Master’s thesis

Implementation of a Rough Knowledge Base System Supporting Quantitative Measures

by
Robin Andersson

LITH-IDA-EX–04/056–SE

2004-06-11
Master’s thesis

Implementation of a Rough Knowledge Base System Supporting Quantitative Measures

by
Robin Andersson

LiTH-IDA-EX–04/056–SE

Supervisor: Aida Vitória
Department of Science and Technology
at Linköpings universitet

Examiner: Prof. Jan Małuszyński
Department of Computer and Information Science at Linköpings universitet
Abstract

This thesis presents the implementation of a knowledge base system for rough sets [Paw92] within the logic programming framework. The combination of rough set theory with logic programming is a novel approach. The presented implementation serves as a prototype system for the ideas presented in [VDM03a, VDM03b]. The system is available at “http://www.ida.liu.se/rkbs”.

The presented language for describing knowledge in the rough knowledge base caters for implicit definition of rough sets by combining different regions (e.g. upper approximation, lower approximation, boundary) of other defined rough sets. The rough knowledge base system also provides methods for querying the knowledge base and methods for computing quantitative measures.

We test the implemented system on a medium sized application example to illustrate the usefulness of the system and the incorporated language. We also provide performance measurements of the system.

Keywords: Rough set theory, rough sets, logic programming, knowledge bases, artificial intelligence, uncertain reasoning, incomplete reasoning, quantitative measures.
Acknowledgments

I would like to thank my supervisor Aida Vitória for helpful discussions on the theoretical foundations and for tirelessly providing remarks on this thesis. My examiner Jan Maluszyński has been very helpful, and the encouragement from him has driven me to do my best. I would also like to thank my opponent Jakob Henriksson for numerous discussions on implementation and theoretical issues. Last, but not least, I would like to thank my family and friends for great support during this project.
Contents

1 Introduction 1
  1.1 Background ................................. 1
  1.2 Objective ................................. 2
  1.3 Intended audience ........................... 3
  1.4 Structure of this thesis ...................... 3

2 Rough Set Theory 5
  2.1 Introduction ............................... 5
  2.2 Basic notions ............................... 6
    2.2.1 Information systems .................... 6
    2.2.2 Indiscernibility ....................... 8
    2.2.3 Set approximations ..................... 9
  2.3 Reducts ................................... 12
  2.4 Decision rules ............................. 14
    2.4.1 Quantitative measures .................. 15
    2.4.2 Decision synthesis ..................... 17
  2.5 Further readings .......................... 19

3 Rough Knowledge Bases 21
  3.1 Rough knowledge base systems ............... 21
  3.2 Rough sets within the logic programming framework .... 22
    3.2.1 Logic programs ........................ 23
    3.2.2 Viewing decision systems as logic facts ........ 25
    3.2.3 Defining rough relations with logic rules ....... 27
3.3 The rough language ................................. 28

4 Implementation ................................ 35
  4.1 Design choices ....................................... 35
     4.1.1 Prolog system .................................... 35
     4.1.2 Language modifications ............................ 36
  4.2 System overview ...................................... 36
  4.3 Prolog implementation ............................... 40
     4.3.1 Prolog server .................................... 40
     4.3.2 Compilation of rough clauses ...................... 41
     4.3.3 Evaluation of rough queries ....................... 41
     4.3.4 Scanner ........................................... 44
     4.3.5 Rewriting rough statements into Prolog code ...... 44
  4.4 User interface ......................................... 53

5 Application Example ..................................... 57
  5.1 Prognostic problem .................................... 57
  5.2 Data ................................................ 58
  5.3 Classification of patients ............................. 59
  5.4 Approximating rough relations via VPRSM .......... 60
  5.5 Avoiding scintigraphic scan .......................... 63

6 Discussion ............................................... 67
  6.1 Portability ............................................. 67
  6.2 Preserving consistency ............................... 68
  6.3 Performance measurements ............................ 69
     6.3.1 Compilation of rough clauses ...................... 70
     6.3.2 Evaluation of rough queries ....................... 72
  6.4 Conclusions ............................................ 76
  6.5 Future work ............................................ 76

A The Deathmi Decision System .......................... 79
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>An information table.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>A decision table.</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>An example with 45 potential flu patients.</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>Computed quantitative measures for the induced decision rules.</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>A decision table with its corresponding collection of rough facts.</td>
<td>27</td>
</tr>
<tr>
<td>4.1</td>
<td>Number of lines in the core parts of the system implementation.</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Attribute definitions.</td>
<td>58</td>
</tr>
<tr>
<td>5.2</td>
<td>Indiscernibility classes that are both in the lower approximation of the explicit negation of DeathmiApprox and in the upper approximation of Deathmi.</td>
<td>63</td>
</tr>
<tr>
<td>5.3</td>
<td>Migrating patients</td>
<td>66</td>
</tr>
<tr>
<td>6.1</td>
<td>The best, worst, and average computation time for different compilation requests.</td>
<td>72</td>
</tr>
<tr>
<td>6.2</td>
<td>The best, worst, and average computation time for the evaluation of non-ground rough queries.</td>
<td>75</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Equivalence classes in the universe and their relationship with the concept $X$ ........................................ 10
2.2 Reduction of attributes in an information system ............ 12

4.1 System overview ........................................... 37
4.2 Fact DCG .................................................. 47
4.3 Upper clause DCG ........................................ 47
4.4 Lower clause DCG ......................................... 48
4.5 Rough literals DCG ....................................... 50
4.6 Support DCG ............................................... 51
4.7 Strength DCG ............................................... 51
4.8 Coverage DCG ............................................. 52
4.9 Accuracy DCG ............................................. 52
4.10 Compilation of rules ....................................... 53
4.11 Evaluation of queries ..................................... 54
4.12 Knowledge base viewer ................................... 56
Chapter 1

Introduction

This master’s thesis was written as the final project of the Computer Science Program for the fulfillment of a Master of Science in Computer Science at the University of Linköping, Sweden. All work presented in this paper was performed at the Theoretical Computer Science Laboratory (TCSLAB), Department of Computer and Information Science (IDA). The examiner was professor Jan Małuszynski\textsuperscript{1} and it was supervised by Aida Vitória\textsuperscript{2}. The result of this work, in form of an implemented rough knowledge base system, is accessible on the web page: “http://www.ida.liu.se/rkbs”.

1.1 Background

\textit{Rough set theory} was developed by Zdzislaw Pawlak in the beginning of the 1980s [Paw82]. It is an extension of set theory, that makes it possible to deal with uncertainty and vagueness in the classification of objects. The theory has grown very popular and it has been a powerful tool in numerous application areas. However, the existing rough set techniques and software systems based on them usually do not provide natural support

\textsuperscript{1}Department of Computer and Information Science, University of Linköping.

\textsuperscript{2}Department of Science and Technology, University of Linköping.
for incorporation of background knowledge. Moreover, useful problem specific techniques introduced in rough set literature in an “ad hoc” way lack the generality for being applied to other problems. There is thus a need for a more general framework extending basic rough set theory [VDM03a]. To address this problem, [VDM03a, VDM03b] define a language based on rough set notions and integrated within the logic programming paradigm. Viewing decision tables as a set of logic facts gives a basis for extending rough sets (rough relations) to rough logic programs. Within the logic programming framework it is possible to define new rough relations implicitly by logic rules. The expressions “rough set” and “rough relation” are used interchangeably in this work.

1.2 Objective

The aim of this master project is to implement a system supporting rough knowledge bases based on the ideas described in [VDM03a, VDM03b] and briefly summarized in chapter 3 of this thesis. The implemented system shall support the following.

- Definition of rough relations (or rough sets) by sets of rough facts

- Definition of new rough relations by combining different regions (e.g. lower approximations, upper approximations, or boundaries) of other defined rough relations. The ability of incorporating quantitative measures when defining new rough relations shall also be supported.

- Querying information about rough knowledge, such as concept classifications, computation of quantitative measures, and computation of indiscernibility classes in rough regions.

Moreover, the system shall also

- be easy to use for people not familiar with the logic programming paradigm, and

- be developed with focus on usability.
1.3 Intended audience

This thesis is intended for people literate in basic computer science and mathematical notations. No prior knowledge of rough set theory is needed to understand the contents. Some background knowledge of mathematical logic and logic programming will be helpful for better understanding of the integration of rough set theory within the logic programming framework. However, useful references to logic and logic programming are given.

1.4 Structure of this thesis

This thesis includes the following chapters:

Rough Set Theory In this chapter we cover the basic notions of rough set theory needed for the rest of the thesis. We discuss the concepts of decision systems, indiscernibility, rough approximations and decision rules. We also give a brief overview on reducts and reduct computation.

Rough Knowledge Bases The focus of this chapter is on the integration of rough set theory within the logic programming framework. The concepts of rough knowledge bases and rough knowledge base systems are introduced. We discuss how rough sets can be seen as a collection of logic facts and how it is possible to define rough relations by other rough relations using logic rules. The chapter ends with the the syntax of a proposed language.

Implementation The implementation chapter covers the main contribution of this project. We present the implementation of a rough knowledge base system in Prolog. We provide definite clause grammars for the rewriting of statements in the rough language to internal Prolog code and present the implemented user interface.

Application Example In this chapter we provide examples that show the use of the implemented rough knowledge base system and the power of the language.
1.4. Structure of this thesis

Discussion The thesis ends with the discussion chapter. We discuss pros and cons of the implementation choices and provide performance measurements of our implemented system.
Chapter 2

Rough Set Theory

During the past two decades a rapid growth of interest for rough set theory has emerged. Research groups have adopted the theory, integrating it in many wide-spread research areas. Many real world applications have also been developed. This chapter gives an introduction to rough set theory which constitutes a foundation for the work that will be presented in the following chapters.

2.1 Introduction

Rough set theory [Paw82, KPPS98, Paw92, Paw97, PS00, SP97] was developed in the beginning of the 1980s by Zdzislaw Pawlak. The development of this theory coincided with the surge of interest in areas such as artificial intelligence, machine learning, pattern recognition, and expert systems. These foundations were mainly focusing on designing algorithms to deal with practical problems related to machine reasoning, perception or learning. The origin of rough set theory during this period of time turned out to be the missing link in many of the above mentioned areas and many researchers integrated the theory in their research. A number of applications has since been developed and the original theory has been extended by several researchers. As a practical tool, it has been witnessed to be a
Rough set theory is an extension of mathematical set theory. The purpose of rough set theory is to consider uncertainty in the classification of objects. In mathematical set theory, membership in a set is defined such that each object in the considered universe either belongs to the set or to its complement. In reality, the available information about a given object is often not sufficient for its definite classification. Many applications of artificial intelligence, for example, deal with sets which are either not fully known or very complex to represent. The rough set theory makes formal analysis of such situations possible.

2.2.1 Information systems

In data analysis one can represent knowledge about objects as an information system [KPPS98, PS00].

**Definition 2.2.1 (Information system):**

An information system is a pair \( \mathcal{I} = (U, A) \), where \( U \) is a finite non-empty set of objects, called the universe, and \( A \) is a finite non-empty set of attributes. Subsets of \( U \) are often called concepts.

