data type, weight says how important the attribute is relative to the others, and finally the local similarity function refers to a similarity function used to compare two instances of this attribute.

Compound attributes collect simple attributes, and has a name and global similarity function. The name can again be anything, while the global similarity function is a function calculating the collected similarity of all simple and compound attributes below it in the case structure.

jColibri stores the case structure in an XML file, which can be written manually or generated by a GUI tool. The GUI makes it easy by listing only the available data types and similarity functions, and it will also load ontologies and let the system designer select concepts directly when building the case structure. An example case structure from [Sti06] follows.

[1] The query has the same structure as a case, and will be compared to the other cases as if it was a case.

```
<Case concept="carCase">
  <Description globalSim="Average">
    <SimpleAttributeConcept localSim="Equal" name="colour"
         relation="hasColour" type="Concept" weight="0.1"/>
    <SimpleAttributeConcept localSim="Equal" name="batteryStatus"
         relation="hasBatteryStatus" type="Concept" weight="0.8"/>
    <SimpleAttributeConcept localSim="Equal" name="engineStatus"
         relation="hasEngineStatus" type="Concept" weight="0.5"/>
  </Description>
  <Solution globalSim="Average">
    <SimpleAttributeConcept localSim="Equal" name="ignoreIt"
         relation="solutionOf" type="Concept" weight="1.0"/>
    <SimpleAttributeConcept localSim="Equal" name="rechargeBattery"
         relation="solutionOf" type="Concept" weight="1.0"/>
  </Solution>
  <Result/>
  <Reasoner>
    <Type>PELLET</Type>
    <Source>
       src/jcolibri/application/CreekExample/Ontology.owl
    </Source>
  </Reasoner>
</Case>
```

The case structure is for a car being described with attributes `colour`, `batteryStatus` and `engineStatus`. The solution can either be `ignoreIt` or `rechargeBattery`. The case structure is created to retrieve car cases directly from an OWL file by using the PELLET reasoner.

[SRDG05] provides more details about the case structures in jColibri.

3.1.5 Connectors and Case Bases

When loading cases into the system, jColibri uses a two-layer model which can be seen in figure 3.2 taken from [jColibri]. A connector is an object which has the ability to access and retrieve cases from a specific case persistency when given the case structure, and give those cases to the CBR system in a standardized way. Because of this, jColibri can deal with any case persistency as long as a connector is provided.

The first layer is the case persistency and can be plain text, XML, ontologies, a relational databases or anything else we have a connector for. Since all connectors feed cases into the system in the same way, it does not matter to the CBR system which case persistency is used. The second layer is the in-memory organization.

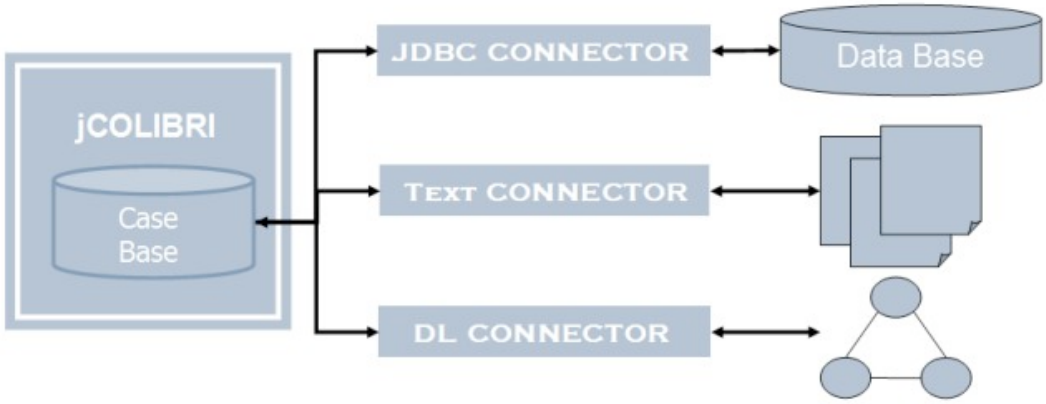

Figure 3.2: The jColibri connector architecture

3.1.6 Helper Functions

Helper functions assist the PSMs when they try to accomplish tasks, and may be domain dependent although they do not have to be. Similarity functions are the most important helper functions, but there may also be others. Although jColibri does not include any other helper functions than the similarity functions at this time, an example could be adaption functions.

Similarity Functions

When comparing two cases, the PSMs use the similarity functions to compare each attribute. The local similarity functions are used for simple attributes, and global similarity functions are used for compound attributes. The *local similarity* and *global similarity* values given to attributes in the case structure like explained in the previous section, must refer to the name of an implemented similarity function. When using the GUI tool to create the case structure, this constraint is taken care of by only letting the user select similarity functions that are available and implemented. Implemented similarity functions may be unavailable to a certain application if the function is part of a package that is not enabled. E.g., a textual similarity function like `TokensContained` is not available unless the textual extension is enabled.

Following CBROnto, jColibri has a `CBRSimilarity` abstract class which implements `CBRTerm`. It represents the similarity concept inside the Java framework. A `CBRSimilarity` has a name and a class name which refers to the

class implementing the similarity function. There are two classes extending `CBRSimilarity`, and those classes are of course `CBRGlobalSimilarity` and `CBRLocalSimilarity`.

The actual similarity functions are imported by the global or local similarities, and they are implementations of the interface `SimilarityFunction`. This interface does not exist in CBROnto, but is part of jColibri to ensure that the similarity functions are implemented in a consistent way. Each similarity function must have a `compute` method taking two individuals as parameters, and then returning the similarity between the two individuals as the data type `double`.

The similarities also have parameters, and they are implemented a class called `CBRSimilarityParam`. The similarity parameters simply have a name and the value. These parameters are imported by `CBRSimilarity` to make them available to both global and local similarities. It is typically used to set variables in functions that depend on, e.g., the size of the case base or anything else that may be used to compute the similarity.

3.1.7 Tasks and PSMs

This section describes the tasks and methods on a conceptual level, before looking at the implementation. These components describe the structure of the CBR system, and hence also the behavior. They are very important and centralized in the jColibri system.

Conceptual

CBROnto has a task decomposition structure like mentioned in section 2.3. Everything starts with a *root task* which is decomposed into the four well known CBR tasks from [AP94]: retrieve, reuse, revise and retain. Each of these are decomposed or solved directly by PSMs, and this process does not stop until all tasks are either decomposed or have been assigned a resolution PSM. See Figure 3.3 from [Cha90] and [Dia02].

The CBR system's designer will do this process manually, and again the GUI is of great help. When selecting a task, the designer is given a list of methods with the competence to solve the task. The competence is defined in the ontologies, and decomposition methods also have the new subtasks defined here. If a decomposition method is selected, the new subtasks will appear immediately in the GUI.

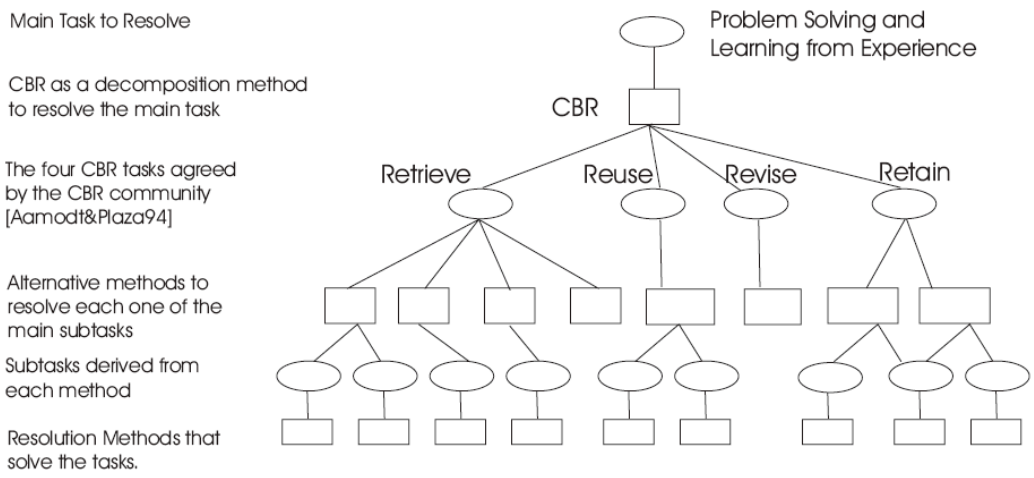

Figure 3.3: The CBR task and method structure

Normally, a method is divided into three main parts: competence, operational specification and requirements. The competence says which tasks the method is able to accomplish (*what* can be solved). The operational specification describes the method's way of delivering a specific competence (*how* something can be solved). Finally, the requirements describes which knowledge is needed to achieve the given competence by going through its reasoning process.

The competence and requirements of a method are described using ontologies, which provide two advantages. A formal description giving precise meaning combined with reasoning support is the first advantage. The second, which is also a general focus in jColibri, is that it enables reuse since they can be used by different systems.

In CBROnto there is a method concept, and each method is an instance of it. The internal reasoning process of the methods are not formalized in the ontologies, but each method has an associated Java class which implements it. The methods have a name, informal description, type and a relationship to a number of other concept instances.

In jColibri, the name is equal to the class implementing the method. This creates an association between the ontology and the Java implementation. How it is implemented will be discussed in the following section. The informal description can be anything.

The method type can be either resolution or decomposition. In the case of decomposition, it will also have a set of subtasks, which will be solved but other methods later. To represent a method's competence, it has a relation

to an instance of the task concept it is able to solve.

The methods may also have several parameters, which may typically be a case structure, a connector or simply an attribute from the `Description` or `Solution` concepts of the case (see section 3.1.4).

Instances of the task concept only has a name and a description. They are identified by their name.

Implementation

Again following CBROnto, the classes `CBRTask` (tasks) and `CBRMethod` (methods) implements the `CBRTerm` (terms) interface.

A task object is really a prototype task which cannot be used directly in an application. Instead, a new instance must be created by cloning the prototype. A task object has a task name, description and name of the task instance. Optionally, it has a reference to a method assigned to solve the task. This reference is kept up to date at all times.

A method object has a name, informal description, instance name, type and a boolean value saying whether or not the method's implementation is available. Similarly to the tasks, also methods are prototypes and needs to be cloned before being used. Every method must have an `execute` method with the context as a parameter, and also the context as return type.

A method has a set of parameters. These parameters are implemented in an own class called `MethodParameter`, and imported to a method through class methods. A parameter has a name, description, data type and an object with its value. Parameters also have some restrictions implemented in a class called `MethodRestrictions`. This class will typically make sure that a specific parameter does not have more than a certain amount of maximum occurrences, and not less than a minimum amount of occurrences.

As discussed earlier, there are two types of methods: resolution and decomposition. This is implement in a simple class called `MethodType`, which says whether a certain method solves a task directly or if it decomposes it into subtasks.

When solving a task, the task's *solve* method should be executed and given the current context. What happens next is that the method instance assigned to the task is executed and given the context (if no PSM is assigned to solve the task, the context is returned unchanged). The method returns the updated context which the task further returns to the core. Since an application is configured to handle everything through the core like discussed

in section 3.1.2, it is easy to see that a task is solved by a method, and that the effects it has is applied to the application's context and kept up to date at all times.

3.1.8 Creating and Executing an Application

Finally, as both a summary and to improve the understanding, we will look at how a simple application is created and executed step by step. Specific details about the code logic are left out.

To create a parallel to later chapters of this report, the following example will be based on a previous project [Sti06] which used the same domain as this project. It is a car domain created in the main example of the VolveCreek system, which will be studied later. In [Sti06], the car model was exported as ontologies which jColibri is able to import, but the export mechanism in VolveCreek simplified the original model. This simplification was necessary because of a fundamental issue related to representation. Nevertheless, the project ended up with an application which will be used in this section.

Creating the Application

The first thing we have to do, is to give the application a name, and specify which packages it will be using. The name can be anything, and the selected packages must of course exist. The GUI provides a list of existing packages we can chose from.

Second, we have to create the case structure. This structure is essential and will be used several times throughout the application's execution. Since we have a file with ontologies exported from the VolveCreek example, we will create a case structure based on the ontology. jColibri can read the OWL file and present the hierarchical structure to the user. From here, it is possible to map a concept from the ontologies to an attribute in the case structure. Figure 3.4 shows how this was done in [Sti06]. The case structure is exported and saved in an XML file.

To be able to load the cases into the system from a case persistency (see section 3.1.5), we normally need to configure a connector as the next step. This is a mapping between attributes in the case structure and the fields in the case persistency. When using ontologies, however, we do not need a connector. Instead, the case structure does the same job since we have already mapped each attribute of the case to a concept found in the ontologies.

