NTNU – Trondheim
Norwegian University of
Science and Technology

CREEK and Description Logics

Mads Opheim

Master of Science in Computer Science
Submission date: June 2014
Supervisor: Agnar Aamodt, IDI
Co-supervisor: Roar Fjellheim, Computas AS

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

Problem description

CREEK is a system and a method thoroughly described, but at the moment without a publicly available implementation. In this thesis, the core of the system will be implemented in a new way.

CREEK is a system for diagnosis and problem solving intended for open domains, and uses a frame-based knowledge representation based on prototype theory and a prosedural semantics. What would be the consequences of replacing that type of frames with Description Logics in CREEK? Can this be implemented within a reasonable scope? Which consequences will this have, in terms of benefits and limitations?

The work will include a demonstration of the implemented system within a suitable domain, and an evaluation of the results.

This report is licensed under a Creative Commons
Attribution 4.0 International License.
http://creativecommons.org/licenses/by/4.0/

Abstract

Creek is a system for diagnosis and problem solving. It makes heavily use of general knowledge, and is intended for open and weak theory domains. The system uses this general knowledge in combination with experiences from previous cases to solve new problems.

We have implemented the core parts of Creek inside the Colibri studio framework, and shown that it is both possible and realistic to create a thorough Creek implementation. This has resulted in a system named Colibreek.

In this implementation, we have replaced the knowledge representation in Creek, switching from the original frames to description logics. This has proven successful, and promising for further development.

We have tested our system on the turbine sensor domain, and shown how our system can be used for condition monitoring on sensor-monitored equipment such as turbines. By doing this, we have also shown how the system can solve real problems people, companies and organisations have.

Samandrag

Creek er eit system for diagnostisering og problemløysing. Det kvilar tungt på bruken av generell kunnskap, og skal løyse problem innanfor domene som er opne og med svak, usikker teori. Systemet bruker denne generelle kunnskapen saman med erfaringane frå tidlegare hendingar for å løyse nye problem.

Vi har implementert dei sentrale delane av Creek innanfor rammeverket Colibri studio, og vist at det er både mogleg og realistisk å lage ein heilskapleg Creek-implementasjon. Dette har resultert i eit system vi har kalla Colibreek.

I denne implementeringa har vi erstatta kunnskapsrepresentasjonen i Creek, frå rammer originalt til skildringslogikk. Dette har vist seg å vera suksessfullt, og eigna for vidareutvikling.

Vi har testa systemet vårt på turbindomenet, og vist korleis systemet kan vera veleigna for tilstandsovervaking av sensor-overvaka utstyr som turbinar. På denne måten har vi også vist korleis systemet kan løyse reelle problem folk, verksemder og organisasjonar har.

Preface

This thesis describes the work done in my master's project in the course *TDT4900 - Computer and Information Science, Master Thesis*, at the Department of Computer and Information Science at Norwegian University of Science and Technology (NTNU).

The thesis is written for the company Computas AS, and I would like to thank everyone at Computas for good discussions, help and cooperation at every crossroad.

I would like to thank my supervisor Agnar Aamodt, and my co-supervisor Roar Fjellheim, for invaluable feedback, guidance and sharing willingly of their insight. A special thanks also to Dmitry Ivanov at ConocoPhillips Norway AS for providing me with test data.

I would also like to thank all the participants in the ENIOC project for feedback and encouragement.

Mads Opheim

Oslo, June 12, 2014

Contents

1 Introduction and overview **1**
- 1.1 Background and motivation . . . 1
  - 1.1.1 Why CREEK? . . . 1
  - 1.1.2 Creek and description logics? . . . 3
- 1.2 Goals and research questions . . . 4
- 1.3 The structure of this report . . . 4

2 Methodology **5**
- 2.1 Search process . . . 6
- 2.2 Study selection process . . . 7
  - 2.2.1 Quality assessment . . . 7
- 2.3 Technical methodology . . . 8
  - 2.3.1 Are we developers or designers? . . . 8
  - 2.3.2 Dynamic architecture . . . 8
  - 2.3.3 Agile? . . . 9
  - 2.3.4 Choice of develoment-driver . . . 10
  - 2.3.5 Tools and libraries . . . 11

3 Theory and background **13**
- 3.1 Case-based reasoning . . . 13
- 3.2 Knowledge-intensive CBR . . . 15
- 3.3 Knowledge representation . . . 17
  - 3.3.1 Domains . . . 18
  - 3.3.2 Frames . . . 19
  - 3.3.3 Description logics . . . 20
  - 3.3.4 Ontologies . . . 21
  - 3.3.5 Semantic networks . . . 22
- 3.4 Creek . . . 25
  - 3.4.1 The CBR cycle in Creek . . . 26
  - 3.4.2 Knowledge representation in Creek . . . 29

3.4.3 Creek and frames 31
3.4.4 Creek and reasoning 31
3.5 CBR frameworks of today 33
3.6 myCBR 33
3.7 Colibri 34
3.7.1 The Colibri Studio architecture 35
3.7.2 Colibri and representation 37
3.8 The domain - turbines 37
3.8.1 Condition monitoring 37
3.8.2 Our domain 38
3.8.3 Use of CBR in condition monitoring 41
3.8.4 Existing CBR systems in oil and gas 41
3.9 Applications of Colibri 42
3.10 Existing combinations of elements of Colibri and Creek 42

4 Architecture **44**
4.1 Characterising parts of Creek? 44
4.2 Why starting with an implementation framework? 46
4.3 Choice of framework - why Colibri? 48
4.4 Uniform knowledge 49
4.5 Our architecture 50
4.5.1 Introduction 50
4.5.2 Major architectural choices 51
4.5.3 Domain independence 52
4.5.4 Our architecture - part by part 52
4.5.5 The architectural dependency on Colibri 55

5 Colibreek **56**
5.1 Using Colibreek 56
5.1.1 Configuring Colibreek for your domain 56
5.2 Implementational principles - a system to be reused 57
5.3 Implementational choices 58
5.3.1 Retrieve 58
5.3.2 Constants and configurations 59
5.3.3 Relevance factor 59
5.3.4 Similarity measurement 60
5.4 The implemented architecture 61
5.4.1 Step-by-step 62
5.4.2 The representation of cases and the domain 63
5.5 A walkthrough of the system - under the hood 64
5.5.1 CBR reasoner 64

5.5.2 Activator 64
5.5.3 Explainer 66
5.5.4 Focuser 66
5.5.5 Explanations to the user 66

6 Application of Colibreek on our domains **69**
6.1 How to use Colibreek - as an end user 69
6.2 Our choice of ontology 70
6.3 What is in a case? 71
6.3.1 Travel recommender 71
6.3.2 Turbine sensor domain 72
6.4 Runtime analysis 73
6.5 Presentation to the user 74
6.5.1 A Colibreek run, step by step - from the user's perspective 76

7 Evaluation **79**
7.1 Evaluation through the five-stage model 79
7.1.1 Stage 1 - task and view 79
7.1.2 Stage 2 - refine the view to a method 80
7.1.3 Stage 3 - implement it 81
7.1.4 Stage 4 - design experiments 81
7.1.5 Stage 5 - run experiments 83
7.2 Colibri and test-driven development 83
7.3 Adapting Colibreek to a new domain 84
7.4 Turbine domain test results 85

8 Discussion and conclusion **87**
8.1 Frames vs DL in Colibreek 87
8.2 DL in Creek within a reasonable scope? 88
8.3 Decision-support for turbines in DL-Creek 89
8.4 Achieving the goal 89
8.5 Conclusion 89
8.6 Further work 90

A Licenses **j**

B How to run **k**

List of Figures

2.1 The TDD cycle 10

3.1 The CBR cycle 14
3.2 The knowledge-intensiveness-scale 16
3.3 An example of a semantic network 23
3.4 The SSN ontology, visualised in Protege 24
3.5 Activate, explain, focus in each CBR step 26
3.6 Another view on the activate-explain-focus cycle 27
3.7 Creek's four knowledge modules 29
3.8 Creek's knowledge hierarchy 30
3.9 The reasoner-choosing process in Creek 32
3.10 An overview of the Colibri studio architecture 36
3.11 The colibri engine on cycles 36
3.12 The onshore reliability centre 38
3.13 The issue types 39

4.1 Time spent on application vs. infrastructure 47
4.2 The architecture of Colibreek 53
4.3 A diagram showing the sequence of a run of the system 54

5.1 The architecture of the implemented system 62

6.1 The case solution structure for the turbine domain 73
6.2 Resource-consuming methods for turbine sensor domain 74
6.3 Distribution of resource-demand between our modules 74
6.4 Distribution of resource-demand in the explain module 74
6.5 Explanation and solution 75
6.6 Run: Namespaces and classifying 76
6.7 Run: Input from the user 77
6.8 Run: Colibri's realizing-print 77
6.9 Run: The query displayed to the user 77

6.10 Run: A case and explanation presented to the user 78
6.11 Run: The proposed solution and solution case 78

7.1 Evaluation results for the turbine AGB domain 85

List of Tables

2.1 Sources 6

4.1 Priorities 45

7.1 Questions for stage 1 80
7.2 Questions for stage 2 80
7.3 Questions for stage 3 81
7.4 Questions for stage 4 82
7.5 Evaluation types 82
7.6 Questions for stage 5 83
7.7 Leave-one-out-results 86

Chapter 1

Introduction and overview

1.1 Background and motivation

The fundamental idea behind this thesis is that we as software developers today reinvent far too many wheels. Instead of reinventing the wheel over and over again, we should have a conscious view on taking advantages of what developers prior of us have done and how our problem can be solved in an effective and suitable way.

1.1.1 Why CREEK?

Why are we doing the work described in this thesis? Why do we think CREEK[1] is still interesting and worth spending time, money, work and resources on?

Creek is a system for solving diagnosis and repair problems, intended for expert users and complex problems. It is a system for case-based reasoning, heavily inspired by the way humans solve such problems.

We believe in the underlying idea of Creek, that a huge degree of general knowledge in combination with the cases is a good way to solve many CBR problems.

Opheim[Opheim, 2012] did a study of the state-of-the-art as of autumn 2012, and found several findings that are interesting for us, bearing in mind our

[1] Abbreviation for "Case-based Reasoning through Extensive Explicit Knowledge", we will in the rest of our report use Creek rather than CREEK, for the sake of readability.

belief in Creek as a promising system with useful mechanisms. Firstly, there is no implementation of Creek today available to the public. The private software firm Verdande Technology has developed a software, DrillEdge, based on the earlier implementation of Creek called TrollCreek, and which the company owns.

We believe that there are several research topics related to the overall Creek framework not yet elaborated and tested, and that there is a potential of improving the Creek architecture further, and have ourselves more ideas than time of how this could be done. In section 8.6, we will list the ideas time and resources prohibited us to include in this report. With so many ideas to test, this would be significantly more feasible if there were a program skeleton to build from. We intend to provide this skeleton in a new system design and implementation, based on the openly published Creek architecture and core mechanisms.

This skeleton would serve at least two purposes: it would lower the threshold of even considering Creek as a system to improve, and it would make it easier to play around with, tweaking, testing and improving the system. In order to be able to achieve the purpose with this thesis - to test Creek with description logics - we obviously need to have an implementation of (parts of) Creek. Since there is no implementation available, we need to make this ourselves. We hope to do this process easier for coming students, researchers and developers interested in improving Creek by providing this skeleton. Here it is also worth mentioning that Creek is a comprehensive system, and there are probably a lot of working hours required in order to implement the entire system. We will in this thesis do our share.

Opheim[Opheim, 2012] also found that Creek introduced several characteristics hardly seen in any other CBR system, neither when the system was designed or now. The crucial role of general knowledge and the active-explain-focus cycle are good examples of this. We concluded then that no system beared in it much enough of Creek to satisfy us.

Another important conclusion we made then, was that CBR development of today usually starts from frameworks. The leading frameworks seemed promising to build a possible Creek implementation from.

We would also point out that Creek is useful. We have mentioned that we believe that Creek is a both interesting and promising system, but as important, we also believe that Creek can solve actual real world problems. Verdande Technology's DrillEdge software already mentioned is a good proof of this, it is used in oil drilling all over the world. Verdande Technology has

also adapted DrillEdge to as different domains as finance[Verdande, 2014a] and healthcare[Verdande, 2014b], where their solutions also are in real use, though not in the same propagation as DrillEdge is.

We are convinced that Creek can suit many other domains and tasks as well. In this thesis, we use Creek to help solving a real world problem for a huge corporation, with a potential saving of millions of dollars a year. Shokouhi et al.[Shokouhi et al., 2010] gives a brief overview on how CBR can be used in the oil well domain, and we can easily imagine Creek applied in all these sub-domains, as well as in the mentioned healthcare and finance, but also law and other domains as well.

The sum of these considerations lead us to the conclusion that a new implementation of Creek was desirable. It is beyond our scope - of reasons due to time and resources - to implement the entire Creek system, we will focus on implementing essential parts.

1.1.2 Creek and description logics?

Our topic is angled on testing Creek with description logics. We will test whether it is even possible within a reasonable scope to implement, and second, if it is desirable. What will the benefits be, what will the disadvantages be?

Creek is by design based heavily on frames, as originally described by Minsky[Minsky, 1975] as the way of representing knowledge. In the time gone after Creek was launched, description logics have become increasingly popular, and we have the strong impression that most of the systems created today uses some kind of description logics.

Hence, it may be a bottleneck for further Creek development that it is so heavily based on frames. We fear that this may repel developers from using Creek.

We will not dive deep into the disagreement between the entuasiasts on respectively frames and description logics, but are eager to find out the possibilities of combining Creek and description logics. During the work on this thesis, we intend to experience whether this will be a failure, a success or something in-between.

1.2 Goals and research questions

Goal: Study how the core of the Creek system can be adapted to a description logics representation.

Research question 1: What would be the consequences of replacing frames with description logics in Creek? Which consequences will this have, in terms of benefits and limitations?

Research question 2: Can DL be implemented in Creek within a reasonable scope?

Research question 3: How can decision-support for reported turbine failures be implemented within a DL-based Creek system?

1.3 The structure of this report

Chapter 1 sets off this thesis with the methodological overview just presented.

Chapter 2 describes our methodology - how we work.

Chapter 3 gives an introduction to the theory and background information needed to understand the rest of the report and the system produced.

Chapter 4 sketches the architecture of our system.

Chapter 5 depicts the implementation we have done of Colibreek

Chapter 6 shows how Colibreek is applied on the domains travel recommender and turbine sensors.

Chapter 7 evaluates our work.

Chapter 8 concludes the report and lines up further work to be done.

Chapter 2

Methodology

We will in this chapter describe how we work in order to answer the research questions and achieve the goal outlined in section 1.2.

We have investigated the state-of-the-art, we have done background research on Creek (the system we are going to implement) and we have had a look on the Colibri framework (the framework we are going to use).

From what we learned during this research phase, we have prioritised our backlog and worked on the highest prioritised topics.

In section 1.1.1, we indicated that it could demand working hours far beyond our resource limitations to implement the entire Creek architecture. Consequently, we here present a multi-level version of our scope:

- In this thesis, we *describe* most of the central parts of the Creek architecture.
- We have sketched out how most of those can be implemented in our system in our high-level *architecture*.
- In our *implementation*, we have focused on making fewer, but better elements.
- We *test and evaluate* a subset of the implementation.

This way, we hope to assist future developers in using our work as a springboard. We believe that this upside-down pyramide-alike structure indicated by the list will be clear and clarifying on what is done and what is left out at each step.

Our approach is a combination of a model/abstraction and a design/experimental. Through modeling and an abstract description of our system, we form the foundation for answers to the research questions, while the complete answers as well as the achievement of the goal will come through the system we design and implement with use of the design/experimental approach.

A further elaboration on *what* we have prioritised is listed in table 4.1.

2.1 Search process

In the work leading to [Opheim, 2012], we did a study of the state-of-the-art and a lot of background research. This research gave us knowledge, and also made us get familiar with many articles related both to that work as well as the issues concerned in this thesis. We will use and reference these articles when needed during the work.

Of course, we have not found every interesting or rewarding article during that work. Hence, we will need to search for more information during the process.

Our search sources are listed in table 2.1. In general, we have found Google Scholar[1] to be the most suitable starting point when searching for articles. We tend to find what we are looking for when searching here, and if not, we try the other sources listed below.

ACM Digital Library	`http://portal.acm.org/dl.cfm`
CiteSeerX	`http://citeseerx.ist.psu.edu`
IEEE Xplore	`http://ieeexplore.ieee.org/Xplore/guesthome.jsp`
ScienceDirect	`http://sciencedirect.com`
Springer Link	`http://springerlink.com`
ISI Web of Knowledge	`http://isiknowledge.com`
Wiley Inter Science	`http://interscience.wiley.com`

Table 2.1: Sources

[1] `http://www.google.com/scholar`

2.2 Study selection process

In [Opheim, 2012], we defined criteria for considering an article relevant and good enough to be taken into consideration. We were satisfied with these criteria, so we have chosen to stick with the behind-laying ideas also this time. As our domain and goal has changed a bit, the criteria also needs to be updated. We consider an article relevant if it fulfills the following criteria:

- The article regards our topic
- It tells us something useful about Creek, Colibri studio or mechanisms used in one of those
- It is of acceptable quality.

Easily said, the articles should give information relevant for taking steps towards achieving our goal. We notice that the last point is somewhat fluid, but in short, it means that we should see study selection criterias in context with the quality assessment heuristics, specified below.

2.2.1 Quality assessment

We have made heuristics for evaluating the quality of articles[2], and tested our reading material against those.

1. Are the results reproducible?
2. Are the presented findings supported by the test evidence?
3. Is there a clear and precise methodology description?
4. Is the study reinventing the wheel? In other words: does the article provide any new information?

These are not hard criterias, as different types of articles requires different levels of formality. For example, the tutorial on jColibri2[Díaz-Agudo et al., 2008] have proven very useful without being scientifically appropriate according to strict definitions.

[2] Equal to those we used in [Opheim, 2012]

2.3 Technical methodology

2.3.1 Are we developers or designers?

Section 3.7.1 describes how Colibri studio is aligned for both developers and designers - with two different interfaces and underlying ideas.

The dichotomy between developers and designers leave us in some manners in a dilemma - with our purposes in mind, are we really designers or developers? Luckily, this is a straw man question, as there is no contrary between the two approaches. We hereby choose to use whatever is most suitable at any given point. Our intention in this work is to study the usage of Creek with description logics, not dive into technical details on how every part of it is technically done.

2.3.2 Dynamic architecture

Depending on the choice of development methodology, the initial amount of architecture varies a lot. The range varies all the way from *draw the final architecture at the beginning* to *the architecture will grow organically when needed*. Beck and Andres[Beck and Andres, 2004] propose a two-step approach: first, find a metaphor for the system. They advocate that as high-level as the architecture is at the initial level, it will hardly give any value anyway. During the first iteration, the developers should implement a set of tasks that enforces them to make a skeleton of the whole system. This is then the initial architecture.

Beck's proposal is intended for a team of developers actually doing work for a customer, whereas we are our own customer as well as a one-man team. However, we believe in the idea of making an architecture that changes when the requirements change or substantial new insight is gained during implementation. We chose to start with a quite simple architecture, and evolving it as required during development. The result is illustrated in figure 4.2 and described in chapter 4 . When specifying this architecture, we also pointed out the direction for the work for ourselves. This procedure is close to the one suggested by Brown[Brown, 2014], which emphasises heavily that a thoughtful, but coarse, software architecture early on is essential for successful projects.

2.3.3 Agile?

For the last few years, there seemingly have been a boom in the flora of development philosophies. Agile methods[3] have gained steadily more acceptance and popularity, and the iterative, incremental development process in these methods facilitate collaboration and adaptation, according to its supporters.

We considered a conscious approach to methodology, and quickly found that the established agile methodologies were mainly intended for groups of developers working for a customer. As we in a sense are our own customer, and work alone, neither of the standard methodology frameworks Scrum[4] nor Extreme programming (XP) [Beck and Andres, 2004] are directly applicable. A good illustration of this is that XP is considered to have 12 practices, of which we found five[5] to be completely meaningless in our setting. However, we have chosen to stick to the remaining principles.

These are test-driven development (as elaborated below), continuous integration, refactoring, coding standards, simple design, system metaphor and sustainable pace.

Continuous integration means that you should integrate your code into the main branch as soon as possible.[6]. *Refactoring* is to improve the design of code you already have: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written"[Fowler et al., 1999][p.9]. *Coding standards* is simply to write all your code with the same syntax and coding principles. *Simple design* says that you should design for what you need today, not what you might (or might not) need tomorrow. The simpler the design, the easier to maintain. A *system metaphor* tells the story about the system in a way that is easy and non-technical to understand. *Sustainable pace* means that you should not work overtime more than one week in a row, and generally limit the amount of required hours - spend more productive hours instead of just more hours. Each of these practices are elaborated more in [Beck and Andres, 2004].

[3]Summarised in the Agile manifesto, see `http://agilemanifesto.org/`

[4]`http://www.scrumalliance.org/why-scrum`

[5]Pair programming, planning game, small releases, collective code ownership, whole team

[6]Trivial for single-person-projects like ours, more crucial for bigger projects

2.3.4 Choice of develoment-driver

Test-driven development (TDD) is the most crucial part of our methodology. When doing test-driven development, you first write a test - a failing test. Then you write exactly enough code to make the test pass, in the easiest possible way. Third, you refactor the code, while making sure the tests are still passing. When done refactoring, it is time to write a new test. This cycle, illustrated in figure 2.1, repeats until you have implemented the desired functionality.

The tests we develop this way are called *unit tests*[Osherove, 2011] - we test a unit in isolation.

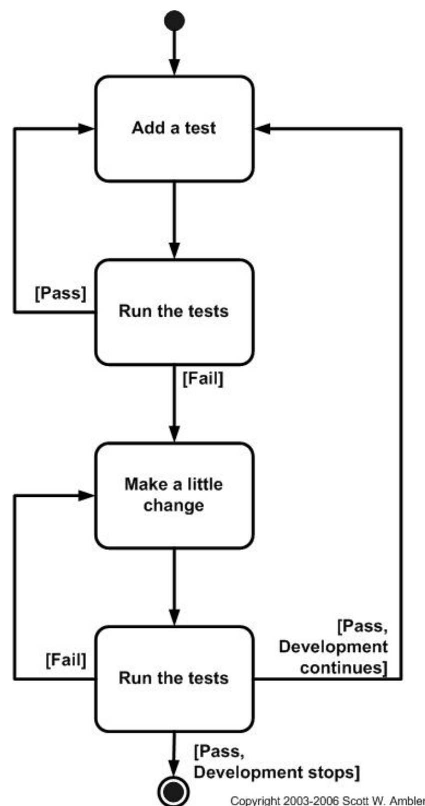

Figure 2.1: **The TDD cycle**, adapted from [Ambler, 2 13]

TDD has two appliances: for developers and for acceptance. Acceptance-TDD is often also called behaviour-driven development (BDD). Here, code is written to achieve a specified behaviour from a users perspective. Developer-TDD is a more technical approach with smaller tests. These two appliances

can be combined, in fact, if you are to perform BDD, it is our opinion that you should also perform developer-TDD.

We have used developer-TDD in our work, motivated in two factors: First, simply to learn it. Second, to make sure that our system behaves the way we want. When tests run and check the behaviour for each few lines of code, it is easy to notify when an error is introduced, and it is satisfactory to have this insurance when writing new code or refactoring: the system still behaves the way we intended. This means that all of our production code is covered by tests, which reduces the risk of errors significantly.

The TDD approach is though impeded by our use of the external framework Colibri studio, which will be further described in section 3.7. While the use of frameworks have many advantages, there is an obvious disadvantage that one does not have full control over them, neither the full overview on how they work.