Information about objects is represented by a set of attributes with associated values. An attribute \( \alpha \in A \) is a partial function \( \alpha : U \to V_{\alpha} \), where \( V_{\alpha} \) is the value set for \( \alpha \). An information system can be represented by an information table, where the rows in the table are objects in the universe and the columns correspond to the attributes.

Consider, as an example, the information table 2.1, where \( U = \{p1, p2, p3, p4, p5, p6\} \) is a set of patients and \( A = \{\text{headache}, \text{muscle pain}, \text{temperature}\} \) are the attributes corresponding to the symptoms of a patient. Every row can be seen as information about a specific patient. For example, patient \( p5 \) is characterized by the attribute value set \( \{\text{(headache,yes)}, \text{(muscle pain,no)}, \text{(temperature, high)}\} \).
An information table can be seen as a set of training examples in machine learning. Each training example is then connected with a decision that classifies the example into a predefined class. To cope with this, an information system is extended with a set of decision attributes [KPPS98, PS00].

**Definition 2.2.2 (Decision system):**
An information system $I$ extended with a set of decision attributes $D$, such that $I = (U, A, D)$ and $D \cap A = \emptyset$, is called a decision system.

In a decision system, the attributes in $A$ are called conditional attributes. Decision attributes may take several values, though binary outcomes are rather frequent. Decision systems are often represented by decision tables.

<table>
<thead>
<tr>
<th>headache</th>
<th>muscle pain</th>
<th>temperature</th>
<th>flu</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>no</td>
<td>yes</td>
<td>high</td>
</tr>
<tr>
<td>p2</td>
<td>yes</td>
<td>no</td>
<td>high</td>
</tr>
<tr>
<td>p3</td>
<td>yes</td>
<td>yes</td>
<td>very high</td>
</tr>
<tr>
<td>p4</td>
<td>no</td>
<td>yes</td>
<td>normal</td>
</tr>
<tr>
<td>p5</td>
<td>yes</td>
<td>no</td>
<td>high</td>
</tr>
<tr>
<td>p6</td>
<td>no</td>
<td>yes</td>
<td>very high</td>
</tr>
</tbody>
</table>

Table 2.1: An information table.

<table>
<thead>
<tr>
<th>headache</th>
<th>muscle pain</th>
<th>temperature</th>
<th>flu</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>no</td>
<td>yes</td>
<td>high</td>
</tr>
<tr>
<td>p2</td>
<td>yes</td>
<td>no</td>
<td>high</td>
</tr>
<tr>
<td>p3</td>
<td>yes</td>
<td>yes</td>
<td>very high</td>
</tr>
<tr>
<td>p4</td>
<td>no</td>
<td>yes</td>
<td>normal</td>
</tr>
<tr>
<td>p5</td>
<td>yes</td>
<td>no</td>
<td>high</td>
</tr>
<tr>
<td>p6</td>
<td>no</td>
<td>yes</td>
<td>very high</td>
</tr>
</tbody>
</table>

Table 2.2: A decision table.
In decision table 2.2, we extend information table 2.1 with the decision attribute \textit{flu}, i.e. \( D = \{ \text{flu} \} \). The value of the decision attribute shows the diagnosis of a patient, i.e. whether or not the patient has the disease flu. The example originates from [Paw97].

**Definition 2.2.3 (Decision class):**

Let \( I = (U, A, D) \) be a decision system. Every \( d_i \in D \) partitions the universe \( U \) in \( |V_{d_i}| \) classes \( X_1, \ldots, X_k \). Each class \( X_j (j \in \{1, \ldots, |V_{d_i}|\}) \) is called a decision class.

### 2.2.2 Indiscernibility

Objects that have the same values of the conditional attributes are called indiscernible (inseparable). Patients, for example, can have the same set of symptoms but different diagnoses. For instance, patients \( p2 \) and \( p5 \) in decision table 2.2 are examples of such a situation. Rough set theory takes into account indiscernibility between objects through the notion of an indiscernibility relation [KPPS98, PS00]. The indiscernibility relation is used to describe the fact that it may not be possible to separate certain objects in the universe by using the information given by the attributes.

**Definition 2.2.4 (Indiscernibility relation):**

Let \( I = (U, A) \) be an information system and let \( B \subseteq A \). The indiscernibility relation \( IND_I(B) \) is defined as:

\[
IND_I(B) = \{(x, x') \in U^2 | \forall \alpha \in B, \alpha(x) = \alpha(x') \}.
\]

If \( (x, x') \in IND_I(B) \), then \( x \) and \( x' \) are indiscernible with respect to the attributes in \( B \). The subscript \( I \) in \( IND_I(B) \) is often omitted if it is clear which information system we have in mind.

Note that the indiscernibility relation is reflexive, i.e. an object in \( U \) is indiscernible from itself. It is also symmetric, i.e. if \( (x, x') \in IND(B) \) then \( (x', x) \in IND(B) \). Moreover, it is transitive, i.e. if \( (x, x') \in IND(B) \) and \( (x', x'') \in IND(B) \) then \( (x, x'') \in IND(B) \). Relations with these characteristics are called equivalence relations. The equivalence class of an object \( x \in U \) consists of all objects \( y \in U \) such that \( (x, y) \in IND(B) \). The equivalence classes obtained from \( IND(B) \) are denoted by \([x]_B\), with \( x \in U \).
From information table 2.1 we have that:

\[ \text{IND}(\{\text{headache}\}) = \{\{p_1, p_4, p_6\}, \{p_2, p_3, p_5\}\} , \]
\[ \text{IND}(\{\text{musclepain}\}) = \{\{p_1, p_3, p_4, p_6\}, \{p_2, p_5\}\} , \]
\[ \ldots \]
\[ \text{IND}(\{\text{headache, musclepain, temperature}\}) = \{\{p_1\}, \{p_2, p_5\}, \{p_3\}, \{p_4\}, \{p_6\}\} . \]

In the last case above, the patients \( p_2 \) and \( p_5 \) are indiscernible regarding all the conditional attributes. However, their values for the decision attribute are different. A decision system that has indiscernible objects with different values of the decision attributes is called \textit{inconsistent} [KPPS98, SP97].

To formalize these ideas, we introduce the notion of a \textit{general decision}.

**Definition 2.2.5 (General decision):**

The \textit{general decision} \( \delta_I(x) \) over \( A \) is defined as:

\[ \delta_I(x) = \{i \mid \exists x' \in U, (x', x) \in \text{IND}_I(A) \text{ and } d(x) = i\} . \]

Consider a decision system \( I = (U, A, D) \). If, for all \( x \in U \), \( |\delta_I(x)| = 1 \) then the decision system is \textit{consistent}. Otherwise, \( I \) is \textit{inconsistent}.

### 2.2.3 Set approximations

In classical set theory we cannot represent inconsistent decision systems in a convenient way. Let \( I = (U, A, D) \) be a decision system. Consider then figure 2.1 where \( U \) is the universe, \( X \subset U \) is a concept, and \( [x_i]_A \) are equivalence classes constructed by \( \text{IND}_I(A) \) (definition 2.2.4). Each equivalence class in figure 2.1 is represented by a square. The equivalence class \([x_2]\) is contained in the concept \( X \), i.e. the objects of \([x_2]\) are members of \( X \). Moreover, \([x_1]\) is outside the concept \( X \), i.e. the objects of \([x_1]\) are not members of the concept. The problematic case, which cannot be conveniently described by set theory, comes with the ambiguity of equivalence class \([x_3]\). This class is partly inside and partly outside the concept \( X \), i.e. the objects in \([x_3]\) are only possible members of the concept. Some objects in \([x_3]\) may be members of \( X \) but others may not, although they are indiscernible using the available information (i.e. the condition attributes \( A \)). Rough set theory can deal with such cases by approximating classical sets to cover either certain or possible members.
2.2. Basic notions

Figure 2.1: Equivalence classes in the universe and their relationship with the concept $X$

Definition 2.2.6 (Rough set approximations):
Let $I = (U, A, D)$ be a decision system, $B \subseteq A$, and $X \subseteq U$. The sets $\underline{B}(X)$ and $\overline{B}(X)$ [KPPS98, PS00] are defined as:

\[ \underline{B}(X) = \{ x \in U \mid [x]_B \subseteq X \}, \]
\[ \overline{B}(X) = \{ x \in U \mid [x]_B \cap X \neq \emptyset \}, \]

where $\underline{B}(X)$ and $\overline{B}(X)$ are called the lower $B$-approximation of the concept $X$ and the upper $B$-approximation of $X$, respectively. The set

\[ \overline{B}(X) = \overline{B}(X) - \underline{B}(X) \]

is called the $B$-boundary region of $X$.

If $\overline{B}(X) = \emptyset$ then $X$ is crisp (exact) and if $\overline{B}(X) \neq \emptyset$ then $X$ is rough (inexact). The boundary region $\overline{B}(X)$ represents the ambiguity in information about objects in $X$ and therefore includes all the inconsistent objects in the concept $X$. Whenever it is clear from the context which attribute set is being used it is preferably omitted from the expression. In our example we would denote $A(X)$ by $\underline{X}$, $\overline{A}(X)$ by $\overline{X}$, and $\overline{A}(X)$ by $\overline{X}$.

Concepts are often connected to a certain outcome of the decision attribute. If the cardinality of the value domain of the decision attribute is
binary then, one value is considered as positive and the other one as negative. Value domains with higher cardinality can also be considered, where a set of attribute values are regarded as negative, but only one as positive.

In the previous example with the flu patients (table 2.2), a positive concept could be \( X = \{ p_i \in U \mid \text{flu}(p_i) = \text{yes} \} \). The corresponding negative concept would then be \( \neg X = \{ p_i \in U \mid \text{flu}(p_i) \neq \text{yes} \} \). Given the sets \( X \) and \( \neg X \) one gets the following approximative sets:

\[
X = \{ p_1, p_3, p_6 \}, \\
\overline{X} = \{ p_1, p_2, p_3, p_5, p_6 \}, \\
\neg X = \{ p_4 \}, \\
\overline{\neg X} = \{ p_2, p_4, p_5 \}.
\]

Obviously, \( \overline{X} = \overline{\overline{X}} \) and \( \overline{\neg X} = \overline{\neg \overline{X}} \). The boundary region then becomes:

\[
\overline{X} = \overline{X} \cap \overline{\neg X} = \{ p_2, p_5 \}.
\]

Moreover, \( X \) is rough since \( \overline{X} \neq \emptyset \).

**Definition 2.2.7 (Rough set):**

Let \( I = ( U, A, \{ d \} ) \) be a decision system. A rough set \( S \) is defined as a pair:

\[
S = ( \overline{A}(S), \overline{A}(\neg S) ),
\]

where \( \neg S = U - S \).

Another way of defining a rough set is via a membership function [KPPS98, PS00] which gives the conditional probability \( P(x \in X \mid x \in [x]_B) \).