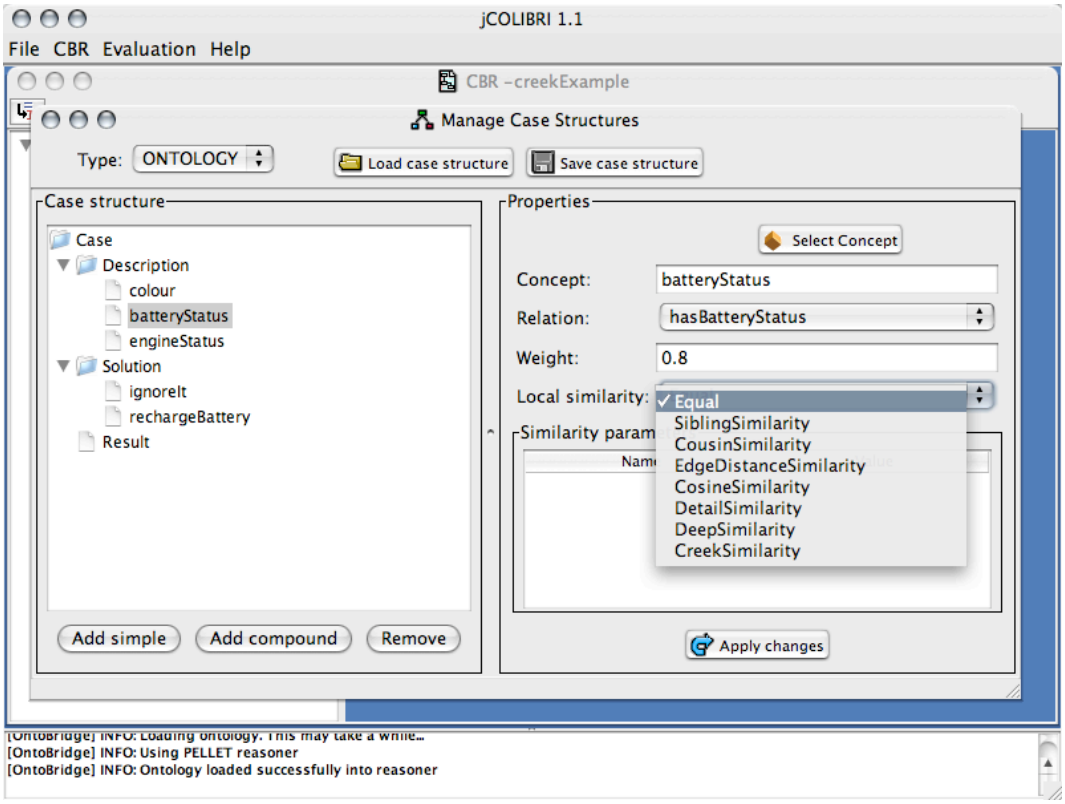

Figure 3.4: Creating the Case Structure

Configuring the application mainly consists of creating the task structure, and assign methods to solve them. Adding new tasks to the tree structure is done by having a decomposition method for one of the tasks, creating any number of subtasks. The GUI makes this simple by listing methods for each task, and it is then given that they are available and have the competence to solve the task. The system designer can then create instances of the methods, and continue this process until all tasks and subtasks are solved. Depending on the type of method, it may also be necessary to provide the value of some parameters. A typical parameter may be the case structure or the connector, but it can also be anything else.

In this example we have 5 resolution tasks. The first is in the precycle task list that loads cases into the system from the ontologies like described in section 3.1.5. This method needs the location of the case structure as a parameter. The second resolution task, and the first task of the CBR cycle, is to obtain a query from the user. This query also needs the case structure as a parameter, as the query looks exactly like a case, and will be compared to other cases later in the retrieval process. The query should typically look like the problem we wish to solve. Retrieve was the only step

of the four well known CBR steps configured for this application, and consist of three subtasks. First, all the working cases are selected to be used for comparisons. The second subtask of retrieve is to compute the similarity between the query and each of the working cases that were loaded into memory. Finally, the last subtask is to select the best task which will be the *retrieved case*, which could be used further in a reuse step had the application not been a pure retrieval system. It would also be possible to select several of the best cases, as that is up to the selection method and the later adaption methods.

Assigning methods to solve a task is done through the core instance. Although the GUI has made sure that chosen methods have the required competence and are given the parameter values they need, this instance does a final check. A GUI should always help the user with such constraints, but it should not be responsible for their satisfaction. It is also possible to write an application without using the GUI, so this final check is necessary. The core first checks if the method is compatible with the task, then adds the method instance to the context for execution, and finally updates the state. These components were explained in section 3.1.2.

Once the configuration is completed, jColibri generates the application. This is based on an application template, and the result will be a Java class which is completely separated from the jColibri framework. The application will communicate with jColibri only through its core instance as we will learn in the next step where the application is executed.

Executing the application

The first thing which happens when an application created with jColibri is executed, is that a new core is created and initialized. As mentioned in section 3.1.2, the core consist of a state, a context and a some packages. The initialization of the core does four main things which are listed below.

- Load all wanted packages by using the package manager. Loading a package consist of first flushing all the registries, and then loading them with everything available in the wanted packages. There are five registries, each having one type of component: prototype method registry, prototype task registry, local similarity registry, global similarity registry and data type registry.
- Creates a new *state* which only has one task, and that task is the concept type *root task*. The instance of *state* will be kept up to date as new tasks are added later on.

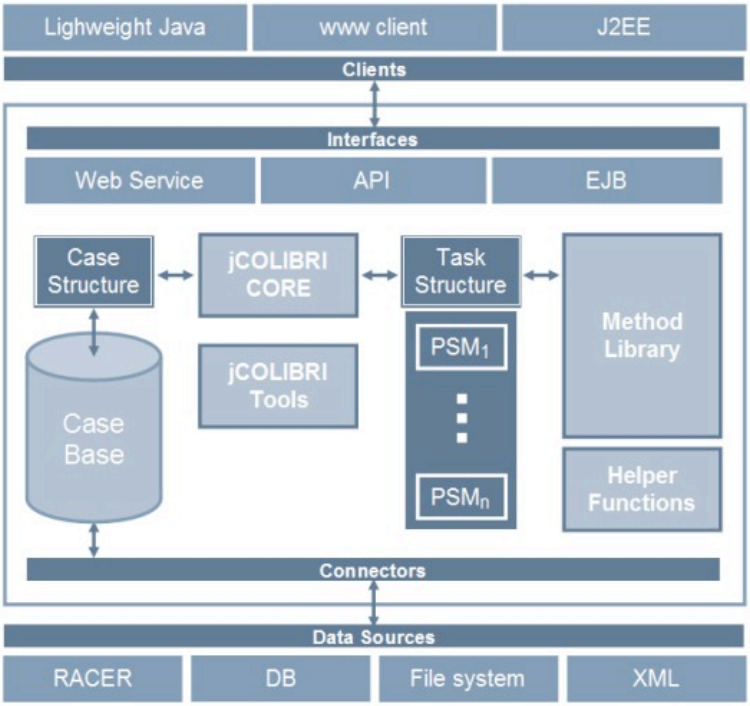

Figure 3.5: An overview of the jColibri architecture

- Creates a new *context* which will be used by the methods later when they accomplish the tasks. Each method will return the updated context when their execution process has completed. This means that it will always will be up to date at any moment during the application's execution.
- Each application has three lists of tasks which are given a root task. The three lists are as follows:

  1. The *precycle task list* contains tasks to be accomplished before the CBR can commence. Typically involves loading cases from the case persistency and other tasks preparing the system.
  2. The *CBR cycle task list* will typically have the three well known CBR reasoning steps: retrieve, reuse, revise and retain.
  3. The *postcycle task list* has tasks that are done after the CBR reasoning is completed.

Since the application is fully configured, the task structure can be traversed and each method can be executed. The traversal behaves like a Depth First Search (DPS). Each PSM's *execute* method will be invoked as each task is being solved, and this method must return the updated context. When all

tasks are resolved, the application is finished. Depending on the configured tasks we will end up with some kind of result. In this example, the case most similar to the query will be selected.

A model of the entire jColibri architecture can be seen in figure 3.5 taken from [jColibri].

3.2 VolveCreek

VolveCreek is the Java implementation of the Creek system introduced in chapter 2.4. Since VolveCreek is well known to the readers of this report, the analysis will be less extensive than that of jColibri. The VolveCreek version used in this project is incomplete, as it is a development snapshot from Volve AS.

As already introduced in section 2.4, the representation of VolveCreek is a semantic net consisting of entities and relations. They are defined by the ontologies, and implemented in the VolveCreek framework in interfaces and classes.

Figure 3.6 shows an example of a semantic net and one of its frames. The semantic net is in the background, partly covered by the frame. The *colour* entity which is used for this example frame, is highlighted in the semantic net (red dot). As we can see in both the semantic net and the frame, *colour* is an instance of the top level *Symbol* entity, and it has two instances called *red* and *blue*.

In this chapter, the case will be given extra attention in an own section, as its representation has changed recently. Cases are no longer nodes in the semantic net. Specific components will also be given extra attention as they are important to reach the project goals, and some are very recent development. Finally, we will create an run an example application.

We will first look at the ontologies, and then describe the implementation of entities and relations in the following sections.

3.2.1 Ontologies

VolveCreek has a top level ontology which defines our view of the world in which our application will be executed. The model contains knowledge about what exists, and how those things relate to each other. At the very

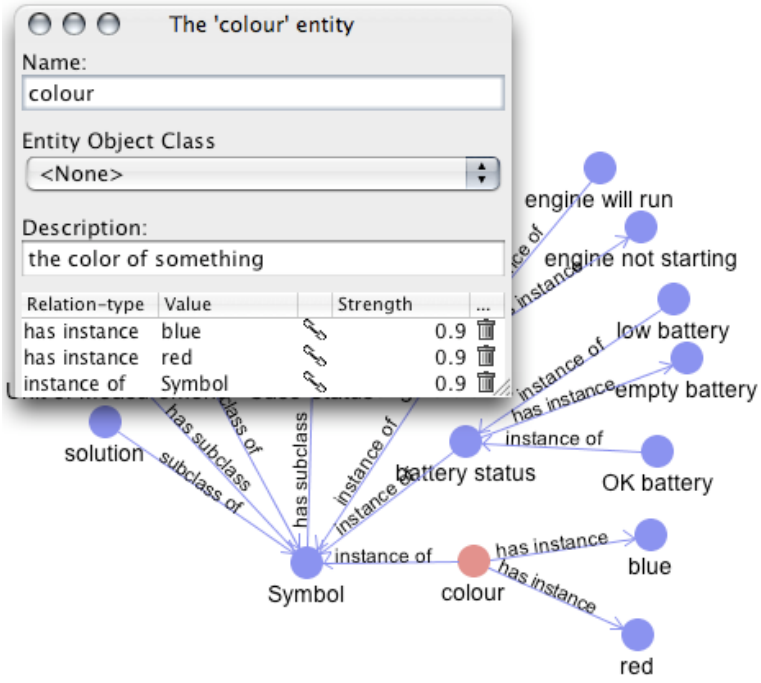

Figure 3.6: An example semantic net from VolveCreek, and a frame

top level, we have the *Thing* concept. Everything in the world is a *Thing*, meaning that it is the most general term in the model.

VolveCreek has implemented an interface which can be used by the knowledge model, and it contains the top level ontology. Typically, a new knowledge model is the first thing to be created in an application, and it is initialized with such an ontology. The ontology used in this initialization is either the aforementioned top level ontology, or an extension of it. This ensures that the ontology includes everything that is completely necessary. Necessary things are the establishment of entities, relations and other things needed by the representation and basic inference system.

The model which will initialize the knowledge models in this project is a case model. This model establishes things like as cases and other things common to all CBR applications.

Below the top level, VolveCreek currently has a mid-level which is specific to a certain type of usage. It can be a specific domain, but it will be rather general concepts from the domain if so. The concepts are of course mapped to the top level. Further, there is a lower level which is mapped to the middle level. This is a domain specific vocabulary, which is used when defining cases and other domain specific entities and relations.

3.2.2 Entities

The entities are nodes in the semantic network. They have references to all relations coming to and from it, and that collection will define a *frame*. All data for an entity is accessed through an *entity data* interface. It specifies that each entity data can be encapsulated by several entity objects, and that all manipulations must be done from the entity object or a knowledge model.

There are several types of entities defined in the ontology, such as numbers, strings, and URLs.

3.2.3 Cases

In earlier versions of VolveCreek, cases used to be nodes in the semantic network, and they were implemented as a type of entity. Recent changes has taken the cases out of the semantic network, and they are now separate objects. The case structure has recently been defined by a DTD[2], as a formal description of their structure is needed when using XML to deal with the cases.

Looking at the XML tree structure, the *case* element can have any number of entries and sections. Entries can be seen as leaf-nodes in the tree, and sections collect the entries by being internal nodes. The root node is simply *case*.

A case element has two required attributes: name and status. The status can either be solved, unsolved or processed. The sections of a case, which may contain more entries or sections, only has one required attribute which is its name. An entry may contain a symbol value or a data value.. A symbol value is typically taken from ontologies, while a data value can be, e.g., letters and numbers.

Entries have six attributes of whom two are required: parameter and source. The parameter identifies which which part of the case the entry is representing a value for, and the source is from where the value was obtained. An example can be the color of a car with value red found by a human observer, as we will see in an example in section 3.2.7. The remaining four attributes of an entry are all implied: data confidence, statistical weight, expert relevance and learned relevance. Symbol value which an entry may contain, is simply some parsed character data originating from the model.

[2]Document Type Definition

Data value which the entry may also contain is parsed character data as well, but must in addition have a value type saying what kind of data we are dealing with (data type).