When we are building our system quite tightly integrated with Colibri studio, we encounter difficulties in some of our testing - it is challenging to write tests that are comprehensive and realistic enough to be useful, while still being short, readable and maintainable.

We have met this difficulty by emphasising that the code *we* write should be highly modular and easily testable, as we will amplify in section 5.2.

2.3.5 Tools and libraries

We have used a set of tools and external libraries during our work. As we are satisfied with all of them, we here briefly present the tools we have used, and a short justification of why.

Eclipse

Eclipse[7] is an integrated development environment mainly used by Java developers. Colibri studio is delivered inside Eclipse.

[7] `http://eclipse.org`

Infinitest

Infinitest[8] is a test-runner-plugin for Eclipse and IntelliJ[9]. Every time we save code, Infinitest runs all the tests associated with the code we change. This ensures us continuous testing, and makes it fast and easy to discover errors and mistakes.

Emma

Emma[10] is an Eclipse-plugin for checking test coverage - it computes how much of your code that are tested through your tests, both with regards to lines and branches. It also visualises very clearly what of your code is tested and what is not.

Google Guice

Google Guice[11] is a library for the dependency injection. It makes it easy to break dependencies between classes, and helps us loose up the coupling inside our system.

Mercurial

We have used Mercurial[12] as our version control system. Every time we have made major changes, we have saved them to Mercurial. Hence, the code is securily stored and easy to rollback to. We have used the plugin MercurialEclipse[13], that integrates Mercurial support in Eclipse.

[8] `http://infinitest.github.io/`
[9] Another IDE mainly intended for Java. `http://www.jetbrains.com/idea/`
[10] `http://emma.sourceforge.net/`
[11] `https://code.google.com/p/google-guice/`
[12] `http://mercurial.selenic.com/`
[13] `http://javaforge.com/project/HGE`

Chapter 3

Theory and background

3.1 Case-based reasoning

Case-based reasoning (CBR) is the response from artificial intelligence to how humans tend to reason[Kolodner, 1993]. In this situation we are in now, we choose what to do affected by what we did last time we experienced a similar situation, and what that action lead to. We scan our mind for similar episodes and remember those most similar. We use what we did then when deciding what to do now. We note the differences from the previous episode to this one, and consider what effects these make to the solution - we adjust the previous solution and then concludes on our solution this time. We will remember this solution the next time we are in a similar situation.

A *case* is in our sense a particular problem situation. This case has a description and a solution, and possibly a justification and a result as well. The description is usually a collection of findings, also called features. Cases can in certain systems have other properties in addition to those, but every case has at least a description and a solution.

CBR was a major newcoming in the AI society as most AI systems depend entirely on generalised or general knowledge. In CBR, much of the knowledge is extracted from specific, concrete situations - cases[Aamodt and Plaza, 1994]. CBR is suitable for continuous learning, as this is exactly what happens when new cases occur.

Standard CBR follows the same type of reasoning. Aamodt and Plaza[Aamodt and Plaza, 1994] defines what has now become the de facto standard way of describing CBR at a high-level, rendered in figure 3.1. It defines a CBR

cycle constructed from “the four RE’s”: Retrieve, reuse, revise, retain.

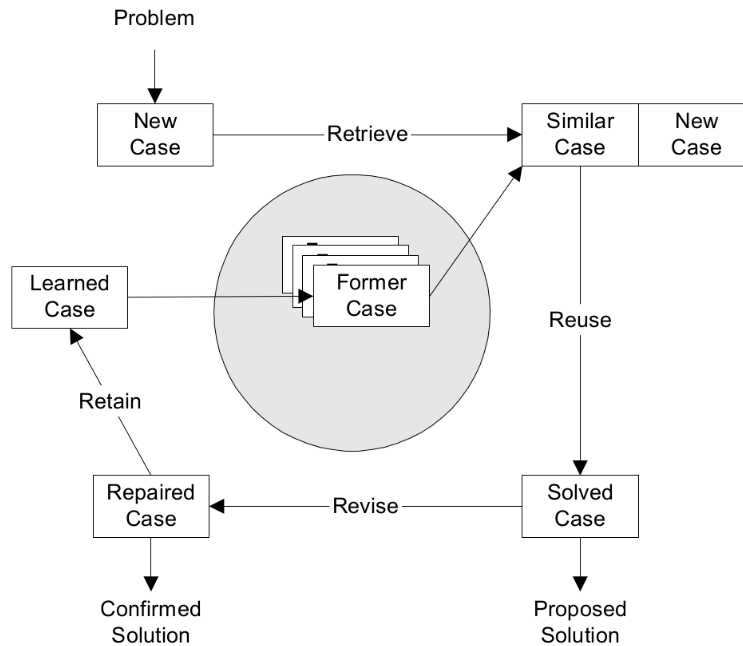

Figure 3.1: **The CBR cycle**, adapted from [Aamodt and Plaza, 1994]

Retrieve searches the case base, and returns the case or cases most appropriate to the new situation. Often, these cases are described as the most similar ones, which is often, but not necessarily, true[Leake, 1996]. The cases should be those which can prove most helpful in finding a good solution to the incoming case. This can of course be the most similar ones, but it can as well be more different cases that are easy to adapt or that provide better explanations. In practice, we have the impression that most systems stick to the *most similar* heuristics.

How many cases retrieve should extract varies from system to system. In many systems, it is perfectly useful with one case, while others require several. This is influenced by among others how much the system emphasises explanations, how complex the solutions suggested by the systems are and how complex the case descriptions are.

Reuse uses information from the cases found in retrieve to solve the current problem. This phase transforms the *previous* solutions from retrieve into a *current* solution to our problem. This can be done by simply copying the solution from the most relevant case, or it can be adapted.

Revise can be seen as the evaluation step of the CBR cycle. In this phase, the solution proposed by reuse is evaluated and tested. If it is successful, then it is passed on to retain, otherwise it is repaired through the use of domain-specific knowledge. In many system, revise is left as an exercise to the user.

Retain could also be named *learning*. If the case is sufficiently different from the most relevant previous case, either the new case is stored in the case base or a previous case is updated with the new information from this case. The system learns from this new situation.

Several CBR types exist, which emphasise different qualifications: textual CBR, CBR for recommendation systems, CBR for education, for planning and so on. CBR systems can be distinguished along several axes:

- How much communication is required between the system and the user?
- How much emphasis is put on explanations?
- How much knowledge beside the cases is contained in the system?

3.2 Knowledge-intensive CBR

Our approach in the previous section to how humans reason was correct, but misguiding. It is definitely so that we reason with the experiences from previous cases, but we also make our decisions based on how we think the world is linked together. Our understanding of the world uses to interplay with the experiences we have made in similar previous situations.

Most humans share a common "base" of knowledge about the world which is rather domain-independent. *A woman is a person*, *a person is a mammal* or *a tree is a part of a forest* are typical examples of this type of knowledge. This knowledge is seldom used directly when we solve problems, but is always there at the back of our heads when we reason. Every now and then such relations and general knowledge turns out useful when we are adapting or parsing solutions as well.

The other part of this picture is the specific things we know about the domain in question. The more we know about the domain, the higher the chance that we can use what we know during problem solving.

CBR systems solve many different problems. For some problems, including much case-independent knowledge would be overkill. Other problems can be

solved remarkably better, faster or easier through the use of extra knowledge. A rule of thumb is that the value of extra knowledge correlates with complexity.

It is worth noting here that every system contain knowledge about the domain, this is used in most similarity measures. Also cases contain domain knowledge, so to be specific: when we are talking about general knowledge, we mean *generalised* and *explicitly* represented such.

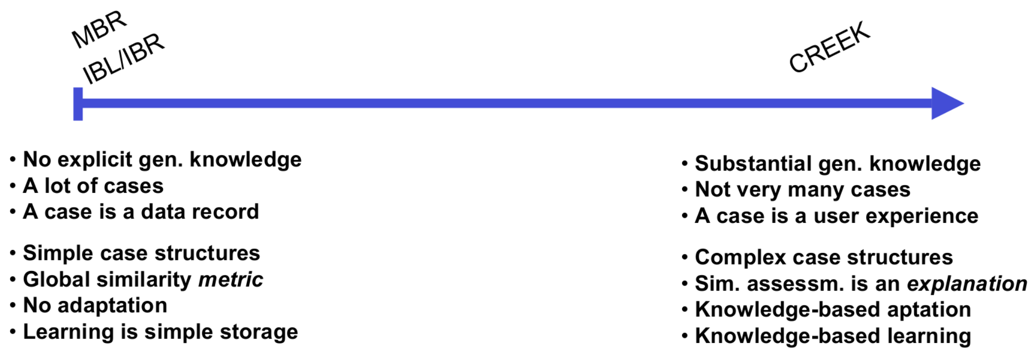

Figure 3.2: **The knowledge-intensiveness-scale**, adapted from [Aamodt, 2004]

Figure 3.2 denotes two extremes in the scale of knowledge-intensiveness (instance-based reasoning on the less-side and Creek on the much-side), and lists characteristics of both. The figure illustrates that it is a scale between the two extremes, which is an important aspect to bear in mind.

A benefit in making knowledge-poor systems, is that one avoids the knowledge acquisition bottleneck of encoding, formulating and representing the general domain knowledge. Opheim[Opheim, 2012] elaborates this benefit further. The rule of thumb that says that *the less complexity, the less risk of making errors* is also a good argument that no more general knowledge than necessary to solve the problem should be added.

Some claim that “The more knowledge is embedded into the system, the more effective is expected to be”[Díaz-Agudo et al., 2000][p.1]. We are not at all sure that this is true, but we recognise that it for many systems is true up to a certain degree. Smyth and Cunningham[Smyth and Cunningham, 1996] promotes heavy-weight arguments against this theory, and states that after the casebase has reached a *saturation* a point, the efficiency decreases rather than increases when new cases are added. This is a long discussion in the discipline, called the *utility problem* or *swamping problem*[Francis and

Ram, 1993], but the underlying premise that always holds is that one should have a conscious thinking on what knowledge to embed into the system and how this is done.

Case-independent knowledge allows to build more intricate and advanced systems and similarity mechanisms. Through the use of it, CBR systems can compare cases and findings with regards to both pragmatics and semantics, not just syntactically[Aamodt, 1994a, 2004]. It can also be directly used for adapting a previous case. When figuring out what of the new case to learn and retain, it can be used[Aamodt and Langseth, 1998; Aamodt and Skalle, 2004].

To reason with this non-case-knowledge is really a kind of model-based reasoning (MBR), thus not strictly case-based reasoning. This means that doing CBR with the support of external knowledge in fact is a combination of CBR and MBR. On the other hand, one can say that a CBR without any external knowledge in fact is instance-based reasoning (IBR)[Aha et al., 1991]. It is important to keep in mind that knowledge can also be incorporated in the structure of the cases, and thus that a CBR system with only cases can still be knowledge-intensive.

The conclusion we draw from this is that there are no watertight bulkheads between the different types of reasoning, but that they overlap with each other. Recio-García et al.[Recio-García et al., 2014] pinpoints that knowledge-intensive CBR is at its best when few cases are available, but much knowledge about the domain exists.

3.3 Knowledge representation

How to represent knowledge is a challenging area for designing artificial intelligent systems. How can the ideas and thoughts we have for our system be transformed into something a computer can understand? What will be the benefits and disadvantages of choosing one representation rather than another? These are among the questions the system designer needs to address before deciding how to represent the knowledge.

How we represent knowledge affects other parts of the system as well. Davis et al.[Davis et al., 1993][p.17] claims that knowledge representation in fact is an answer to the question “in what terms should I think about the world?” Language influences the way humans think[Boroditsky, 2011], and this of course also affects when the language in question is a programming language.

Davis et al.[Davis et al., 1993] also states that this is one of five roles a knowledge representation plays. It is also a theory about reasoning. How we model and represent also affects what can infer and how and about what we can reason. This statement is closely related to the previous one, and with the next, whichh states that knowledge representation is useful for computation.

In addition, a knowledge representation is a surrogate for what it is a model for. It is also a way for humans to describe the world, and it is through the use of it we tell the computer how the world is. There is an obvious interplay between these two roles.

The idea of case-based reasoning originate from Roger Schank and his work with dynamic memory and memory organisation packages[Schank, 1982], structures that have much in common with Minsky's frames[Minsky, 1975].

Over the years, frames has become the industry standard for closed-world-systems, and description logics for open-world-systems. A discussion of the background for this is beyond our scope, but we believe that this is mostly because of the benefits we describe in the upcoming sections.

3.3.1 Domains

The world can be divided into different domains along many axes. In the CBR context, two of them are important: weak vs strong and open vs closed.

In a strong domain, relations are well-defined and polite and there is little to no uncertainty. The weaker the domain, the more uncertainty. In weak domains, communication with the outer world is necessary.

When a domain is closed, we assume that all relations and all facts of the domain are encoded. This is not the case in an open domain. Aamodt[Aamodt, 1994a] says that "An open domain is a domain which cannot be realistically modelled unless the problem solver's relationships with the external, changing world are anticipated by the model." In other words: we realise that we can hardly model the entire domain precisely. Hence, we define that all that we can consider true or false is stated in the knowledge base, either directly or through a series of statements. Anything not included in this do we not know whether is false or true.

This is called the *open world assumption*, and is followed by any open domain. A closed domain will adhere to the *closed world assumption*, where everything not stated in the knowledge based is considered *false*. As a consequence, one

cannot easily transform between open and closed domains, so the decision on which assumption to stick to should be taken consciously.

While doing reasoning, which of these domain characteristics present will affect and be constraints for the reasoning. In a weak and open domain, inference and deductive proofs are hardly the way to go, as the domain may change or new information is discovered. Hence, instead of the true-false dichotomy, statements are more or less probable or more or less supported[Aamodt, 1994a]. Consequently, open domains need more general knowledge than the closed ones. On the other extreme: If we have a strong, closed domain with enough cases, general knowledge would turn out excessive[Aamodt, 1994a].

Certain techniques are appropriate for certain types of domains. CBR can be used for all types of domains, but seems most useful for open and weak theory domains. For closed and strong domains, every possible reasoning can be represented directly without the uncertainty support or the support for new situations that CBR can give.

3.3.2 Frames

A frame, as introduced by Marvin Minsky[Minsky, 1975], is a datastructure for representing a concept or a prototypical situation. Each frame contains information of several kinds: what is this, how can I use it, what can I expect to happen next, what do I do if these expectations are not met and possibly also other types of information.

We can see the frame as a network, consisting of nodes and relations. This network has two parts, where the top part consists of the statements that are common for every instance of the frame. The lower part are for the values that vary between instances, each of these values fill an acquired "slot". The slot can have constraints the assigned value must meet. The value can be a terminal value or it can be a subframe.

A benefit with frames is that it defaults with inheritage, so that subframes inherit properties from their superframes.

Collections of related frames are put together in *frame systems*. Central for frames from the start on was that they should be flexible and non-formal[Nebel, 1999].

The structure of a frame is formed by the domain it represents, and is typically close to the way domain experts think about the domain. Constraints

for allowed values can be domain-defined. A constraint is defined by a so-called *facet*.

Description logics (DL) grew out of the work of concretising and implementing frames[Nebel, 1999], as the common perception of frames had deviated away from Minsky's original version. As semantics were more and more specified, central issues such as default inheritance, the procedural semantics and the flexibility started to fade out. Hence, frames changed into a structure with declarative, logic-based semantics, without default inheritance, strictly defined form and where statements are either true or false - corresponding with the closed world assumption. Thus the assumption seen for instance in [Wang et al., 2006], that frames is only suitable for closed domains.

Especially the need for a precise and defined semantics was a central motivation for introducing DL.

3.3.3 Description logics

Description logics are made up of two types of building blocks: terminological and assertional statements, denoted as TBox and ABox. The TBox is the terminology or the vocabulary of the domain, and defines concepts and roles, such as *Mother is Parent and Woman*, and *hasParent(Child,Parent)*. ABox has statements about individuals, such as *Woman(Anne)* and *hasParent(Anne,Nils)*[Baader and Nutt, 2003].

This way, a strictly defined semantics is defined and reasoning can be done through a specified set of rules.

We bear in mind that DLs are not a single language or logic, but a family of different logics sharing a central set of characteristics and differing on other aspects.

A central concept in DL is *satisfiability* - in other words: whether or not there exists a possible model with all the defined statements that is consistent. A DL reasoner does this check and looks for one such model, while the frame-reasoner[1] checks if the one given model is consistent. Another difference between a frame- and a DL-reasoner is that the DL-reasoner itself can determine subclasses, while these have to be explicitly defined in frames. DL can also determine whether an instance belongs to a class: it does so if it satisfies all the class' necessary and sufficient properties. This is not the case

[1] In this entire paragraph, when saying frames, we mean the deviated, non-Minsky sense of frames

with frames. An instance in DL can belong to several classes, while it belongs to exactly one in frames[Lillehaug, 2011]. An example of this could be a new mobile phone, which in DL can belong to both the classes MobilePhone and SmartPhone, while in frames, it can be a member of only one of them.

Another difference between frames and DL is important to bear in mind: with frames, you cannot add something to a frame unless there is an explicitly reserved place for it in its template, while in DL, you can add it as long as it does not introduce a contradiction in terms[Wang et al., 2006].

OWL

The most common description logics representation used is Web Ontology Language (OWL)[W3C, 2012b]. The recent version of OWL, OWL 2, exists in three different profiles, OWL 2 EL, QL and RL[W3C, 2012c], each of which is intended for different scenarios. OWL 2 EL is the standard profile suitable for most problems, while OWL 2 QL is excellent for dealing with SQL databases and lightweight ontologies. OWL 2 RL is the best for operating on data in RDF triples.

Another term frequently used when speaking of OWL nowadays is *OWL 2 DL*. This is a term describing ontologies that satisfy syntactic conditions provided in [W3C, 2012a]. The other type of semantics available is RDF-based semantics, which semantics are linked directly to RDF graphs. These are informally named *OWL 2 Full ontologies*.

The OWL 2 DL ontologies are, as implied by their names, adhering to description logics semantics.

Lillehaug[Lillehaug, 2011] and we share the feeling that the CBR community is moving from frames to using OWL. The emergence of the semantic web, along with OWL's clear specification and formal semantics are significant parts of the explantation for this change.

3.3.4 Ontologies

An ontology in our sense is a data model for what entities a domain consists of, and how they are related[Gruber, 1993]. The data model has a formally defined meaning and is processable by a machine[Hitzler et al., 2011]. This way, ontologies make assumptions about the domain explicit, and allow for

unambiguous communication between different systems. The intention behind them is to provide a shared and common understanding of the domain, which can be communicated between systems and humans[Fensel, 2001].

The ontology can list both the abstract and the concrete entities - both the concept *person* and the person *Marie* are plausible parts of a person ontology. Also attributes and relations are included: the ontology states that a typical person has two legs, and that Marie is a woman. It also states that woman is a subclass of person. Ontologies can describe one or more domains, and several ontologies can be merged or imported and then used together.[2]

Different ontologies can have different purposes. The most noticeable ontology types are for the representation of knowledge about domain, metadata, generic, representational or method-and-task[Fensel, 2001].

Some ontologies do not include individuals, these are typically intended to be reused across applications. A typical example is the SSN ontology[3], which is an ontology for sensors. If we are to use this ontology to model sensors, we typically import the SSN ontology and supply it with our own individuals.

3.3.5 Semantic networks

Semantic networks are among the predecessors of description logics, and a semantical network is a typical way of representing knowledge.

A semantic network represents knowledge as a graph, and is put together of nodes and links. Each node is a conceptual unit, and a link is the relation between two units[Lehmann, 1992].

Each node and each link has a meaning, which is the reason for the *semantic* part of the name. This is illustrated in figure 3.3. Note the spectrum of relations and the different meanings a relation in such a network can have. Other properties than this label can also be put on a link, as we will see examples of when we discuss this aspect of Creek.

Most of the advantages of using semantic networks comes through reasoning, as the reasoning can make use of knowledge not directly represented. For example, if *a leads to b*, and *b leads to c*, then it can be reasoned that *a leads to c*. This lessens the bottleneck of acquisition dramatically.

[2]Although this is not necessarily trivial.

[3]`http://www.w3.org/2005/Incubator/ssn/ssnx/ssn`

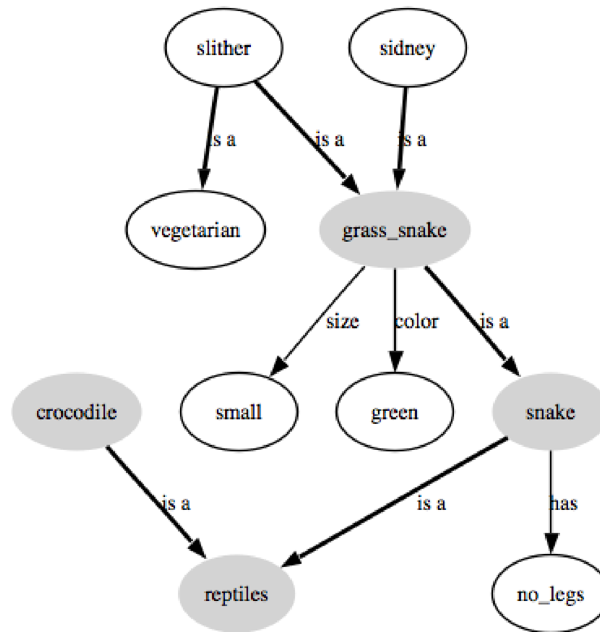

Figure 3.3: **An example of a semantic network**, adapted from [Lalanne, 2008]

Semantic networks are traditionally tightly linked with frames, and it is common to speak about those in the same breath. A prototypical example of this is seen in [Baader et al., 2005][p.230], "DLs differ from their predecessors, such as semantic networks and frames, in that they are equipped with a formal, logic-based semantics."

DL as a semantic network

We find it important to state that description logics and semantic networks are a very feasible combination. Sowa[Sowa, 2013] points out that the semantic web is in fact a giant semantic network, and the semantic web is represented with description logics.

Semantic networks are essential to Creek, and if we were to exchange them with another structure, large efforts would have to be made. We mentioned earlier in this section the strong relationships traditionally seen between such networks and frames. This can be confusing and obfuscating, and it could be easy to think that this would make it hard to combine semantic networks and description logics. We were ourselves unsure on this right until we started

thinking of how a description logics representation really is.

DL represents knowledge as nodes and relations, and when putting them together, we acquire a network where the nodes and links themselves carry meaning - *a semantic network*. Figure 3.4 illustrates this: it represents the SSN ontology[4], and each node is a class in the ontology. Each link is a relation between two classes, and different types of relations are marked by different arrows. For instance, subclass-of is marked by solid purple line, has-property by a dotted purple line and has-input dotted turquiose line.