**Definition 2.2.8 (Rough membership function):**

A rough membership function is defined as:

\[
\mu^B_X : U \rightarrow [0, 1], \text{ such that } \mu^B_X(x) = \frac{|X \cap [x]_B|}{|[x]_B|}.
\]

The rough membership function quantifies the degree of relative overlap between the set \( X \) and the equivalence class to which \( x \) belongs.
Given the membership function (definition 2.2.8), one can define the approximative sets as:

\[
\begin{align*}
\mathcal{B}(X) &= \{ x \in U \mid \mu_X^B(x) = 1 \}, \\
\overline{\mathcal{B}}(X) &= \{ x \in U \mid \mu_X^B(x) > 0 \}, \\
\underline{\mathcal{B}}(X) &= \{ x \in U \mid 0 < \mu_X^B(x) < 1 \}.
\end{align*}
\]

### 2.3 Reducts

Let \( I = (U, A) \) be an information system and \( A = \{a_1, a_2, a_3\} \). Figure 2.2 illustrates how selections of attributes in \( A \) change the partitioning of \( U \) into different equivalence classes.

<table>
<thead>
<tr>
<th>a1,a2,a3</th>
<th>a1,a2</th>
<th>a1,a3</th>
</tr>
</thead>
<tbody>
<tr>
<td>[x1]</td>
<td>[x1]</td>
<td>[x1]</td>
</tr>
<tr>
<td>[x2]</td>
<td>[x2]</td>
<td>[x2]</td>
</tr>
<tr>
<td>[x3]</td>
<td>[x3]</td>
<td></td>
</tr>
<tr>
<td>[x4]</td>
<td></td>
<td>[x4]</td>
</tr>
</tbody>
</table>

Figure 2.2: (a): All the original attributes in \( A \) are kept, yielding four different equivalence classes. (b): Attribute \( a_3 \) is removed from the attribute set \( A \), yielding only two equivalence classes. (c): Attribute \( a_2 \) is removed from the attribute set \( A \), yielding four different equivalence classes (the same as in case (a)).

From figure 2.2 one can see that the attribute sets \( \{a_1, a_3\} \) and \( A \) yield the same set of equivalence classes. This means that the attribute \( a_2 \) is not needed to discern the objects in \( U \). The attribute set \( \{a_1, a_2\} \), on the other hand, corresponds to fewer equivalence classes. It is thus only needed to keep the minimal number of attributes that preserve the indiscernibility relation. Such a reduced set of attributes, called a reduct, preserves the partitioning of the universe and does not change the classification of objects.
when compared with the original set of attributes [KPPS98, PS00, Paw01]. Unfortunately, the problem of finding the reducts, which have been thoroughly investigated, is NP-hard [SR92]. However, there are relatively fast algorithms to find reducts that rely on heuristics.

**Definition 2.3.1 (Reduct):**
Let $I = (U, A)$ be an information system. A reduct is a minimal set of attributes $B \subseteq A$ such that $IND_I(B) = IND_I(A)$ (definition 2.2.4).

Reducts can be computed through the creation of a *discernibility matrix* and the application of a *discernibility function* on this matrix [KPPS98, SP97].

**Definition 2.3.2 (Discernibility matrix):**
Let $I = (U, A)$ be an information system. A discernibility matrix of $I$ is a symmetric $n \times n$ matrix of elements $c_{ij}$. Every element $c_{ij}$ consists of the set of attributes that discern object $x_i$ from object $x_j$. Hence, $c_{ij}$ is defined as:

$$c_{ij} = \{ \alpha \in A | \alpha(x_i) \neq \alpha(x_j) \}, \quad i, j = 1, \ldots, n .$$

**Definition 2.3.3 (Discernibility function):**
A discernibility function $f_I$ of an information system $I = (U, A)$ is defined as:

$$f_I(\alpha_1^*, \ldots, \alpha_m^*) = \bigwedge \{ \bigvee c_{ij}^* | 1 \leq j \leq i \leq n, \ c_{ij} \neq \emptyset \},$$

where $c_{ij}^* \subseteq \{ \alpha^* | \alpha \in c_{ij} \}$ and $\alpha_1^*, \ldots, \alpha_m^*$ are boolean variables corresponding to the attributes $\alpha_1, \ldots, \alpha_m \in A$.

The following example shows the discernibility function corresponding to the matrix induced from information table 2.1. The attributes $h$, $m$, and $t$ denote *headache*, *muscle pain*, and *temperature*, respectively.
2.4 Decision rules

A row in a decision table can be seen as a decision rule. A decision rule is an if then statement on the form if \( f \) then \( g \), represented as \( f \rightarrow g \). For example, consider patient \( p1 \) in decision table 2.2. The information (i.e. the attribute values) for this patient forms the following decision rule.

\[
\text{if (headache, no) and (musclepain, yes) and (temperature, high) then (flu, yes)}
\]

We can of course create decision rules for the other patients as well.

There are different approaches for inducing decision rules in decision systems. In [Ste98], the approaches are divided into three categories of algorithms:

1. algorithms inducing the minimal set of rules,
2. algorithms inducing the exhaustive set of rules, and
3. algorithms inducing the satisfactory set of rules.
The first category is focused on describing the objects in the universe using the minimum number of necessary rules. The second one tries to generate all possible decision rules in the simplest form. To this category of algorithms one finds the classical algorithms in rough set theory. The third category of algorithms gives as a result the set of decision rules which satisfy given a priori user requirements.

In the following sections formal theory regarding decision rules and quantitative measures are covered [Paw01]. A method for finding the exhaustive set of decision rules [Paw01, SP97, Ste98] is also presented.

2.4.1 Quantitative measures

Let $I = (U, A, D)$ be a decision system. A set of formulas $For(B)$ is associated with every $B \subseteq A$. Every formula $f \subseteq For(B)$ is built up from standard logical connectives and consists of attribute pairs $(\beta, v)$, $\beta \in B$ and $v \in V_\beta$. With every formula $f \in For(B)$, $||f||_I$ is defined as the set of objects $x \in U$ that satisfy $f$ in $I$. $||f||_I$ denotes the meaning of $f$ in $I$ and is formally defined as:

$$
|| (\beta, v) ||_I = \{ x \in U | \beta(x) = v \}, \forall \beta \in B, v \in V_\beta ,
$$

$$
|| f \lor g ||_I = ||f||_I \cup ||g||_I ,
$$

$$
|| f \land g ||_I = ||f||_I \cap ||g||_I ,
$$

$$
\sim ||f||_I = U - ||f||_I ,
$$

$$
|| f \rightarrow g ||_I = (U - ||f||_I) \cup ||g||_I .
$$

A formula $f$ is true in $I$ if $||f||_I = U$ and a decision rule $f \rightarrow g$ is true in $I$ if $||f||_I \subseteq ||g||_I$. The left hand side of the rule $f \rightarrow g$ (with respect to $\rightarrow$) is called the antecedent of the rule, and the right hand side (with respect to $\rightarrow$) is called the conclusion. An object $x \in U$ satisfies a rule $f \rightarrow g$ if $x \in ||f \rightarrow g||_I$ and it satisfies the antecedent of the rule if $x \in ||f||_I$.

Several quantitative measures are usually associated with decision rules. We consider the quantitative measures support, strength, accuracy, and coverage.

The support of a decision rule is the number of objects that match both the antecedent and the conclusion. It is an estimate of the number of objects that are predicted correctly by the rule.
Definition 2.4.1 (Support):
Let $I = (U, A, D)$ be a decision system. The support of a decision rule $f \rightarrow g$ in $I$ is defined as:

$$\text{Support}(f \rightarrow g) = \text{card}(|f|_I \cap |g|_I),$$

where $\text{card}$ denotes the cardinality of a set.

The strength of a decision rule indicates how often objects in the universe satisfy the rule.

Definition 2.4.2 (Strength):
Let $I = (U, A, D)$ be a decision system. The strength of a decision rule $f \rightarrow g$ in $I$ is defined as:

$$\text{Strength}(f \rightarrow g) = \frac{\text{Support}(f \rightarrow g)}{\text{card}(U)}. $$

The accuracy of a decision rule expresses the fraction of objects satisfying the antecedent of the rule that also satisfy the conclusion. Hence, the accuracy of the decision rule $f \rightarrow g$ expresses how trustworthy the indiscernibility class described by $f$ is in drawing the conclusion $g$.

Definition 2.4.3 (Accuracy):
Let $I = (U, A, D)$ be a decision system. The accuracy of a decision rule $f \rightarrow g$ in $I$ is defined as:

$$\text{Accuracy}(f \rightarrow g) = \frac{\text{Support}(f \rightarrow g)}{\text{card}(|f|_I)}.$$ 

We may also consider the opposite, i.e. the fraction of objects satisfying the conclusion of the rule that also satisfy the antecedent. This quantitative measure is called coverage. The coverage of the decision rule $f \rightarrow g$ expresses how well the indiscernibility class described by $f$ describes the conclusion $g$.

Definition 2.4.4 (Coverage):
Let $I = (U, A, D)$ be a decision system. The coverage of a decision rule $f \rightarrow g$ in $I$ is defined as:

$$\text{Coverage}(f \rightarrow g) = \frac{\text{Support}(f \rightarrow g)}{\text{card}(|g|_I)}.$$ 

2.4.2 Decision synthesis

In inconsistent decision systems, the exhaustive set of decision rules can be induced using the upper and lower approximations [KPPS98, Paw01, SP97, Ste98]. Using this approach, one categorizes decision rules as either exact or approximative. For every decision class, exact decision rules are generated from the lower approximation. Exact decision rules are of the form:

\[
\text{if } \bigwedge \{(\alpha_j, V_{\alpha_j}) | \alpha_j \in A\} \text{ then } (d = i), i \in V_d.
\]

Approximative decision rules are generated from the upper approximation [Paw01, SP97]. Approximative decision rules are of the form:

\[
\text{if } \bigwedge \{(\alpha_j, V_{\alpha_j}) | \alpha_j \in A\} \text{ then } (d = i) \lor (d = j) \lor \ldots \lor (d = m),
\]

where \(i, j, \ldots, m \in \delta_f(x), x \in U\).

<table>
<thead>
<tr>
<th>headache</th>
<th>temperature</th>
<th>flu</th>
</tr>
</thead>
<tbody>
<tr>
<td>yes</td>
<td>normal</td>
<td>8</td>
</tr>
<tr>
<td>yes</td>
<td>very high</td>
<td>15</td>
</tr>
<tr>
<td>no</td>
<td>high</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2.3: An example with 45 potential flu patients.