3.2.4 Relations

Relations are the links between nodes, and defines what the relationship is between them. Each relation has an associated *relation type* defining what kind of relation it is. The relation type also has an inverse so the relationship can be interpreted in both directions. The relations are extremely important to the system.

Similarly to the entities, all data for a relation is accessed through a *relation data* interface. Each relation data can be encapsulated by several relation objects, and all manipulation must be done from the relation object or the knowledge model.

Examples of relations existing in the case model are *has section*, *causes*, *implies*, *has similarity to* and so forth. The inverse relations would be, e.g., *caused by* and *implied by*. Example usage could be that an empty battery attribute of a case may have a relation *causes* linking it to an engine that will not start. Because of this, it is possible to assume with a certain probability, that this car case will not start because of its battery state. If we already know that it does not start, which is probably why it is a problem case, we can go the other way and use it to assume that the battery is the reason for the car not starting if it is indeed a flat battery.

3.2.5 Reasoning

Reasoning in VolveCreek is a three-step process which could potentially be done for each of the four steps in the CBR cycle. Only *retrieve* and partly *reuse* are implemented at this stage, so the rest would have to be done by domain experts at this point, but that is likely to change. Since there are not many new changes to this model, some of the below is taken from [Sti06] which was written by the same author as this report.

1. *Activate* relevant parts of the semantic network (knowledge structures);
2. *Explain* the hypothesis (candidate facts);
3. *Focus* (select) one of them and make it the conclusion.

There are several good reasoning mechanisms in VolveCreek, and they work particularly well in open and weak theory domains. VolveCreek uses abductive reasoning (inference to the best explanation) which is a process where the explanation which makes most sense (based on the known facts) is chosen. Such reasoning can never be monotonic, as that would fail to adjust the explanation when new knowledge enters the system.

VolveCreek supports inheritance and even plausible inheritance. With plausible inheritance, we can have inheritance without having one concept defined as a subclass or instance or the other. VolveCreek adds up and compares the weights assigned to relations transferring other relations, and concludes whether it is plausible or not to inherit a given relation. *Causes* is typically a relation which can lead to plausible inheritance.

With default reasoning, which is support in VolveCreek, it is also possible to draw conclusions from the lack of contradicting evidence. This may happen unless there is a local value overriding an inherited value. Since it is all non-monotonic, such a conclusion will be invalidated once contradicting evidence enters the system.

What happens during each of the three steps will be described more in section 3.2.7, which presents a complete example with the retrieve CBR step.

3.2.6 Comparison Controller

The comparison controller is an important component in VolveCreek. It decides which similarity measure, transformation method and attribute weight metric we will use in comparison operations. These three components are described below.

Similarity Measure

Similarity measures are functions used to compare two case entries. A case entry may have symbol or data values like described in section 3.2.3. Which attribute of the entry that should be compared is given as a parameter together with the two case entries.

The similarity function will return a value between 0 and 1, ranging from no similarity to completely equal.

Transformation Method

A transformation method will transform the structure of a case. Typically, it may expand the section of a case by adding entries found through plausible inheritance. Like mentioned earlier, this can be done by following the *causal* relation and its inverse. The degree of belief to the new entry is equal to the strongest path supporting it.

Attribute Weight Metric

The attribute weight metric defines the relative importance of a case entry or section. The weight range from 0 to 1, where 0 means irrelevant, and 1 is very important (essential).

The scale is linear, meaning an entry with weight 0.2 is twice as important as an entry with 0.1.

3.2.7 Creating and Running a VolveCreek Application

We will now describe how an example application can be created, and what happens during execution. VolveCreek has a nice editor which makes it significantly more user friendly. The system designer can use this editor to create most of the below, however it will be described closer to the code level for understanding. [BSAB04] contains a good introduction where the Creek Knowledge Editor is used.

Creating a Model

Creating a knowledge model is the first thing to do. The new model is not completely empty, but contains a top level ontology. It will establish entities, relations and relation types which the representation and the rest of the system is implemented to use. The model which will be used in this project is the case model. It is a rather simple model extending the basic model, which has only the minimum requirements. Like mentioned earlier, the case model adds all kinds of CBR components.

Adding a Vocabulary

Now that we have a model with some top-level terms, it is time to create a domain. Before describing the domain, we need a vocabulary to do so.

These new components will be mapped to the already existing top-level model. One example could be *colour* which we have looked at earlier, and which can be seen highlighted in the semantic net in figure 3.6.

To accomplish this, we first create a new entity called *colour*, and add it to our model. We map it to our top-level model by adding an *instance-of* relation between *colour* and *Symbol*. The latter is one of the most general things in our model. In addition, we set *colour* to be a subclass of *attribute*, so it can be used to describe cases (see 3.2.3). The relation used here is *subclass-of*.

Finally, we need some instances of the new entity *colour*. This is also done by adding *instance-of* relations between *colour* and other entities such as *red* or *blue*. The two colors must be added to the model in the same way we added *colour* itself, before we create the new relations.

This process is repeated for everything we want to have in our domain. In this project the vocabulary also includes an *engine status*, *battery status*, *age* and several others which are used to describe the car domain.

Adding a Causal Model

We now use our new vocabulary to describe some domain specific behavior, before adding cases. This is the *causal model*. One example is that an empty battery will prevent the engine from start. In other words, it will cause the engine not to start. This is done by creating a relation between a new entity representing the event that the engine will not start, and *empty battery* which exists in the model already as an instance of *battery status*. The special relation type *causes* is defined by our model.

This is done for all parts of our model that may cause a specific type of behavior. The causal relations also have a weight like other relations, describing how likely it is to cause the given effect.

Adding Cases

When adding cases to the model, we first create a type of case which will generalize all our specific cases. It may simply be called *car case* which suits this example. We could create many types of cases like that, and use it to categorize our more specific cases. As usual, we map it to our top-level model by creating an *instance-of* relation between the newly created component, and this time *case* which is defined by our case model as an

essential component of CBR. Several cases are now created by added them to the model, given the type *car case* plus a unique name.

Further, we want to describe each case in details. Recently developed is an own class taking care of this, and we can use it in combination with the case model. This is basically sections and entries described in section 3.2.3. To create an entry for a case, we simply use what we need from our existing model, and sections collect several entries. An example may be that we give *car case 1* an entry *colour* with the value *red*.

Configuring the CBR Reasoning

Reasoning in VolveCreek is well developed, but only the retrieve step of the CBR cycle is fully implemented. Currently, the reasoning can be done by giving the retrieve reasoning step an unsolved case. This case will be used during execution to find cases of the same type from the model, and compute similarities between them. There is no configuring of the reasoning besides making sure that the retrieve step is invoked and that it is given the unsolved case. The rest is already coded into VolveCreek, but is likely to become more dynamic as the software evolves.

Running the Application

When an application is executed, the first thing which happens is the addition of a vocabulary, causal model and cases. This is already explained, so we will skip to the reasoning in the retrieve step.

When the retrieve step is created, it is given an unsolved case as a parameter. This unsolved case should be solved using solution already existing in the case base. The case being most similar to the unsolved case has the solution we are looking for, and hence the unsolved case is compared to all cases being of the same type as itself.

When each case is compared to the unsolved case, a new *case comparison* is created and given the two cases as input parameters. In addition, it will also be given a comparison controller, which is basically a set of rules saying which components of the cases should be using which similarity functions. All components of the two cases are then compared using various similarity functions, and their collected similarity is used as the final similarity value between the two cases. All results are stored in a vector collecting all similarities, and they are sorted by similarity value with the most similar case first.

Finally, the *focus* step shrinks the number of case comparisons in the vector by leaving only those that are found to be relevant. This is done by using a threshold. The case or cases producing the highest similarity and hence not being filtered out by the focus step will form the retrieved case. This case's solution may be reused in the next step, which is not yet implemented.

3.3 Comparing VolveCreek and jColibri

Because of the project goals, it is essential to compare VolveCreek and jColibri. The comparison will shed light on possibilities for reuse of jColibri's components in the VolveCreek system. The comparison will, like the previous chapters, look at one component of the system at a time. After an overview in this chapter, the next will construct a possible solution to accomplish the project goals.

A rough estimate of how difficult or easy each component will be to import is also provided.

3.3.1 Representation

The representation is a fundamental issue, and there is no easy solution to this problem. A big part of this is the type of domains the systems are intended for. jColibri is intended for strong domain theories, where everything is logic-oriented. VolveCreek is intended for weaker and more open domain theories, where default reasoning[3] may be used.

Description Logics (DL) is the knowledge representation language used by jColibri, and it can be translated to first-order predicate logic. The development of DL emerged from the lack of clear semantic rules in semantic networks. jColibri does not have default reasoning, which is the oposite of VolveCreek which uses default reasoning. VolveCreek's representation language is called CreekL [Aam94Nov], which uses frames to describe the nodes in a semantic network.

A knowledge model from VolveCreek was imported into jColibri in [Sti06]. The first step was to export the model as ontologies represented by the Web Ontology Language (OWL). Unfortunately, this export mechanism is forced to exclude certain things from the original model. After every export,

[3]Default reasoning is to assume something because of the lack of contradicting evidence. Systems using default reasoning should never use monotonic logic, as the system must change when new evidence is discovered.

the resulting model represented using OWL will be a simplification if the original VolveCreek model.

This simplification does not have to be extensive, however, and certain domain models may not lose anything at all. As mentioned about the VolveCreek system earlier, its strengths are within open and weak theory domains. It is models from such domains that will suffer the most if exported.

There are several things that can be done to improve this, but they are not strictly related to the fundamental representation, so they are described in later sections. Solving the representational issues directly seems improbable. Both representations have their strengths, and they both cover their own arenas much better than the other. Perhaps the best alternative is to have both representations available in one system.

3.3.2 Model

Both systems are using a top-level ontology, and attaches more and more specific terms to the top-level terms. jColibri has CBROnto, while VolveCreek has a case model. These two serve much of the same purpose, and they have many common terms.

The approach of jColibri is to use CBROnto to describe everything related to CBR, and use other ontologies to describe the rest of the world. CBROnto is represented using OWL, and this is an advantage. This means that it will be able to share knowledge with other projects related to the Semantic Web[4], which is a result of international efforts to create a standards for web content which can be interpreted and used by software agents. Ontologies that are domain independent are mapped to CBROnto. Further, it is possible to create domain specific ontologies and extend this hierarchy any way necessary.

VolveCreek may not be able to cooperate with Semantic Net projects quite as easily because of the representation, but it can cooperate with many projects. We call jColibri's approach an advantage because the Semantic Web effort includes W3C recommendations [W3C01] [DGGG05]. VolveCreek's ontology has many of the qualities found in CBROnto, but it is not as mature yet. VolveCreek's approach can most likely match CBROnto, and potentially use some of the possibilities with its representation to offer something unique that is not possible with CBROnto.

[4]Specifications included under Semantic Web is the Resource Description Framework (RDF), RDF Schema, Web Ontology Language (OWL) and others.

3.3.3 Cases

The case structures are rather similar. They can both be seen as a tree structure, getting a similarity function assigned to each node at some point. jColibri has local similarity functions for leaf-nodes, and global similarity functions for the root and internal nodes. VolveCreek has an entry comparison for leaf-nodes, section comparison for internal nodes and a case comparison for the root nodes. Which similarity function will be used in VolveCreek is decided by the comparison controller.

If not used directly, the case structures can certainly be translated from one system to the other. CBROnto guides the case representation in jColibri, and the structure is stored in an XML file. VolveCreek is using a DTD to define how cases may be constructed.

The value of each case component is a symbol value or a data value in VolveCreek. Symbol values are taken from the model, while the data values can be of any known data type. jColibri is not very different, and although using a slightly different approach, the attribute values should be quite easy to work with if both systems have the data types used to represent them. VolveCreek does not divide the case into a description, solution and result like jColibri, but something equivalent can be done. The case has constants defining if it is solved, unsolved or currently processed, and the description, solution and result can be given in the case's data. How all of this will be solved is not completely certain at the time of writing this report. The new version of VolveCreek is making quite dramatic changes to the case representation, and the changes are not completed.

3.3.4 Comparison Components

jColibri's distributed architecture makes it very pleasant to work with. It is easy to get an overview of how the system will execute, and which comparison component will be used. The similarity function is specified in the case structure, attached to each attribute and the Description concept.

jColibri's GUI makes it easy to chose from a list of available and implemented functions, assuming the user knows which one should be used. It could be an advantage if jColibri also filtered out similarity functions that does not work with the values of certain attributes, but in all fairness this is something the system designer should be able to sort out.

VolveCreek was earlier not quite as well organized, but has come a long way with the recent development. The new comparison controller takes care of

assigning similarity measures to entries based on their attribute type. This is a more flexible system than that of jColibri, since we have the potential to implement a number of controllers and not specify the similarity measure explicitly like done in the jColibri case structures. On the other hand, we could use several case structures to do the same thing in jColibri.

Although it is too early to see the full potential of the comparison controller this early in its development, it does appear to be a very good idea which perhaps also jColibri could benefit from. All in all, however, this does not represent any problems for the import of components, as the assignment of similarity functions are not tangled within the rest of the code.