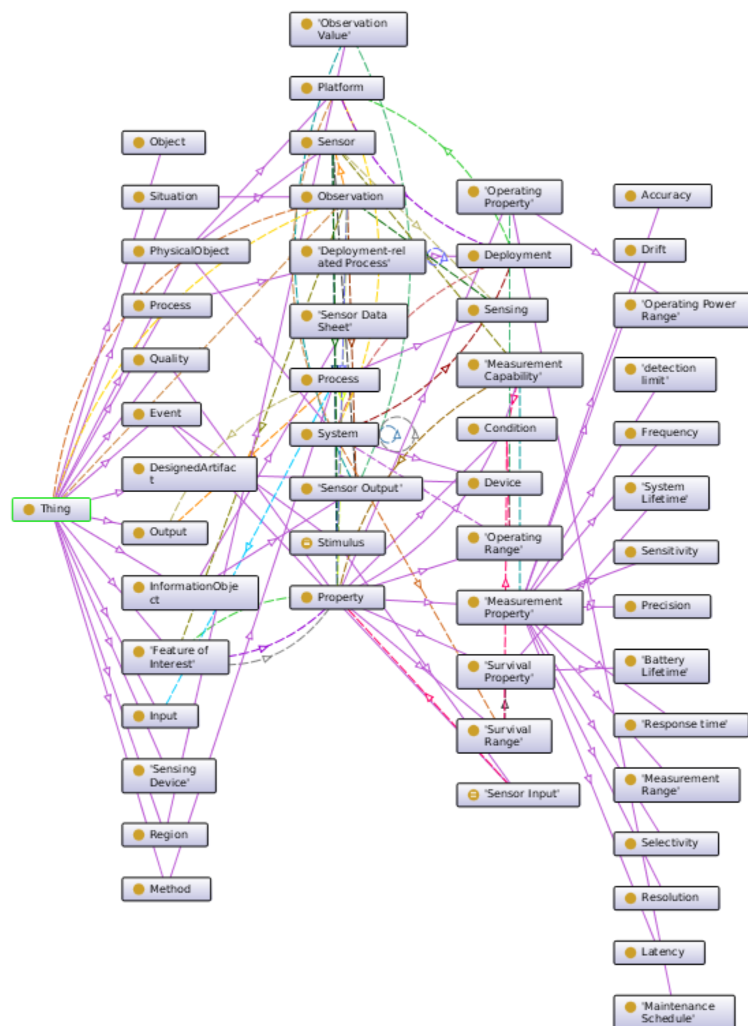

Figure 3.4: **The SSN ontology, visualised in Protege**

[4] http://www.w3.org/2005/Incubator/ssn/ssnx/ssn

3.4 Creek

Opheim[Opheim, 2012] described Creek as follows: "Creek is a CBR system proposed in [Aamodt, 1991] making excessive use of general knowledge, as indicated by its name. Creek is primarily intended for solving diagnosis and repair problems, in the broad sense of these terms: not knowing what to make for dinner fits into this classification. It is a mixed-initiative system: it asks the user when the system is itself not capable of providing plausible enough explanations. Thus, we will define Creek as an expert system, since the system assumes that the user knows the answer when itself lacks some information or is unable to do the appropriate reasoning.[5] Creek fits into a sub-group of expert systems called *decision support systems* – the interaction with the user is dynamic and the focus is on finding a satisfactory solution the user will agree to and understand. We will give an intro to the Creek system, with more extensive focus on the parts distinguishing it from other CBR systems, and focus on the explanatory consequences of those.[6]" Note that when the term *expert system* is used here, it is in the meaning *targeted at expert users*.

In this thesis, we use *Creek* as a term for the method framework. Readers familiar with the implementation history of the framework will recall that a subset of this framework has been previously implemented. This implementation was made by researchers at NTNU in cooperation with the company Trollhetta AS, and named *TrollCreek*. Insight in TrollCreek can be obtained from Brede et al.[Brede et al., 2006]. TrollCreek formed the basis for DrillEdge, as shown in section 1.1.

TrollCreek evolved and was eventually adopted by the company Verdande Technology. Verdande Technology has developed upon TrollCreek to what is now the DrillEdge system.[7]

[5] Although one may argue that a good-enough trained and elaborated Creek system eventually will contain so much knowledge that it no longer needs this type of feedback. We consider this though a very hypothetic situation which will hardly occur in real use. Hence, Creek is in our opinion an expert system.

[6] The Creek system is primarily described in [Aamodt, 1994a] and [Aamodt, 2004], and more detailed in [Aamodt, 1991]. These may be considered the sources for this entire section.

[7] `http://www.verdandetechnology.com/energy/products-and-services/drilledge-software/`

3.4.1 The CBR cycle in Creek

This section is in its whole taken as it was from the work we did in [Opheim, 2012].

Each[8] of the steps in the CBR cycle is in Creek decomposed into three substeps: activate, explain and focus. Creek uses the same basic tactic for each of the three steps retrieve, reuse and retain, as visualised in figure 3.5.

Figure 3.5: **Activate, explain, focus in each CBR step**, adapted from [Aamodt, 2004]

Activate takes as input the input data (findings, also known as features), task, goal and constraints for the solution, and returns everything (every concept) considered relevant for solving the task. It then does two separate, but similar searches in the semantic network: one finds the relevant concepts through spreading activation [see Lehmann, 1992] along the different relations. This starts from both the start (initial findings) and the end (initial goal), and each creates a so-called sphere through the spreading. Easily said, every node in both of these spheres, as well as every node on the path between the initial findings and the initial goal, is activated. The other uses the input findings or the findings from the activation-spreading to find potentially relevant former cases. The relations along which activation spreads, can be of several types: some are generic, while others may be specified during the modelling process. Examples of generic relations are taxonomic types such as sublass-of and instance-of, causal and associational such as correlates-with.

Explain takes the activation-output as its input, and its output is a set of possible solutions with associated explanations. It follows the activated paths that have the strongest cumulated strength, using the default relation explanatory strength described above. It is worth noting that these strengths may vary with different contexts, a relation's explanation may be more useful or valuable in one situation than another.

[8] Well, as Aamodt[Aamodt, 1994a] points out, it makes sense to view revise as a non-existing step in Creek, as it is not handled by the system at all.

The explain step has a set of defined evaluation strategies in order to evaluate each explanation, but it might ask the user whether a given hypothesis is good enough or not, in the situations where the system cannot determine this itself. Obviously, explain returns only hypotheses that are better than a specific threshold.

Focus takes this set with explanation-attached solution hypotheses as its input, and returns the best solution. It handles both provided constraints (e.g. resource demands of the solution) and priorities (e.g. try the cheapest solutions first) in its reasoning.

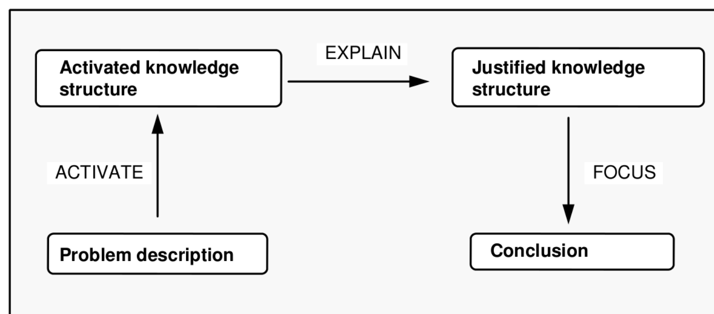

Figure 3.6: **Another view on the activate-explain-focus cycle**, adapted from [Aamodt, 1991]

Retrieve

In the retrieve step, activate will first spread as described in the general section, and consider a finding relevant for a case in the case base if the case is mentioned in the finding's *list of cases (with associated relevance value) this finding is relevant for.*

Then, explain will try to improve the match between the input case and those cases activated. For each activated case, it first tries to explain away the findings provided in only one of the retrieved cases[Aamodt, 1994a]. It finds paths between the unmatched findings of the input case and the activated, and finds the strength of this path. If there are several parallell paths, it sums them up. This way, the system determines how similar to syntactically different findings actually are. If a finding considered relevant is missing in the input case, Creek now asks the user, and adjusts case matching values according to the response.

Focus usually, but not necessarily, chooses the case with the highest explanatory value, or rejects them all. If the designer has provided the system with limitations (regarding for example resource demands for specific solutions in the real world), Creek is able to take this into consideration.

Reuse

In the reuse process, activation starts from the solution case provided from retrieve, and spreads out to all expected findings from this case, as well as the "risky consequences"[Aamodt, 1994a] of this solution. Explain consideres whether every expected finding either is confirmed or considered irrelevant, if not, it tries to modify the solution. If explain is missing some information about a finding, it asks the user. This may be about both relevance and requesting more information. When this information is general (domain) knowledge rather than case-specific, the explanation mechanism checks whether the newly added information contradicts with existing knowledge, if so, it demands an explanation before the information is added. It will also identify ambiguities or mistakes[Aamodt, 1991].

Focus checks whether the proposed solution fits with the provided constraints, and then presents it to the user for confirmation. The solution along with an explanation is provided to the last step, retain.

Retain

Activate finds the parts of the new case and explanation that may be worth saving, as well as updated or new information on other concepts obtained from the user during the retrieval and reuse processes. This information may consider cases or the general knowledge, and is thus reducing the problem of the knowledge acquisition bottleneck.

Explain decides whether this case is to be saved as a new case or not. This is only necessary if the case is substantially different from every case in the case base. Regardless of this result, explain will find *what* information to save – the relevant findings and the strongest explanation. If there is not one explanation considered strong enough, several parallell explanations will be retained. If the system considered the case special enough to be stored as a new case, it computes how predictive and important each of the finding is for the case, thus finding the relevance factors.

Focus makes a case frame for those structures that are to be saved, and saves them as well as their indices. It also updates the above-mentioned relevant-cases-list for each affected finding.

3.4.2 Knowledge representation in Creek

We have earlier in this thesis demonstrated how important general knowledge is in Creek. This knowledge is represented by four modules, as displayed in figure 3.7.

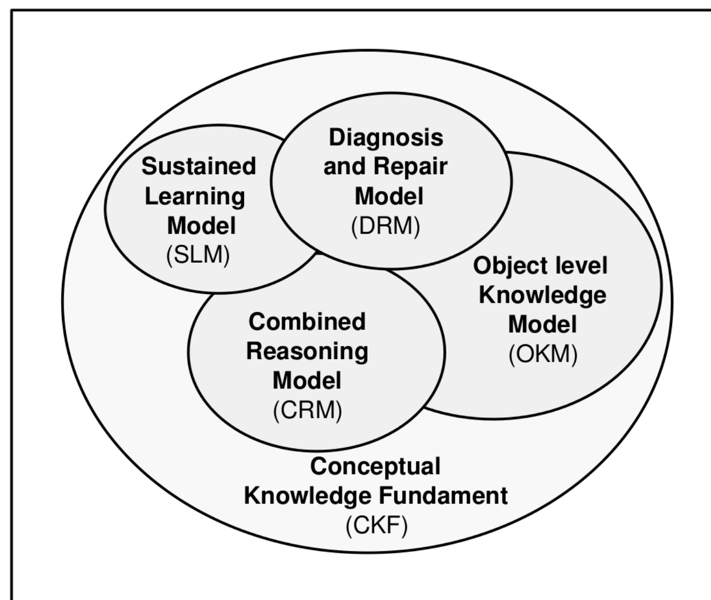

Figure 3.7: **Creek's four knowledge modules**, adapted from [Aamodt, 1991]

- The object-level knowledge model (OKM) - this means concepts, relations, object-level rules and solved cases
- A model for diagnosis and repair (DRM) - when to look for more info, when to apply which domain knowledge et cetera
- A reasoning meta-level model for choosing which reasoner to be used (Combined reasoning model, CRM)
- A reasoning meta-level model for knowledge-supported case based learning, and rules to direct the process for matching and generalising (Sustained learning model, SLM)

We emphasise that the usefulness of this four-part-structure depends on several interesting premises. There must be several reasoners available in the system for the CRM to be useful, which has not been the case in TrollCreek, for instance.

Creek represents every piece of knowledge in the same structure and using the same type of representation - regardless of whether the piece is a case, a finding in the case or a part of the specific or general knowledge. Note the division between specific and general knowledge: Creek has a high-level set of generic concepts that are generic and domain-independent, and is a highly knowledge-intensive system, as illustrated in figure 3.2. These concepts are for instance *thing*, *symbol*, *concept* and *relation*. This type of concepts are supplemented by domain-specific concepts. If our domain of interest is travel, concepts as *hotel*, *airport* and *vacation-length* are typical concepts here. How the knowledge fits together is shown in figure 3.8.

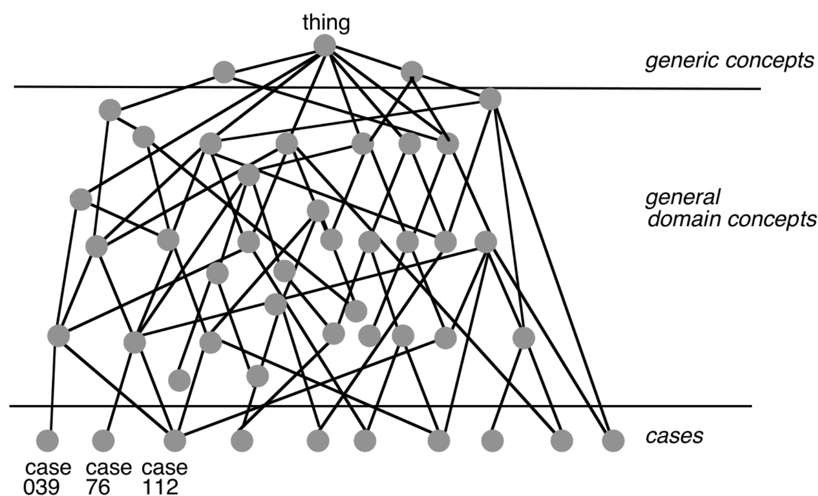

Figure 3.8: **Creek's knowledge hierarchy**, adapted from [Aamodt, 2004]

Creek's knowledge is represented in a semantic network where every node represents an entity in the OKM and every link is a relation, as described in section 3.3.5.

Creek contains many different types of relations, some are generic and some are domain-specific. Generic relations include subclass-of, part-of, member-of, instance-of and caused-by. Every relation in Creek has its inverse, so when you declare that A is a subclass of B, you also define that B is a superclass of A.

Every node in the semantic network is a frame. Here, we must note that a relation type is also a node in the network, as the semantics of a relation is also a part of the domain model and should be modelled accordingly.

3.4.3 Creek and frames

Creek uses frames, with the justification of being flexible, expressive and transparent[Aamodt, 1991]. It allows for many different types of relations and is in general very flexible, although somewhat restricted in order to still be able to infer. This trade-off was one of the crucial problems with frames, as we discussed in section 3.3.2. The frames used in Creek are of the Minsky type, and hence, Creek sticks to the open world assumption. This fits well with the intention of the system: to reason with open, weak domains.

Creek's frames are represented in a system-specific frame language named CreekL, where every concept is a frame, containing slots, facets, value expressions and value fields. Each relation type is specified in the network, and each relation is symmetric. Creek does inference with several techniques, with the basic one being default inheritance[Aamodt, 1990].The semantics are mostly clearly defined, but with a trade-off between flexibility and ability to infer[9].

However, problems with the use of frames as the knowledge representation suddenly occured. As frames deliberately is structured as the domain, there is hardly any common ground for reusing frames or easily implementing other systems within Creek. The semantics are also left to the developer to establish, as it is a procedural semantics. An attempt to solve these problems is seen in [Aamodt and Langseth, 1998], where bayesian networks are introduced within Creek.

3.4.4 Creek and reasoning

Creek is, as can be assumed by its name, mainly a CBR system. However, the Creek architecture and also some work on the system introduces the support for model-based reasoning (MBR) and rule-based reasoning (RBR)[Aamodt, 1991; Skalle et al., 2013b; Shokouhi et al., 2009].

Original Creek uses RBR and possibly MBR as fall-back reasoners when CBR fails - that is, when the CBR reasoner is unable to propose a sufficient suitable

[9] For the entire justification for the trade-off, see [Aamodt, 1991][p.148].

solution[Aamodt, 1991]. Typical occasions when this happens is when the case base contains few or non-covering cases, the adaptation knowledge is imperfect or the new case is very special. Later work has attempted to combine MBR and CBR[Shokouhi et al., 2009][10].

Creek starts out by trying CBR - if the system fails to find an acceptable solution, it tries to adapt or modify it to become good enough. If the solution still not is acceptable, Creek resigns from solving this situation with CBR and tries RBR instead. If RBR can not find a good enough solution neither, pure MBR is tried. This reasoning-choosing process is seen in more detail in figure 3.9.

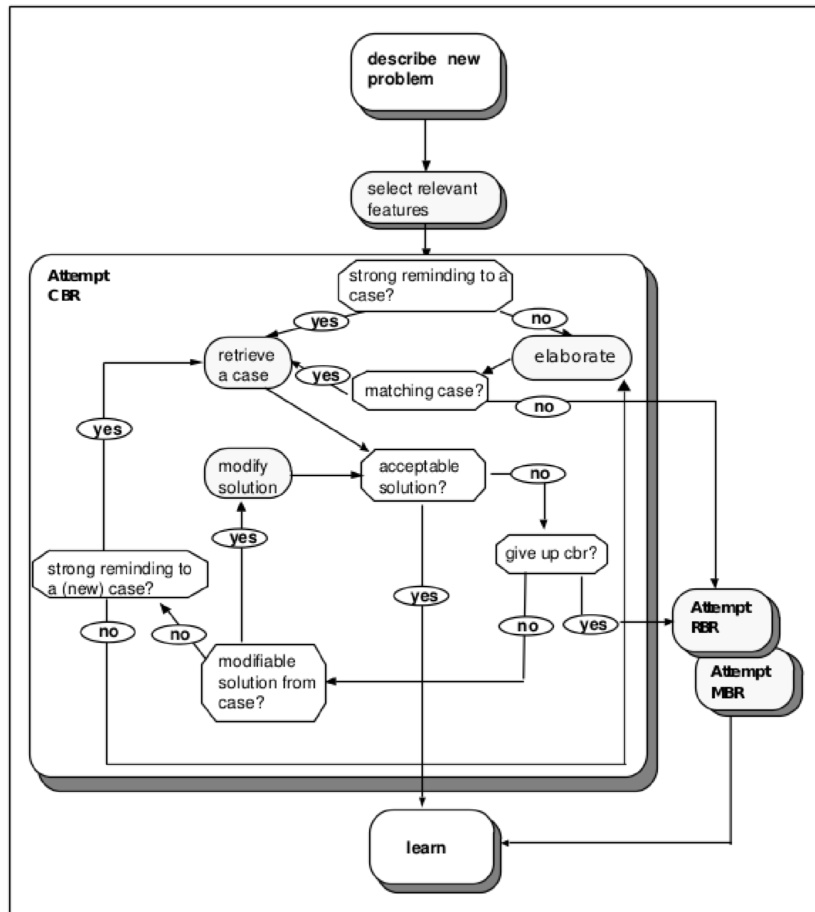

Figure 3.9: **The reasoner-choosing process in Creek**, adapted from [Aamodt, 1991]

[10]An interesting discussion is whether knowledge-intensive CBR with much general knowledge in fact is MBR combined with CBR.

3.5 CBR frameworks of today

First step towards achieving our goal of studying how the core of Creek can be adapted to a description logics destination, is to define where to start. We intend on implementing this core, and will have to define where we start our implementation. Should we start from scratch, modify an existing system or start from an existing framework?

The answer to this question is not obvious up-front. However, we have earlier done a study on this[Opheim, 2012], and considered especially the frameworks or systems INRECA[Bergmann et al., 1998], CAT-CBR[Abásolo et al., 2002], IUCBRF[Bogaerts and Leake, 2005], CBR*Tools[Jaczynski and Trousse, 1998], Colibri studio and myCBR. Since then, we have become aware of eXiT*CBR, a distributed CBR tool for the healthcare domain[Pla et al., 2013]. eXiT*CBR seems promising and includes a lot of promising techniques, we are especially curious on their approaches to multi-agent systems and distributed CBR. However, eXiT*CBR is deliberately domain-specific, and it seems also technically directed on other types of problems than we intend to solve.

We concluded in our previous study that Colibri studio and myCBR are two very promising frameworks for developing a Creek implementation within, and we have not seen any indications of other frameworks or systems more suitable. Hence, we will here have a closer look at these two.

3.6 myCBR

myCBR is a software development kit and a tool for similarity-based retrieval[Stahl and Roth-Berghofer, 2008]. This classification clarifies where the focus of the framework lies, it has a heavy focus on the similarity measures.

myCBR was originally implemented as a Protégé[11] plugin, but the team has later on developed a stand-alone implementation, and this is being steadily phased in, while the plugin is no longer supported[Bach and Althoff, 2012].

The motivation leading to myCBR was the lack of a way to easily build fast CBR applications, with a special emphasis on teaching and tutoring, research and small projects for business use[Stahl and Roth-Berghofer, 2008].

[11] An ontology editor, available from `http://protege.stanford.edu/`

The framework is also well suited to be combined with the usage of other frameworks, as data can be easily imported to and exported from myCBR. This is especially worth noting for Colibri studio, as it is easy to see these two frameworks as competitors for market share. This is true, but not the entire truth: the systems have different strengths and different areas of focus. myCBR is particularly directed at the knowledge modeling phase, while Colibri easily said is intended for the rest of the development process.

myCBR can perfectly fine be used in combination with Colibri studio. The developer teams behind the two systems presents how in Roth-Berghofer et al.[Roth-Berghofer et al., 2012], where they both provide the motivation for doing so and how it can be done.

3.7 Colibri

Colibri Studio is a platform for creating all types of CBR systems[Recio-García et al., 2014]. Colibri studio is the most recent version in a series of different Colibri versions; the platform has in previous versions been named both COLIBRI and jColibri[Recio-Garcıa et al., 2006]. Colibri studio is designed to be used by both developers and designers, and facilitates different approaches for those rather different groups of users.

The idea motivating Colibri is to share knowledge[Recio-García et al., 2013]. The more boilerplate code included in Colibri, the less effort is needed when someone wants to create, test or play with a CBR system. Examples of boilerplate code can be the engine to run the CBR cycle[12], the most common similarity mechanisms or the intermediaries between code and storage medium. All of these are needed in almost every CBR system, and having this code supplied with the framework saves remarkable amounts of time and - more difficult to measure, but not less important - motivation when systems are to be implemented. We note as a side observation that this idea is close to what we sketched as our motivation in section 1.1.1.

At the time of writing, the vast majority of the Colibri framework is written by the inner core of framework developers, the Group of Artificial Intelligence Applications (often abbreviated GAIA) at Universidad Compluense de Madrid, but some extensions created by other research groups are also made available. More external extensions are explicitly desired from the GAIA group[Recio-García et al., 2013], and the website is facilitated for it.

[12] As defined in [Aamodt and Plaza, 1994]

In this thesis, we consequently use the terms Colibri and Colibri studio: Colibri when we mention the framework, and Colibri studio when talking about the concrete implemented software. To be precise, Colibri studio is the implementation of the top layer, built on top of jColibri, which is the lower level[GAIA, 2014]. As Colibri studio is delivered as a coherent package, we have found that keep distinguishing between the terms Colibri studio and jColibri in this thesis would give more confusion than benefit.

3.7.1 The Colibri Studio architecture

Colibri studio is intended to be used by both developers and designers. For designers, the framework provides a black-box-architecture, while developers can make usage of a white-box-architecture[Díaz-Agudo et al., 2008].

Developers here essentially means programmers: the white-box-architecture has no graphical tools, but is intended for programming. Designers have a set of graphical tools making it possible to create CBR applications without writing many lines of code. In Colibri studio terminology, *the bottom layer* is for programmers and *the top layer* for designers, as graphically shown in figure 3.10. While the elements in each block complement each other, it is fully possible to leave unwanted parts out and replace with one's own desired parts, at least in the programmer's layer. For instance, we have replaced the similarity measurement from Colibri, as it hardly fits with Creek. We have chosen to use most of the remaining parts of this layer in our implementation.

The key idea behind this architecture is to divide the core from the user interface, after bad experiences on unintended mixing of these in the first implementation of Colibri[Díaz-Agudo et al., 2008].

Cases are represented as Java beans - that is, classes where every public attribute has a set and get method[Recio-Garcıa et al., 2006]. This leaves freedom to the developer to design cases the way most appropriate for the domain and problem.