Inspired by the example with the potential flu patients (table 2.2) a similar fictive example is given in table 2.3 to illustrate the theory presented in this section. Forty-five patients with symptoms connected to the disease flu have been examined and a medical expert has made a diagnosis of their outcome for the disease, i.e. if they have flu or not. For instance, eight patients with headache and normal temperature have flu but ten patients with the same symptoms have not been diagnosed to have flu. Given the information from the decision table above, the following approximative decision rules can be derived:

1. \(\text{if } (\text{headache, yes}) \land (\text{temperature, normal}) \text{ then } (\text{flu, yes}) \lor (\text{flu, no})\)
### 2.4. Decision rules

2. \( \text{if } (\text{headache, yes}) \land (\text{temperature, very high}) \text{ then } (\text{flu, yes}) \lor (\text{flu, no}) \)

3. \( \text{if } (\text{headache, no}) \land (\text{temperature, high}) \text{ then } (\text{flu, no}) \)

Every approximative rule can be seen as two new rules. This facilitates the computation of quantitative measures.

1. \( \text{if } (\text{headache, yes}) \land (\text{temperature, normal}) \text{ then } (\text{flu, yes}) \)
2. \( \text{if } (\text{headache, yes}) \land (\text{temperature, normal}) \text{ then } (\text{flu, no}) \)
3. \( \text{if } (\text{headache, yes}) \land (\text{temperature, very high}) \text{ then } (\text{flu, yes}) \)
4. \( \text{if } (\text{headache, yes}) \land (\text{temperature, very high}) \text{ then } (\text{flu, no}) \)
5. \( \text{if } (\text{headache, no}) \land (\text{temperature, high}) \text{ then } (\text{flu, no}) \)

In Table 2.4 the quantitative measures accuracy, coverage, support and strength are computed for the previously induced rules.

<table>
<thead>
<tr>
<th>rule</th>
<th>support</th>
<th>accuracy</th>
<th>coverage</th>
<th>strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>( \approx 0.44 )</td>
<td>( \approx 0.35 )</td>
<td>( \approx 0.18 )</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>( \approx 0.56 )</td>
<td>( \approx 0.45 )</td>
<td>( \approx 0.22 )</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>( \approx 0.68 )</td>
<td>( \approx 0.65 )</td>
<td>( \approx 0.33 )</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>( \approx 0.32 )</td>
<td>( \approx 0.32 )</td>
<td>( \approx 0.16 )</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1</td>
<td>( \approx 0.22 )</td>
<td>( \approx 0.11 )</td>
</tr>
</tbody>
</table>

Table 2.4: Computed quantitative measures for the induced decision rules.

One can for example see that

- approximately 44% of the patients with \textit{headache} and \textit{normal temperature} have the disease \textit{flu}.
- approximately 65% of the patients with \textit{flu} have \textit{headache} and \textit{very high temperature}.
- approximately 16% of all the patients in the observed universe have \textit{headache, very high temperature} but not \textit{flu}.
2.5 Further readings

The interested reader is encouraged to consult [KPPS98] for a tutorial that covers a great part of what has been done in the field of rough set theory. It also presents numerous applications of rough set theory. For more discussions on reducts, primarily the power of *dynamic reducts*, consult [Baz98].
Chapter 3

Rough Knowledge Bases

In this chapter we present the theoretical background and motivation for the implementation of a rough knowledge base system. The concepts of rough programs, introduced in [VDM03a], and rough knowledge bases are briefly discussed. Furthermore, a language for defining rough sets with quantitative measures and constructing rough queries is overviewed [VDM03a, VDM03b].

3.1 Rough knowledge base systems

The concepts of a knowledge base and a knowledge base system are defined by The Free On-line Dictionary of Computing\(^1\) as:

**knowledge base**: “A collection of knowledge expressed using some formal knowledge representation language. A knowledge base forms part of a knowledge-based system.”

**knowledge-based system (KBS)**: “A program for extending and/or querying a knowledge base.”

\(^1\)The Free On-line Dictionary of Computing, http://www.foldoc.org/, editor Denis Howe, supported by Imperial College Department of Computing.
With the above concepts in mind we define informally the notion of a *rough knowledge base*. A *rough knowledge base* is a collection of *rough knowledge* expressed with a language that caters for *explicit* and *implicit* definitions of rough sets. A rough set can be explicitly defined by *rough facts* that represent a decision table. Implicitly defined rough sets are obtained by combining other defined rough sets by using *rough clauses*.

For the implementation of a rough knowledge base system, we need a language for the representation of rough sets. With this language one shall be able to express knowledge in form of rough facts or rough clauses. The language shall also support *rough queries*.

### 3.2 Rough sets within the logic programming framework

In this section we present the notions of rough sets used in our framework. We discuss how a decision system can be seen as a collection of rough facts and how it is possible to define rough sets by combining other rough sets by using rough clauses. These clauses can possibly contain quantitative measures as constraints.

We start with presenting the notion of rough sets used in our framework [VDM03b]. Consider an information system \( I = (U, A) \). Every object in \( U \) (e.g. every row in an information table) is associated with a tuple of attributes. We assume that this tuple is the only way of referring to the object. Hence, different individuals described by the same tuple are indiscernible.

**Definition 3.2.1 (Rough set):**

Let \( I = (U, A = \{a_1, \ldots, a_n\}) \) be an information system. A *rough set* (or *rough relation*) \( S \) is a pair of sets \( (\overline{S}, \overline{\overline{S}}) \) satisfying conditions (i) and (ii).

(i) The elements of sets \( \overline{S} \) and \( \overline{\overline{S}} \) are expressions of the form \( \langle t_1, \ldots, t_n \rangle : k \), where \( \langle t_1, \ldots, t_n \rangle \in \prod_{a_i \in A} V_{a_i} \) and \( k \) is an integer larger than zero.
(ii) The following implications are true:

\[ \langle t_1, \ldots, t_n \rangle : k \in S \Rightarrow \forall k' \neq k (\langle t_1, \ldots, t_n \rangle : k' \notin S) , \]
\[ \langle t_1, \ldots, t_n \rangle : k \in \overline{S} \Rightarrow \forall k' \neq k (\langle t_1, \ldots, t_n \rangle : k' \notin \overline{S}) . \]

The rough complement of a rough set \( S = (S, \overline{S}) \) is the rough set \( \neg S = (\overline{S}, S) \).

Note that definition 3.2.1 differs from definition 2.2.7 in the previous chapter. The difference is that we consider a rough set to be a collection of tuples, not a set of objects in the universe. Moreover, each tuple can be seen as describing an indiscernibility class. The complement of a rough set as presented in definition 2.2.7 is defined as \( \neg S = U - S \), while in our framework a rough set \( S \) divides the universe in four regions: \( S, \overline{S}, \overline{\overline{S}}, \) and the remaining part of the universe not contained in any of those.

For simplicity, we write \( t \) to designate a general tuple \( \langle t_1, \ldots, t_n \rangle \).

An element \( t : k \in S (t : k \in \overline{S}) \) indicates that the indiscernibility class described by \( t \) belongs to the upper approximation of a rough set \( S (\neg S) \) and that this class contains \( k > 0 \) individuals that are positive examples of the concept described by \( S (\neg S) \). The lower approximation of a rough set \( S \) is then defined as:

\[ \underline{S} = \{ t : k_1 \in S | \forall k_2 > 0, t : k_2 \notin \overline{S} \} \]

and the boundary region is defined as:

\[ \overline{S} = \{ t : k_1 : k_2 | \exists k_1, k_2 > 0, t : k_1 \in S \text{ and } t : k_2 \in \overline{S} \} . \]

Next, we briefly cover the notions of logic programs and extended logic programs needed in the following theory.

### 3.2.1 Logic programs

The ability of defining rough sets in terms of other ones is fundamental for the construction of rough knowledge bases [VDM03a]. The language used to define new rough sets, presented in detail in sections 3.2.2 and 3.2.3, is compiled in the language of extended logic programs [PA92] that can easily
be executed in a Prolog system. In this section we briefly review the main notions underlying extended logic programs. Chapter 4 is devoted to the compilation issues.

The paraconsistent semantics of extended logic programs [SI95] provide two forms of negation, explicit and default, allowing both open-world and closed-world reasoning. Explicit negation describes negative evidence, e.g. negative examples in a decision table. Default negation, on the other hand, allows reasoning with lack of information, needed when defining lower approximations of rough sets. Under the paraconsistent semantics, information and its explicit negation can simultaneously hold. This is crucial in rough set theory for the concept of boundary regions.

We now recall the syntax of logic programs\(^2\), covering only the needed parts used in the following sections.

The alphabet of the language of logic programs consists of the following classes of symbols:

- **variables** which will be written as alphanumeric identifiers beginning with capital letters
- **constants** which are numerals or alphanumeric identifiers beginning with lower case letters
- **predicate symbols** which are alphanumeric identifiers starting with lower case letters, e.g. \( p \), with an associated **arity** \( n \geq 0 \), denoted \( p/n \)
- **logical connectives** which are \( \land \) (conjunction), \( \neg \) (explicit negation) and **not** (default negation)

Conjunctions are often written as the comma character (,). The syntax is built up from ordinary first order **atoms**. An atom is a predicate symbol with a number of **terms** specified by its arity. It is written as \( p(t_1, \ldots, t_n) \), where \( p/n \) is a predicate and \( t_i, 1 \leq i \leq n \), are terms. A term is either a variable or a constant. An atom with all terms being constants is called **ground**. The set of all atoms is denoted \( At \). An **objective literal** \( L \) is either an atom \( A \in At \) or its explicit negation \( \neg A \). The set of all objective literals

\(^2\)For an introduction to logic programming see [NM95].
is $OLit = At \cup \neg At$, where $\neg At = \{\neg A \mid A \in At\}$. A default negated literal $L$ is denoted $not L$.

**Definition 3.2.2 (Program clause):**
A program clause is an expression

$$L_0 \leftarrow L_1, \ldots, L_m, not L_{m+1}, \ldots, not L_n,$$

where each $L_i \in OLit$ and $0 \leq m \leq n$.

The left hand side of the clause (with respect to $\leftarrow$) is called the head and the right hand side of the clause (with respect to $\leftarrow$) is called the body. A program clause is an implication of the form $body \Rightarrow head$, i.e. if $body$ is true then $head$ is also true. The implication is logically equivalent to the disjunction $\neg body \lor head$, i.e. the disjunction is false if and only if the head is true and the body is false. A program clause with an empty body is called a fact. If the clause instead only has a body then it is called an integrity constraint. An integrity constraint $\leftarrow body$ represents the implication $body \Rightarrow false$. If all the literals in a program clause are ground then it is called a ground clause.

**Definition 3.2.3 (Extended logic program):**
An extended logic program (ELP) $\mathcal{P}$ is a set of program clauses and integrity constraints.

### 3.2.2 Viewing decision systems as logic facts

Consider a decision system $I = (U, A, \{d\})$ where $d$ is a binary decision attribute. Decision systems are often represented by decision tables. A tuple $t$ in the decision table describes an indiscernibility class in $I$. A decision table can be seen as an alternative representation of a rough set $D = (\overline{D}, \overline{\neg D})$. An expression $t : k_1 \in \overline{D}$ corresponds to $k_1 > 0$ lines (expressing $t$) in the table with positive outcome for the decision attribute. An expression $t : k_2 \in \overline{\neg D}$ corresponds to $k_2 > 0$ lines (expressing $t$) with negative outcome for the decision attribute.