The interfaces for similarity measures in the two systems are also very similar. jColibri's similarity function interface has a *compute* method returning the data type *double* based on the input of two individuals. VolveCreek's similarity measure interface has a *similarity* method also returning a double based on two case entries and an attribute saying which attribute of the two case entries should be compared. If VolveCreek simply sent the attribute values instead of the entries, it would be the same as that of jColibri.

It is rather safe to already now conclude that we can quite easily make the jColibri similarity functions work in VolveCreek.

3.3.5 Problem Solving Methods

VolveCreek does not facilitate a lot of PSMs. It is not obvious how, e.g., jColibri's textual extension would be implemented for VolveCreek. It could surely be done, but there is no organization set up for it at this stage in the development. The closest thing would be the abstract class *CBRReasoningStep* being extended by *RetrieveResult*.

VolveCreek does not have a task hierarchy like jColibri, so there is no task versus method competence. VolveCreek developers are likely to create some kind of organization for methods as they come further in the development process. Some of the advantages VolveCreek could gain from this is discussed later in this project report.

jColibri on the other hand, has several problem solving methods, and they are well integrated into the system as extensions of the abstract class `CBRMethod`. New methods, like other components, can be added to the system by placing them in packages as discussed in section 3.1.2.

The methods also have a standard way of communicating with the rest of the system in jColibri, but again VolveCreek has not come that far in

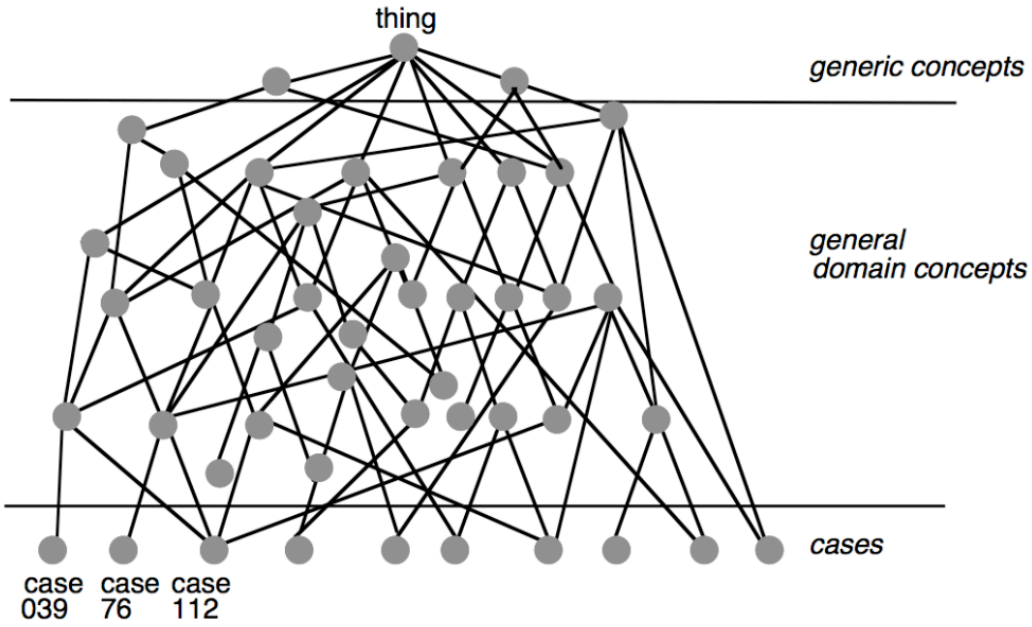

Figure 3.7: The VolveCreek domain

development which makes it hard to compare them. jColibri methods are described in section 3.1.7.

Importing methods to VolveCreek may be require some efforts since they are very centralized in jColibri and communicate with many other components. At the same time, the VolveCreek system is not well prepared to welcome the new components.

3.3.6 Transforms

VolveCreek has a big advantage supporting plausible inheritance. Practically, this is done by transforming cases before comparing them using a transformation method. The transformation uses the *causal* relations and its inverse, which is found in the case model. When such a relationship is found between two symbols, an inferred entry is added with a weight equal to the strongest path supporting it. This is not available in jColibri.

3.3.7 Reuse

Reuse is an important focus in jColibri, but it is also possible with several parts of VolveCreek. Figure 3.7 is a figure of VolveCreek's domain taken

from [Aam04]. The three levels, like described in section 3.2.1, are the top level concepts, the general domain concepts mapped to the top level, and finally the cases describing specific problems which are mapped to the general domain knowledge.

If we look at what is implemented in each system, we can argue that VolveCreek has a focus in a branch rather low in that figure although being present over the whole scale. It does have very general concepts in place in the ontologies, but the implementation is for now focused a bit domain specific, or at least towards specific types of systems. The implementation is generalized as much as possible, however, without slowing down the development process significantly. jColibri on the other hand has focused on the the top level. This enables a broader reuse of both knowledge and code. jColibri's textual and web extensions are examples of things that are a bit lower on the figure. Both systems has the potential to be a complete solution over the whole board.

Both systems are able to reuse its own knowledge and code, but because of the representational issues, jColibri has problems reusing things from VolveCreek. This was studied in [Sti06], and it is clear that we lose some information or knowledge in the transition. The other way around may be a bit harder to do in a general way, but we should not have the same loss of information or knowledge. jColibri's distributed architecture should make it much easier for VolveCreek to reuse its components. This is something which will come clear in later chapters, as the key goals of this project is to do just that.

3.4 Summary

jColibri is the Java implementation of the COLIBRI system. Major components are:

- The core consisting of a state, context and packages;
- Data types such as numbers and strings;
- Connectors and case bases;
- Helper functions such as the similarity functions;
- Tasks and methods which together configure and guide the execution of a jColibri application.

VolveCreek is the Java implementation of the Creek system. The implementation used in this project is a development snapshot from Volve AS, and is incomplete. The most important aspects of VolveCreek to this project are the ontologies, entities, relations, reasoning capabilities and the comparison controller.

A comparison between the two systems shows that the main issue is representation, while several of the major components are fairly similar. jColibri has come further in its development, and some its components does not yet exist in VolveCreek. It is likely that VolveCreek will be developed in a direction which will eventually cover most of what jColibri offers.

Reuse is a huge focus in jColibri, and hopefully this will make it easier for VolveCreek to reuse its components. The other way around is harder, and some things are lost in the transition [Sti06].

Chapter 4

Construction

This chapter describes how jColibri components can be imported to VolveCreek, and how we can create an application to demonstrate our results. It will be implemented and evaluated in the next chapters.

The components should be usable inside the existing VolveCreek system, so tasks will not be imported. VolveCreek would benefit from having a task hierarchy like jColibri, and this will be discussed later. The following three components are the focus in addition to the demonstrator system itself:

- Helper functions
- Data types
- Methods

From the comparison in chapter 3.3, we concluded that the systems are quite similar in many ways, but they are also built on different foundations. Using the components directly is not possible without some kind of bridge between the two systems. The bridge will have to take care of the representational issues somehow. The more general the solution is, the more useful it will be.

We will start each section by identifying the requirements for the component, before constructing a possible solution. To construct a mechanism which enables VolveCreek to use jColibri components, we must look at what the components are dependent on to be able to execute successfully. Finally, we will look at the construction of the demonstrator system where these components will be used.

Requirements from the VolveCreek system will be taken care of once the components are available to the system. Note that the solution attempts to

import already existing components from jColibri. Some restrictions may not necessarily be with the jColibri system itself, but with the existing components.

4.1 Helper Functions

Similarity functions are the only helper functions we can import since there are no other helper functions in jColibri yet. The functions are implemented almost the same way in both systems, so it should be fairly easy to import them. We will need a way to represent the VolveCreek symbol and data values as jColibri individuals. This can be done by using a *wrapper*.

This wrapper should wrap the VolveCreek component, and implement jColibri's `Individual` interface. If this new implementation of the `Individual` interface is able to access the VolveCreek values, then the similarity functions should be possible to use directly.

Each jColibri similarity function implements the `SimilarityFunction` interface, which means that it will have a `compute` method taking two individuals are parameters. At the end of its execution, it will return a *double* data type which indicates the similarity between the two individuals. This value is between 0 (no similarity) and 1 (equal).

The implemented similarity functions are naturally implemented to compare two values of data types existing in jColibri. This means that we must make sure that the data type from VolveCreek is in the same format and that it is compatible with the similarity function. Much of this is also taken care of in the similarity function itself, by returning 0.0 similarity if the data types cannot be compared by that specific similarity function. This is only a way to make the code error-tolerant however, and we should still make sure that we are using the right data type.

To make all jColibri similarity functions available, we can implement a new similarity measure for VolveCreek. The similarity measure constructor can get the name of the jColibri similarity function as a parameter, and use it to create an instance of it (provided that it exists). jColibri similarity functions also use parameters as described in section 3.1.6. To use them, we should also be able to send a list of parameters to the similarity measure. The parameters can then be used when creating an instance of the jColibri similarity function.

A lot of this is very similar to what jColibri does when using its own similarity functions. The class `CBRSimilarity` has a method `getSimilFunction`

which does much of what we desire to do in our new VolveCreek similarity measure. This can be used as an inspiration also in the implementation phase.

4.2 Data Types

We will try to import the data type `Text` from jColibri's textual extension.

Data types are implemented as own classes in jColibri, and they can be accessed directly. To use jColibri's data types in VolveCreek, we have to create a new entity type, and also add it to the ontologies. The configuration files explained in section 3.1.3 can not be used in VolveCreek. The implementation should be fairly straight forward, but must be done to each data type in a similar fashion to what is already done with native Java data types used in VolveCreek.

jColibri has a *DataFilter* data type which is located in the textual extension. This data type stores data in a general way using hash-tables instead of class attributes. A type of data filter is `Text`. It is composed by a collection of `Paragraph` instances. Each paragraph contains a collection of `Sentence` instances, and each sentence by a collection of `Token` instances. Each of these are implemented in separate classes. The textual extension will be used several times in this project, and it is described in further detail in [RDGW05].

When creating the new entity type for `Text`, the most important method is `matches` in addition to some constructors. This method is used to check if a given entity matches the representation to this entity type. This can be checked by making sure that a value exists, and that it is an `instance-of` the jColibri `Text` data type.

After constructing the entity type and defining how it should be matched, the `Text` data type class in jColibri can be accessed directly.

One problem which is related to saving the model once a text entity type is used, is that it needs to be *serializable*. The data types are not serializable in jColibri, but have to be if we want to save the model as a binary file which is what VolveCreek does. A simple modification of the `Text` class in jColibri solves this issue. We let the class implement `Serializable`. We do not want to change anything in jColibri since the components should be imported like they are implemented, but this is the only change to jColibri in this project and it is a trivial matter.

4.3 Problem Solving Methods

PSMs require more consideration than other components because of their centralized location within the jColibri system. See section 3.1 for more details on the methods. Their operations cooperate with several other components, and this complicates the import.

We will first look at two different approaches, and then describe the construction in greater detail for one of them. The first approach focuses on a strict import of the methods, while the second focuses on making them usable within VolveCreek.

4.3.1 Import Focus

Following the kind of approach done with other components, we can try to import methods from jColibri into VolveCreek directly. Since the methods use several other components, we also have to import those components for the methods to execute successfully.

The most important component is the context, which in turn contains the equally important collections of cases. Both can be specialized by extending existing jColibri classes and interfaces to make them work with the VolveCreek system. Once we have a context, a case base and perhaps a case evaluation list, we can actually start to execute the methods. None of these components are difficult to specialize (extend) or wrap in some way.

When looking at specific methods, however, a serious problem arise. All methods are implemented to use very specific aspects of the jColibri system directly during execution. Although we can provide all methods with what they need to execute, their execution will only be partial unless we continue to import almost everything from the jColibri system. Most likely, it is also necessary to modify the methods, which is not something we want to do unless it is a very small and trivial matter. This includes the individuals and other things implemented to support jColibri's representation, and it is also soon obvious that we are running into a representational issues sooner or later. These issues will have to be solved rather specifically for each type of method or application, and some cannot be solved. This is a well known issue with these two systems, and it was also the major difference in section 3.3.1. Future extension packages are also likely to be difficult to import using this approach.

While this is still interesting for many applications, we basically have to import the whole jColibri system before we are done. It does not seem like

an attractive option to first wrap and specialize many components, and not end up with something that can be used. At least not when knowing that we can deal with the representation right away, and then use the jColibri components as they are. This is covered by the remaining sections of this chapter, which attempts to use the jColibri system in a small temporary jColibri environment without strictly importing them.

4.3.2 Usage Focus

The essence of this solution is that we can create a minimal native environment for the jColibri methods to execute in, and apply the result back to VolveCreek once the execution has finished. In other words, instead of importing almost everything from the jColibri system to VolveCreek, we rather move some data over to a small jColibri application which executes and returns the result.

A typical method will be modifying parts of the context somehow, and return the updated context. Normally, since this is CBR, the cases are either modified, transformed or we could even end up with completely new cases. After a method has finished executing, the resulting cases can be found in the context as *working cases*, i.e., the cases we are currently working on. These cases may be used to update the original cases that are located in the case base or a case persistency.