Persistence is handled through the external library Hibernate, which supports both data bases and XML files, and is flexible and widely used. Here, it is worth noting that cases are stored differently for persistence than in-memory, while Hibernate handles the persistence, it is a choice for the designer in each system to handle the in-memory-storing. Colibri studio offers weak, wide boundaries for how the cases should be stored in-memory.

When Colibri is to run the CBR system, it does this through a three-step-

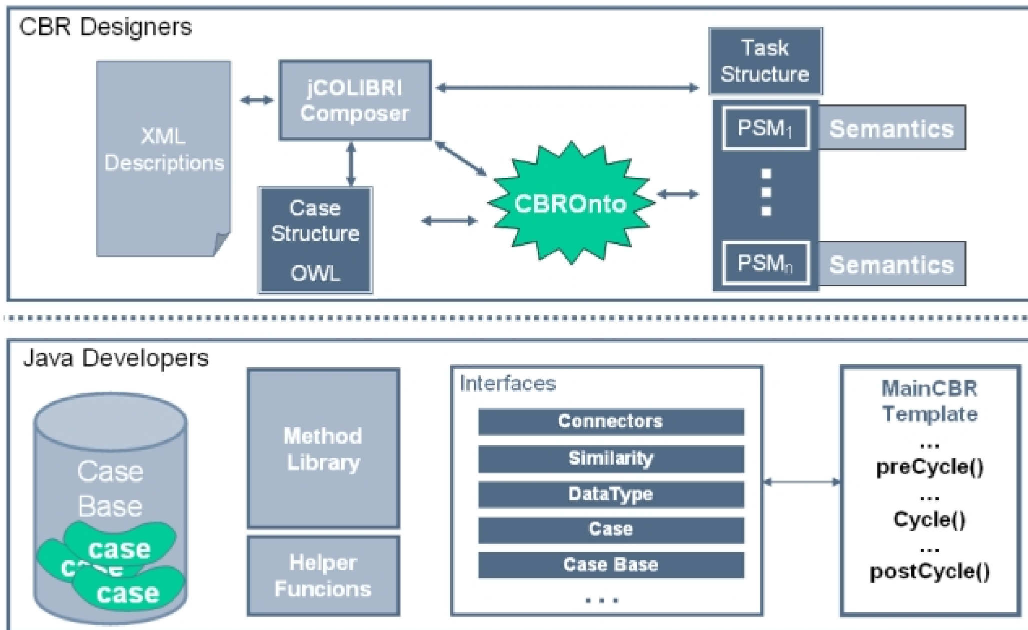

Figure 3.10: **An overview of the Colibri studio architecture**, adapted from [Díaz-Agudo et al., 2008]

mechanism: the case base and the knowledge base is loaded and set up in the *pre-cycle*. Then, the reasoning is done in the *cycle*. Whether the cycle step is to be done one or several times is a decision left to the implementer. Finally, the case base and knowledge base is closed and the system shuts down in the *post-cycle*. This is illustrated in figure 3.11, where "connector" is the class handling the knowledge base.

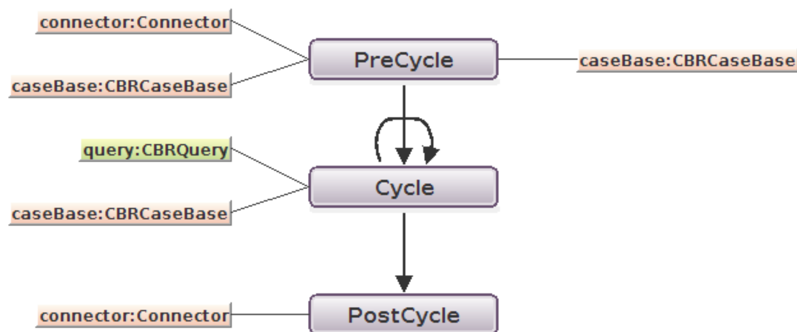

Figure 3.11: **The colibri engine on cycles**

3.7.2 Colibri and representation

Colibri allows the user to choose among three ways of storing cases: in a data base, in plain text or in an ontology[Recio-García et al., 2014]. For ontologies, they are handled through the ontology-managing library OntoBridge[13][Recio-García et al., 2014]. Colibri supports several techniques especially intended at knowledge-intensive CBR systems, though few of them are directly corresponding to any Creek technique.

Colibri supports ontologies in OWL[Recio-García and Díaz-Agudo, 2007]. When a developer specifies the structure of cases in a new CBR system, and decides that the system should use ontologies, the structure is automatically mapped into OWL and saved. This structure should of course correspond with the structure of the cases in the case base, and when this is set up, the developer no longer notices that she is handling description logics. In practice, this is an exaggeration, but it illustrates how well Colibri hides this layer from the developer.

3.8 The domain - turbines

3.8.1 Condition monitoring

Our research question 3 sounds "How can decision-support for reported turbine failures be implemented within a DL-based Creek system?" This will be a condition monitoring (CM) problem. Condition monitoring is the discipline of monitoring state of operational machinery whilst in action[SKF, 2014], or in other words, monitoring sensor values regarding this machinery, with the purpose of identifying possible errors or faults.

Condition monitoring makes planning, predictions and maintenance easier and can minimise the time and resources spent on repairing[Jardine et al., 2006]. It can make it possible to schedule repairs so that they can be executed when suitable - e.g. several at the same time[Gill, 2009].

Condition monitoring is highly linked with condition-based maintenance (CBM) - condition-based maintenance is significantly more demanding without the monitoring[Bengtsson et al., 2004]. CM makes CBM a realistic and useful strategy for performing cheap and on-demand maintenance.

[13] http://gaia.fdi.ucm.es/research/ontobridge

3.8.2 Our domain

We have a cooperation with the petroleum corporation ConocoPhillips Norway for this part, they provide us with example data and domain insight. Hence, we will use real world data for this part, handed by ConocoPhillips Norway's Onshore Reliability Centre (ORC). ORC is located in ConocoPhillips Norway's centre in Stavanger, and can be seen in figure 3.12.

Figure 3.12: **The onshore reliability centre**

We will look at gas turbines running offshore, providing electricity to the Ekofisk field and the surrounding, ConocoPhillips-driven fields. We will look at errors reported from these, and try to classify reported failures. Operators at ORC classify reported failures in one of four categories. The types are also seen in figure 3.13, and are as follows:

1. Sensor issue
2. Data issue
3. Software issue
4. Mechanical issue

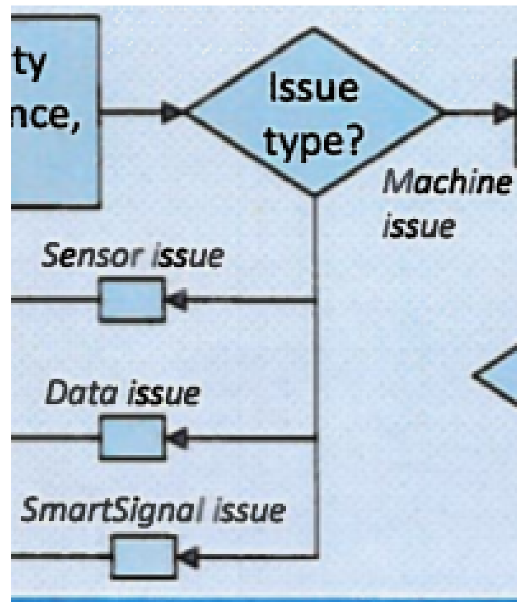

Figure 3.13: **The issue types**, adapted from a ConocoPhillips flow chart

Each issue type is handled in a specific way. The three first issue types are to be reported and fixed, but no further action is required from the ORC operators than logging and forwarding the error messages to the appropriate recipient.

Mechanical issues are more severe - they can make harm to the turbine if not handled properly. A such issue can in principle be anything from overheating to fire to a loose screw. This can be both expensive and harmful to the environment and workers, and is of huge importance to avoid. At the same time, it is regarded as really expensive to stop the turbine, as this will cause a production stop possibly lasting for days. ConocoPhillips estimate this to cost millions of dollars, and have structures and strategies to make as few production stops as possible while at the same time keeping safe and with everything in shape.

To illustrate: Skalle et al.[Skalle et al., 2013c] emphasise that oil and gas drilling generally costs a lot of money, and hence, every reduction of production stops or other cost-driving problem situations, is really valuable and very quickly gives a positive return on investment. They estimate the potential savings from a global big-scale propagation of DrillEdge to be more than $2.1 billion a year, given the premise that non-productive time can be reduced by 5%. We see from this that every decrease, even by a thousandth, of non-productive time is considered worth many millions. Replication of

sensors and other equipment is an important part of this work, as is prediction and the usage of condition monitoring as such. Examples we have seen from ConocoPhillips supports this emphasis.

The earlier an issue is discovered, the better. This makes it possible to schedule the repair to a suitable time, e.g. combined with other repairs, when a spare part is delivered or when the right engineer is at place.

Each turbine has tens of sensors attached to it, monitoring the status of the turbine and attached generators, compressors et cetera. A sensor can monitor for instance the temperature, the pressure or the vibration.

To discover what type of issue a new issue is, is relatively easy for a skilled and experienced operator: the three non-critical types are easy to spot:

- If it is a *sensor issue*, only one of several correlating, connected sensors start giving changed values, e.g. if only one of two duplicated vibration sensors change.
- If it is a *data issue*, floating numbers from the sensors suddenly all become integers.
- If it is a *software issue*, the results from one of the monitoring software programs start showing values not corresponding to the reality offshore, and different from the other programs.

This is an almost prototypical example of reasoning that is easy to do for humans, but not at all that easy for a computer. As case-based reasoning is heavily based on how humans reason, it is tempting to see the problem from a CBR angle.

Several subtasks can be extracted from this domain as well: to determine whether a given situation is an error situation or not, determine if it is a specific error type (e.g. mechanical) or to determine which type of error a given situation is.

Whether a situation is a potential mechanical error or not can be seen from the sensor values and how much and how they have changed from previous values. Trend considerations and history are important utilities here. A significant drop or rise is typically a warning light.

As we see, to distinguish a mechanical issue from a sensor issue involves regarding several values and the degree of correlation.

3.8.3 Use of CBR in condition monitoring

CBR has been used for a variety of condition-based maintenance or condition monitoring applications. Bengtsson et al.[Bengtsson et al., 2004] used CBR with success on gearboxes of industrial robots, a problem that tends to show by that the sound of the box differs from normal. Expert users with long experience are required to do this classification, so the domain is feasible for CBR[Bengtsson et al., 2004]. Yang et al.[Yang et al., 2004a] proposed a combination of CBR with neural networks for early fault diagnosis, with the example domain electric motors. Yang et al.[Yang et al., 2004b] proposed to combine CBR with Petri nets, showed on induction motors.

However, while our examples just given show that CBR is being used in condition-based maintenance, there are notably fewer applications of CBR for rotating equipment.

3.8.4 Existing CBR systems in oil and gas

The petroleum industry is an industry with a lot of resources and with a demand and desire for continuos improvement. Accordingly, there exist several CBR applications for the domain already. We have already discussed DrillEdge (in section 1.1), which is based on Creek. The initial Creek, as described in [Aamodt, 1991], also had oil well drilling as its example domain. WellCase[Mendes et al., 2003] is a system for petroleum well design, using CBR to guide designing new drilling wells. Genesis[Kravis and Irrgang, 2005] is also intended for the well design problem, and provides a very interesting approach to case adaptation: it adapts from several cases rather than just one, following the lines pointed out by Smyth et al.[Smyth et al., 2001].

Popa et al.[Popa et al., 2008] made a system for choosing which type of cleaning technique to use for sanded/seized well failures, which is a problem particularly similar to ours. We notice that their efforts seemed successful. Shokouhi et al.[Shokouhi et al., 2009] used CBR and the Creek framework on hole cleaning problems, with promising results on building a knowledge-intensive CBR system on this domain.

An extensive overview of CBR systems in the petroleum industry is given in [Skalle et al., 2013a], and in [Shokouhi et al., 2010].

3.9 Applications of Colibri

A list of systems making use of different versions of the Colibri frameworks can be found at http://gaia.fdi.ucm.es/research/colibri/people-users-apps.

Martín and León[Martín and León, 2010] used Colibri to create a system for search in a digital library, by using the ontology functionality of Colibri to provide a recommender system.

Mikos et al.[Mikos et al., 2011] applied Colibri on the thermoplastic injection molding process domain. They combined CBR with agents to do rapid problem solving within a manufacturing context, using a basic CBR implementation.

Water stress problems arisen within different regions can have important similarities, Kukuric et al.[Kukuric et al., 2008] built a CBR application using Colibri to provide comparison between such cases.

A combination of CBR and Bayesian networks is tried within [Bruland et al., 2010] and [Bruland et al., 2011]. It is implemented within Colibri and the BN-software Smile[14], and in the first version also with myCBR.

These few examples, chosen among many, show some of the breadth of applications implemented within the Colibri framework.

3.10 Existing combinations of elements of Colibri and Creek

We are not the first ones seeing the potential benefits of combining Colibri and Creek. As both are intended for knowledge-intensive CBR, directed at expert users, it is natural to explore whether they can be beneficially combined.

Fidjeland[Fidjeland, 2006] created an OWL ontology for Creek. He used the then-existing Creek implementation called jCreek, and made it capable of importing OWL ontologies into the Creek system as well as exporting them from the system. While some information got lost in the translation, the approach proved to be promising and for his usage sufficient. This approach

[14]Developed at the University of Pittsburgh, available from http://genie.sis.pitt.edu/

may very well have proven to be adequate, provided some more development on both his work and jCreek. However, translation and import/export is never as good as skipping this middle layer and encoding directly into OWL, so when jCreek no longer were distributed, also this work became excessive. His work on defining a meta ontology for Creek has though proven to be inspiring and useful for providing a future full-scale Creek version, and should be brought forward during that work.

Stiklestad[Stiklestad, 2007] did an effort with the goal of importing jColibri components into Creek. The work was done in close cooperation with Volve, now known as Verdande Technology. He also made a more general analysis of Creek and jColibri, and looked on how Creek could be integrated into jColibri. This is directed on jColibri 1.0, which is now replaced with the entirely changed jColibri2 version series. This work was to a large degree directed at Volve/Verdande Technology, and in a situation where both Colibri and Verdande Technology were far from where they are now. We do however note his conclusion that integration of Creek in Colibri would be the most interesting for "an academic version of Creek"[Stiklestad, 2007][p.77].

Chapter 4

Architecture

Our system is an implementation of Creek within the Colibri studio framework. As Creek is a framework as well as a specific method, this means that what we do is developing a framework which is a framework inside a framework.

We intend that our resulting system will appear to users as a single, coherent framework, providing the best from both worlds. As this is not trivial, we have put certain effort into the structure of our system.

Furthermore, we hope that our system will be subject to further development, as it is by no means completed work. We have for instance pointed our efforts only on the retrieve part of the CBR cycle, leaving the implementation of the remaining phases for further work. We are aware that this development should be done, and have facilitated the system for easily extension. If a developer wants to implement the reuse step in our system, we believe that the architecture is well adapted for this as well as other expected extensions.

4.1 Characterising parts of Creek?

What caracterises Creek? What parts do we need to implement if we are to say that what we implement is the Creek system? What are we required to do to swap out frames with description logics? These are questions we need to answer in advance.

We have made a priority list for which Creek parts we focus on, prioritised on a scale from highest (1) to lowest (6), rendered here:

Priority	**Part**
1	The activate-explain-focus-structure[Aamodt, 1994a]
1	The tight relation between general knowledge and cases[Aamodt, 1994b]
2	Flexibility on relations - several types supported[Aamodt, 1991]
3	Explanation - and justification[Aamodt, 1994a]
3	All knowledge (both cases, general and domain-specific knowledge) represented in the same structure[Aamodt, 1994b]
4	Fallback-structures - use model-based or rule-based reasoning when CBR fails[Aamodt, 1991]
4	Mixed-initiative - ask the user for input when the system's explanation is insufficient[Aamodt, 1991]
5	Several of the cycle steps[Aamodt and Plaza, 1994]
5	A thorough implementation of indexing[Aamodt, 2004]
5	Plausible inheritance[Aamodt, 1994b]
6	The four reasoning modules[Aamodt, 1991]

Table 4.1: Priorities

Of course, this is no definitive list of what Creek *is*, but we find the sum of these items to be a good limited functional specification for a full Creek system.

Plausible inheritance is worth mentioning especially, as it through Sørmo's work[Sørmo, 2000] was elaborated more. This allows for properties to be inherited through all types of relations, e.g. that a part-of-relation implies that the location is inherited. A thorough inclusion of this into Creek would reduce the knowledge acquisition bottleneck[1]. An example of how this can be utilised is seen in [Aamodt, 2004].

The priority of the parts is done by a combination of sorting on how central the part is for the Creek system, the notion of whether or not the part is within the scope for this work and what parts are most useful for reaching our goal and answering our research questions.

This also means that functionality *not* on this list will not be prioritised.

On non-functional requirements, we find these important to mention:

[1]This bottleneck is described in section 3.2.

- Code quality: for a framework, code quality is important
- Runtime: in general, the system should be as fast as possible. The architecture presented makes it possible to create a fast system
- Resource usage: the same counts here as for runtime
- Adaptability: it should be easy to adapt the system to a new domain

4.2 Why starting with an implementation framework?

We have as mentioned chosen to build Colibreek on top of the existing Colibri framework. This choice hides many low-level details from our sight and directs our focus to the higher, more application-oriented levels of framework development. Colibri handles file input/output, casebase initialisation, ontology-to-code-conversion, the execution engine et cetera.

Colibri handles much of the description logics handling, so that we as developers to a small degree have to consider that the reasoning we are doing actually computes with description logics. Colibri and its built-in tools abstracts away DL in abstractions and methods.

This means that we do not ourselves control every part of our system, and that any changes made by the Colibri studio developers may cause our system to malfunction. This is a tradeoff one always has to do when using external frameworks, but it is reinforced by the integrated role Colibri plays in our system.

For most frameworks, the typical way our systems communicate with them is through one or a very few strictly defined interfaces we make ourselves, covered by tests, so that we know exactly what results we expect from the framework. Internal changes made in the framework or even changes in the framework's interface will cause few or no problems for our system, as we at most have to update at one single point. If the Colibri studio developers suddenly changes a central concept or a central interface, our system might face big following changes.

These risks are worth taking, the way we see it. If we were to *not* use Colibri studio, we would spend a significant amount of our total time dealing with low-level issues already addressed in the framework, and we would thus lose corresponding amounts of value-producing time. Figure 4.1 is a good

illustration of this. Maybe even more important, we find the Colibri studio to be of high quality, and providing a suitable and solid ground for our system.

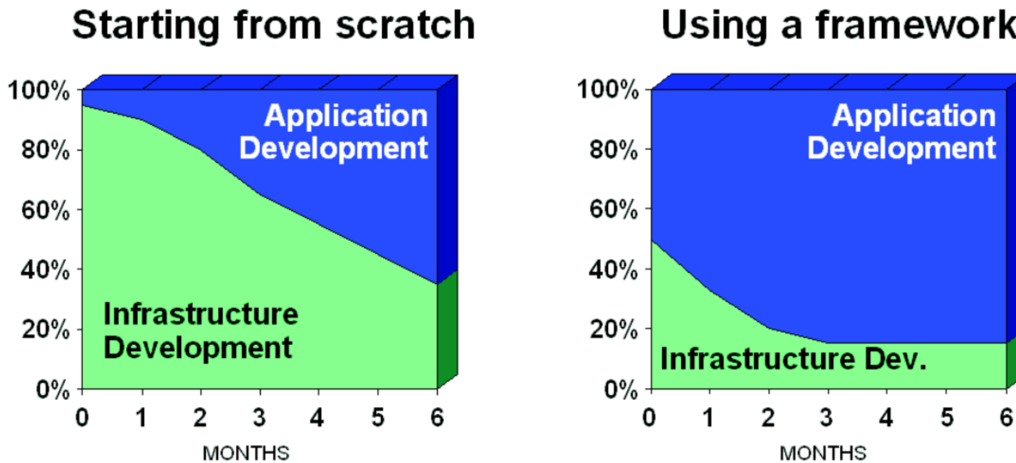

Figure 4.1: **Time spent on application vs. infrastructure**, adapted from [Recio-Garcıa et al., 2006]

In general, the idea of frameworks corresponds well with the premise we started this thesis with: avoiding to reinvent the wheel. Frameworks can make a non-expert able to write more complex applications[Abdrabou and Salem, 2008], and they tend to provide a well-established way of solving more generic parts of the problems. A framework can be seen as a semi-complete application suitable for being specialised to solve specific problems[Bello-Tomás et al., 2004]. In other words: a framework is an application where the developer only needs to specify a defined, limited number of classes to get an application solving her specific problem[Jaczynski and Trousse, 1998].

Even when familiar with these benefits from using frameworks, we find it important to state that frameworks are tools suitable in many, but far from all situations. If the framework is not appropriate for the problem you are trying to solve, do not use it. By *appropriate* here, we think of a wide spectrum of criteria - and they boil down to the simple question “can you solve your problem easier and/or better with the framework or without it, given your constraints?” Interesting aspects to look at here can be how similar your problem is to what the framework is intended at, how complex your problem is, how familiar you are with the domain and technology in hand, lifetime of your product et cetera. A discussion and calculation on the cost of reusing code in general is provided in [Brodwall, 2014].

4.3 Choice of framework - why Colibri?

As we have indirectly said already, we have chosen Colibri studio as the framework we build our system on. Chapter 2 presented myCBR and Colibri studio in some depth, and mentioned several other frameworks as well. We believe that the choice of tools - and a framework is a tool - in fact can be reformulated into "Which tool(s) is the best suitable to help me solve *this* particular problem?" The answer may also totally acceptable be *none*.

We try to create a CBR software development platform that makes it easy for developers to create new knowledge-intensive CBR systems. We try to do as much of the generic, non-business-logic programming as possible, and to make application-specific usage of our software as easy as possible. We will make it obstacle-free to introduce a new problem domain into our system.

These criteria can probably be met by several frameworks. We have found Colibri studio to be the most suitable. We bear in mind that myCBR and Colibri studio perfectly fine can be combined, and that modelling can be done in myCBR, then imported to Colibri studio. While this is clearly beneficial for many systems, and enables better and more thorough representation, we have found it to be a bit overkill for our project.

Non-functional properties contributed to the choice of Colibri as well: it is still subject to maintenance, and it is frequently updated and improved. The Colibri team pointed in [Díaz-Agudo et al., 2000], published in 2000, now 14 years ago, out some of the main lines for what eventually became Colibri, and the first version was released in 2004, described in [Recio-García and Díaz-Agudo, 2004]. It has been subject to improvements and refinement since then, and we find it reasonable to believe that it has now reached a fairly stable level. The latest version, Colibri studio, was released in 2012[2].

We also note that Colibri studio is open source software, so the code is available for us. This means that we can dive into and see for ourselves what it actually does and how it functions. We find this the only really trustworthy way to know what other people's code does, including side effects et cetera, an approach we share with Atwood[Atwood, 2012]. Of course, one can understand code by reading documentation or writing tests, but you can never be entirely sure of potential side effects. On the other hand, reading code is demanding, and to read all the code of the frameworks we are using would take serious amounts of our time. We have therefore landed on the

[2] available at `http://sourceforge.net/projects/jcolibri-cbr/files/COLIBRIStudio/`

compromise that we read the code when we are unsure about what is going on and how, and otherwise trust our tests and the Colibri team's provided documentation.

Note also that Colibri studio uses a lot of other frameworks as well. In section 1.1 we put a lot of focus on how the use of frameworks and reuse of software prevents us from reinventing the wheel over and over again. The Colibri team seem to share this view[Recio-García et al., 2013], and use well-established frameworks for parts of the framework's functionality. The list of used frameworks is presented at `http://gaia.fdi.ucm.es/research/colibri/jcolibri`, and includes the persistence framework Hibernate[3] and the OWL-reasoner Pellet[4].