Studying decision tables from a logic programming perspective one can view each row in the table as a logic fact. The predicate symbol of
such a fact denotes the outcome (positive or negative) of the decision attribute. From a decision table representing the decision system $I = (U, A = \{a_1, \ldots, a_n\}, \{d\})$, one can derive logic facts on the form:

\[
d(t_1, \ldots, t_n),
\]

\[
\neg d(t_1, \ldots, t_n),
\]

where each $t_i$ denotes a value of the conditional attribute $a_i$. The latter fact describes the explicit negation of the rough relation $D$. The same tuple can describe both positive and negative examples of the rough relation.

The support of $d(t_1, \ldots, t_n)$ ($\neg d(t_1, \ldots, t_n)$) corresponds to the number of lines in the decision table having positive (negative) outcome of the decision attribute $d$.

A decision table can be encoded as a collection of rough facts. A rough fact describes a tuple in the upper approximation of a rough relation and is on one of the two forms:

\[
\overline{d}(t_1, \ldots, t_n) : k_1,
\]

\[
\overline{\neg d}(t_1, \ldots, t_n) : k_2,
\]

which describe a tuple in the rough region $\overline{D}$ with support $k_1$ and a tuple in the rough region $\overline{\neg D}$ with support $k_2$, respectively.

As an example let us consider the decision system $Walk$ in table 3.1 [KPPS98]. The decision system $Walk$, with its decision attribute $Walk$ explicitly defines the rough relation $Walk$. Note the similarities and differences of the printed names of the decision system, rough relation and decision attribute, as this naming convention will hold throughout the rest of this thesis.

In table 3.1, one can see that both $\overline{Walk}(31-45, 1-25)$ and its explicit negation $\overline{\neg Walk}(31-45, 1-25)$ holds, which is possible within the paraconsistent semantics. Two objects (06 and 07) are in the same indiscernibility class and in the same decision class which yields the support 2. The decision system $Walk$ represents the following rough set.

\[
Walk = \{\langle 16-30, 50 \rangle, \langle 16-30, 26-49 \rangle\},
\]

\[
\overline{Walk} = \{\langle 16-30, 0 \rangle, \langle 46-60, 26-49 \rangle\},
\]

\[
\overline{Walk} = \{\langle 31-45, 1-25 \rangle\}.
\]
### 3.2.3 Defining rough relations with logic rules

The previous section introduced the basic idea of viewing decision systems as a collection of rough facts. The rough facts, induced from a decision table, are used to explicitly define a rough relation. Our definition of a rough knowledge base require that the rough language is able to express definitions of rough relations in terms of other rough relations. This can be done within the extended logic programming framework. Rough relations can be defined by other ones with the use of *rough clauses*.

As an informal example, once again consider the decision system *Walk* in Table 3.1. Its corresponding rough relation *Walk* can be used to define a new rough relation *Walk*(−*Age*), which corresponds to the original rough relation but ignores the conditional attribute *Age*. The rough clauses needed for the definition of *Walk*(−*Age*) are the following:

\[
\begin{align*}
\text{Walk}(\text{LEMS}) & \leftarrow \text{Walk}(\text{LEMS}, \text{Age}). \\
\text{¬Walk}(\text{LEMS}) & \leftarrow \text{¬Walk}(\text{LEMS}, \text{Age}).
\end{align*}
\]

Rough clause 3.1 (3.2) capture the positive (negative) upper approximation of *Walk*(−*Age*). This set includes all the objects from the upper approximation of *Walk* (¬*Walk*). When not considering the conditional attribute *Age* the original indiscernibility classes may change, as shown in section

<table>
<thead>
<tr>
<th></th>
<th>Age</th>
<th>LEMS</th>
<th>Walk</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>16-30</td>
<td>50</td>
<td>Yes</td>
</tr>
<tr>
<td>o2</td>
<td>16-30</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>o3</td>
<td>16-30</td>
<td>26-49</td>
<td>Yes</td>
</tr>
<tr>
<td>o4</td>
<td>31-45</td>
<td>1-25</td>
<td>No</td>
</tr>
<tr>
<td>o5</td>
<td>31-45</td>
<td>1-25</td>
<td>Yes</td>
</tr>
<tr>
<td>o6</td>
<td>46-60</td>
<td>26-49</td>
<td>No</td>
</tr>
</tbody>
</table>
| o7 | 46-60 | 26-49 | No   | }
2.2.2. From the new rough relation we get the following regions:

\[
\begin{align*}
\textit{Walk}_{\text{(-Age)}} &= \{\langle 50 \rangle \}, \\
\overline{\textit{Walk}}_{\text{(-Age)}} &= \{\langle 0 \rangle \}, \\
\textit{Walk}_{\text{(-Age)}} &= \{\langle 1-25 \rangle, \langle 26-49 \rangle \}.
\end{align*}
\]

Note that the lower approximation of \( \textit{Walk}_{\text{(-Age)}} \) only includes the tuple \( \langle 50 \rangle \), i.e. this is the only tuple that describes an indiscernibility class whose members have consistent membership to the positive decision class \( \textit{Walk} \). The boundary region of \( \textit{Walk}_{\text{(-Age)}} \) covers the tuples associated with ambiguous decisions, i.e. in both indiscernibility class \( \langle 1-25 \rangle \) and \( \langle 26-49 \rangle \) it is possible to find different objects belonging to decision class \( \textit{Walk} \) and to its complement \( \overline{\textit{Walk}} \).

The ideas introduced so far will be further extended; new rough relations may be defined in terms of more than one rough relation and quantitative measures can also be incorporated. More complex examples will be presented in chapter 5.

### 3.3 The rough language

In this section we formally present the language for defining rough relations with quantitative measures and constructing rough queries [VDM03a, VDM03b]. The formal definitions of the syntax are mixed with some explanatory examples for better understanding. The semantics of the language without quantitative measures is, for the interested reader, covered in [VDM03a].

Rough facts encode rough relations explicitly defined by a decision table, as discussed in section 3.2.2. Rough clauses, introduced in section 3.2.3, are on the other hand, used to implicitly define new rough relations obtained by combining different regions of other rough relations.

**Definition 3.3.1 (Rough fact):**

A rough fact is any statement of the form:

\[
\overline{\beta}(\alpha_1, \ldots, \alpha_n) : \kappa ,
\]
where $\beta \in \{p, \neg p\}$ denotes the rough relation $P$ or $\neg P$, with $n$ conditional attributes $a_1, \ldots, a_n$ and values $\alpha_i \in V_{a_i}$ ($1 \leq i \leq n$). The constant $\kappa (>0)$ denotes the support of the fact.

If the fact has the form $p(\alpha_1, \ldots, \alpha_n) : \kappa (\neg p(\alpha_1, \ldots, \alpha_n) : \kappa)$ then $\kappa$ indicates the number of objects in the indiscernibility class $\langle \alpha_1, \ldots, \alpha_n \rangle$ that have positive (negative) outcome for the decision attribute.

As an example, consider the following rough facts:

$$\bar{r}(c_1, c_2, c_3) : 5.,$$
$$\neg\bar{r}(c_1, c_2, c_3) : 8..$$

These facts state that the indiscernibility class described by the tuple of attribute values $\langle c_1, c_2, c_3 \rangle$ has 5 individuals that are positive examples of the rough relation denoted by $r$, designated as $R$, and it has 8 individuals that are negative examples of $R$ (or positive examples of $\neg R$).

The expression $\beta(t_1, \ldots, t_n)$ (in definition 3.3.1), where $\beta$ (possibly negated) denotes a rough relation, is called a rough literal. The other possible forms of rough literals are: $\beta(t_1, \ldots, t_n)$ and $\neg\beta(t_1, \ldots, t_n)$. The terms $t_i$ in the rough literals are either variables or constants (denoted as $\alpha_i$ in definition 3.3.1).

In section 2.4.1 we defined quantitative measures for decision rules. With a set of attributes $A = \{a_1, \ldots, a_n\}$, expressions of the forms $p(t_1, \ldots, t_n)$ and $\neg p(t_1, \ldots, t_n)$ can be seen as the decision rules $(a_1, t_1) \land \ldots \land (a_n, t_n) \rightarrow (p, yes)$, and $(a_1, t_1) \land \ldots \land (a_n, t_n) \rightarrow (\neg p, yes)$, respectively.

**Definition 3.3.2 (Quantitative measure):**

A **quantitative measure** is any of the following:

- **support**: $\text{supp}(p(t_1, \ldots, t_n))$,
- **accuracy**: $\text{acc}(p(t_1, \ldots, t_n))$,
- **coverage**: $\text{cov}(p(t_1, \ldots, t_n))$,
- **strength**: $\text{strength}(p(t_1, \ldots, t_n))$,

where $p$ denotes the existing rough relation $P$. The same holds for $\neg p$, denoting the rough relation $\neg P$. 
3.3. The rough language

Note that the quantitative measures are applied on a tuple (of variables and/or constants) in a specific rough relation.

Quantitative measures can be used as constraints in rough clauses. A *quantitative measure constraint* is formally defined by the following definition.

**Definition 3.3.3 (Quantitative measure constraint):**
A quantitative measure constraint is any of the two forms:

\[ m(p(t_1, \ldots, t_n)) \text{ relOp } k, \]
\[ m_1(p_1(t_1, \ldots, t_n)) \text{ relOp } m_2(p_2(t_1, \ldots, t_n)), \]

where \( p, p_1 \) and \( p_2 \) are predicate symbols denoting a rough relation, \( m, m_1, \) and \( m_2 \) are any of \( \text{supp}, \text{acc}, \text{cov} \) or \( \text{strength} \), and \( k \) (\( > 0 \)) is a rational value. The operator \( \text{relOp} \) is either \( < \), \( > \), \( = \), \( \leq \) or \( \geq \).

**Definition 3.3.4 (Rough clause):**
A rough clause is any formula of the following two forms:

\[ \overline{\beta}(t_1, \ldots, t_n) :\neg[\tau,F] R_1, \ldots, R_m, C_1, \ldots, C_l, \]  
(3.3)
\[ \beta(t_1, \ldots, t_n) :\neg[\tau,F] R_1, \ldots, R_m, C_1, \ldots, C_l, \]  
(3.4)

where \( \beta \) is either \( p \) or \( \neg p \) (for some predicate symbol \( p \)), \( t_i \) (\( 1 \leq i \leq n \)) are attribute terms, \( R_j \) (\( 1 \leq j \leq m \)) are rough literals, and \( C_k \) (\( 1 \leq k \leq l \)) are quantitative measure constraints such that all variables occurring as their arguments should also appear in some \( R_j \). \( F \) is a *support-combining function* that determines how the support of the newly defined rough relation is obtained from the support of the rough relations in the body of the clause. The available support-combining functions are *sum*, *min* and *max*. If the body only has one rough literal then \( F \) is optional (actually not needed) and often set to \( \bot ([\tau,\bot]) \). The constant \( \tau \in [0,1] \) (often set to 1) is a rational number representing the *trust* in the body of the clause. The *trust* is the fraction of the calculated support of the body that should be considered as support for the rough region being defined (i.e. in the head).