A method may also simply retrieve a few cases based on some filter, and this collection of cases may be used for further processing. In fact, most methods are using the working cases instead of the case base directly, because they assume some kind of retrieval method to filter out unnecessary cases that does not need that method's type of processing. The retrieval methods will of course use the case base directly.

To get cases into the case base from the case persistency, a method using a connector will typically be executed before the retrieval method to fill the case base with cases if they is not already there. This is normally done in the precycle, which happens before the CBR cycle (see sections 3.1.8 and 3.1.8 where the precycle task list is used and accomplished by such methods).

This means that we need to create an environment with a context and cases. The context and the cases should be easy to work with for the jColibri methods, while they should also reflect the current situation of the VolveCreek application. The jColibri representation should be used.

Since a VolveCreek application will be trying to use a jColibri component at

some time during execution, we need some way of having an updated context at that time. Since the cases and the state of a VolveCreek application changes continuously during execution, there is no reason to initialize or keep the context updated at all times. Instead, it would be better to load the cases into the context when we need them, and use the result after the method has executed. This way, we can also minimize the amount of cases we have to transfer to the case base. This can be compared to some kind of retrieval function, but it will go both ways and we can use the jColibri `Connector` interface to achieve what we want.

Connectors can work with any case persistency like explained in section 3.1.5. Implementing a custom connector for VolveCreek is the best solution, and once a connector is in place, the jColibri methods can work with the normal context using the connector as a bridge between the two systems.

Note that this is because the specific jColibri methods themselves use the cases. It would not be necessary to do all of this if we just wanted to invoke a jColibri method from VolveCreek in a general way and not worry about what it actually does.

Inspired by jColibri's application template, we can create a class to deal with the execution of a jColibri method from the VolveCreek system. Such a class would have the following requirements.

- Have a constructor taking at least two parameters: an instance of the jColibri method we wish to execute and some kind of filter (an entity type or something else we can use as a filter);
- Use a special VolveCreek connector to retrieve the wanted cases;
- Create a context where the cases will be kept during execution, and initialize it with the cases we want to apply the method on;
- Execute the method affecting the context;
- Use the VolveCreek connector to transfer the affected cases back to the VolveCreek model;

4.3.3 Method Construction with Usage Focus

First we must create the custom connector which can import VolveCreek cases into the jColibri context. This new connector should extend jColibri's *Connector* interface. The most important methods of this new connector class are the ones fetching cases from VolveCreek and translating them to

jColibri cases, and others taking jColibri cases and storing them in a VolveCreek format. The method first will be used before the method execution, and the second will be used after the execution to let VolveCreek know about the results. To lighten this operation we only send a minimum amount of cases. Most of the time the system designer should be able to limit the number of cases quite a lot. This filter may, e.g., be an entity type, which is frequently used as a filter in VolveCreek.

For each VolveCreek case, we can construct a new jColibri case with an identification equal to the case's name. We may then continue by iterating through all entries and section of the case, and transform them into jColibri attributes (described in section 3.1.4). This can be done using a recursive method since both systems' case structures are basically the same (trees).

The case attributes may have two different types of values: symbol value (from the model) or data value (any data type). We must deal with the two types a bit different, but it is unlikely that it will be a significant problem. VolveCreek stores cases in XML structures, but they are also available directly from the application during execution. Where the connector fetches the cases does not really matter, and we can create several connectors to suit different requirements.

Once the cases are in the context, we can execute the method instance and give it the context as a parameter. The method should now execute successfully and affect the cases in our context.

The selected method which will be used in this project is called *Stemmer-Method*. The stemmer method takes a `Text` data type, and transforms each word (token) to its stem (base/root form). The stemmer is actually another project called SnowBall[1], which jColibri uses through its textual extension. A definition of SnowBall from its website reads:

> Snowball is a small string-handling language, and its name was chosen as a tribute to SNOBOL, with which it shares the concept of string patterns delivering signals that are used to control the flow of the program.

Practically for this project, we will, e.g., see the words like *"writing"* be stemmed to *"write"*, and the same will happen to *"writes"* and other variation. To do this, we can use the new data type imported in the previous section. A case can be given an entry with a text value, and later go through the stemmer method. Here we also have the possibility to filter cases based on their entries, as we obviously do not have to transfer cases without a

[1] http://snowball.tartarus.org/

text entry if we want to use the stemmer.

After the method has been executed, the cases in our context have been changed. We will need a method to update our VolveCreek knowledge. This method should take the values from the jColibri individuals, and place them in VolveCreek components. It is likely that we will need many variations of this method. It really depends on the method used to affect the cases.

An important issues which surfaces when we are about to transfer the updated values back to VolveCreek, is to make sure that data types and symbol types are created appropriately. There are many things to worry about here, but this project will not solve everything. The most important thing in this project is to see if we can do this at all, and these issues are mostly practical and their solutions are fairly obvious.

It is also a question whether or not we want the changes in our model, or if we just wanted to see the result of it as a temporary calculation. Some custom work is likely to be necessary for each new type of application, but it should not be extensive.

4.4 Demonstrator System

The demonstration is based on the example application which is included in VolveCreek. This example has been used throughout this project and [Sti06], and section 3.2.7 provides a description of how it is build. We will be extending this application with our new components:

- A method called `StemmerMethod`;
- Similarity functions called `Equal`, `Interval`, `TokensContained`;
- A data type called `Text`.

Since the example application already has some cars with attributes, we can extend these cases with more attributes. First off we need to apply the similarity functions on some values of the correct data types. The `Equal` similarity function simply checks if two individuals are equal or not, and returns either 1 (equal) or 0 (not equal). The function uses the method `java.lang.Object.equals` to do this, unless a value is a `StringEnum` in which case `java.lang.String.equals` will be used instead. We can test any data type using this similarity function, so we do not have to add any new attributes.

The `Interval` similarity function works with numbers, and the example application already has an attribute for that as well. The `age` attribute

can be used. `Interval` computes $1 - \frac{|x-y|}{INTERVAL}$, where `x` and `y` are the two numbers being compared, and `INTERVAL` is a number defining how large the interval in which they are compared is. Some difference between `x` and `y` may not mean a whole lot if `INTERVAL` is large.

`TokensContained` should be tested on a `String` value having several words (tokens). Since we want to check how many words two attributes have in common, we must make sure that at least one word can be found in all attributes of this kind. The similarity value is computed by checking how many tokens they have in common, and dividing that number by the total number of tokens.

One attribute should also use the new `Text` data type. This is needed to both test the stemmer and the data type itself. A stemmer can potentially improve the case matching by making two words be recognized as the same words even though they are written in different times or in plural. To test this we can run a similarity function before the stemming, and then another after the stemming. If we chose words for this attribute that are not the same before stemming, but become the same after the stemming, we have proven a point. If this can be done, we have also shown that the text data type itself is working in Creek.

By running the resulting application, we will be able to evaluate whether the imported components are working as they should or not.

4.5 Summary

We look at the construction of three components: helper functions, data types and methods. Tasks are not included because we want to use the components in existing VolveCreek applications, and we are not able to use tasks there.

The only helper functions we can import are the similarity functions. We are able to construct a general solution which will enable us to use any similarity function of jColibri through one similarity measure implemented in VolveCreek. A wrapper is used to represent case entries as individuals before they are compared by the similarity function.

There are no general solution for data types, so we must implement one at a time. The jColibri data type classes can be used directly, but we must add a new entity type in VolveCreek. We must also add the new data type to the ontologies and edit some other VolveCreek components.

The methods require more work than the other components because of their centralized position within the jColibri system. We look at two ways of solving it. First, we have a solution which focuses on the import. This is possible, but when looking at specific methods it is clear that they are not very useful after having been imported. Instead, we go for the second solution which has a usage focus. This solution creates a minimal jColibri environment in which the method can be executed, before applying the changes back to VolveCreek. To achieve this, we create a connector to transfer cases, and a class inspired by the jColibri application template to create the environment.

A demonstrator system will be based on the VolveCreek example. Two similarity functions can tested on existing attributes, while the rest requires that we extend the application a little bit. The new data type and method also requires some work, but can be tested fairly easily.

Chapter 5

Implementation

The implementation chapter describes how the steps outlined in the construction chapter was implemented to create a demonstration. Code snippets introduced by explanations are provided.

5.1 Helper Functions

The construction phase describes a very general solution to import all jColibri similarity function through one VolveCreek similarity measure. The class implemented to take care of this extends VolveCreek's `SimilarityMeasure` interface. First of all, we will look at the constructor and the class variables.

In the code snippet below, we assume that the `name` is equal to a Java class name with full path. If it does not exist, it will be caught in an exception later. The parameters are also assumed valid, and of the type `CBRSimilarityParam` which is implemented in jColibri. This is all we need, and the constructor is shown below.

```
1 protected String name;
2 protected List<CBRSimilarityParam> parameters;
3
4 public JColibriSimilarityMeasure(String name, ArrayList params) {
5   this.name = name;
6   this.parameters = params;
7 }
```

Each similarity measure has a `similarity` method, and it is here that we have to be careful. This is where the actual similarity is being computed. This method must get two case entries and an attribute as parameters. The case entries have the values we want to give to the jColibri similarity

functions, so we need to put them in `Individual` objects. This is done in a class called `CaseEntryIndividual`, which implements the jColibri `Individual` interface and wraps a VolveCreek `CaseEntry`.

The implementation of the wrapper is not complicated. It is very similar to `SimpleIndividual` which is implemented in jColibri. The main difference is the constructors, which are specialized to deal with VolveCreek case entries. The value of the individual is set to either the symbol value or the data value of the case entry, depending on which of the two is provided. Only one will be provided for each entry. The rest of this new individual is the same as the original jColibri `SimpleIndividual`.

Now that we have two individuals, we are almost ready to invoke a jColibri similarity function. Before we do that, we must create an instance of the wanted similarity function, and we must give it the list of parameters. The below code takes care of this. Note that this approach is very similar to that of jColibri itself, and the code below is similar to a method in jColibri's `CBRSimilarity`, which also has the same functionality.

```
1  public SimilarityFunction getSimilarityFuncion() {
2    Class cl;
3    SimilarityFunction similFunc;
4    Iterator it;
5    HashMap<String, Object> map;
6    CBRSimilarityParam param;
7    try {
8      cl = Class.forName(this.name);
9      similFunc = (SimilarityFunction) cl.newInstance();
10     if (parameters != null) {
11       it = parameters.iterator();
12       map = new HashMap<String, Object>();
13       while (it.hasNext()) {
14         param = (CBRSimilarityParam) it.next();
15         map.put(param.getName(), param.getValue());
16       }
17       similFunc.setParameters(map);
18     }
19   return similFunc;
20   }
21   catch (java.lang.ClassNotFoundException cnfe) { }
22   catch (java.lang.InstantiationException ine) { }
23   catch (java.lang.IllegalAccessException ile) { }
24   return null;
25 }
```

The above method creates a new instance of the similarity function, adds the wanted parameters, and catches all possible exceptions which may be thrown. The `catch` blocks could be used to write to the log or something. Now that we have a similarity function and the two individuals that are needed, the `similarity` method can be completed. The method is shown below.

```
public double similarity(
              Entity attribute, CaseEntry entryA, CaseEntry entryB) {
  if (entryA == null || entryB == null)
    return 0.0;
  CaseEntryIndividual a = new CaseEntryIndividual(entryA, attribute);
  CaseEntryIndividual b = new CaseEntryIndividual(entryB, attribute);
  SimilarityFunction simFunc = getSimilarityFuncion();
  double eval = 0.0;
  if(simFunc != null){
    eval = simFunc.compute(a, b);
  }
  return eval;
}
```

If all goes well, a *double* value representing the similarity between the two entries is returned. If not, the similarity is assumed to be zero (irrelevant).

5.2 Data Types

Following are code snippets and explanations which covers the import of the new data type `Text` from jColibri to VolveCreek. A new entity type is implemented.

In the new entity type we need a constructor. The constructor will take the knowledge model, the text value and a description as parameters. It will create the new entity and add it to the knowledge model. The entity will be given the identification `"TextEntity#"` followed by a unique number. The description will be attached to the entity. In addition, it will be associated with the `Text` object as its entity data.

```
public TextEntity(KnowledgeModel model, Text text, String description) {
  super(makeEntity(model, description), true);
  setEntityObject(text);
}

private static Entity makeEntity(KnowledgeModel model, String description) {
  Entity entity = null;
  int i = model.entitySize()+1;
  while(entity == null){
    try {
      entity = new Entity(model, "TextEntity#"+i, description);
    }
    catch(NameAlreadyExistException e) {
      i++;
    }
  }
  return entity;
}
```

We also need a constructor to be used when we want to create a new entity value of an already existing entity type. Parameters are the knowledge

model, the text object and the entity type it will belong to. An `instance-of` relation is used in the model to associate the entity with its type.

```
public TextEntity(KnowledgeModel model, Text text, Entity type) {
  this(model, text, "");
  try {
    addRelation(BasicModel.INSTANCE_OF, type);
  }
  catch(NoSuchRelationTypeException e) {
    e.printStackTrace();
  }
}
```

Although the code is short, the following method is very important. It will define how entities are matched before defined and treated as a text entity type. We simply make sure that the value exists and that it is an instance of the `Text` class.

```
public static boolean matches(Entity ent) {
  return  (ent.getEntityObject() != null)
           && (ent.getEntityObject() instanceof Text);
}
```

In order to make the VolveCreek system display the new entity type in the GUI elements both in results after execution and in the CKE[1], we need to add some code supporting it several places. It is not very flexible that we have to add this directly, and it should be some kind of configuration files in XML for this, but that is likely to improve as the VolveCreek software matures. The case writer and parser also needs to recognize the new data type. Following is a list of places where changes were made during implementation. The changes are minor, and some are just for convenience. We will not go into further detail regarding these changes in the report.