4.4 Uniform knowledge

Also worth adding is how Colibri studio is formed by thoughts similar to ours: it is designed to be a general-purpose CBR system, and to make it easy to create new systems on top of. We have most of the time found Colibri studio to be playing with us, not against us. A good example for this is found in the knowledge representation: Colibri allows for different types of this, as persistance can be done to flat files, databases or ontologies according to the developer's wishes. We base our work on ontologies, and we found it easy to do this inside Colibri.

We also emphasise the importance of having every piece of knowledge represented in the same structure and in the same way. This makes our system consistent, and lightens the developing burden signficantly. It also makes it a lot easier to write clean code. We remember from table 4.1 that this consistency is high on our prioritised list, so it could potentially be a deal-breaker if Colibri enforced us to violate this principle.

Luckily, Colibri studio with ontology *represents knowledge uniformly by default!* To achieve our goal at this point was simply to do what the framework encouraged us to do. To be more precise, we want to have both the cases and the general knowledge represented in the same semantic network, with every piece of information represented in triples. We wanted "May" in *May subClassOf Spring* to be the same *May* instance as the one in *Case24 hasseason May*. This makes both inference and reasoning a lot simpler, and is

[3] `http://hibernate.org/`
[4] `http://clarkparsia.com/pellet`

a central part of Creek.

4.5 Our architecture

In this chapter, we have chosen to discuss our choices as we describe them. This way, the reader will be able to follow our reasoning and choices as we go.

4.5.1 Introduction

Colibreek, our Creek implementation in Colibri, is built from modular building blocks. We will here describe the architecture of the system as we intend it to be with all parts implemented, as this gives the reader a better overview, while at the same time illuminates the consistency of the system.

We have built the architecture so that it corresponds to the different steps of the execution.

First, a new case, also denoted a query, is brought into the system. This can be done manually or through an automated mechanism. Skalle et al.[Skalle et al., 2013a] test Creek with real-time data and find it promising, a similar approach can prove to be feasible for Colibreek as well. The query can be complete or incomplete, we account for some missing case values.

Some configuration is then done, in the core engine. This includes loading the case base and loading the ontology, if this is not done already. Then the query is passed on to the reasoner.

The system is preconfigured with what type of reasoner to use. Our architecture supports whatever type of reasoner, as it takes a query as input and can return a solution proposal as output. That said, we have built the system for case-based reasoning, and have had CBR in mind during the entire process, so there ought to be a bias towards CBR in our system. A developer intending to use Colibreek with e.g. model-based reasoning should bear this in mind. She should also bear in mind that Colibri studio is intended solely for case-based reasoning[Díaz-Agudo et al., 2000], which leads to illogicalities. A good example of this is that the loading of case bases is done in the core engine, regardless of the type of reasoner chosen.

For a case-based reasoner, the reasoning is divided into the four RE's known from Aamodt and Plaza[Aamodt and Plaza, 1994]: Retrieve, reuse, revise

and retain. Note that revise is not a step included in the original Creek[Aamodt, 1994a], but we find it perfectly fine to include in the architecture, and from this architectural point of view, it is similar to the other three phases. Each of the RE's have its own activate-explain-focus-cycle, as we described in section 3.4.1. For each phase, the exact steps done in activate, explain and focus differ, but the overall structure is the same.

Each step benefits from Colibri's built in DL reasoner, as well as other tools included in Colibri.

When the case that is the proposed solution is found, the reasoner handles this to the executer. The executer then presents the solution to the user. Exactly how this is done is implementation-dependent, the simple default is printing to the console.

4.5.2 Major architectural choices

This little review of our architecture illuminates two essential architectural choices. First, it is flexible, and different implementational solutions can be chosen for the majority of the steps. In order to make the framework suitable for solving many different problems of different types with different techniques, we will allow for as wide application as possible, while still making it easy to stick to the default settings. This duality is non-trivial, but we believe that our approach fulfills the intention. Through modality and relying on sharply defined edges, the number of classes and methods to replace is minimised.

Second, much of the overall structure for each RE-step is the same, so we have represented them as one in the architecture. However, there are of course differences when set on a lower level. This counts especially for revise. The three other steps are complementary described in the Creek context (in [Aamodt, 1991] and [Aamodt, 1994a], for instance), while revise deliberately has been considered a step not part of Creek. We will focus on the retrieve step in our detailed design and implementation, and we believe that illustrating how this is done combined with the insight of how the other steps are intended makes the reader herself able to elaborate her intended step.

4.5.3 Domain independence

As the diagram of our architecture - figure 4.2 - indicates, the architecture is made as domain-independent as possible. This is a natural consequence of our desire to create a framework that should be applicable to a wide range of domains. We consider this to be one of the central advantages of our framework. This also means that taking Colibreek from a framework to a runnable system in the normal case is to implement your domain logic and tell Colibreek to use it.

This also means that Colibreek is agnostic to which ontologies it reasons with, as long as the developer has connected the choicen ontologies to the system correctly in the few required seams.

All of the domain logic is collected in one place, which makes it easy to add a new domain to the system and be sure that nothing is forgotten. Doing it this way, we also lower the risk of tangling the domain logic into our framework.

During the development, the architecture changed, as expected. In section 2.3.2, we sketched out how we were to elaborate the architecture.

An important aspect of the architecture is that most entities in it correspond to central parts of the domain - the architecture is domain-oriented rather than technology-oriented.

4.5.4 Our architecture - part by part

The architecture of our system is illustrated in figure 4.2.

The flow starts with a *new case*, also denoted a query, that comes into the core engine. The system is agnostic to how this new case comes - whether it is through input from the user, from another system or from a file, for instance.

The *core engine* constructs the system parts. It loads the case base into memory, initialises the ontology connector (for connecting with the knowledge base) and then passes the new case along with the case base to the reasoner.

The *reasoner* can be either model-based, rule-based or case-based, as is symbolised by the light-grey arrows.

If the chosen reasoner is the case-based, the *CBR reasoner* will use the case base, the rest of the knowledge base and the query to reason. It will follow

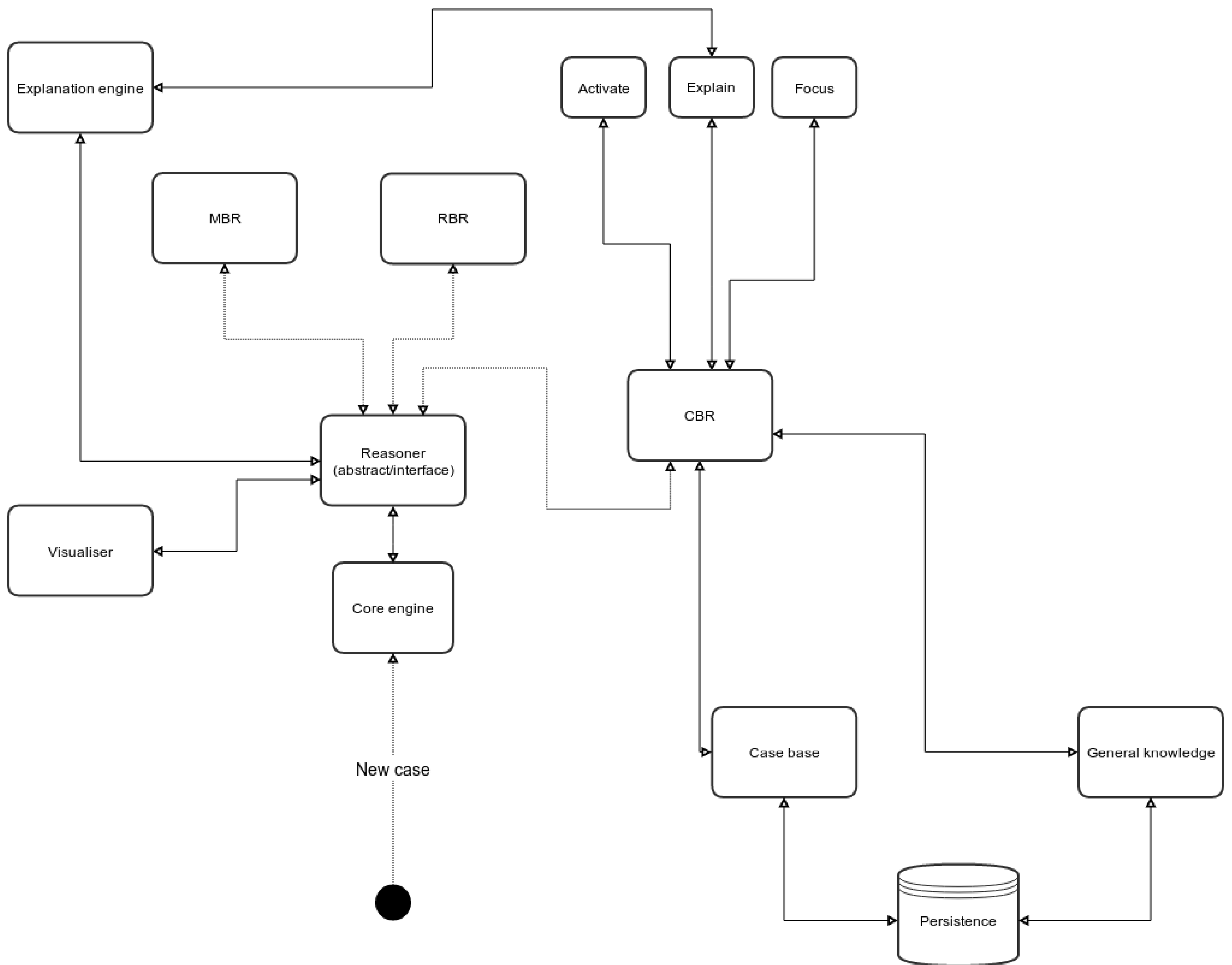

Figure 4.2: **The architecture of Colibreek**

the activate-explain-focus cycle.

The case base and the general knowledge are obtained through tools from *Colibri studio*.

First, the *activator* activates the possibly relevant cases and a broad, possibly-relevant part of the knowledge base - a rough extraction.

The *explainer* will receive the activated area and the cases from the activator, and explain each case more thoroughly. It will find the most relevant cases and provide explanations for each of these.

The *focuser* will pick the most relevant case provided from explain.

This cycle will be repeated, in a slightly different matter, for the additional CBR cycle steps reuse and retain.

Whatever reasoner chosen, the system needs to explain its results to the user. This is the responsibility of the *explanation engine*. When use of CBR, this task is left to the explainer.

Finally, the *reasoner* passes the task of displaying the results to the user on to the *visualiser*. The visualiser displays, in a way specific per domain, the results and the explanation to the user.

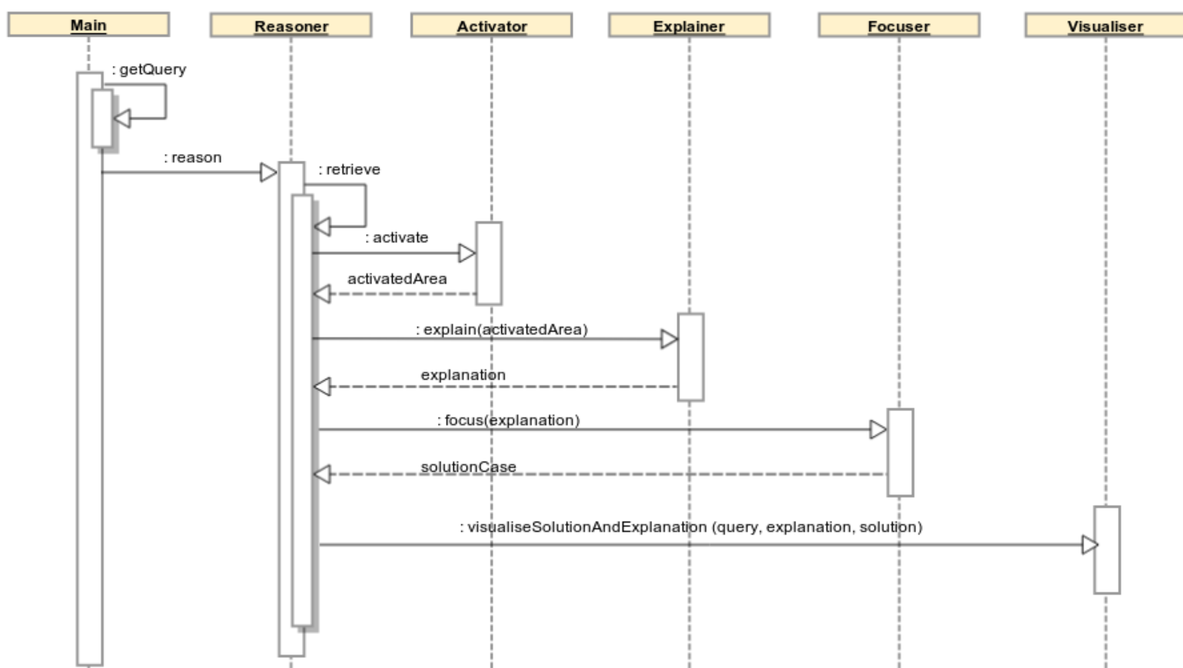

Figure 4.3: **A diagram showing the sequence of a run of the system**

This flow is visualised in figure 4.3, and each step will be described in more details in section 5.5.

4.5.5 The architectural dependency on Colibri

Section 4.2 revealed the tight dependency between our system and Colibri studio, coming as a consequence of choosing Colibri as the framework for our implementation. This is definately true, but the degree of dependence is not a priori set. The established best-practice approach to using external tools, as we described in section 4.2, enforces sharp boundaries between our code and the tool code. We create our own entry and exit points from our code, and at this entry point, the conversion and communication between our and their code is done. There is no structural nor basic problem preventing us from doing this also with the Colibri code.

However, we have tried to stick as much to the Colibri code and ways of doing things, as far as they do not conflict with the Creek way or our essential structure. The reason for this is partly to lower the threshold for developers already familiar with the Colibri structure. This means for instance that we accept and include the Java Bean structure[Recio-García et al., 2014], even though we personally ideally would consider replacing this mechanism. It also means that changing the essential Colibri classes can hardly be envisaged.

This justification is supplied with the fact that our Colibreek is already thoroughly dependent on Colibri in order to run at all. This means that it is hardly a current issue to replace Colibri studio with another framework or just our own code. Even if so is to be done, the amount of classes we explicitly rely on is limited, around ten.

The Colibri classes are no obstacles for our automated testing either, so this is no argument for encapsulating them. The opposite here are some of Java's built-in classes, for instance Scanner (the utility for receiving keyboard strokes from the user). Scanner can not be easily replaced by a testing dummy, so Scanner would need to be encapsulated for testing to be possible.

This in sum makes us accept the direct dependencies between our code and Colibri classes.

We admit that we are not totally convinced that this choice is the best, but we find it reasonable and at the moment, it fulfills our needs. A developer not satisfied with this choice can of course do the required changes and create sharp boundaries.

Chapter 5

Colibreek

In chapter 4, we described the architecture and the high-level structure of our framework. In this chapter, we will go into more detail and describe our realisation of the architecture. We have implemented parts of the architecture, and will in this chapter describe our implementation.

The intention behind Colibreek is to be a solid framework where each delivered part is of high quality. Future developers should be able to rely on that we have made a trustworthy platform for further development.

5.1 Using Colibreek

5.1.1 Configuring Colibreek for your domain

When a user wants to use Colibreek to solve the problem in her domain, it is important that Colibreek is configured to use the settings for that domain. This is done through changing *one line* in the system, and is in a production or test setting done during setup, so the end user will not have to worry about this. The rest of the configuration is also done during the setup.

Colibreek is implemented with the use of subclassing for domain-specific settings, methods and variables. We ship the system with two domain implementations, *TravelRecommender* and *TurbineSensor*. These illustrate how to adapt Colibreek to the domain you need, and the specific implementational details related to the domains are depicted in chapter 6.

When you are to introduce a new domain, you have to make a few more changes. You have to:

1. Create a package in colibreek.domain with a describing name
2. Implement the classes in that package according to what you need. See the existing domain packages for inspiration and guidance.
3. Add your domain in the Domains enum
4. Make a method for your domain and create a case in the Colibreek-Module class
5. Change the first line in DomainChooser to use your domain

The second step in this walkthrough is of course the most complex one. This also includes acquiring your desired ontology and case base, and creating a XML config file for it. Good examples of how to do this are the already implemented ones. These tend to follow the same pattern.

The implementation of Colibreek is shipped with two domains initially implemented, both as a security net for ourselves not to tangle domain logic into other parts of the framework, as a proof of concept for further developers, and to test that the framework supports both of these rather different domains.

5.2 Implementational principles - a system to be reused

Throughout this entire work, we have emphasised the aspect of *reuse*, an aspect we also have encorporated into our code base.

Colibreek is by intention made to be highly modular and testable, as we find these important qualities of reusable code. It has also been important for us to build a system that is easy to understand and easy to modify. We believe that breaking down the system in modular parts, and using small building blocks makes this possible. Section 4.5.5 emphasised how our use of Colibri studio complicates this, and how we are sticking to Colibri standards whereever suitable.

Inner consistency is of particular importance in a framework. We have tried to be consistent on coding standards, naming, structuring and so on. Our heuristics have been towards short methods, describing names, limit use of

null, make use of dependency injection and a few more. The best documentation for such things is the code itself, so we encourage you to read the code to get the whole picture.

Also, a natural consequence of using test-driven development is a high test coverage. We have ensured that all the important and all the risk-associated code is fully covered by tests. We have included three types of tests: unit tests, integration tests and acceptance tests, unit tests and acceptance tests as described in section 2.3.4. Integration tests test the communication between some of the units of the system.

5.3 Implementational choices

5.3.1 Retrieve

The CBR paradigm advocates four phases in the CBR cycle, and Creek leaves out the revise phase from the system's consideration. Each phase of the cycle depends on the outcome from the previous step, and a CBR system with only a subset of the phases would be limited.

While the CBR process clearly is a cycle, the character changes when we break it down to the situation for each new case. For a new case, the process starts with retrieval and sums up with retain[1].

We have chosen to prioritise the implementation of phases in decreasing order from first to last, starting with retrieve. The main reason for this is usefulness - for an expert user, the outcome from retrieve often is of value in itself.

Hence, we will limit our work to the retrieve phase. Beck and Andres[Beck and Andres, 2004] state that there are four variables regulating software development: cost, time, quality and scope. For us are cost and time defined, so the two variables we can control are quality and scope. If we lower our quality requirements, we will break the confidence given to us by those interested in extending or using our system. We take the consequence of this, and limit our scope to what needed in order to provide the expected level of quality.

[1]At least when not taking the setup into consideration.

5.3.2 Constants and configurations

Colibreek divides between two types of constants and configurations: the domain-specific and the domain-independent. The domain-dependent are specified within either the domain package or within the class *DomainDependentConfigurations*, which every domain must subclass. The threshold value for activation of a case, which type of relations that should lead to application and the algorithm for comparing date/time are examples of configurations set in this class. The values provided in this class are standard values and methods that tend to be common for many domains, but they can be overridden when needed.

Every domain is responsible for providing its configurations, such as paths to its ontology and case base, types of findings in the case description and how the case solution should be presented to the user.

Domain-independent configurations are in general placed where it is needed in the code base. Some constants are gathered at program-level, mainly the generic, standard OWL URLs.

5.3.3 Relevance factor

Creek originally has the concept of relevance factor, computed from the combination of importance and predictive strength. Importance is the degree of necessity, and predictive strength is the degree of sufficiency, of a finding describing a stored case[Aamodt, 1994a]. We have chosen to implement only the relevance factor from this, as we found this sufficient for our use. The extension to include also the two other factors should though be quite tiny.

Each domain offers a slightly different set of relations, and different findings, so the relevance factors must clearly be defined per domain. The original Creek approach is to define relevance factors per finding-case pair, while we have chosen to assign the same relevance factors for every case related to the finding. Beck and Andres[Beck and Andres, 2004] point out that when you are unsure today of whether you will need a future tomorrow, the best typically is to wait until tomorrow to see if you *really* need it then. If you do, you implement it tomorrow. This way, you do not spend time on doing work which very well may prove to be misguided, overengineered or even useless. We have found experimentally that our data do not require the possibility to specify relevance factor per case, so we have chosen not to include it. This implementation can be changed when - or even if - needed.

When developing a framework, as we are, this rule of thumb needs to be taken with a pinch of salt, but we still find the underlying idea to be true. As Beck and Andres point out, if you keep your code in a good state and your architecture as simple as possible, the cost will be relatively small if you are to implement this feature later on. If you intend to do this, use the *Finding* class as your starting point, then you should have a pretty good go.

You may however easily change the relevance factors for the findings, as these are given in the domain-specific setup class you specify for the domain, in the *CaseDescription*-subclass. The relevance factor is a floating number between 0 and 1.

5.3.4 Similarity measurement

In [Aamodt, 1994a], the similarity measurement between two findings in Creek is described concisely. The measurement is done entirely on categorical values, meaning that also numbers and date-times would have to be cast into categorical values, despite this being both illogical and ineffective.

We have extended and tweaked the algorithms for this, and provided three different comparison algorithms. We believe this to be a substantial improvement of this part of Creek, and would encourage further work on Creek to include this advance.

Numerical values

If the finding is a numerical value - e.g. a decimal, double or float - we do a mathematical comparison. The exact algorithm for this is given in section 5.5.5.

In Creek, numerical values are grouped into qualitative values, such as 'high', 'medium' and 'low'. We have improved this algorithm by allowing for actual computation with numbers in order to determine how similar they are. This way, this pre-compilation of knowledge is significantly reduced.

However, limitations in Colibreek enforce us to still group the numbers. Colibreek insists to represent all findings as *instances*. A consequence of this is that numbers must be rounded to a limited number of decimals[2], while it

[2] How many decimals depends on the spreading of the values for this finding, in our turbine domain we have successfully used 0 or 1.

would have been both easier and more straight-forward to use an associated data type such as double.

Nevertheless, the reasoning is done automatically, and all we have manually done is round the numbers. The algorithm we use in Colibreek would be exactly the same if we were to not round numbers as well, the input data is the only point of change. Hence, despite the Colibri-introduced limitations, we see this as a great improvement.

Datetime

We use a specific algorithm to handle date or time as well[3]. The exact algorithm for this is given in section 5.5.5.

We have chosen to leave the exact algorithm for comparing times domain-specific. For some domains, only the dates are interesting, while others emphasise minutes and seconds. We have as a consequence formulated a default algorithm, and made it suitable for overriding.

Categorical values

The similarity between categorical values is computed as described in section 5.5.5. It uses a simplified version of the algorithm proposed in [Aamodt, 2004], as it simply computes the length of the path between the two findings. The relation strength concept is deliberately let out of our implementation, mainly because it turns out to be challenging and time-consuming to implement it within the Colibri studio limitations. The same counts for the parallell paths, as our way of doing this also relies heavily on Colibri studio. These simplifications leave us with a simple, straight-forward computation.

5.4 The implemented architecture

Chapter 4 illustrated and described the architecture of the full Colibreek system. Here, we present the architecture of the implemented parts of it. It can and should be seen as a subset of the full-system architecture we presented in section 4.5.

[3] It would be technically legit, but logically way off, to handle dates as numbers.