A rough clause like \( p(X,c) :\neg [0.8,\bot] q(X,c) \). could be used if the user strongly doubts the reliability of the information carried by 20% of the examples belonging to any indiscernibility class that only has positive examples of \( Q \) and for which the second attribute has value \( c \) [VDM03b].
Consider the following rough clause:

\[ \overline{p}(X_1, X_2) :- \left[ \tau, F \right] q(X_1, X_2), \overline{r}(X_1, X_2). \]

Assume that there are two indiscernibility classes described by tuple \( \langle c_1, c_2 \rangle \); one indiscernibility class is contained in \( Q \) and the other belongs to \( \overline{R} \). Function \( F \) is then used to combine \( \text{supp}(q(c_1, c_2)) \) with \( \text{supp}(\overline{r}(c_1, c_2)) \).

If \( \langle c_1, c_2 \rangle : k_2 \in Q \) and \( \langle c_1, c_2 \rangle : k_3 \in \overline{R} \) then \( \langle c_1, c_2 \rangle : k_1 \in P \), where \( k_1 = \left\lfloor \tau \times F(k_2, k_3) \right\rfloor \).

We now give the definition of a rough program.

**Definition 3.3.5 (Rough program):**

A rough program is a finite set of rough facts and rough clauses.

The heads of formulae 3.3 and 3.4 are rough literals denoting the upper and lower approximation of a rough relation, respectively. The head of a rough clause cannot refer to the boundary region of a rough relation. However, this is not a real restriction as shown in the following example.

To exemplify the previous theoretical definitions a small example of a rough program \( P \) [VDM03b] is given.

\[
P = \{ \overline{p}(X_1X_2) :-[1, \min] q(X_1X_2), \overline{r}(X_1X_2). , \overline{p}(X,c) :-[1,\_]\overline{q}\overline{1}(X,c). , \overline{q}(a,c) : 2. , \overline{r}(a,c) : 3. , \overline{\overline{r}}(a,c) : 4. , \overline{q}\overline{1}(a,c) : 3. , \overline{\overline{q}}\overline{1}(a,c) : 7. \}
\]

The body of the first rough clause represents the intersection of the lower approximation of the rough relation \( Q \) and the boundary of the rough relation \( \overline{R} \). From this clause together with the rough facts of \( P \), stating that \( \langle a,c \rangle : 2 \in Q \) and \( \langle a,c \rangle : 4 : 3 \in \overline{R} \), it can be concluded that \( \text{supp}(p(a,c)) \geq 1 \times \min(2,4) \). The support-combining function \( \min \) is applied to \( \text{supp}(q(a,c)) = 2 \) and \( \text{supp}(\overline{r}(a,c)) = 4 \), which yields the
value 2. The second and third rough clause together state that if an indiscernibility class belongs to the boundary of the rough relation $\neg Q_1$ and its second attribute has value $c$ then it also belongs to the boundary of $P$. The restriction of not allowing rough literals referring to the boundary region of a rough relation in the head of a rough clause can thus be simulated with these two clauses. Moreover, $\text{supp}(q_1(a,c)) = 3$ individuals should be considered as representing positive examples of $P$, while $\text{supp}(\neg q_1(a,c)) = 7$ individuals should be considered as representing negative examples of $P$. Putting all together, it can be concluded that $\text{supp}(p(a,c)) = \min(2,4) + 3 = 5$ and $\text{supp}(\neg p(a,c)) = 7$.

As can be seen in definition 3.3.4, quantitative measures can be used as constraints in the body of a rough clause. Moreover, they can also be used as assignments in rough queries for the calculation of interesting values.

**Definition 3.3.6 (Quantitative measure assignment):**
A **quantitative measure assignment** is any statement of the form:

$$K = m(p(t_1, \ldots, t_n)),$$

where $K$ is a variable to be instantiated with the computed value of the quantitative measure $m$ (i.e. $\text{supp}$, $\text{acc}$, $\text{strength}$, or $\text{cov}$) applied on $p(t_1, \ldots, t_n)$.

**Definition 3.3.7 (Rough query):**
A **rough query** with respect to a rough program $P$ is either an expression of the form

$$Q_1, \ldots, Q_n,$$

or

$$C = \text{classify}(p(t_1, \ldots, t_n)),$$

or

$$C = \text{classify}(\neg p(t_1, \ldots, t_n))$$

Each $Q_i$ ($1 \leq i \leq n$) is either a rough literal, a quantitative measure assignment or a quantitative measure constraint. $\text{classify}$ denotes a classification procedure. $C$ is a variable that shall be instantiated with the result of the classification.
The classification query requests a prediction for the decision class to which a new individual \( i \) described by tuple \( \langle t_1, \ldots, t_n \rangle \) may belong. To answer such a query the following strategy is used. All decision rules that match the description of \( i \) cast a number of votes corresponding to their support. Let \( \theta \) be the total number of votes casted by all decision rules. The number of votes obtained for each decision class is then summed and divided by \( \theta \). We obtain in this way a certainty factor \( C_F \) for each decision class. The prediction corresponds to the decision class with the highest certainty factor. The result of the classification request \( \text{classify}(p(t_1, \ldots, t_n)) \). (\( \text{classify}(\neg p(t_1, \ldots, t_n)) \).) is the pair \( (p = \text{yes}, C_F) \), \( (p = \text{no}, C_F) \), or \( (p = \text{unknown}, C_F) \). The last case corresponds to the situation where no decision rule is fired, i.e. when there are no decision rules that match the tuple \( \langle t_1, \ldots, t_n \rangle \).

If the rough query is non-ground then it requests all instantiations of the attribute variables in \( Q_i \). If the query, on the other hand, is ground then it requests the truth value (yes or no) of the query.

Consider again the rough program \( P \) (example 3.5) and the following rough queries with their computed answers.

- **What are the strengths of the decision rules in \( \neg R \)?**

  **Rough Query:**
  \[ \neg r(X_1, X_2), \ K = \text{strength}(\neg r(X_1, X_2)) \ . \]

  **Answer:** \( K = 0.5714, X_1 = a, X_2 = c \)

  The variables \( X_1 \) and \( X_2 \) are instantiated with the values \( a \) and \( c \), respectively. The strength is computed as explained in definition 2.4.2, i.e. \( K \) is instantiated with the value of dividing \( \text{supp}(\neg r(a,c)) = 4 \) with the sum of the total supports for \( \neg R \) and \( R \). This gives \( K = \frac{4}{4+3} = 0.5714 \).

- **What is the accuracy of the decision rule described by \( p(a,c) \)?**

  **Rough Query:**
  \[ K = \text{acc}(p(a,c)) \ . \]
3.3. The rough language

Answer: $K = 0.4167$

The computation of the quantitative measure accuracy is described in definition 2.4.3.

- *Is the indiscernibility class \( \langle a, b \rangle \) a member of the rough region \( P \)?*

Rough Query:
\[ p(a,b). \]

Answer: no

The above query is ground and thus requests the truth value for the query. The answer is no, i.e. the indiscernibility class \( \langle a, b \rangle \) does not belong to \( P \). In fact, \( \langle a, b \rangle \) is not a member of any region of a rough relation defined in \( P \).

- *What is the predicted decision for an individual described by the tuple \( \langle a, c \rangle \) in rough relation \( P \)?*

Rough Query:
\[ K = \text{classify}(p(a,c)). \]

Answer: \( K = (p = \text{no}, 0.5833) \)

\( K \) is instantiated with the predicted decision \( \text{no} \), i.e. the decision in \( P \) with the highest support for the tuple \( \langle a, c \rangle \) is \( \text{no} \). The certainty factor is calculated as the total support for the tuple with decision \( \text{no} \) divided by the total support for the tuple, i.e. the certainty factor is \( \frac{7}{5+7} \approx 0.5833 \).
Chapter 4

Implementation

This chapter describes the implementation of a rough knowledge base system, motivated and theoretically introduced in chapter 3.

4.1 Design choices

Before digging into the implementation details, we will first discuss some design choices.

4.1.1 Prolog system

The rough knowledge base system is mainly implemented in Prolog. As we consider rough sets within the logic programming framework this choice seemed obvious. There are numerous distributions of Prolog systems, more or less adapted to the ISO Prolog standard\(^1\). It is of course beneficial to implement a system in a standardized Prolog language as this does not restrict the choice of the Prolog interpreter. We have chosen to use the XSB Prolog 2.6 system\(^2\) [SSW+03], a freely available open source software


\(^2\)http://xsb.sourceforge.com/.
that conforms to the ISO standard. Most of the Prolog code is imple-
mented using the ISO Prolog standard. We have, however, chosen to use
some libraries that are specific to XSB Prolog. For instance, the socket
communication library of XSB Prolog is used even though methods for this
type of communication are not covered by the ISO standard. The reason
for incorporating such methods anyway is because it seemed as the most
convenient solution for the communication with the Java implemented user
interface. Moreover, we use the methods provided by XSB for exception
handling. These methods are more easily used and incorporated in our
Prolog implementation than the ISO methods for exception handling. The
exception handling methods provided by standard Prolog can of course
replace the XSB specific parts if the implementation is to be ported to
another Prolog system. This will, however, require some reconstruction of
the implementation.

4.1.2 Language modifications

The syntax of the rough language was covered in chapter 3. The different
approximation identifiers; \( \overline{p}, \underline{p} \) and \( \overline{p} \) of a rough relation \( P \), are for usability
reasons changed to:

\[
\overline{p}(T_1, \ldots, T_n) \Rightarrow \text{upper}(p(T_1, \ldots, T_n)),
\underline{p}(T_1, \ldots, T_n) \Rightarrow \text{lower}(p(T_1, \ldots, T_n)),
\overline{p}(T_1, \ldots, T_n) \Rightarrow \text{boundary}(p(T_1, \ldots, T_n)).
\]

These rough literals can also be constructed for the explicit negation of
\( p \). The explicit negation will be written using \( \sim \) (tilde), e.g. \( \sim p \) will be
changed to \( \sim p \).

4.2 System overview

The kernel of the system is implemented in Prolog and it is further dis-
cussed in section 4.3. This kernel forms the actual rough knowledge base
system (RKBS). It handles the rough knowledge base (RKB) and supplies
to it methods for modification, creation and querying the represented rough
knowledge. It is a user-associated process that receives requests, acts accordingly, and outputs results corresponding to these requests. The Prolog engine is a stand-alone program that can be used as is but, regarding the usability and accessibility for users outside the logic programming community and for the benefit of avoiding installation of a local Prolog system, we have chosen to add a front-end to it (see figure 4.1).

The user front-end is implemented in Java\(^3\) and consists of a Java server and a collection of Java servlets. The servlets handle the direct communication with the end-user through a web page. Having the system accessible on the World Wide Web improves usability and accessibility of the RKBS. On the web page, an end-user can request the RKBS for compilation of rough clauses. The user can also query the rough knowledge base for indiscernibility classes, classification of new individuals, and computation of quantitative measures related to a certain rough relation. The web front-end makes it possible to graphically overview the knowledge base and the

---

\(^3\)http://java.sun.com/
computed results in a comprehensive way (section 4.4).