- The case model and associated constants
- The case parser
- The case writer
- A new constructor for case entry to take the new data type as a parameter

The new data type will be tested in the next chapter.

[1] VolveCreek Knowledge Editor

5.3 Problem Solving Methods

We will start by creating the environment in which our methods will be executed. It is called `JColibriApp`, and it is much like a jColibri application. The difference is that it does not use tasks, and hence we are not using a core object either. The implementation is not fully developed, but it illustrates its points. A few simplifications have been made compared to the solution explained in the construction phase.

First off, we create the constructor. It takes two parameters, which are an instance of the jColibri method and a case. The case represents the filter, as we can use this case to only fetch cases of its type. This also ensures that they are possible to compare, which may often be a very important point. The constructor then initializes the three main variables we will be using: context, knowledge model and the connector. The cases are retrieved by the connector in `retrieveAllCases` (also shown in the code below), and then put into the context by `setCases`. The method is then executed.

```
1  public JColibriApp(CBRMethod method, SeparatedCase case1) {
2    this.context = new CBRContext();
3    this.connector = new VolveCreekConnector();
4    this.km = case1.getKnowledgeModel();
5    SeparatedCase[] cases = this.km.getCases();
6    try {
7      this.connector.init2(cases);
8      this.context.setCases((List<CBRCase>)this.connector.retrieveAllCases());
9    } catch (InitializingException e1) {
10     e1.printStackTrace();
11   }
12   executeMethod(method);
13 }
14 public Collection<CBRCase> retrieveAllCases() {
15   ArrayList<CBRCase> list = new ArrayList();
16   for (int i = 0; i < cases.length; i++){
17     CBRCaseRecord cbrcase = new CBRCaseRecord(cases[i].getName());
18     CaseEntry[] entries = cases[i].getEntries();
19     for (int j = 0; j < entries.length; j++){
20       // if symbol value
21       if (entries[j].getSymbolValueAttribute() != null){
22         cbrcase.addAttribute("" + entries[j].getID(), entries[j]
23             .getSymbolValueAttribute(), entries[j]
24             .getStatisticalWeight(), null);
25       }
26       // if data value
27       if (entries[j].getDataValue() != null){
28         Object value = entries[j].getDataValue();
29         cbrcase.addAttribute("" + entries[j].getValueType().getName(),
30             value, entries[j].getStatisticalWeight(), null);
31       }
32     }
33     list.add(cbrcase);
34   }
35   return list;
36 }
```

Before we execute the method, we need to set some parameters. This implementation will only set the necessary parameters. It is possible to go further and let each method define its own parameters, but this was not given any attention in this project. It will always process the cases. If a specific method should be given some other parameters from the user, it would not be a lot of work to create a way to let the system designer send an array with values and have them added to the parameter hash map.

```
public void executeMethod(CBRMethod method){
  HashMap parameters = new HashMap();
  parameters.put("Process Cases", true);
  parameters.put("Process Query", false);
  method.setParameters(parameters);
  try{
    method.execute(this.context);
  } catch (ExecutionException e) {
    e.printStackTrace();
  }
}
```

Once the method has executed, we can use the result to update our cases. As mentioned earlier, time did not allow for implementation both ways here.

5.4 Demonstrator System

We will look at one component at a time as we place them in the demonstrator system.

5.4.1 Using the New Data Type

To use the new data type, we add an attribute `desc`, just like we added `colour` in section 3.2.7. This is an abbreviation for *description*, but we will just use it to attach some text to each car. The text will contain words that can be stemmed by the `StemmerMethod` later and become identical. It does not really matter what this text is, as we are just testing it. The code below shows that `desc` will be an instance of `Text` which was added to the case model in section 3.2.7, and it is an attribute.

```
...
Entity text = new Entity(km, "desc", "the description of the car");
text.addRelation(SeparatedCaseModel.INSTANCE_OF,
SeparatedCaseModel.TEXT);
text.addRelation(SeparatedCaseModel.SUBCLASS_OF, attribute);
...
```

The first car is given the text *"run"*, the second car is given *"runs"* and the third car is given *"running"*. These text values are added to `desc` entries, like shown below. The `text` variable contains *"run"* since this is *Car Case 1*, while *"Human Observation"* is a value describing how the value was obtained. The latter is not important here.

```
...
case1.addEntry(km.getEntity("desc"), "Human Observation", text);
...
```

We will not add this attribute to the causal model. If it had been added, however, it would not have affected this demonstration in any way.

5.4.2 Using the new Similarity Functions

To use our new similarity functions, we implement a new comparison controller. This is a very convenient way to make sure that our attributes will be compared using a jColibri similarity function.

The new comparison controller will be called `DemoComparisonController`, and the only thing we will change from the default controller is the `getSimilarityMeasure` method. We explicitly find attributes of the car, and assign a similarity function to them. Below is one example which is assigning the jColibri `Interval` similarity function to the `age` attribute. Notice that we are also sending this function a parameter.

```
...
if(attribute.getName().equals("age")){
  CBRSimilarityParam param = new CBRSimilarityParam("INTERVAL", "15");
  ArrayList arrayList = new ArrayList();
  arrayList.add(param);
  // This function computes: sim(x,y) = 1 - (|x-y|/interval)
  simMeasure = new JColibriSimilarityMeasure(
                "jcolibri.similarity.local.Interval", arrayList);
}
...
```

Matching the name of an attribute directly is OK in this demonstration, but normally it would be smarter to match something a bit more general like an entity type. We can apply any similarity function we want, although we should of course consider the data type of an attribute before we apply it. If the data type is not right, then we will typically just end up with a similarity of 0.0.

Following is a list of case entries and which similarity function they have assigned in this demonstration. All similarity functions are implemented for jColibri, but now used by VolveCreek.

- `age` is a number, and has the `Interval` similarity function assigned;
- `words` is a collection of words, and has the `TokensContained` similarity measure assigned;
- `desc`, and all other attributes already in the application, will be using the `Equal` similarity measure.

5.4.3 Invoking a Method

To invoke the method, we only have to create a new `JColibriApp` (see section 5.3), and give it an instance of `StemmerMethod` and a case. The case was only chosen as a way to filter which cases we will be transferring through the connector.

```
...
new JColibriApp(new StemmerMethod(), km.getCase("Car Case 3"));
...
```

Like discussed earlier, the implementation is missing a way to affect the VolveCreek cases after the jColibri method has been executed. It is not necessarily difficult to implement it, but the implementation phase ran out of time. To cover this hole, we create a method printing the results instead. The method printing the results will be invoked right after the stemming is completed. Code for this method follows.

```
public void outputResults(){
  ArrayList<CBRCase> cases = (ArrayList<CBRCase>) context.getCases();
  Iterator iter = cases.iterator();
  int i = 0;
  while(iter.hasNext()){
    CBRCase case1 = (CBRCase) iter.next();
    Text val = (Text) case1.getDescription().getRelation("Text").
                                getTarget().getValue();
    Collection tokens = val.getTokensList();
    Iterator itern = tokens.iterator();
    while(itern.hasNext()){
      Token token = (Token) itern.next();
      System.out.println("The original token was: "
                                      + token.getData(Token.COMPLETEWORD));
      System.out.println("The stemmed token is: "
                                      + token.getData(Token.STEMMEDWORD));
    }
  }
}
```

The code simply loops through the list of cases, and then through the token list for each case. For each token it prints both the complete (original) word and the stemmed word. The idea is that all stemmed words should be identical if the complete word is the same.

5.5 Summary

Following the construction phase, we implement the import the three components we are interested in: helper functions, data types and methods. Helper functions are implemented as planned. The `Text` data type is also implemented as planned, with some unforeseen work in the case writer and parser. The method is also imported, but with some simplifications. The connector only works one way, and the evaluation phase will have to use a rather basic method which prints the results instead of applying them back to the VolveCreek application.

In addition we implement the demonstrator system, which is the VolveCreek example application extended to use the imported components.

Chapter 6

Testing

The code produced during the implementation phase will now be evaluated by running the demonstrator system. This chapter present and evaluates its results. A further discussion about the solution in general is provided in the next chapter.

First off are the results from the comparisons performed by the similarity functions.

6.1 Similarities

Figure 6.1 shows the similarities computed by the similarity functions. `words` is the first attribute in the list, and it is given a similarity value of 0.6 using the `TokensContained` similarity function. The description of the first case (called A in the figure) is *"five six two four"*, while the second case (called B in the figure) has *"one two three four five"*. Of the five tokens in the second case's attribute, the first case contains three. Since $\frac{3}{5} = 0.6$, we can conclude that the similarity functions is working as it should.

There are several attributes using the `Equal` similarity function: `solution`, `battery status`, `colour`, `starter engine won't turn`, `engine status`, `desc` and `starter engine turns slowly`. These needs to be identical for the similarity function to give a similarity value of 1, or else they will be given 0. We could call this a boolean function, but the return value is a *double*. As we can see in Figure 6.1, some of them are equal, and some are not. The results are correct.

A special case is `desc`, which is using our new `Text` data type. The values should have been stemmed and the results applied back to the cases, but

Entry-comparisons:

Parameter	Similar...	Description A	Description B
words	0.6	words: five six two four null	words: one two three four five null
solution	0	N/A	solution: recharge battery
battery status	0	battery status: empty battery	battery status: low battery
colour	0	colour: blue	colour: red
starter engine wo...	1	starter engine won't turn: starter engin...	starter engine won't turn: starter engin...
engine status	1	engine status: engine not starting	engine status: engine not starting
desc	0	desc: running null	desc: run null
age	0.133	age: 16 null	age: 3 null
starter engine tur...	1	starter engine turns slowly: starter eng...	starter engine turns slowly: starter eng...

Figure 6.1: Results from the similarity functions

as we already know, the implementation phase ran out of time and did not complete this. Since a method was created to at least print the results, we will evaluate what the results would have been in section 6.2 where the method is tested and evaluated.

Finally, we have the `age` attribute. Its similarity function is `Interval`, and the value are 16 and 3. This function also uses a INTERVAL number, which was set to 15 which can be seen in the section 5.4.2. Placing these numbers in the formula, we get $1 - \frac{|16-3|}{15} = 0.133...$ This is the same as shown in Figure 6.1, so we now know that all similarity functions have produced the desired results.

6.2 The Method and the Data Type

The `StemmerMethod` was selected as our method component, and it is supposed to stem words stored in the `Text` data type. We created a case attribute called `desc` to store such a value, and although the implementation is only partial, the method printing the results produce the following:

```
INFO: StemmerMethod BEGIN
INFO: StemmerMethod END
The original token was: run
The stemmed token is: run
The original token was: runs
The stemmed token is: run
The original token was: running
The stemmed token is: run
```

The code printing this is presented in section 5.4.3. Above, we can see that the method begins stemming, and then ends before the results are printed. The first case had the value "run", which was already stemmed. The second

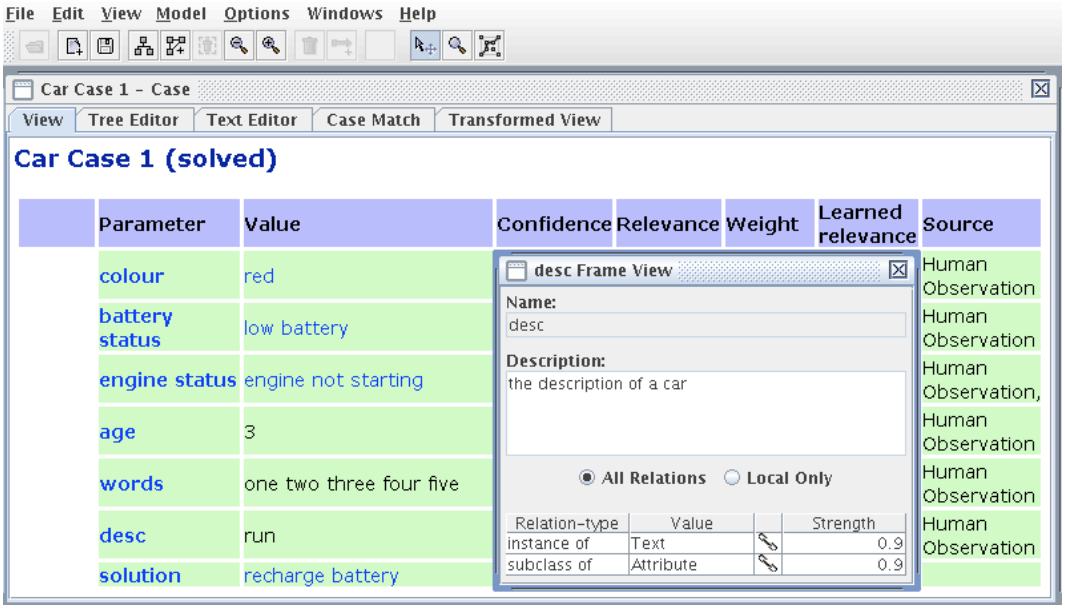

Figure 6.2: Screen shot from the VolveCreek Knowledge Editor with the demonstrator system loaded

case had "runs", and we can see that it was stemmed to "run". The same happened to the third case's "running" value. All stemmed tokens are identical, which is what we wanted.