5.4.1 Step-by-step

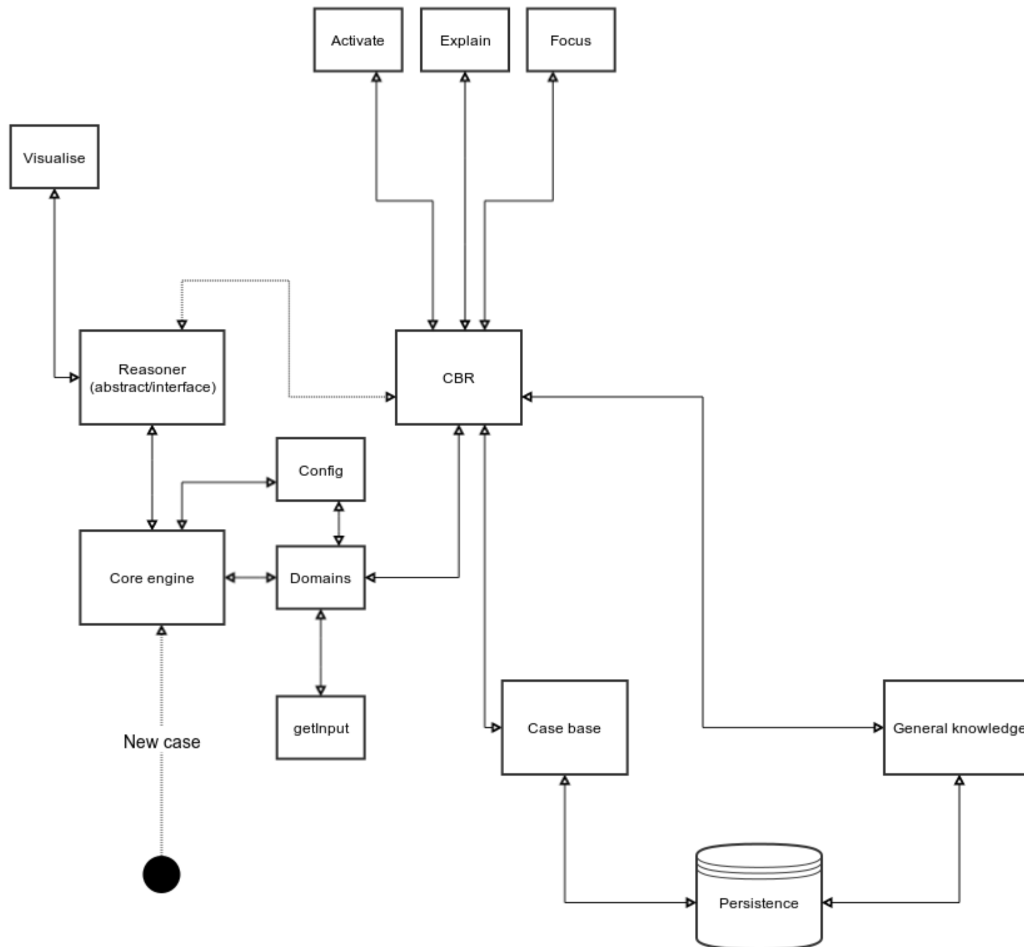

Figure 5.1: **The architecture of the implemented system**

The implemented architecture is visualised in figure 5.1. It is a specialisation of the general Colibreek architecture seen in figure 4.2. The notable differences originate from the implementational choices done during implementation.

The implemented architecture focuses solely on the case-based reasoner, and the explanation engine is an integrated part of the explain step. These are the two simplifications of the architecture, while we have also included some details that are suitable at this level. These are seen right next to the core engine: config, domains and getInput.

getInput is the module that lets the user give input to the system. In the

present implementation, it lets the user specify the new case she wants to run through the system, but it may also be extended later on when implementing other CBR cycle steps.

The *domains* module controls which domain the user has chosen to run Colibreek on, as well as the consequences of this choice: the domain-specific parts of the system, such as case structure and configuration files.

The *config* module contains matching thresholds, default OWL URLs and which relation types that should lead to activation.

The remaining modules behave as described in section 4.5.4.

5.4.2 The representation of cases and the domain

The domain is represented in Colibreek through five classes, including one trivial. The four classes describes respectively the case description, case solution, domain-specific system configurations and input-output-configurations.

The case description consists of a variable per finding, get-and-set-methods for each, and a method to get all findings.

The input-output-configurations specify how the findings are put into the system, and how the results of a run is displayed to the user.

A discussion of interest here is to which degree the system should accept incomplete cases - that is, cases where one or several findings are undefined. We have chosen to allow this, and to simply ignore these findings. A path to walk for improving the matching could be to lower the total match degree for each missing finding. A part of the task for the non-implemented CBR-cycle steps is to fill these undefined finding-slots.

We are also unsure whether the system should allow for slots to be filled by classes rather than instances - say for instance in the travel recommendation domain: if you know that a vacation was during the summer, but not whether it was in June or July, should it be saved into the system simply as "Summer"? We are unsure, and have thus chosen not to support this at this point of implementation. Extending the system to tackle this could be done later on if found needed.

5.5 A walkthrough of the system - under the hood

5.5.1 CBR reasoner

The CBR reasoner class coordinates the reasoning. It exposes the *reason* method, which accepts the query as input and calls on the implemented reasoning steps - at the moment, this is just retrieve.

Retrieve first checks if this query is already present in the case base. This is not an actual issue for neither of our domains, but we guess that it might be for other domains. If the query is already present in the case base, we return that case and the reasoning is finished.

In the usual cases where the query is not already saved, the reasoner fetches the relations in the semantic network, through the use of Colibri's methods. The relations, the case base and the query is provided to the activator. The resulting activated subnetwork is passed on to the explainer. The explanations are handed to the focuser, and when the focuser returns its solution, it is the visualiser that presents the results to the user. Each of these steps are described in more detail below.

5.5.2 Activator

This class handles the activate step. If the number of statements in the graph provided as input[4] to the activator is lower than a domain-specific limit, the entire graph is activated, to avoid overhead.

Activate follows the Creek standard way. Activation is spread through the network, starting from the findings of the query. The algorithm is like this:

- All the links to or from the findings of the query are at first activated.
- Every case with at least one finding equal to one in the query is found.
- For each of these cases, all the relations directly linked to its findings are activated.
- Then, the activation spreads from each now activated node and both up and down in the network. This algorithm loops through every statement having the findings so far activated as its *object*, and checks every

[4]Typically the knowledge base.

statement having this finding as its *subject*. If this relation is of a domain-specific pre-defined type, the statement is activated. This process is done recursively until the top node is reached.

In the original Creek, there was proposed a default set of relations to activate along[Aamodt, 1991][p.280]. The flexibility needed in order to support a variety of ontologies means that this list has to be domain-specific. For our testing domains, we have chosen a subset of relations consisting of *subClassOf*, *subPropertyOf*, *topDataProperty* and *hasDataValue*. Summarised, we can say that the last item in our list can be expressed as: activate recursively from the query's and the activated case's findings to the top node, as long as the relations are of a specific type.[5]

Activator finishes off its work by making sure that all cases it activates matches the query to a sufficient degree - the two cases have a similarity of at least the domain-specific, manually set threshold value. A question here arises: how is this similarity computed? On the one hand, the results we get out will be better the more sophisticated the computation is. On the other hand, we want to make this computation efficient: we do this computation on often the bulk of the case base. This tradeoff is difficult.

We have chosen a rather simple similarity measure: how many findings are equal between the old case and the query, weighted with relevance factors? The precise formula is given in equation 5.1. The computational efforts of this compared to running the system without the activator was, according to our integration tests, approximately the half. The threshold as well as the aggregation levels for each finding can be adjusted to tweak this algorithm to include the desired cases. Note that we have assumed that the number of findings are the same for every case in the domain.

$$\frac{\sum_{i=1}^{n} relevanceFactor_i * findingIsEqualInCaseAndQuery_i}{\sum_{i=1}^{n} relevanceFactor_i} \tag{5.1}$$

[5] As always, the best documentation is the source code itself. To get the precise and definite view of the algorithm, look at the source code. Appendix A points to where the code can be found.

5.5.3 Explainer

The explainer receives the graph with the activated statements and cases from the activator. Explain looks at every activated case, and does a more sophisticated comparison in order to find a more precise similarity. The explainer compares, for each activated case, each finding to its equivalent in the query, and finds their similarity and the path between them in the graph. How this is done is described in more depth in section 5.5.5.

The explainer creates an explanation for each of the activated cases. This explanation contains a reference to the case and the collection of all the finding similarities. Each finding similarity contains the finding class it describes, the similarity value, the trace and the relevance factor for that class. The explainer returns the explanations, and is done with its work.

5.5.4 Focuser

Focus takes as input the set of explanations from the explainer, and finds the single most suitable case to solve the problem in question. In our system, as in Creek, the job done by the focuser is trivial. It chooses the most similar case from the input, or if none is considered similar enough, it returns the empty case, saying that no solution was found.

5.5.5 Explanations to the user

Colibreek, as Creek, is primarily targeted at expert users. The explanations provided by our systems take this into account. Sørmo et al.[Sørmo et al., 2005] found that explanations can be categorised into four boxes:

- Reasoning trace - what are the steps in the reasoning?
- Justification - why is the system reasoning that way?
- Strategic - how does the system reason? What is its strategy?
- Terminological - to explain terms and concepts from the domain

While novice users typically want terminological or justification explanations, experts wants to know about the reasoning trace and the strategical decisions. Colibreek compares the query to a case in the case base by comparing finding by finding.

For each finding, the system will compute how similar the two cases are with regards to just this finding class. We have three different strategies for doing similarity measurement in Colibreek, depending on the data type of the finding in consideration.

Numerical values

If the findings are numeric, the system will compute how similar the two findings are by using the formula 5.2. *Max* and *min* in this equation are the highest and the lowest, respectively, number of this class registered in the ontolology. For example, in our turbine sensor domain, the highest registered exciter field voltage is 3.9. Max is thus 3.9. The lowest registered is 2.9, and this is min.

$$1 - \frac{|FindingInNewCase - FindingInActivatedCase|}{Max - min} \tag{5.2}$$

Date or time values

If the finding is a date or a time, the system does a specific type of comparison between the two findings - it compares year to year, month to month and so on. The default algorithm is as follows:

```
Start with similarity 0.0
If year is equal, similarity += 0.25
If month is equal, similarity += 0.2
If date is equal, similarity += 0.2
If hour is equal, similarity += 0.2
If minute is equal, similarity += 0.15
If second is equal, similarity += 0.0
```

Categorical values

If the finding is neither a number nor a date-time, the system makes use of the graph. Every finding is a separate node in the system's semantic network. The corner case first: if the finding is the same in both the new and the old case, the similarity is simply 1 and no more reasoning is done. Otherwise, the system finds the shortest possible path between the two findings in the graph. This is done by finding the lowest common ancestor of the two nodes.

This ancestor can however be a concept too generic to provide any useful information about the similarity, typically when the ancestor is a node that is common for all instances of a class. This can be the class itself or it can be *thing*. When no ancestor lower than this level is found, it means that the findings are not similar at all. The similarity returned is then 0.0 and the path is discarded.

Two findings with some, but not total similarity, will have its similarity degree computed based on how long the path between them is. The shorter the path, the higher the similarity, and the equation is given in 5.3. This computation is a simplification from Creek, as Creek utilises relational strengths when computing this similarity[Aamodt, 2004]. However, to incorporate relation strengths within Colibri as is of today, proved out to be too much labour for too less benefit.

$$\frac{1}{length - of - the - path} \tag{5.3}$$

Presentation to the user

When the system has finished its reasoning, the results and explanation must be presented to the user. How this is done is domain-specific. How this is done in the implemented domains, can be seen in section 6.5.

Chapter 6

Application of Colibreek on our domains

This chapter describes the application of the general problem-solving tool Colibreek on the domains *travel recommender* and *turbine sensors*.

6.1 How to use Colibreek - as an end user

To use Colibreek, the user solely has to enter the query she wants to solve. A *query* has the same structure as the description part of the case, and will during reasoning be handled the same way. Let us see how this works out in practice, we look at how an end user can use Colibreek with our turbine sensor domain.

The user hits the run button, and Colibreek starts. The user is then questioned on the properties describing her new turbine sensor case: the exciter field voltage of the turbine generator, the power value of the generator, the id of the generator in question, at what time the case happened, and a few more variables related to the generator (exciter field current, reactive load, phase-A-current and phase-B-current). The system then does its reasoning, and presents the solution to the user along with an explanation.

Keep in mind that Creek is supposed to be used by expert users. Hence, it is important that the explanation provided justify how the system reached its conclusions[Sørmo et al., 2005]. We have solved this by presenting to the user the 10 most similar cases, including findings and solutions, as well as the single most similar case. For different categorical findings in this case,

Colibreek presents how it computed the finding's similarities with the query. If for instance the query has a finding for month with value July, and the solution case has June, the system states that July is a summer month, and June is a summer month, hence they are quite similar. This is an application of the mechanisms described in section 5.5.5.

6.2 Our choice of ontology

A crucial factor for success for every knowledge-intensive CBR system is the availability and application of general knowledge. This general knowledge is within DL represented in ontologies, as discussed in section 3.3.4. As there are a diversity of available ontologies, the choice of ontologies to use should be taken consciously.

Ontologies are both general-purpose and more domain-specific. We have tried our system using solely domain-specific ontologies as well as with the combination of domain-specific and general-purpose. In addition comes of course the possibility to create your own ontologies.

We have considered both the domain-specific ontologies Semantic Sensor Network (SSN)[1] and ISO 15926[2], as well as the general-purpose ontologies DOLCE Ultra Lite (DUL)[3], Suggested Upper Merged Ontology (SUMO)[4] and COSMO[5]. The general-purpose ontologies are used in combination with the domain-specific ones.

It seems obvious that a domain-specific ontology should in all but the most generic situations be used. We have found SSN to be simple as well as suitable for our condition monitoring domain, so have chosen to stick to it for our turbine sensor domain. When testing to combine it with an upper-level ontology, we have noticed that each of the three ontologies mentioned made the system slower. SUMO and COSMO to a remarkable degree, while the difference was smaller with DUL, though still approximately doubling the runtime.

To be able to make use of the upper-level-degree ontology, it must be connected with the domain-specific ontology. This will demand insight in the

[1] http://www.w3.org/2005/Incubator/ssn/ssnx/ssn
[2] http://15926.org/
[3] http://www.loa.istc.cnr.it/ontologies/DUL.owl
[4] http://www.ontologyportal.org/index.html
[5] http://micra.com/COSMO/

ontologies and some time. SSN and DUL is initially coupled from the SSN developer team, so advantage can here be taken from combining these two.

There are advantages and benefits for whatever choice made, and the major tradeoff is between time and the potential for improved precision. This tradeoff did for our domain tilt towards sticking to only SSN, but this varies with the domain

6.3 What is in a case?

The general properties listed in section 5.4.2 holds for the concrete domain-specific case structures as well. Here, we will describe the case structure for the travel recommender and turbine domain, respectively.

6.3.1 Travel recommender

The travel recommender domain is a prototypical domain for CBR systems, and our case structure here is taken from the Colibri studio documentation[6].

A travel recommender case consists of a description and a solution. The description structure is as follows:

- mainConcept - also known as caseId.
- HolidayType - e.g. *bathing* or *wandering*
- NumberOfPersons
- Transportation - e.g. *plane* or *car*
- Destination - e.g. *Egypt* or *Turkish riviera*
- Duration - number of days
- Season - divided by months, so e.g. *May* or *December*
- Accommodation - e.g. *Holiday flat* or *Three stars*

The solution is simpler:

- mainConcept

[6] More precisely, "test 10", available at `http://gaia.fdi.ucm.es/files/people/juanan/jcolibri/doc/api/jcolibri/test/test10/package-summary.html`

- Price

All values are categorical. Price, duration and numberOfPersons could advantageously have been numerical values, but this is not originally supported by Colibri, and we chose to put our categorical-to-numerical-effort in the turbine domain rather than the travel domain.

6.3.2 Turbine sensor domain

The data for this domain are provided by ConocoPhillips Norway, and the choice of findings are defined based on the data received from them.

The monitored turbines are covered with a lot of sensors, and the data amounts are huge. Each type of mechanical issues are discovered by a different set of sensors. We found during development that every extra parameter introduced extended the runtime dramatically, so we have limited the number strictly. In order to make the scope practically manageable, we have chosen to refine the implementation to cover a specific type of mechanical issues.

The structure of a case description is as follows:

- CaseID - categorical
- Timestamp - with year, month, date, hours, minutes, seconds. Seconds are excessive for the test data. Datetime
- GeneratorID - the ID of the monitored generator. Categorical
- GeneratorPower - numerical
- ExciterFieldVoltage - numerical
- Frequency - numerical
- ExciterFieldCurrent - numerical
- ReactiveLoad - numerical
- PhaseACurrent - numerical
- PhaseBCurrent - numerical

The potential problems are typically found when the proportion between the generator power and exciter field voltage values is wrong. The last five findings are also properties of the generator at the given time, and are part

of the case description to minimise bias and ensure the sustainability of the system.

A case solution is also here a simpler structure:

- CaseID - categorical
- IsAlarmingSituation - i.e. is this a situation where it seems probable that something is wrong? The user should take a deeper look into this situation.

The structure of the Java class representing the case solution can be seen in figure 6.1.

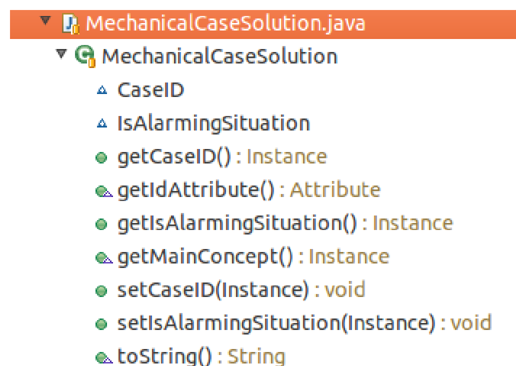

Figure 6.1: **The case solution structure for the turbine domain**

6.4 Runtime analysis

The figures 6.2, 6.3 and 6.4 show from different angles what the computational-heavy parts of the system are, and the distribution of CPU usage among them. The numbers are accumulated through a run of each of the 423 cases for the turbine sensors.

6.2 shows that much of the computation is demanded from the Jena reasoner, as can be seen if you look at the package each method belong to - a vast majority of the most consuming methods belong to various *com.hp.hpl.jena* packages. Of course, the Jena methods are linked together with and used within our own-made Colibreek methods, so these numbers should only be taken as an indication. For instance, the *toString* and *hasNext* methods are used within several of our algorithms. Thus, the figure should be seen in connection with figure 6.3 and 6.4.

Hot Spots - Method	Self time [%]	Self time	Self time (CPU)
com.hp.hpl.jena.reasoner.rulesys.impl.LPBRuleEngine.**checkForCompletions** ()		867 833 ms (43,3%)	867 833 ms
colibreek.reasoner.cbrreasoner.steps.activate.ColibreekCBRActivator.**activateAllStatementsLinkedToThisFinding** ()		199 188 ms (9,9%)	199 188 ms
com.hp.hpl.jena.reasoner.rulesys.impl.Generator.**addConsumer** ()		194 985 ms (9,7%)	194 985 ms
com.hp.hpl.jena.graph.impl.LiteralLabelImpl.**toString** ()		180 789 ms (9%)	180 789 ms
com.hp.hpl.jena.util.iterator.WrappedIterator.**hasNext** ()		104 998 ms (5,2%)	104 998 ms
jcolibri.datatypes.Instance.**toString** ()		83 696 ms (4,2%)	83 696 ms
com.hp.hpl.jena.util.iterator.NiceIterator.**close** ()		55 765 ms (2,8%)	55 765 ms
com.hp.hpl.jena.graph.Triple.**hashCode** ()		47 278 ms (2,4%)	47 278 ms
com.hp.hpl.jena.reasoner.rulesys.impl.ConsumerChoicePointFrame.**nextMatch** ()		42 670 ms (2,1%)	42 670 ms
com.hp.hpl.jena.util.iterator.NiceIterator$1.**hasNext** ()		23 087 ms (1,2%)	23 087 ms
colibreek.reasoner.cbrreasoner.steps.explain.SimilarityUtils.**isANumber** ()		19 429 ms (1%)	19 429 ms
com.hp.hpl.jena.graph.Triple.**sameAs** ()		12 981 ms (0,6%)	12 981 ms
com.hp.hpl.jena.reasoner.rulesys.impl.LPInterpreter.**close** ()		12 012 ms (0,6%)	12 012 ms
colibreek.reasoner.cbrreasoner.steps.explain.SimilarityUtils.**getLocalNameFromIndividual** ()		9 002 ms (0,4%)	9 002 ms
com.hp.hpl.jena.graph.impl.GraphBase.**find** ()		8 909 ms (0,4%)	8 909 ms
colibreek.reasoner.cbrreasoner.steps.activate.ColibreekCBRActivator.**getStatementsFromActivatedCases** ()		8 002 ms (0,4%)	8 002 ms
com.hp.hpl.jena.util.iterator.FilterIterator.**hasNext** ()		7 788 ms (0,4%)	7 788 ms
com.hp.hpl.jena.reasoner.transitiveReasoner.TransitiveGraphCache.**find** ()		6 703 ms (0,3%)	6 703 ms
com.hp.hpl.jena.graph.Node.**hashCode** ()		6 412 ms (0,3%)	6 412 ms
com.hp.hpl.jena.reasoner.rulesys.impl.LPInterpreter.**<init>** ()		6 290 ms (0,3%)	6 290 ms
com.hp.hpl.jena.reasoner.rulesys.impl.Generator.**runCompletionCheck** ()		6 209 ms (0,3%)	6 209 ms

Figure 6.2: **Resource-consuming methods for turbine sensor domain**

Figure 6.3 shows the distribution between the modules of our system. As we can see, the explain step is by far the most resource-consuming step. A further break down of the explain step is found in figure 6.4.

Method		Time	Time (CPU)
colibreek.reasoner.cbrreasoner.ColibreekCBRReasoner.**retrieve** ()		2 005 491 ms (100%)	2 005 491 ms
colibreek.reasoner.cbrreasoner.ColibreekCBRReasoner.**activateExplainFocus** ()		2 004 583 ms (100%)	2 004 583 ms
colibreek.reasoner.cbrreasoner.steps.explain.ColibreekCBRExplainer.**explain** ()		1 465 469 ms (73,1%)	1 465 469 ms
colibreek.reasoner.cbrreasoner.steps.activate.ColibreekCBRActivator.**activate** ()		531 681 ms (26,5%)	531 681 ms
colibreek.reasoner.cbrreasoner.StatementExtractor.**getInitialStatements** ()		3 415 ms (0,2%)	3 415 ms
colibreek.ExplanationVisualiser.**visualise** ()		3 017 ms (0,2%)	3 017 ms
colibreek.reasoner.cbrreasoner.steps.focus.ColibreekCBRFocuser.**focus** ()		1 000 ms (0%)	1 000 ms
Self time		0,000 ms (0%)	0,000 ms
colibreek.reasoner.cbrreasoner.ColibreekCBRReasoner.**findTheExistingSolutionIfTheCaseIsAlreadySolved** ()		907 ms (0%)	907 ms

Figure 6.3: **Distribution of resource-demand between our modules**

Method		Time
colibreek.reasoner.cbrreasoner.steps.explain.CaseSimilarityComputer.**findPathAndThenComputeSimilarity** ()		1 458 767 ms (72,7%)
colibreek.reasoner.cbrreasoner.steps.explain.PathHandler.**findPathBetweenTwoFindings** ()		1 272 699 ms (63,5%)
colibreek.reasoner.cbrreasoner.steps.explain.PathHandler.**findPathBetweenAandB** ()		792 063 ms (39,5%)
colibreek.reasoner.cbrreasoner.steps.explain.PathHandler.**atLeastOneFindingLacksClass** ()		252 887 ms (12,6%)
com.hp.hpl.jena.ontology.impl.IndividualImpl.**getOntClass** ()		214 057 ms (10,7%)
colibreek.reasoner.cbrreasoner.steps.explain.PathUtils.**getIndividualFromName** ()		13 192 ms (0,7%)
colibreek.reasoner.cbrreasoner.steps.explain.PathHandler.**returnsPathWithoutExcessiveStatements** ()		498 ms (0%)
Self time		0,000 ms (0%)
colibreek.reasoner.cbrreasoner.steps.explain.CaseSimilarityComputer.**computeSimilarity** ()		186 068 ms (9,3%)

Figure 6.4: **Distribution of resource-demand in the explain module**

6.5 Presentation to the user

The results from the reasoning are presented through plain text. When the system is done reasoning, it will present to the user an explanation and a solution.