Each user request is processed by the main Java servlet, RKBServlet, and redirected via socket communication to the Java server, called RKBServer. This server manages the possible multitude of simultaneous user requests by letting RKBServerThreads handle the communication with each user (see figure 4.1). Communication through sockets is beneficial in the sense that the RKBServer does not need to be running on the same location as the Tomcat server\(^4\) which handles the execution of the servlets. It also makes the implementation easier than other methods, as the focus of the implementation is not on the Java front-end but on the Prolog implementation. The use of Java threads (RKBServerThreads) implies communication separation of different users. Each user gets the correct behavior of the rough knowledge base system as if it was operating for that user only. With a unique identifier\(^5\) for every user it is possible to map each user to the correct RKB, in form of a pair of a RKBToXSBClient and a XSB process running the rough knowledge base. The communication between the RKBToXSBClient and the Prolog engine is done through sockets. This form of communication is easily implemented and stable in Java and XSB Prolog [SSW+03].

Algorithm 4.2.1 represents the pseudo code for the evaluation of a user request in the rough knowledge base system. It shows the main steps of execution in the system from a user giving a request to the system until the user receives feedback.

<table>
<thead>
<tr>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java servlets</td>
</tr>
<tr>
<td>Java front-end</td>
</tr>
<tr>
<td>Prolog engine</td>
</tr>
<tr>
<td><strong>Total</strong></td>
</tr>
</tbody>
</table>

Table 4.1: Number of lines in the core parts of the system implementation.

---

\(^4\)For more information regarding the Tomcat server and the Apache Jakarta Project see: http://jakarta.apache.org/tomcat/.

\(^5\)Every user request is sent together with a user unique session identifier generated by the Tomcat server.
Algorithm 4.2.1 The pseudo code for the evaluation of a user request.

\textbf{Evaluate Request}(Request, UserId)

1. **RKBServlet:**
   2. Send Request together with UserId to the RKBServer
   3. Wait for feedback

2. **RKBServer:**
   3. Start new RKBServerThread for the communication with
   4. the RKBServlet
   5. Redirect Request and UserId to the RKBServerThread

3. **RKBServerThread:**
   9. if UserId represents a new user
      10. then Tell the RKBServer to create a new RKBToXSBClient
      11. for the communication with XSB Prolog
      12. else Get the RKBToXSBClient associated with UserId
      13. for the communication with XSB Prolog
      14. Send Request to the RKBToXSBClient
      15. Wait for feedback

4. **RKBToXSBClient:**
   17. if The user is new
      18. then Start a new XSB Prolog process that handles the
      19. rough knowledge base associated to the user
      20. Set up sockets for communication with XSB Prolog
      21. Send Request to the newly created or already
      22. existing XSB Prolog process associated to the user
      23. Wait for feedback

5. **XSB Prolog:**
   25. Evaluate Request
   26. Report result to the RKBToXSBClient
   27. The result is then propagated from the RKBToXSBClient
   28. to the RKBServerThread and finally to the user via
   30. the RKBServlet
4.3 Prolog implementation

Algorithm 4.2.1 described how a user request is handled by the rough knowledge base system. However, the actual evaluation of user requests is handled by the methods implemented in Prolog. The Prolog implementation consists of the following core parts:

- a single-client server that handles the socket communication with the RKBToXSBClient,
- a scanner which tokenizes input requests,
- definite clause grammars for the mapping of input tokens and the rewriting of rough statements into internal Prolog code,
- a framework for compilation of rough clauses,
- a framework for evaluation of rough queries,
- methods for reporting results, such as answers to rough queries and feedback after compilation of rough rules, and
- methods for detecting errors, such as syntax errors, undefined predicate errors, and integrity constraint violations (model failures).

4.3.1 Prolog server

The Prolog server is a single-client server that waits for incoming requests through sockets from the RKBToXSBClient. A request is of the form compile_file(Inf,Out), the main method in the Prolog implementation. The parameters of compile_file/2 are explained below.

Inf A file consisting of a request type (rules or queries) together with the rough rules or queries written in the rough language.

Out The file where the result(s) of modifying or querying the RKB is reported.

The pseudocode of compile_file/2 is presented in algorithm 4.3.1.
Algorithm 4.3.1 Main method. If the request type is `filetype(rules)` (`filetype(queries)`), then redirect execution to `Compile_Rules/2` (`Evaluate_Queries/2`).

<table>
<thead>
<tr>
<th>Compile_File(Infile, Outfile)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Open Infile for reading</td>
</tr>
<tr>
<td>2 Open Outfile for writing</td>
</tr>
<tr>
<td>3 Read first line of Infile</td>
</tr>
<tr>
<td>4 if line = filetype(rules).</td>
</tr>
<tr>
<td>5 then Call Compile_Rules(Infile, Outfile)</td>
</tr>
<tr>
<td>6 else if line = filetype(queries).</td>
</tr>
<tr>
<td>7 then Call Evaluate_Queries(Infile, Outfile)</td>
</tr>
</tbody>
</table>

4.3.2 Compilation of rough clauses

As mentioned in section 4.3.1, the current request type is specified in the input file. Hence, when parsing this file the type of operation is distinguished. If the request type is compilation of rough rules, then the behavior of the rough knowledge base system is described by algorithm 4.3.2. Its main functionality is to read lines from the input file, tokenize the input, match the tokens with a definite clause grammar (DCG) and extend the rough knowledge base with the generated Prolog code. The main DCGs used in the system are described in section 4.3.5.

4.3.3 Evaluation of rough queries

Algorithm 4.3.2 describes the sequence of execution when the request type is compilation of rules. On the other hand, if the request type corresponds to evaluation of queries then, as stated in algorithm 4.3.1, the execution proceeds according to algorithm 4.3.3. Its main functionality is to read lines from the input file, tokenize the input, try to match the tokens with the DCG `compound_query/4` and instantiate the variables of the query by evaluating the generated Prolog code in the rough knowledge base.
Algorithm 4.3.2 Compilation of rough rules

**COMPILE_RULES(Infile, Outfile)**

1. while not end_of_file(Infile) do
2.   Read new line from Infile
3.   Tokenize line by rkb_scan/4
4.   if tokenization was not successful then
5.     Remember syntax error cause
6.     Start over with next line in Infile
7.   if the tokens match the DCG: lower_clause/6 then
8.     Generate Prolog rule
9.     Generate integrity constraint
10.    else if the tokens match the DCG: upper_clause/5 then
11.       Generate Prolog rule
12.      else if the tokens match the DCG: fact/5 then
13.        Generate Prolog fact
14.       else Remember syntax error cause
15.      Start over with next line in Infile
16.     Assert the generated Prolog code to the KB
17.     Remember the newly defined predicate p and the rule/fact
18.     Generate general predicate rules for p (section 4.3.5)
19.     Assert the generated rules to the KB if not already asserted
20.    if an integrity constraint violation has occurred then
21.      Remember model failure cause
22.      undo statements: (19), (20) and (22)
23.     exit loop
24.   if the parsing has been error-free then
25.     Write the known predicates in the KB and
26.      the rules or facts that defines them to Outfile
27.   else Write the errors and causes to Outfile
Algorithm 4.3.3 Evaluation of rough queries

Evaluate Queries(\textit{Infile}, \textit{Outfile})

1. while not end\_of\_file(\textit{Infile})
2. do
3. \hspace{1em} Read new string from \textit{Infile}
4. \hspace{1em} Tokenize string and identify variables to instantiate
5. \hspace{1em} if tokenization was not successful
6. \hspace{2em} then Remember syntax error cause
7. \hspace{2em} Start over with next string in \textit{Infile}
8. \hspace{1em} if the tokens match the DCG: \texttt{compound\_query/4}
9. \hspace{2em} then Generate Prolog code for evaluation
10. \hspace{2em} else Remember syntax error cause
11. \hspace{2em} Start over with next string in \textit{Infile}
12. \hspace{1em} Try to evaluate the generated Prolog code and find all
13. \hspace{2em} instantiations of the variables occurring in the parsed query
14. \hspace{2em} if an undefined predicate error occurs during evaluation
15. \hspace{3em} then Remember the undefined predicate and location
16. \hspace{2em} else if the evaluation fails due to syntax error
17. \hspace{3em} then Remember syntax error cause
18. \hspace{2em} else Write the result of the evaluation to \textit{Outfile}
19. \hspace{1em} if syntax or undefined predicate errors have occurred
20. \hspace{2em} then Write the errors and causes to \textit{Outfile}
4.3.4 Scanner

When compiling rough rules or evaluating rough queries the strings read from the input file need to be scanned and tokenized. The main reason why the scanner \texttt{rkb\_scan/4} is needed is for easier parsing of user input. The scanner takes as input parameter a string for tokenization and returns the generated tokens, the number of lines in the string, and all occurring variables in the string.

Quantitative measures applied on rough literals are separated due to the fact that Prolog cannot identify variables occurring in a \textit{functor} inside another \textit{functor}, e.g. the innermost parts of the term \texttt{strength(p(A,B))}, using unification. For instance, consider the expression \texttt{strength(p(A,B))} occurring as a term in the Prolog predicate \texttt{q\_measure/2}. \texttt{q\_measure/2} is used for identifying quantitative measures. If \texttt{q\_measure/2} is defined by the rule \texttt{q\_measure(strength(P),Code) :- \ldots}, then \texttt{P} cannot be unified with the functor \texttt{p(A,B)} in expression \texttt{strength(p(A,B))} so the expression has to be separated.

Moreover, a number of keywords such as \texttt{upper}, \texttt{lower}, \texttt{boundary}, and quantitative measure names such as \texttt{strength}, \texttt{cov} (coverage), \texttt{acc} (accuracy) and \texttt{supp} (support), are identified and the resulting tokens are modified for easier parsing.

4.3.5 Rewriting rough statements into Prolog code

The internal representation of rough literals differs from the actual rough language. Consider the literal \texttt{p(t)}, where \texttt{p} denotes the rough relation \texttt{P} and \texttt{t} represents a tuple in \texttt{P}. The following new internal predicates for \texttt{p(t)} are introduced:

\texttt{p^*(t,k)} which says that the tuple \texttt{t} belongs to \texttt{\overline{P}} and the support of \texttt{p(t)} is \textit{at least} \texttt{k},

\texttt{p(t,k)} which is similar to the predicate above except that the support of \texttt{p(t)} is \textit{exactly} \texttt{k}, and

\texttt{p^?(t)} which says that the tuple \texttt{t} belongs to \texttt{\overline{P}} but it does not keep any information about the support.
The above presented predicates are similarly constructed for the explicit
negation of $p(t)$ (i.e. $\neg p(t)$). Whenever a new predicate $p$ appears in the
head of a rough clause or in a fact, the system will generate the following
general predicate rules.

$$p(X,K) :- \text{bagof}(K', p^*(X,K'), L), \text{sum}(L,K).$$

$$\neg p(X,K) :- \text{bagof}(K', \neg p^*(X,K'), L), \text{sum}(L,K).$$

$$p^\pi(X) :- p(X,\_).$$

$$\neg p^\pi(X) :- \neg p(X,\_).$$

Before presenting the DCGs used for parsing of rough clauses and rough
queries, we first recall the concept of definite clause grammars. Definite
clause grammars are extensions of context free grammars that have been
proven useful for describing natural and formal languages and that may be
conveniently expressed and executed in Prolog. A DCG rule is executable
because it is just a notational variant of a logic rule that has the following
general form:

$$Head \longmapsto Body,$$

with the declarative interpretation that “a possible form for $Head$ is $Body$”. The
procedural interpretation of a DCG rule is that it takes an input list
of tokens, analyses some initial portion of that list, and then produces the
remaining portion (possibly enlarged) as output for further analysis.