If we had used the Equal similarity function on the stemmed values of `desc`, it would have returned 1 instead of 0. The VolveCreek GUI could still have shown the original tokens as each Token has both an original and a stemmed version, but assuming some work on the GUI, the line showing the `desc` attribute in figure 6.1 could then be as follows:

```
desc      1      desc: run [running]      desc: run [run]
```

This would mean a match between *"running"* and *"run"*, both stemmed to *"run"*, for the attribute `desc`.

Since the values were added in VolveCreek and treated by the code of both systems without incidents, the `Text` data type must also be working properly. The model is also saved successfully because of the serialization fix applied in section 4.2. If we open the model generated and saved during the execution of this demonstration, we can, e.g., view *Car Case 1* like in Figure 6.2. The Creek Knowledge Editor is used to view the model. The columns covered by the *"desc Frame View"* are not set in the demonstration.

6.3 Summary

We expand the example application with our three new components to form a demonstration. The new components are the new data type `Text`, the method *StemmerMethod* and the similarity functions `TokensContained`, `Interval` and `Equal`. To test them, we add a new entry to the car cases with a Text value. In addition, some other entries are added to test the similarity functions.

The similarity functions are computing the right similarities on our old and new case entries. The new method and data type is also working as the Text values are stemmed by the method and giving the correct output.

Chapter 7

Evaluation and Discussion

This chapter will discuss two solutions. The first solution has been the focus of this report so far. The second solution addresses the last goal presented in the introduction chapter, related to bringing VolveCreek to jColibri as an extension. As mentioned earlier, the latter solution is interesting because of jColibri's goal of formalizing CBR and becoming a standard for CBR system development.

7.1 Importing jColibri Components to VolveCreek

In this project it has been shown that it is possible to import several of the major components of jColibri into VolveCreek. The outstanding and essential issue is representation, but this was not part of the goal in this project, although it has been discussed throughout this report and [Sti06]. A perfect solution to the representation problem is unlikely to exist. We can argue that the representational issues are addressed at some level, but a complete solution was not attempted. Perhaps it is not even necessary or wanted to merge the representations either. Maybe we are better off having one system with both representations, giving us the best of both worlds. The second part of this chapter will to some degree end up with such a hybrid system, after the VolveCreek system is placed as an extension in jColibri and brings its own representation. More about this in section 7.1.

The connector in this project is translating from one representation to the other, and we are also using some wrappers. In [Sti06], a simplified model was exported as OWL, and the export of methods was also explained (some

of that work influenced section 7.1). In both cases we have the situation where everything is not necessarily possible to translate, wrap or export. However, the solutions are still good enough to be very practical in several areas. An obvious example is that we can use this approach to test various aspects of jColibri from VolveCreek in a very cheap way, which is interesting in itself because it gives the developers information that may be very valuable and expensive to obtain otherwise.

We will now discuss the import of each component.

7.1.1 Helper Functions

The similarity functions are probably the most successfully imported components in this project, and they were also a very high priority. It is possible to improve the solution quite a bit, but the demonstration shows the important points.

Since these functions are using values directly in both systems, they are much easier to work with compared to other components. A similarity function is rarely concerned with anything but the value itself, although it can sometimes be interesting to scale the results or otherwise influence the values based on the application in general or its state.

With jColibri similarity functions you can sent parameters to affect the similarity measures, while in VolveCreek the comparison between two values will always be the same. An example could be seen in the `Interval` similarity function imported from jColibri in this project, where a variable was used to scale the similarities. It would be a good idea for VolveCreek to implement something equivalent.

The comparison controller plays an important role in VolveCreek, and as mentioned earlier it appears to be a good idea with such a component. This component may have a lot of potential, and could work as an important layer between the case components and the similarity functions. If it is not there, we are more dependent on the similarity functions than if we had a layer to control it. With jColibri defining which similarity functions that should be used in the case structures, it does appear to be less flexible. jColibri does gain a lot of flexibility by having implemented classes to take care of parameters, and although it has a very thorough distributed architecture, it could potentially become better by adapting some of VolveCreek's recent development.

7.1.2 Data Types

Data types are implemented in a very straight-forward way in both systems. We showed that importing the `Text` data type was possible, and that we had to implement a new entity type before using it. When the entity type was implemented, the new data type could be used directly.

It would be preferable to have a general way to import all data types, and it may actually be possible. It really depends on the data types, as some aspects like serialization might become problematic. We can conclude that it is not a lot of work to import them, however.

The VolveCreek system is not well organized when it comes to some components, and the data types are perhaps the most obvious case. VolveCreek would benefit from organizing components in different packages and making them available to the rest of the system through some kind of interface. Right now, adding new data types requires editing the source several places. This is something which is normally solved when the software matures, however, and it is not important in the beginning.

7.1.3 Methods

The methods were harder to import than other components because of their centralized position in the software. The solution chosen, which had a focus on usage, has both positive and negative sides.

A method may do anything to any part of a system, and hence it is hard to create a general solution to import them. The solution implemented in this project works well with many methods, but not everyone. Those that cannot be used with this kind of approach are not interesting for VolveCreek at this point, however.

Methods assigned to tasks just to configure the execution while doing nothing to the context are not interesting, and they only exist to guide the application execution in jColibri as well. Methods using helper functions may be interesting, but since the helper functions can be used directly, they are not really necessary although often convenient. Methods accomplishing various tasks in specific extensions are very interesting on the other hand. The stemmer method imported in this project is a good example of such as method, an as we saw, we often have to import other components like data types and similarity measures to really make use of it.

It is not necessarily just methods created for jColibri that are interesting for

VolveCreek, but methods that enable jColibri to use external projects. The stemmer is yet again a good example. jColibri is able to use an external stemmer in its system, and VolveCreek would like to do that as well. By reusing jColibri's code, VolveCreek can use the stemmer without having to implement everything. It is of course an advantage to be able to use external projects directly instead of going through jColibri, but at least for testing purposes and comparison of solutions, this is a very cost effective solution.

By enabling reuse of jColibri components, VolveCreek gains access not only to jColibri components but also to all other projects which jColibri can work with. Because of jColibri's focus on standardization and close relation to efforts such as the Semantic Web, we may be looking at quite a few projects as development continues. This is a very important motivation.

Finally, there has been some progress during the last period of this project, which did not make it into this report nor the implementation. It appears that it would in fact be possible to import methods in a more direct way, and that it would not have caused us to modify existing methods directly. This has not been implemented or tested, but the conceptual idea does seem to work. That said, the solution used in this project is not bad, and it does serve its purpose as a possible solution. The conclusion that a strict import was not usable, however, may not have been completely accurate. The fact that another solution would require more work was accurate, however the amount of additional efforts may not be as big as previously estimated. Time did unfortunately not allow for a thorough investigation, and further details are left to future projects.

7.2 VolveCreek Extending jColibri

We will now look at an alternative solution, which is to move VolveCreek into jColibri as an extension. Since jColibri has come further in its development, this seems like an attractive alternative in a possible integration or cooperation between the two systems. This approach seems very logical in many ways because of jColibri's focus.

Because of the earlier project [Sti06] and the way jColibri was explored and studied in preparation for this project, some of the ideas regarding having VolveCreek as a jColibri extension is partly tested with experimental code. This section will attempt to explain this approach by describing how a possible implementation could be done. This is based on conceptual ideas, but explained by going through an approach that was attempted.

7.2.1 Models

The efforts related to the models are strictly representational, and this was worked on in [Sti06]. The solution presented was good, but not complete since the model was slightly simplified. Since the representation is unlikely to be completely solved like mentioned in the beginning of this section, it is important to see if they can somehow live side by side.

The jColibri context could potentially have a VolveCreek knowledge model, and the VolveCreek extension could use that model for its strengths. A specialization of the `CBRContext` would be necessary, and some translation between the two representations would also become necessary. This is a similar functionality to the connector implementated in the previous solution.

7.2.2 VolveCreek Components in jColibri

We will check how well VolveCreek can function as an extension of jColibri by looking at one component at a time. There are not many of each component in VolveCreek, so possible changes to them would not be a huge effort, but we will go through them to see what is needed.

Similarity Measures

The VolveCreek similarity measures will have to become helper functions, and more specifically they must implement the *SimilarityFunction* interface. In section 3.3.4 we found out that jColibri and VolveCreek are very similar when it comes to these functions, so all we need is to merge the two interfaces `SimiliarityFunction` and `SimilarityMeasure` which is not a major operation.

A good solution would be to have both `compute` and `similarity` methods in each, so they could be called with both VolveCreek entries from the knowledge model now in the context, and with jColibri individuals.

Some other comparison components in VolveCreek could still be used like they are today, but they would have to be moved into methods. This is explained later.

Data Types

Data types would have to be configured like other data types in jColibri, and they also need an editor. VolveCreek does not have any data types that are custom to the system, so this is not an issue. E.g. we could have configured the URL data type with the following XML code located in `/config/creek/datatypes.xml`:

```
<DataType>
  <Name>URL</Name>
  <Class>java.net.URL</Class>
  <GUIEditor>jcolibri.gui.editor.URLEditor</GUIEditor>
</DataType>
```

The class `jcolibri.gui.editor.URLEditor` would have to be implemented.

Methods

We would like to have the VolveCreek methods defined through CBROnto, and we would like to give them competencies to solve certain tasks. The definition is not a problem, but we must change the implementation. Each method must implement the `CBRMethod` interface, which basically means that it must have an `execute` method. VolveCreek does not have a lot of methods either, so we do not have any serious issues with many existing components. The next section will exemplify how it can be done and also defines the methods through CBROnto.

All in all, we do not have any major issues with components, as they are few and can be changed slightly to work with jColibri. The challenge is to make the idea behind the VolveCreek system work in jColibri, not its system components.

7.2.3 Example Application

This example does exactly the same as the original VolveCreek example. The first thing we have to do, is to place the entire VolveCreek source in `/src/jcolibri/extensions/creek`, refactor all paths and create a configuration file for the extension. The configuration follows:

```
<jCOLIBRIPackage>
  <name>VolveCreek Components</name>
  <description>Contains components from VolveCreek</description>
```

```
  <path>creek</path>
  <methods>methods.xml</methods>
  <tasks>tasks.xml</tasks>
  <datatypes>datatypes.xml</datatypes>
</jCOLIBRIPackage>
```

When this is present, we will get an option to enable the extension when we open the jColibri GUI and start making a new system.

We will separate the code from the VolveCreek example application and put them into methods implementing CBRMethod to illustrate how this can be done since we do not have any such methods in VolveCreek. There are others we could have used, but it would have been the same operation. The code for the methods are copied directly from the VolveCreek example file, and placed in `execute`. This is necessary, and it is the main effort needing to be completed outside the definitions. Following is a list of methods and a description of what they do.

- `CreekPreCycleMethod` does nothing by returning the unchanged context;
- `AddAttributesMethods` adds the attributes to our model;
- `AddCausalModelMethod` adds the causal model as defined by the VolveCreek example;
- `AddCasesMethod` adds the car cases;
- `SolveCreekCBRMethod` executes the `RetrievelResults` reasoning step implemented for VolveCreek;
- `CreekPostCycleMethod` does nothing by returning the unchanged context.

Since we have now implemented several methods, we want to define them through CBROnto, give them competencies and hence we also need some tasks which they can solve. Following is one task followed by a method which has the competence to solve it. This is defined in `/config/creek/tasks.xml` and `/config/creek/methods.xml`.

```
<Task>
  <Name>AddCases</Name>
  <Description>Add cases</Description>
</Task>

<Method>
  <Name>jcolibri.extensions.creek.method.AddCasesMethod</Name>
```

```
  <Description>Adds the cases</Description>
  <Type>Resolution</Type>
  <Competencies>
    <Competence>AddCases</Competence>
  </Competencies>
</Method>
```

An important point is that the earlier description of the methods are only their implementation. The CreekPreCycleMethod is actually a decomposition method, and that is why it is only returning the unchanged context. Its definition contains the following, which configures the application to use the other methods in the precycle.

```
<SubTasks>
  <SubTask>AddAttributes</SubTask>
  <SubTask>AddCausalModel</SubTask>
  <SubTask>AddCases</SubTask>
</SubTasks>
```

Once the precycle task has been configured to be solved by CreekPreCycleTask, these three subtasks will appear. When configuring them, only one method will show up as available for each, and the methods showing up are those we just added. This is a very basic use of the jColibri system, but illustrates how easy one can migrate from VolveCreek to jColibri without reimplementing everything.

Figure 7.1 shows that the application is configured in jColibri, with some custom methods that were created just to test. If we generate this application and execute it, we get exactly the same result as the CreekExample, which should not be a surprise. We have not done anything big.

The question now is if we can make this work as easy with other types of applications, and how these would be developed. Which representation they would use and other similar questions would have to be answered.