The ideas behind the explanation is described in section 5.5.5. The system presents the query provided, the n[7] cases most similar to the query and the proposed solution.

For each presented case, it explains each feature. This similarity is given as a decimal number, and for the categorical values the path between them is also included. An example of this can be shown from the travel domain as follows[8]:

```
SEASON: similarity is 0.5, through
[[vacation.owl#June, 22-rdf-syntax-ns#type, vacation.owl#SUMMER],
 [vacation.owl#July, 22-rdf-syntax-ns#type, vacation.owl#SUMMER]]
```

How this looks for numerical values is shown in figure 6.5. Perhaps the most important line here is the one describing the proposed solution: *"The case proposed as solution was: CaseID: Case1, isAlarmingSituation: NotAlarmingSituation, with similarity 0.9946666666666667. This is based on the explanation"* If the user's belief in this conclusion is weak, she can easily view the n-1 next most similar cases and their conclusions as well, and what findings that affected the solution.

```
REACTIVELOAD: similarity is 1.0, through [], this case has the finding value 1.7
TIMESTAMP: similarity is 0.2, through [], this case has the finding value 2013-04-14T09:40:00
The case has solution CaseID: Case59, isAlarmingSituation: NotAlarmingSituation

Case: ID146 has similarity 0.920831829497858:
EXCITERFIELDCURRENT: similarity is 1.0, through [], this case has the finding value 1.8
EXCITERFIELDVOLTAGE: similarity is 1.0, through [], this case has the finding value 45
FREQUENCY: similarity is 1.0, through [], this case has the finding value 60.2
GENERATORID: similarity is 1.0, through [], this case has the finding value EE-43-E660
GENERATORPOWER: similarity is 0.8999999999999999, through [], this case has the finding value 3.6
PHASEACURRENT: similarity is 0.9051833122629583, through [], this case has the finding value 351.6
PHASEBCURRENT: similarity is 0.9010554089709762, through [], this case has the finding value 341.6
REACTIVELOAD: similarity is 1.0, through [], this case has the finding value 1.7
TIMESTAMP: similarity is 0.4, through [], this case has the finding value 2013-04-15T00:20:00
The case has solution CaseID: Case147, isAlarmingSituation: NotAlarmingSituation

Case: ID272 has similarity 0.9199138949950355:
EXCITERFIELDCURRENT: similarity is 1.0, through [], this case has the finding value 1.8
EXCITERFIELDVOLTAGE: similarity is 0.9, through [], this case has the finding value 50
FREQUENCY: similarity is 1.0, through [], this case has the finding value 60.2
GENERATORID: similarity is 1.0, through [], this case has the finding value EE-43-E660
GENERATORPOWER: similarity is 1.0, through [], this case has the finding value 3.7
PHASEACURRENT: similarity is 0.9481668773704176, through [], this case has the finding value 363.2
PHASEBCURRENT: similarity is 0.9511873350923484, through [], this case has the finding value 352.8
REACTIVELOAD: similarity is 1.0, through [], this case has the finding value 1.7
TIMESTAMP: similarity is 0.2, through [], this case has the finding value 2013-04-15T21:20:00
The case has solution CaseID: Case273, isAlarmingSituation: NotAlarmingSituation

The case proposed as solution was: CaseID: Case1, isAlarmingSituation: NotAlarmingSituation, with similarity 0.9946666666666667. This is based on the explanation
EXCITERFIELDCURRENT: similarity is 1.0, through []
EXCITERFIELDVOLTAGE: similarity is 0.96, through []
FREQUENCY: similarity is 1.0, through []
GENERATORID: similarity is 1.0, through []
GENERATORPOWER: similarity is 1.0, through []
PHASEACURRENT: similarity is 1.0, through []
PHASEBCURRENT: similarity is 1.0, through []
REACTIVELOAD: similarity is 1.0, through []
TIMESTAMP: similarity is 1.0, through []
The proposed case's description is
EXCITERFIELDCURRENT: 1.8, EXCITERFIELDVOLTAGE: 47, FREQUENCY: 60.2, GENERATORID: EE-43-E660, GENERATORPOWER: 3.7, PHASEACURRENT: 359.1, PHASEBCURRENT: 349.1, REACTIVELOAD: 1.7, TIMESTAMP: 2014-04-13T0
0:00:00,
Your query was
EXCITERFIELDCURRENT: 1.8, EXCITERFIELDVOLTAGE: 45, FREQUENCY: 60.2, GENERATORID: EE-43-E660, GENERATORPOWER: 3.7, PHASEACURRENT: 359.1, PHASEBCURRENT: 349.1, REACTIVELOAD: 1.7, TIMESTAMP: 2014-04-13T0
0:00:00,
 ----------------
The proposed solution for your query is thus CaseID: Case1, isAlarmingSituation: NotAlarmingSituation.
```

Figure 6.5: **Explanation and solution**

[7] In the provided implementation, n is set to 10.

[8] This example is taken from the travel domain, as it contains categorical values suitable for comparison, something the turbine domain does not.

6.5.1 A Colibreek run, step by step - from the user's perspective

We will in this section follow the user through a step-by-step run of Colibreek, on the turbine sensor domain.

Given that the domain is initially chosen - either by code or by user input - to be the turbine sensor domain, the first things presented to the user after she presses run are the mandatory prints enforced by Colibri studio: information about namespaces and classifying, as seen in figure 6.6.

```
14:37:52,085  INFO OntoBridge:215 - Loading Main Ontology: http://www.semanticweb.org/mads/ontologies/2014/2/untitled-ontology-24
14:37:52,610  INFO OntoBridge:227 - Loading Complete
14:37:52,610  INFO OntoBridge:229 - Base Namespace: http://www.semanticweb.org/mads/ontologies/2014/2/untitled-ontology-24#
14:37:52,611  INFO OntoBridge:230 - Namespaces loaded: {=http://www.semanticweb.org/mads/ontologies/2014/2/untitled-ontology-24#, owl=http://www.w3.org/2002/07/owl#, ns=http://creativecommons.org/ns#,
 xsd=http://www.w3.org/2001/XMLSchema#, untitled-ontology-24=http://www.semanticweb.org/mads/ontologies/2014/2/untitled-ontology-24#, rdfs=http://www.w3.org/2000/01/rdf-schema#, ssn=http://purl.oclc.o
rg/NET/ssnx/ssn#, rdf=http://www.w3.org/1999/02/22-rdf-syntax-ns#, terms=http://purl.org/dc/terms/, ace_lexicon=http://attempto.ifi.uzh.ch/ace_lexicon#, DUL=http://www.loa-cnr.it/ontologies/DUL.owl#,
dc=http://purl.org/dc/elements/1.1/}
Classifying 18 elements

Classifying:  5% complete in 00:00
Classifying:  11% complete in 00:00
Classifying:  16% complete in 00:00
Classifying:  22% complete in 00:00
Classifying:  27% complete in 00:00
Classifying:  33% complete in 00:00
Classifying:  38% complete in 00:00
Classifying:  44% complete in 00:00
Classifying:  50% complete in 00:00
Classifying:  55% complete in 00:00
Classifying:  61% complete in 00:00
Classifying:  66% complete in 00:00
Classifying:  72% complete in 00:00
Classifying:  77% complete in 00:00
Classifying:  83% complete in 00:00
Classifying:  88% complete in 00:00
Classifying:  94% complete in 00:00
Classifying:  100% complete in 00:00
Classifying finished in 00:00
Give exciter field voltage
```

Figure 6.6: **Run: Namespaces and classifying**

The user is then allowed to prompt her query - the findings of the new case. If she does not know, she can simply let the text be empty and press enter, then that finding is left undefined, as seen with generator ID in figure 6.7. If she provides a non-recognised input, she is allowed to try again, as seen with frequency in figure 6.7.

The system then does the reasoning, which is usually the most time-consuming part of each run. The print is provided directly from Colibri studio, and is as seen in figure 6.8.

When reasoning is finished, Colibreek repeats your query, to make it easy to manually compare the query with the returned cases. The query is displayed as in figure 6.9.

The system then prints the 10 most similar cases, with description, explanation (if it is a categorical value) and the case's solution. An example of this is seen in figure 6.10.

```
Give exciter field voltage
47
Give generator power
3.6
Give timestamp (Format: YYYY-MM-DDTHH:MM:SS
2013-04-15T00:10:00
Give generator ID

Give frequency
ThisIsANonRecognisedValue
Frequency value not found in ontology, please try again or give empty string to skip.
60.1
Give exciter field current
1.8
Give reactive load
1.7
Give phase A current
302
Give phase B current
292.5
Realizing 18 elements
```

Figure 6.7: **Run: Input from the user**

```
Realizing:  5% complete in 00:00
Realizing:  11% complete in 00:00
Realizing:  16% complete in 00:00
Realizing:  22% complete in 00:00
Realizing:  27% complete in 00:00
Realizing:  33% complete in 00:00
Realizing:  38% complete in 00:00
Realizing:  44% complete in 00:00
Realizing:  50% complete in 00:00
Realizing:  55% complete in 00:00
Realizing:  61% complete in 00:00
Realizing:  66% complete in 00:00
Realizing:  72% complete in 00:00
Realizing:  77% complete in 00:00
Realizing:  83% complete in 00:00
Realizing:  88% complete in 00:00
Realizing:  94% complete in 00:00
Realizing:  100% complete in 00:00
Realizing finished in 00:00
Reasoning finished for new situation
```

Figure 6.8: **Run: Colibri's realizing-print**

```
Reasoning finished for new situation
EXCITERFIELDCURRENT: 1.8, EXCITERFIELDVOLTAGE: 47, FREQUENCY: 60.1, GENERATORPOWER: 3.6, PHASEACURRENT: 302, PHASEBCURRENT: 292.5, REACTIVELOAD: 1.7, TIMESTAMP: 2013-04-15T00:10:00,
This returned the explanation
```

Figure 6.9: **Run: The query displayed to the user**

```
This returned the explanation

Case: ID145 has similarity 0.8655343146902211:
EXCITERFIELDCURRENT: similarity is 1.0, through [], this case has the finding value 1.8
EXCITERFIELDVOLTAGE: similarity is 1.0, through [], this case has the finding value 47
FREQUENCY: similarity is 1.0, through [], this case has the finding value 60.1
GENERATORID: similarity is 0.0, through [], this case has the finding value EE-43-E660
GENERATORPOWER: similarity is 1.0, through [], this case has the finding value 3.6
PHASEACURRENT: similarity is 0.4981036662452595, through [], this case has the finding value 341.7
PHASEBCURRENT: similarity is 0.4934036939313988, through [], this case has the finding value 330.9
REACTIVELOAD: similarity is 1.0, through [], this case has the finding value 1.7
TIMESTAMP: similarity is 1.0, through [], this case has the finding value 2013-04-15T00:10:00
The case has solution CaseID: Case146, isAlarmingSituation: NotAlarmingSituation
```

Figure 6.10: **Run: A case and explanation presented to the user**

Eventually, Colibreek prints the most similar case, and the proposed solution for your query. The findings of both the query and the solution case are also presented, to make it easy for the expert user to at a glance see similarities and differences. The print-style is as in figure 6.11. If no good enough solution is found, the system says so.

```
The case proposed as solution was: CaseID: Case146, isAlarmingSituation: NotAlarmingSituation, with similarity 0.8655343146902211. This is based on the explanation
EXCITERFIELDCURRENT: similarity is 1.0, through []
EXCITERFIELDVOLTAGE: similarity is 1.0, through []
FREQUENCY: similarity is 1.0, through []
GENERATORID: similarity is 0.0, through []
GENERATORPOWER: similarity is 1.0, through []
PHASEACURRENT: similarity is 0.4981036662452595, through []
PHASEBCURRENT: similarity is 0.4934036939313988, through []
REACTIVELOAD: similarity is 1.0, through []
TIMESTAMP: similarity is 1.0, through []
The proposed case's description is
EXCITERFIELDCURRENT: 1.8, EXCITERFIELDVOLTAGE: 47, FREQUENCY: 60.1, GENERATORID: EE-43-E660, GENERATORPOWER: 3.6, PHASEACURRENT: 341.7, PHASEBCURRENT: 330.9, REACTIVELOAD: 1.7, TIMESTAMP: 2013-04-15T0
0:10:00,
Your query was
EXCITERFIELDCURRENT: 1.8, EXCITERFIELDVOLTAGE: 47, FREQUENCY: 60.1, GENERATORPOWER: 3.6, PHASEACURRENT: 302, PHASEBCURRENT: 292.5, REACTIVELOAD: 1.7, TIMESTAMP: 2013-04-15T00:10:00,
 ----------------
The proposed solution for your query is thus CaseID: Case146, isAlarmingSituation: NotAlarmingSituation.


-------------------------------------------------------------
Case id of the most similar case: Case146
alarming: NotAlarmingSituation
```

Figure 6.11: **Run: The proposed solution and solution case**

Chapter 7

Evaluation

To be able to learn from work done, the work should be evaluated, so that the good things are preserved and brought further, while the bad parts are either improved or replaced. We will in this chapter evaluate our process and results. “Evaluation provides a basis for the accumulation of knowledge” [Cohen and Howe, 1988].

7.1 Evaluation through the five-stage model

A five-stage model for how AI research is done is presented in [Cohen and Howe, 1988], along with evaluation aspects for each stage. We will look at each stage and how it fits with our situation. The content in the following tables are all taken from this article.

7.1.1 Stage 1 - task and view

The initial stage one consists of linking the topic to a task, and identify a view on it - that is, a coarse idea on how to do it.

We have throughout the report answered all of these questions - we started the report by justifying our task, we have explained why we consider it important, and have been honest about our system's limitations, scope and intention. Additional research questions have been pointed out explicitly.

1. Is the task significant? Why? 2. Is your research likely to meaningfully contribute to the problem? Is the task tractable? 3. As the task becomes specifically defined for your research, is it still representative of a class of tasks? 4. Have any interesting aspects been abstracted away or simplified? 5. What are the subgoals of the research? What key research tasks will be or have been addressed and solved as part of the project? 6. How do you know when you have successfully demonstrated a solution to the task? Is the task one in which a solution can be demonstrated?

Table 7.1: Questions for stage 1

7.1.2 Stage 2 - refine the view to a method

In this stage, the view from stage one is refined to a particular method - how the task should actually be solved.

1. How is the method an improvement over existing technologies? 2. Does a recognized metric exist for evaluating the performance of your method (for example, is it normative, cognitively valid)? 3. Does it rely on other methods? 4. What are the underlying assumptions? 5. What is the scope of the method? 6. When it cannot provide a good solution, does it do nothing or does it provide bad solutions or does it provide the best solution given the available resources? 7. How well is the method understood? 8. What is the relationship between the problem and the method? Why does it work for this task?

Table 7.2: Questions for stage 2

We have been clear about the scope of our work and our priorities. The system depends especially on the Colibri studio framework, as explained in section 4.5.5.

Our system is based on a recognised existing framework, Creek, and im-

plemented within another recognised framework, Colibri studio. It is thus reasonable to say that the method is good understood. We have in addition discussed our deliberate architectural and implementational choices in their respective chapters.

7.1.3 Stage 3 - implement it

The third stage consists of actually implementing the method described in stage two.

1. How demonstrative is the program?
2. Is it specially tuned for a particular example?
3. How well does the program implement the method?
4. Is the program's performance predictable?

Table 7.3: Questions for stage 3

Our implementation is freely available, so everyone is free to herself look into our implementation and evaluate it. We have done a continuous evaluation by emphasising writing tests, so each feature of the implementation is tested, as is the system as a whole. Our implementation is by choice domain-independent, as described thoroughly in section 5.1.1.

We have done some changes to the original Creek method, including both extensions and simplifications. These are described and discussed in the implementation chapter.

7.1.4 Stage 4 - design experiments

Stage 4 handles the experiments of the newly designed system - this can be seen as a test of the system.

The audience for the system are expert users, as pointed out in section 3.4. The users can be experts on any domain, so the system must be flexible and is domain-independent, exemplified by the two implemented domains, travel recommendation and turbine sensors. The performance criteria for the system naturally varies along the domains, for instance whether real-time is an interesting consideration or not.

1. How many examples can be demonstrated?
2. Should the program's performance be compared to a standard such as another program, or experts and novices, or its own tuned performance? Should the standard be normative, or cognitive validity, or outcomes either from the real world or from simulations?
3. What are the criteria for good performance? Who defines the criteria?
4. Does the program purport to be general (domain-independent)?
5. Is a series of related programs being evaluated?

Table 7.4: Questions for stage 4

Cohen and Howe lists six types of studies that can be used when evaluating:

Comparison	Measures for the program are selected, and both our program and a bottom-line "standard" program solve them. Finally, the measures are compared.
Direct assessment	The system does some work, and then an expert user evaluates the performance
Ablation and substitution	Parts of the system are removed or substituted, enabling analysis of the different parts' role in the system
Tuning	The system is fine-tuned with some data, and tested on other data
Limitation	We test the system at its known limits
Inductive	The system is tested on a new set of problems

Table 7.5: Evaluation types

Of these evaluation types, we evaluate our system with ablation, substitution, tuning and induction. Comparison evaluation proved out of scope - we have no bottom-line system at hand. We consider limitation and direct assessment tests to be more suitable later on, when the system is more completely implemented.

The ablation and substitution evaluation is done in each unit test, where we test specific parts of the system either in isolation or in a stripped-down version of the system.

The tuning and inductive tests will follow in the sections 7.3 and 7.4.

7.1.5 Stage 5 - run experiments

The results from stage four are analysed in this final stage. This stage consists of what is usually considered evaluation.

1. How did program performance compare to its selected standard (for example, other programs, people, normative behavior)?
2. Is the program's performance different from predictions of how the method should perform?
3. How efficient is the program in terms of space and knowledge requirements?
4. Did the program demonstrate good performance?
5. Did you learn what you wanted from the program and experiments?
6. Is it easy for the intended users to understand?
7. Can you define the program's performance limitations?
8. Do you understand why the program works or doesn't work?

Table 7.6: Questions for stage 5

Performance in terms of resource usage has not been prioritised during our work, neither has the user interface. We have by choice made a proof-of-concept where these aspects clearly could be better. However, the system does what it is supposed to do, and gives the expected results.

7.2 Colibri and test-driven development

We have encountered some difficulties using test-driven development while extending an existing framework, Colibri studio. This enforced us to get to know parts of the Colibri framework in depth, and made us discover much of the internal structure of the framework.

Unfortunately, it turned out that crucial parts of Colibri are not optimised for testability - for example, it requires comprehensive rewriting and circum-

vention to write unit tests that do not depend on reading XML files from the file system. Class construction, file reading, setup and initialisation is tangled tightly together. This makes the tests slower, more vulnerable and less pinpointed than they would have been with a better code structure. For every unit test not testing exactly this functionality, it would be better if we could have simulated the file reading.

We did not dive into this refactoring work, and as a consequence, our tests are slower and more vulnerable than desired. However, we have experienced during our progress that the unit tests were very valuable. Whenever we made a change to existing code, the relevant tests ran within seconds, and we could at all times be sure that we did not break existing functionality - and discover this in less than a minute from the change was made.[1]

We draw two major conclusions from this: it can be perfectly fine to do test-driven development inside a non-test-oriented framework, and we had success with test-driven development in this work.

7.3 Adapting Colibreek to a new domain

As Colibreek is emphasised to be domain-independent and flexible towards new domains, it is important to make it easy to adapt to a new domain. We have evaluated the required amount of needed work in order to use Colibreek to help solving problems within another turbine-related field, with the aim of finding turbine AGB failures. These are a type of mechanical failures in a turbine that are different from the ones we have considered so far in this report.

It took us about an hour to make the system run as predicted on this domain, including time fixing errors in the ontology.

This required changes in three existing Java files, the creation of one XML config file, the ontology and case base and a total of five new Java classes. The total amount of new Java lines are approximately 240.

[1] The tools that supported in this are listed in section 2.3.5.

7.4 Turbine domain test results

We have used the leave-one-out-cross-validation strategy to evaluate the results. This strategy involves a rather simple algorithm:

1. Pick and remove one case from the case base
2. Use this case as the query
3. Run the system
4. Check if the solution returned by the system is equal to the case's solution
5. Repeat for every case

We have run this algorithm on our turbine sensor domain, where the case base consists of 423 cases. Of these, the system returned the correct solution in 422 of them. This evaluation took a total of 2241 seconds, or 37 minutes and 35 seconds.

We also ran this evaluation on the new turbine AGB domain, and got results showing 264 successes and 25 failures, as shown in figure 7.1. This indicates that our approach is successful, but that tuning remains, particularly of relevance factors.

```
264
Out of total289
Leaving 25 errors
Success degree: 0.9134948096885813
```

Figure 7.1: **Evaluation results for the turbine AGB domain**

Similar runs for the travel domain return very bad results, which should also be expected: the travel recommender domain makes excessive use of the reuse step and adaptation of the retrieved case, for instance with regards to number of travellers, duration or accomodation level. This adaptation is not present to nearly the same degree for the two turbine domains.

Table 7.7 presents the results of the leave-one-out-runs, as well as the average number of cases activated for every run.

Domain	Total# of cases	Avg # of cases post-activate	Successes	Failures
Travel	125	34	6	119
Turbine sensors	191	423	422	1
Turbine AGB	289	7	264	25

Table 7.7: Leave-one-out-results

Chapter 8

Discussion and conclusion

8.1 Frames vs DL in Colibreek

A central question in our work has been what the consequences would be of replacing frames with description logics in Creek. With Colibri studio, DL is the standard way of representing knowledge for knowledge-intensive CBR systems. Thus, the implementation of DL itself has been trivial, a consequence of Colibri's DL support discussed in section 3.7.2.

To implement frames within Colibri studio would have proven to be a substantially more demanding task.

Our inclusion of direct comparison for numerical values has been an improvement to Creek. Section 5.3.4 highlighted how this functionality has been limited due to the representation. It might be tempting to blame this on DL, but it is due to implementational choices in Colibri.

The most honest answer to our research question on the consequences of replacing frames with DL would be that the consequences are small. Much of this is due to the use of libraries that take care of the direct handling with the ontology. We have experienced that the characteristics of Creek, as listed in 4.1, are perfectly well suitable for a DL representation. We find this theory strengthened by our implementation as well.

The *benefits* of replacing frames with DL in Creek would summarised be:

- Numbers of users - DL is the more widespread representation today
- Colibri support - Colibri supports DL easily

- Defined semantics
- Reusability - ontologies can be reused and merged easily

The major disadvantages are:

- Change - frames is the original representation in Creek, it would have to be replaced
- Frames are flexible

We find the first disadvantage to require some more explanation: in Creek, frames is the initial knowledge representation, and this means that the system is built with this in mind. It also typically means that the vast amount of research assumes frames, so that border cases and tricky questions are examined more thorough with frames than with an alternative. This issue is present for all major changes of a software, but it is easy to forget or to neglect. A conclusion we draw from this fact is that the effect of the replacement should be significant in order to be worth the risk.

8.2 DL in Creek within a reasonable scope?

Our implementation is in itself a proof that it *is* possible to inject description logics in Creek within a reasonable scope. Our angle for dealing with this challenge was that of a proof-of-concept, so there will be other implementational questions and challenges occuring during development intended for production, but the main questions and main challenges are dealt with and can be handled.

That said, a DL implementation requires significant amounts of knowledge to be provided: an ontology and possibly an initial case base. To create an ontology for a domain may be lots of work. The same though counts for creating an equivalent knowledge structure for frames for the same domain. In addition, ontologies can be merged or partly reused, so the amount of required work may be less than initially assumed.

The use of libraries as Pellet, Jena and Ontobridge are preconditions for this conclusion to hold. Were we not to accept these dependencies, a lot more effort would have to be put into this task.

8.3 Decision-support for turbines in DL-Creek

By using this domain as our example domain, we have shown how a DL-based Creek system can be done. The system shows promising results, and it is our belief that Colibreek can, with more hours put into it, turn into a practically useful and value-bringing system for such problems.

Some obstacles must be handled on the way, primarily execution speed. Our system runs with data from a small set of sensors, while a turbine park in real has a lot of such sensors. In order to handle real-time data, the system must be able to run noticeably faster than it is able to do today. The problem with scaling has though been around for CBR systems for a long time, and much research and many techniques have been put into effort to overcome the difficulties.

8.4 Achieving the goal

We have during this work constructed a proof-of-concept partial Creek implementation with description logics as its knowledge representation. This implementation can form the basis for further research or for a more production-oriented version. We have pointed out the directions for how both can be done.

8.5 Conclusion

We have in this project worked to fulfill the goal and answer the research question we listed in section 1.2:

Goal: Study how the core of the Creek system can be adapted to a description logics representation.

Research question 1: What would be the consequences of replacing frames with description logics in Creek? Which consequences will this have, in terms of benefits and limitations?

Research question 2: Can DL be implemented in Creek within a reasonable scope?

Research question 3: How can decision-support for reported turbine failures be implemented within a DL-based Creek system?

We have in this report described the fields of knowledge required to understand our field, and the systems and frameworks we have used. In particular, we have described the Creek system. We have described how we have worked and the results of it, and we have evaluated our work.

We have shown that Creek *can* be implemented with description logics instead of frames, and that the consequences of this are manageable. Section 8.1 discusses the advantages and disadvantages of this switch.

We have provided a proof-of-concept realisation of Creek with DL. In this realisation, central parts of Creek are implemented. We conclude from this that *yes*, it is possible to implement DL in Creek within a reasonable scope. We have used the turbine domain as our domain of choice during implementation and testing, and hence shown how the system can be used for decision-support in this domain.

8.6 Further work

Our work has revealed several lines of potential further work.

More phases

We have described how Colibreek can handle the entire CBR cycle, but the current implementation only handles the retrieve step. To extend Colibreek to also do reuse, retain and possibly revise, would be a substantial improvement.

Revise

We have proposed, in section 4.5, that revise can be included as a step in Colibreek. Further elaboration on how this can be done is required, including specifying how the activate-explain-focus-cycle would fit in. We propose a skeleton of a way revise could be included into Creek, in a six-step algorithm:

1. Add the reused case (temporarily) to the case base
2. Run retrieve again
3. Assure that the correct case is retrieved when re-running. This can possibly be done with a twist

4. Display the retrieved, reused case to the user
5. Let the user accept or decline the solution
6. If the solution is declined, it must be repaired

User interface

A thorough user interface could lift the usability of the system dramatically. This counts for both the presentation of the results and the explanation.

Optimising

To be able to handler bigger amounts of data, work on speeding up the system is needed.

A better algorithm for computing similarity between two cases in the activate step could improve the system's accuracy.

Indexing

The indexing of cases is in Colibreek done through a built-in mechanism in Colibri studio. This differs from the Creek way of indexing[Aamodt, 1991][p.203], adapting Colibreek to use the Creek way of indexing cases is important.

Types of problems

Colibreek is at present intended for diagnosis and problem solving. It would be interesting to see the system directed at other goals - e.g. tutoring or learning.

Transform existing Creek system to DL

Another aspect related to our work is whether an existing Creek system can be ported to use DL within a reasonable scope, including transforming the knowledge base and the cases. We believe that this would be possible, and would like to see a conducted attempt of doing so.

Finish up

Section 4.1 listed our priorities. We have realised most of them, but the remaining ones are still to be implemented.

Relational database

It is worth experimenting with moving the ontology into a relational database. This could very well speed up the application as well as simplifying the structure.

Improve Colibri studio

Particularly section 5.3.4, but also other sections in this report, pointed out how the limitations in Colibri studio restricts us from implementing parts of Creek. We have in this work made a conscious choice of not doing these improvements, but in order to make a more sustainable implementation, Colibri should be improved. The Colibri studio framework should be improved by allowing for the concrete improvements we have pointed out, but also with tests and refactoring.

Replace Colibri studio

For situations where the dependency on and integration into Colibri studio is not desirable, it might be suitable to use Jena and Pellet directly to still have the DL handling. This would give extra work, for instance on replacing connectors and the casebase handling, while on the other hand giving extra flexibility.

Bibliography

Aamodt, A. (1990). Knowledge-intensive case-based reasoning and sustained learning. In *Topics in Case-based reasoning*, pages 274–288. Springer.

Aamodt, A. (1991). *A knowledge-intensive, integrated approach to problem solving and sustained learning*. PhD thesis, University of Trondheim, Norwegian Institute of Technology (NTH).

Aamodt, A. (1994a). Explanation-driven case-based reasoning. In *Topics in case-based reasoning*, pages 274–288. Springer Verlag.

Aamodt, A. (1994b). A knowledge representation system for integration of general and case-specific knowledge. In *Tools with Artificial Intelligence, 1994. Proceedings., Sixth International Conference on*, pages 836–839. IEEE.

Aamodt, A. (2004). Knowledge-intensive case-based reasoning in CREEK. In *Advances in case-based reasoning, 7th European Conference, EC-CBR 2004*.

Aamodt, A. and Langseth, H. (1998). Integrating bayesian networks into knowledge-intensive cbr. In *AAAI Workshop on Case-Based Reasoning Integrations*.

Aamodt, A. and Plaza, E. (1994). Case-based reasoning; foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59.

Aamodt, A. and Skalle, P. (2004). Knowledge-based decision support in oil well drilling; combining general and case-specific knowledge for problem solving. In *Proceedings of ICIIP-2004, International Conference on Intelligent Information Processing*.

Abdrabou, E. A. M. L. and Salem, A.-B. M. (2008). Case-based reasoning tools from shells to object-oriented frameworks. In *Proceedings of the 12th*

WSEAS international conference on Computers, pages 781–786. World Scientific and Engineering Academy and Society (WSEAS).

Abásolo, C., Plaza, E., and Arcos, J.-L. (2002). Components for case-based reasoning systems. In Escrig, M., Toledo, F., and Golobardes, E., editors, *Topics in Artificial Intelligence*, volume 2504 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg.

Aha, D. W., Kibler, D., and Albert, M. K. (1991). Instance-based learning algorithms. *Machine learning*, 6:37–66.

Ambler, S. (2002-13). Introduction to test-driven development (TDD). `http://www.agiledata.org/essays/tdd.html`, permalink `http://perma.cc/56RR-2EK9`. Last accessed 18th of February 2014.

Atwood, J. (2012). *Effective Programming: More Than Writing Code*. Hyperink Programming and Software Engineering Books.

Baader, F., Horrocks, I., and Sattler, U. (2005). Description logics as ontology languages for the semantic web. In Hutter, D. and Stephan, W., editors, *Mechanizing Mathematical Reasoning*, volume 2605 of *Lecture Notes in Computer Science*, pages 228–248. Springer Berlin Heidelberg.

Baader, F. and Nutt, W. (2003). Basic description logics. In Baader, F., Calvanese, D., McGuinness, D., Nardi, D., and Patel-Schneider, P., editors, *The description logic handbook*, pages 43–95.

Bach, K. and Althoff, K.-D. (2012). Developing case-based reasoning applications using myCBR 3. In Agudo, B. and Watson, I., editors, *Case-Based Reasoning Research and Development*, volume 7466 of *Lecture Notes in Computer Science*, pages 17–31. Springer Berlin / Heidelberg.

Beck, K. and Andres, C. (2004). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

Bello-Tomás, J. J., González-Calero, P. A., and Díaz-Agudo, B. (2004). jcolibri: An object-oriented framework for building CBR systems. In Funk, P. and González Calero, P. A., editors, *Advances in Case-Based Reasoning*, volume 3155 of *Lecture Notes in Computer Science*, pages 29–39. Springer Berlin / Heidelberg.

Bengtsson, M., Olsson, E., Funk, P., and Jackson, M. (2004). Technical design of condition based maintenance system-a case study using sound

analysis and case-based reasoning. In *Proceedings of the 8th International Conference of Maintenance and Reliability*. University of Tennessee–Maintenance and Reliability Center, Knoxville, USA.

Bergmann, R., Breen, S., Fayol, E., Göker, M., Manago, M., Schmitt, S., Schumacher, J., Stahl, A., Wess, S., and Wilke, W. (1998). Collecting experience on the systematic development of CBR applications using the INRECA methodology. In *Advances in Case-Based Reasoning – 4th European Workshop, EWCBR-98 Dublin, Ireland, September 23–25, 1998 Proceedings*. Springer Berlin/Heidelberg.

Bogaerts, S. and Leake, D. (2005). IUCBRF: A framework for rapid and modular case-based reasoning system development – report version 1.0. Technical report, Indiana University.

Boroditsky, L. (2011). How language shapes thought. *Scientific American*, 304(2):62–65.

Brede, T., Sørmo, F., Aamodt, A., and Bø, K. (2006). *TrollCreek Tutorial - A Knowledge Modeling Editor and Testing Environment for Knowledge-Intensive Case-Based Reasoning*. Available at `http://www.ipt.ntnu.no/~pskalle/files/pt8500/TrollCreekTutorial.doc`, permalink `http://perma.cc/N7AG-LVV6` last accessed 26th of February 2014.

Brodwall, J. (2014). The economics of reuse. `http://johannesbrodwall.com/2014/03/24/the-economics-of-reuse`, permalink `http://perma.cc/3TX6-Z9S7`. Last accessed 26th of March 2014.

Brown, S. (2014). Five things every developer should know about software architecture. `http://www.codingthearchitecture.com/2014/03/05/five_things_every_developer_should_know_about_software_architecture.html`, permalink `http://perma.cc/PFZ2-B9J7`. Last accessed 25th of March 2014.

Bruland, T., Aamodt, A., and Langseth, H. (2010). Architectures integrating case-based reasoning and bayesian networks for clinical decision support. In *Intelligent Information Processing V*, pages 82–91. Springer.

Bruland, T., Aamodt, A., and Langseth, H. (2011). A hybrid CBR and BN architecture refined through data analysis. In *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*, pages 906–913. IEEE.

Cohen, P. R. and Howe, A. E. (1988). How evaluation guides ai research: The message still counts more than the medium. *AI Magazine*, 9(4):35.

Davis, R., Shrobe, H., and Szolowits, P. (1993). What is a knowledge representation? *AI Magazine*, 14(1):17–33.

Díaz-Agudo, B., González-Calero, P. A., and Informáticos, D. S. (2000). An architecture for knowledge intensive CBR systems. In *Advances in CaseBased Reasoning – (EWCBR00)*, pages 37–48. Springer-Verlag.

Díaz-Agudo, B., González-Calero, P. A., and Recio-García, J. A. (2008). jCOLIBRI2 tutorial.

Fensel, D. (2001). *Ontologies: A silver bullet for knowledge management and electronic commerce*. Springer.

Fidjeland, M. K. (2006). Distributed knowledge in CBR - knowledge sharing and reuse within the semantic web. Master's thesis, NTNU.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the design of existing programs*. Addison-Wesley Reading.

Francis, A. and Ram, J. A. (1993). Computational models of the utility problem and their application to a utility analysis of case-based reasoning. In *In Proceedings of the Workshop on Knowledge Compilation and Speed-Up Learning*, pages 48–55.

GAIA (2014). Colibri studio. `http://gaia.fdi.ucm.es/research/colibri/colibristudio`, permalink `http://perma.cc/4CBV-ZJCC`. Last accessed 18th of March 2014.

Gill, D. (2009). Protecting critical machinery - the value of a complete solution. *Maintenance & Equipment*.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220.

Hitzler, P., Krotzsch, M., and Rudolph, S. (2011). *Foundations of semantic web technologies*. CRC Press.

Jaczynski, M. and Trousse, B. (1998). An object-oriented framework for the design and the implementation of case-based reasoners. In *In 6Th German Workshop on Case-Based Reasoning*, pages 33–42.

Jardine, A. K., Lin, D., and Banjevic, D. (2006). A review on machinery diagnostics and prognostics implementing condition-based maintenance. *Mechanical systems and signal processing*, 20(7):1483–1510.

Kolodner, J. (1993). *Case-based reasoning*. Morgan Kaufmann Publishers.

Kravis, S. and Irrgang, R. (2005). A case based system for oil and gas well design with risk assessment. *Applied Intelligence*, 23(1):39–53.

Kukuric, N., Robijn, F., and Griffioen, J. (2008). The i3S document series: Using case based reasoning for the solution of water stress problems. Technical report, Aquastress.

Lalanne, C. (2008). An introduction to knowledge engineering. `http://www.aliquote.org/articles/tech/KnowledgeEngineering/`, permalink `http://perma.cc/A7RA-Q72L`. Last accessed 14th of February 2014.

Leake, D. B. (1996). *Case-Based Reasoning: Experiences, Lessons and Future Directions*. MIT Press, Cambridge, MA, USA, 1st edition.

Lehmann, F. (1992). Semantic networks. *Computers & Mathematics with Applications*, 23(2–5):1–50.

Lillehaug, M. B. (2011). Explanation-aware case based reasoning. Master's thesis, NTNU.

Martín, A. and León, C. (2010). Expert knowledge management based on ontology in a digital library. In *ICEIS (2)*, pages 291–298.

Mendes, J. R. P., Morooka, C. K., and Guilherme, I. R. (2003). Case-based reasoning in offshore well design. *Journal of Petroleum Science and Engineering*, 40(1–2):47 – 60.

Mikos, W. L., Ferreira, J. C. E., and Gomes, F. G. C. (2011). A distributed system for rapid determination of nonconformance causes and solutions for the thermoplastic injection molding process: A case-based reasoning agents approach. In *Automation Science and Engineering (CASE), 2011 IEEE Conference on*, pages 755–760. IEEE.

Minsky, M. (1975). A framework for representing knowledge. In Winston, P., editor, *The psychology of computer vision*, pages 211–277. McGraw-Hill.

Nebel, B. (1999). Frame-based systems. *MIT Encyclopedia of the Cognitive Sciences, MIT Press, Cambridge, MA*, pages 324–325.

Opheim, M. (2012). CREEK - what have we learned? NTNU, available at https://dl.dropboxusercontent.com/u/577581/Mads%20Opheim%20-%20Creek%20-%20what%20have%20we%20learned.pdf, permalink http://perma.cc/UM3K-AVYG.

Osherove, R. (2011). Unit test-definition. http://artofunittesting.com/definition-of-a-unit-test/, permalink http://perma.cc/9FEG-MCJG. Last accessed 26th of March 2014.

Pla, A., López, B., Gay, P., and Pous, C. (2013). eXiT* CBR. v2: Distributed case-based reasoning tool for medical prognosis. *Decision Support Systems*, 54(3):1499–1510.

Popa, A. S., Popa, C. G., Malamma, M., and Hicks, J. (2008). Case-based reasoning approach for well failure diagnostics and planning. In *SPE Western Regional and Pacific Section AAPG Joint Meeting*. Society of Petroleum Engineers.

Recio-García, J. A. and Díaz-Agudo, B. (2004). An introductory user guide to jcolibri 0.3. Technical Report 144, Dep. Sistemas Informáticos y Programación,Universidad Complutense de Madrid,Spain.

Recio-García, J. A. and Díaz-Agudo, B. (2007). Ontology based cbr with jcolibri. In *Applications and Innovations in Intelligent Systems XIV*, pages 149–162. Springer.

Recio-García, J. A., Díaz-Agudo, B., and González-Calero, P. A. (2013). The COLIBRI open platform for the reproducibility of CBR applications. In *Case-Based Reasoning Research and Development*, pages 255–269. Springer.

Recio-Garcıa, J. A., Dıaz-Agudo, B., Sanchez-Ruiz, A. A., and González-Calero, P. A. (2006). Lessons learnt in the development of a CBR framework. In *Proceedings of the 11th UK CBR Workshop*, pages 60–71.

Recio-García, J. A., González-Calero, P. A., and Díaz-Agudo, B. (2014). jColibri2: A framework for building case-based reasoning systems. *Science of Computer Programming*, 0(79):126–145.

Roth-Berghofer, T., Garcia, J. A. R., Sauer, C. S., Bach, K., Althoff, K.-D., Diaz-Agudo, B., and Calero, P. A. G. (2012). Building case-based reasoning applications with mycbr and colibri studio. In Petridis, M., Roth-Berghofer, T., and Wiratunga, N., editors, *Proceedings of the 17th UK Workshop on Case-Based Reasoning. UK Workshop on Case-Based*

Reasoning (UKCBR-12), located at SGAI International Conference on Artificial Intelligence, December 11, Cambridge, United Kingdom, pages 71–82. School of Computing, Engineering and Mathematics, University of Brighton, UK.

Schank, R. (1982). *Dynamic memory: A theory of learning in people and computers*. Cambridge: Cambridge University Press.

Shokouhi, S. V., Aamodt, A., and Skalle, P. (2010). Applications of CBR in oil well drilling: A general overview. In Shi, Z., Vadera, S., Aamodt, A., and Leake, D., editors, *Intelligent Information Processing V*, volume 340 of *IFIP Advances in Information and Communication Technology*, pages 102–111. Springer Berlin Heidelberg.

Shokouhi, S. V., Aamodt, A., Skalle, P., and Sørmo, F. (2009). Comparing two types of knowledge-intensive CBR for optimized oil well drilling. In *IICAI*, pages 722–737.

Skalle, P., Aamodt, A., and Gundersen, O. E. (2013a). Experience transfer for process improvement. *Engineering Applications of Artificial Intelligence*, 26(9):2206–2214.

Skalle, P., Aamodt, A., Gundersen, O. E., et al. (2013b). Detection of symptoms for revealing causes leading to drilling failures. *SPE Drilling & Completion*, 28(02):182–193.

Skalle, P., Sørmo, F., Aamodt, A., and Gundersen, O. E. (2013c). A real-time decision support system for high cost oil-well drilling operations. *AI Magazine*, 34(1):21–32.

SKF (2014). Condition monitoring. `http://www.skf.com/group/products/condition-monitoring/index.html`, permalink `http://perma.cc/YV3X-K53R`. Last accessed 10th of March 2014.

Smyth, B. and Cunningham, P. (1996). The utility problem analysed. In Smith, I. and Faltings, B., editors, *Advances in Case-Based Reasoning*, volume 1168 of *Lecture Notes in Computer Science*, pages 392–399. Springer Berlin Heidelberg.

Smyth, B., Keane, M. T., and Cunningham, P. (2001). Hierarchical case-based reasoning integrating case-based and decompositional problem-solving techniques for plant-control software design. *Knowledge and Data Engineering, IEEE Transactions on*, 13(5):793–812.

Sørmo, F., Cassens, J., and Aamodt, A. (2005). Explanation in case-based reasoning–perspectives and goals. *Artificial Intelligence Review*, 24(2):109–143.

Sowa, J. F. (1992/2013). Semantic networks. `http://www.jfsowa.com/pubs/semnet.htm`, permalink `http://perma.cc/7JNX-QPQ6`. Last accessed 14th of February 2014.

Stahl, A. and Roth-Berghofer, T. (2008). Rapid prototyping of CBR applications with the open source tool myCBR. In Althoff, K.-D., Bergmann, R., Minor, M., and Hanft, A., editors, *Advances in Case-Based Reasoning*, Lecture Notes in Computer Science, pages 615–629. Springer Berlin / Heidelberg.

Stiklestad, E. (2007). Reusing external library components in the Creek CBR system. Master's thesis, NTNU.

Sørmo, F. (2000). Plausible inheritance; semantic network inference for case-based reasoning. Master's thesis, NTNU, Department of Computer and Information Science.

Verdande (2014a). Financial services. `http://www.verdandetechnology.com/Financial-Services/`, permalink `http://perma.cc/HV95-FKTB`. Last accessed 12th of March 2014.

Verdande (2014b). Healthcare. `http://www.verdandetechnology.com/Healthcare/`, permalink `http://perma.cc/F3X6-Y55U`. Last accessed 12th of March 2014.

W3C (2012a). *OWL 2 Web Ontology Language Direct Semnatics (Second Edition)*. Available at `http://www.w3.org/TR/owl2-direct-semantics/`, permalink `http://perma.cc/3LH-AVKB`, last accessed 24th of March 2014.

W3C (2012b). *OWL 2 Web Ontology Language Primer (Second Edition)*. Available at `http://www.w3.org/TR/2012/REC-owl2-primer-20121211/`, permalink `http://perma.cc/7LJG-TLQZ`, last accessed 24th of March 2014.

W3C (2012c). *OWL 2 Web Ontology Language Profiles (Second Edition)*. Available at `http://www.w3.org/TR/owl2-profiles/`, permalink `http://perma.cc/SN49-5QBJ`, last accessed 24th of March 2014.

Wang, H., Noy, N., Rector, A., Musen, M., Redmond, T., Rubin, D., Tu, S., Tudorache, T., Drummond, N., Horridge, M., et al. (2006). Frames and

OWL side by side. In *9th international Protégé conference – Presentation Abstracts*, pages 54–57.

Yang, B.-S., Han, T., and Kim, Y.-S. (2004a). Integration of art-kohonen neural network and case-based reasoning for intelligent fault diagnosis. *Expert Systems with Applications*, 26(3):387–395.

Yang, B.-S., Kwon Jeong, S., Oh, Y.-M., and Tan, A. C. C. (2004b). Case-based reasoning system with petri nets for induction motor fault diagnosis. *Expert Systems with Applications*, 27(2):301–311.

Appendix A

Licenses

This report is lisenced under a license as given on page II.

The code we have produced ourselves is licensed under a GNU Lesser General Public License, version 3.0.

The details are available at `http://opensource.org/licenses/LGPL-3.0`.

Attached code libraries are licensed according to its respective developers.

The entire code base is available at `https://bitbucket.org/madsop/colibreek-public`. At this page, you can also easily find an easy-runnable, non-setup version of Colibreek.

Appendix B

How to run

When the code is obtained from the resource given in appendix A, the code can be run in two ways: either via the pre-made runnable jar-file, or by importing the Java project into an IDE and running from there.

Bitbucket uses the version control system Mercurial, and the instructions for retrieving the project to your computer is given at the Bitbucket page. This can be done from within Eclipse using the plugin MercurialEclipse. All of these tools are described briefly in section 2.3.5.

The easiest is however to download the runnable jar-file. All the source code and config files[1], as well as the required external code-libraries, are included in this file. To run it, browse to the folder containing the file, and run it with *java -jar colibreek.jar*.

The system is configured to choose domain based on the domain passed by the user. If no parameter is given, it defaults to the travel recommender domain. This means that if you want to run the system on the turbine sensor domain, you write *java -jar colibreek.jar TurbineSensor*. However, as the turbine sensor data are not included, this will not be a success.

[1]Except for the data files from ConocoPhillips, which we are not allowed to distribute