The idea that was suggested earlier where both representations live in the same system appear to be very interesting. This is solved by specializing some of the jColibri components in a similar way to what was done in the previous solution earlier in this project. Creating a new case base which can work with a VolveCreek knowledge model is possible, and it is not a whole lot of work either. In fact, if we specialize the major components of jColibri to work with VolveCreek's knowledge model, we have solved almost everything except the situations where we want to actually use jColibri's representation and reasoning mechanisms. This can also be done, but like in the previous solution, it does require that we translate from one to the

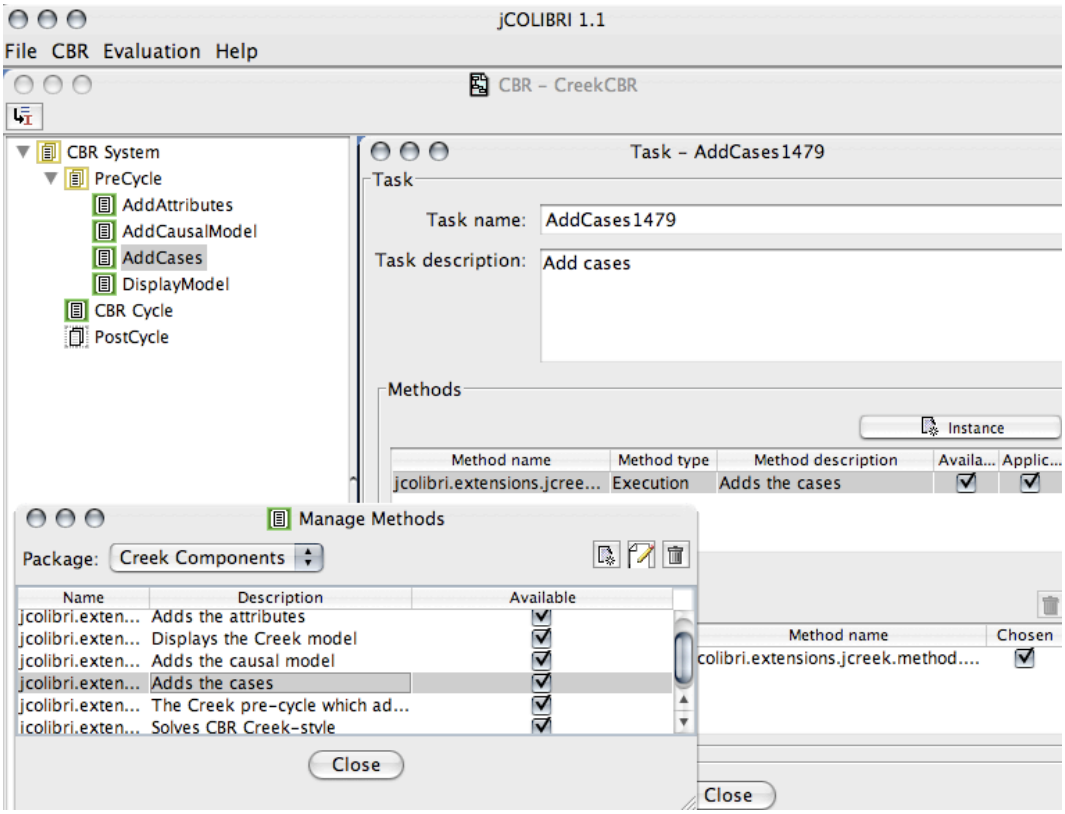

Figure 7.1: Configuring the CreekExample in jColibri

other, and then back again if we want to apply the results.

Although this attempt seemed to work very nicely, it is clear that it has a lot to do with the VolveCreek system not having a lot of components ready. The architecture is there, but since they can also be defined in jColibri without a lot of work, we are not looking at any major issues. jColibri's architecture is written in such a general way that it is actually about to formalize CBR, should we believe our experiences with the system.

For VolveCreek as a comercial product, this may not be quite as interesting as the former solution, and that is why the former solution was given more attention in this project. For academic purposes, it seems like a good idea to start using the jColibri framework.

Chapter 8

Conclusion and Further Work

This final chapter describes further work, and concludes the project.

8.1 Further Work

There are many issues touched by this project that should be further investigated. They can be logically divided into a VolveCreek view and a jColibri view. The VolveCreek view contains work having VolveCreek in focus, assuming a continued development of VolveCreek in the current direction. The jColibri view contains work where the focus is to make VolveCreek a part of jColibri which was discussed in section 7.1. These two views will be addressed in the following subsections.

8.1.1 VolveCreek View

It is clear that a lot of work is yet to be done, and this project is merely a start. The goals of this project was not to finalize a solution, but to identity efforts and exemplify some of the things that are possible. This means that the code produced in this project is not necessarily meant to be used for anything else than exemplification, although generally good solutions have of course been attempted.

Some of the implementation done in this project is rather specific towards the demonstrator system. The systems are large and the components many, so a project like this does not have enough time to work thoroughly through everything. It is perhaps not best to continue development on top of this

project, but rather use it to get started quickly should the results of this project be interesting for further development.

Each component can be improved quite a lot, and perhaps most of all the methods. Very important in this regard is the task component. A considerable amount of time should be used to analyze how this component can be a part of VolveCreek. Even if that is not possible, VolveCreek should still consider implementing something with a similar functionality. VolveCreek does have the `CBRReasoningStep` abstract class with variables such as `state` which can be compared to some things found in jColibri, but it does not provide enough to compensate for the tasks which are more general. A task and method hierarchy with competencies is a very powerful approach. If they are also defined by ontologies in the model then that is even better. jColibri has developed a very good solution here, which VolveCreek could use as an inspiration.

Other components, namely data types and similarity functions, are imported rather well in this project. The solutions should get better error-handling and other things making them easier to use, but other than that we have shown that they can be imported without a lot of problems.

Regarding the representation, there has been suggested several ideas both in this project and other projects. None of these projects have had the representation as its main focus, so perhaps that is what needs to be done. Such a representation-focused project should look at the problem from several different angles, including some kind of integration between them, hybrid solutions or some way of using them all together with an interface to translate between them. The representation has become increasingly advanced lately because of the focus on specific knowledge in addition to general knowledge which has been a focus for a long time. This representation issue should be given a thorough investigation.

8.1.2 jColibri View

The solution where VolveCreek tries to become an extension of jColibri seems to be something that is very possible with what we have today, but it does require a lot of work if we want a flawless integration. First of all, the VolveCreek system must be slightly changed to easier fit within the jColibri framework, but this is not going to cost VolveCreek any functionality. This is strictly a code design issue, and can be worked through fairly easily since VolveCreek has not been finished and is not being used extensively in a lot of projects. In fact, now is a good time to work on it, if we want to do it.

It is very likely that the developers of VolveCreek have plans that go past what this project has discovered, and that this complicates things quite a lot. This solution is mostly interesting for an academic version of Creek.

8.2 Conclusion

This project has analyzed jColibri and VolveCreek, and compared them to find similarities and differences. Based on the comparison, it was shown that it is possible to reuse external library components in the VolveCreek CBR system. The import of three different jColibri components was constructed, implemented and evaluated. One data type, one PSM and three similarity functions.

The construction shows that it is possible to create fairly general solutions for importing components, and that they will work with the existing VolveCreek system. Later chapters, which slightly simplifies what was outlined in the construction, shows that the implementation is not extensive, and that the components can be imported and tested which can give VolveCreek developers valuable information.

The data types and similarity functions are imported in a clean way, while the methods are more painful. The methods would be easier to import had they not been implemented been so centralized in jColibri. It was later realized, however, that the best solution for the methods was perhaps not used in this project.

The evaluation shows that the project goals were accomplished, and that the demonstrator system brings the expected results. In the demonstration, the new data type `Text` is used when adding a new entry to several cases, and the data type is later stemmed by an imported method. Several jColibri similarity functions were applied to VolveCreek case entries, and they returned the correct similarity value.

The final goal of the project was accomplished in section 7.1, where the alternative approach was discussed and partly tested. It was shown that such a solution is possible, and that the cost is not very high. In fact, from an academic point of view, it may even be preferable to use this approach rather than the one having a focus in this report. The approach used in this project is more interesting for VolveCreek as a commercial product, however.

Bibliography

[Aam94Nov] Agnar Aamodt
A Knowledge Representation System for Integration of General and Case-Specific Knowledge.
Proceedings from IEEE TAI-94, International Conference on Tools with Artificial Intelligence. New Orleans, November 5-12, 1994. 4 pages.

[Aam04] Agnar Aamodt
Knowledge-intensive case-based reasoning in Creek.
ECCBR 2004. LNAI 3155, Spinger, 2004. pgs. 1-16.

[AP94] A. Aamodt and E. Plaza
Case-based reasoning; Foundational issues, methodological variations, and system approaches.
AI Communications, 1994, pages 39 - 59.

[Bra04] Stein Erlend Brandser *The jCreek Programmer's Guide*
URL: `http://creek.idi.ntnu.no/docs/jCreek-ProgrammersGuide.doc`

[BSAB04] Tore Brede, Frode Sørmo, Agnar Aamodt, Ketil Bø *A Knowledge Modeling Editor and Testing Environment for Knowledge-Intensive Case-Based Reasoning*
URL: `http://creek.idi.ntnu.no/docs/TrollCreek-tutorial.doc`

[Cha90] Chandrasekaran, B
Design problem solving: A task analysis.
AI Magazine, 11:59–71. 1990

[Creek] Homepage of Creek
`http://creek.idi.ntnu.no/`
Last visit: June 7th, 2007.

[DGGG05] Belen Díaz-Agudo, Pedro A. González-Calero, Pedro Pablo Gómez-Martín and Marco Antonio Gómez-Martín
On Developing a Distributed CBR Framework through Semantic Web Services

Proceedings of Workshop OWL: Experiences and Directions, at International Conference on Rules and Rule Markup Languages for the Semantic Web, 2005

[Dia00] Díaz-Agudo, P. González-Calero
An architecture for knowledge-intensive CBR systems
Proceedings of the 5th European Workshop on Advances in Case-Based Reasoning, 2000

[Dia02] Belen Díaz-Agudo and Pedro A. Gonzalez-Calero
CBROnto: A Task/Method Ontology For CBR
In S. Haller and G. Simmons, editors, Proc. Of the 15 the International FLAIRS?02 Conference. AAAI Press, 2002.

[DINS96] Donini, F. M., lenzerini, M., Nardi, D., and Schaerf, A.
Reasoning in description logics
Pages 191 - 236.
URL: `http://www8.informatik.uni-erlangen.de/IMMD8/Lectures/KRR/reasoning-in-DL.ps.gz`.

[GGDF99] Gomez-Albarran, M., Gonzalez-Calero, P. A., Diaz-Agudo, B., and Fernandez-Conde, C.
Modelling the cbr life cycle using description logics
URL: `http://gaia.fdi.ucm.es/people/pedro/papers/1999_iccbr_mercedes.pdf`.

[Gru93] T.R. Gruber.
A translation approach to portable ontology specifications.
Knowledge Acquisition, 1993, Vol. 5, No. 2, pp. 199 - 220.

[jColibri] Homepage of jColibri
`http://gaia.fdi.ucm.es/projects/jcolibri/`
Last visit: June 7th, 2007.

[NB03] Nardi, D. and Brachman, R. J.
An introduction to description logics
2003. URL: `http://www.inf.unibz.it/~franconi/dl/course/dlhb/dlhb-01.pdf`

[RDGW05] Juan Antonio Recio, Belén Díaz-Agudo, Marco Antonio Gómez-Martín and Nirmalie Wiratunga
Extending jCOLIBRI for Textual CBR
Proceedings of Case-Based Reasoning Research and Development, 6th International Conference on Case-Based Reasoning, ICCBR 2005, pages 421-435.

[RSDG05] Juan A. Recio-García, Antonio Sánchez, Belén Díaz-Agudo, and Pedro A. González-Calero.
jCOLIBRI 1.0 in a nutshell. A software tool for designing CBR systems.
In M. Petridis, editor, Proccedings of the 10th UK Workshop on Case Based Reasoning, pages 20-28. CMS Press, University of Greenwich, 2005.

[SRDG05] Antonio Sánchez, Juan A. Recio, Belén Díaz-Agudo, and Pedro González-Calero
Case structures in jCOLIBRI
Best Poster Award. Twenty-fith SGAI Int. Conf. on Innovative Techniques and Applications of Artificial Intelligence, AI 2005. Cambridge, UK.

[Ste90] Luc Steels
Components of expertise
AI Magazine Volume 11 , Issue 2 (Summer 1990) Pages: 30 - 49

[Sti06] Erik Stiklestad
Sharing Models between TrollCreek and jCOLIBRI
TDT4745 Knowledge Based Systems, Autumn 2006

[SPGKY07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur and Yarden Katz
Pellet: A practical OWL-DL reasoner
Journal of Web Semantics, 5(2), 2007.

[Sør00] Frode Sørmo
Plausible inheritance; semantic network inference for case-based reasoning
Master's thesis, Norwegian University of Science and Technology, Department of Computer and Information Science.

[W3C01] W3C Semantic Web Activity
URL: `http://www.w3.org/2001/sw/`