Consider as an example the DCG rule: $p(X) \longmapsto q(X)$. This grammar
rule will be translated (expanded) by XSB Prolog into:

$$p(X, Li, Lo) :- q(X, Li, Lo).$$

The lists $Li$ and $Lo$ are input list of tokens and output list
of tokens, respectively. Consult [SSW+03] for further readings on DCG
evaluation in XSB Prolog.

It is possible to conveniently incorporate non-DCG predicates in the
body of a DCG rule, if included in curly brackets ($\{\ldots\}$). This facilitates
the construction of more complex grammar rules. In the following discussed
DCGs, the Prolog code part of a DCG is only presented as pseudo code.

The main DCGs used for the rewriting of rough clauses into internal
Prolog code are $\text{fact}/5$, $\text{upper_clause}/5$ and $\text{lower_clause}/6$. After to-
tokenization, a rough fact is (if syntactically correct) matched by the DCG
$\text{fact}/5$ (figure 4.2).
A rough fact is denoted by a rough literal referring to the upper approximation of a rough relation (see section 3.2.2). Rough clauses, on the other hand, can define either lower or upper approximations of a rough relation (see section 3.3). A rough clause defining the upper approximation of a rough relation is rewritten into internal Prolog code using the DCG `upper_clause/5`, presented in figure 4.3.

The operational semantics of a rough clause defining the lower approximation of a rough relation $P$ differs from the upper approximation in the sense that it is required that the indiscernibility classes of $P$ must not belong to $\neg P$. This requirement can be assured by the use of integrity constraints. Hence, whenever asserting new rough clauses or rough facts, the RKBS must check that these constraints are not violated. If a clause or fact violates an integrity constraint then the knowledge base becomes inconsistent and a model failure error is reported to the user. However, we will use model failure rules on the form:

$$\text{model\_failure}(P,\text{Arity}) :- \text{Constraint}. ,$$

as the use of integrity constraints is not supported by XSB Prolog. This does not change the meaning of the constraints but the evaluation of them differs. An integrity constraint can be seen as clause whose head is the atom `false`. We can simulate this by using the above rule and letting the system check that the head of the rule `(model\_failure(P,Arity))` never succeeds. If there are any model in the knowledge base that makes the `Constraint` true then the head will also be true. Hence, we have a model failure.

The DCG `lower_clause/6` for rough clauses defining the lower approximation of a rough relation is given in figure 4.4. The model failure rule generated by `lower_clause/6` (figure 4.4) states that if $\neg P^\pi(T)$ and `Body-Clause` are both true then a tuple unified with the variable T cannot belong to the lower approximation of the rough relation $P$ if the knowledge base should be consistent.

The body of a rough rule is a non-empty sequence of rough literals followed by zero or more quantitative measures (see definition 3.3.4). The DCG `body/5`, not presented in this text, is a grammar for the possible combinations of these parts. The DCGs for rewriting rough literals into internal Prolog code are presented in figure 4.5.
Implementation

\[
\text{fact(Fact,P,Arity)} \leftarrow \\
[\text{[upper], pred(P,T), [:], integer(I),} \\
\{ \\
\quad \text{Fact} \leftarrow P^*(T,I), \\
\quad \text{Arity} \leftarrow |T| \\
\}\].
\]

Figure 4.2: Definite clause grammar for rough facts. \text{Fact} is the generated Prolog code that is added to the KB, \text{P} is the predicate name, and \text{Arity} is the number of attributes in \text{T}.

\[
\text{upper_clause(Clause,P,Arity)} \leftarrow \\
[\text{[upper], pred(P,T),} \\
[:], \text{num(Alpha), [&, [name(F)]]}, \\
\text{body(BodyClause,BodySupp,F)}, \\
\{ \\
\quad \text{Arity} \leftarrow |T|, \\
\quad \text{Clause} \leftarrow (P^*(T,K) :- \text{BodyClause}, \\
\qquad K \leftarrow |\text{Alpha} \times \text{BodySupp}|) \\
\}\].
\]

Figure 4.3: Definite clause grammar for upper approximation clauses. \text{Clause} is the generated Prolog rule that will be asserted in the rough knowledge base. The function \text{F} is propagated to the DCG \text{body/5} and used when combining the support (\text{BodySupp}) of the rough relations occurring in the body of the clause.
lower_clause(Clause,P,Arity,Constraint) \[\mapsto\] [lower], pred(P,T), [\:-\], num(Alpha), [\&, [name(F)]], body(BodyClause,BodySupp,F), 
\{
    Arity \leftarrow \mid T\mid,
    Clause \leftarrow (P^\ast(T,K) :- BodyClause,
        K \leftarrow [\text{Alpha} \times \text{BodySupp}]),
    Constraint \leftarrow (\text{model_failure}(P,Arity) :-
        \neg P^\pi(T), BodyClause)
\}. 

Figure 4.4: Definite clause grammar for lower approximation clauses. 
Clause is the generated Prolog rule that will be asserted in the knowledge base, and Constraint is the model failure rule that cannot be true if the knowledge base should be consistent.

The quantitative measures supported by the rough knowledge base system are support (DCG supp/4: figure 4.6), strength (DCG strength/4: figure 4.7), coverage (DCG cov/4: figure 4.8) and accuracy (DCG acc/4: figure 4.9).

Rough queries are rewritten into Prolog code via the DCG compound_query/4, not covered in this text. This grammar allows any combinations of (one or more) queries on rough regions, quantitative measure assignments or assertions (see definition 3.3.7). A rough query can thus contain one or more of the following single rough queries:

- **upper(p(t))**: (figure 4.5). If the atom p(t) is ground then the query requests the truth value (yes or no) of whether the tuple t is a member of the upper approximation of P. If it is not ground then the query requests all instantiations for the variables in t.
• \texttt{lower}(p(t)): (figure 4.5). As the previous query except that the lower approximation of \( P \) is being considered.

• \texttt{boundary}(p(t)): (figure 4.5). As the previous query except that the boundary region of \( P \) is being considered.

• \( K = \texttt{Measure}(p(t)) \). Instantiate \( K \) with the value computed by the quantitative measure \( \texttt{Measure}(p(t)) \) (\texttt{supp}, \texttt{strength}, \texttt{cov} or \texttt{acc}).

• \( n \ 	exttt{relOp} \ 	exttt{Measure}(p(t)) \). If \( p(t) \) is ground then the answer is the truth value (yes or no) of whether the numerical value \( n \) is in relation \( \texttt{relOp} \ (<, >, =, \leq \text{ or } \geq) \) with the computed value of \( \texttt{Measure}(p(t)) \).

Moreover, the RKBS provides the classification procedure \texttt{classify}, discussed in section 3.3.

If the number of queries in a compound non-ground query is more than one then all equal (by name) variables in the different queries will be unified as in compound Prolog queries. For example, a compound query such as \texttt{upper}(p(A,B)),\texttt{lower}(r(A,B))., will generate as result all instantiations of the variables \( A \) and \( B \) that can be unified with tuples in both the upper approximation of \( P \) and the lower approximation of \( R \).
lower(Code,Support) ← [lower], pred(P,T),
{  
  Code ← (P(T,K), not \(\neg P^\pi(T)\)),  
  Support ← K  
}.  

upper(Code,Support) ← [upper], pred(P,T),
{  
  Code ← P(T,K),  
  Support ← K  
}.  

boundary(Code,Support) ← [boundary], pred(P,T),
{  
  Code ← (P(T,K), \(\neg P(T,_)\)),  
  Support ← K  
}.  

Figure 4.5: Definite clause grammars for the rough literals referring to the upper and lower approximation and the boundary region of a rough relation. Code represents the internal Prolog code that is the result of rewriting the rough literal.
supp(Code,Value) \rightarrow 
[supp], \text{pred}(P,T),
\{
  \text{Code} \leftarrow (P(T,K) ; \text{not } P^\pi(T), K \leftarrow 0)),
  \text{Value} \leftarrow K
\}.

Figure 4.6: Definite clause grammar for the quantitative measure support.

strength(Code,Value) \rightarrow 
[strength], \text{pred}(P,T),
\{
  \text{find all instantiations of } K1 \text{ in } P(X,K1),
  \text{find all instantiations of } K2 \text{ in } \neg P(X,K2),
  \text{Sum1} \leftarrow \text{the sum of all } K1 \text{ instantiations},
  \text{Sum2} \leftarrow \text{the sum of all } K2 \text{ instantiations},
  \text{Code} \leftarrow ((P(T,K0), K \leftarrow K0 / (\text{Sum1 + Sum2})) ;
                \text{not } P^\pi(T), K \leftarrow 0.0)),
  \text{Value} \leftarrow K
\}.

Figure 4.7: Definite clause grammar for the quantitative measure strength.
4.3. **Prolog implementation**

```prolog
cov(Code, Value) ←
[cov], pred(P, T),
{ find all instantiations of K2 in P(X,K2),
  Sum ← the sum of all K2 instantiations,
  Code ← ((P(T,K1), K ← K1 / Sum);
          (not P^π(T), K ← 0.0)),
  Value ← K
}.
```

Figure 4.8: Definite clause grammar for the quantitative measure coverage.

```prolog
acc(Code, Value) ←
[acc], pred(P, T),
{ Code ← ((P(T,K1), ¬P(T,K2), K ← K1 / (K1 + K2)) ;
       (P(T,K1), not ¬P^π(T), K ← 1.0) ;
       (not P^π(T), K ← 0.0)),
   Value ← K
}.
```

Figure 4.9: Definite clause grammar for the quantitative measure accuracy.
4.4 User interface

The user interface consists of the Java front-end and a static HTML page. From the web page it is possible to compile rough clauses, evaluate rough queries and overview the rough knowledge base. The web page is a simple tab-navigation system (see figure 4.10), where the tabs correspond to different modes.

The main mode is “Compile Rules or Queries”, a simple form for input of rough rules or queries from local files, URL, or manually written
4.4. User interface

Figure 4.11: Evaluation of queries.

rough statements in the text area. A user can compile rules via the button “Compile Rules” and query the knowledge base via the button “Query KB”.

The feedback from compiling rules (figure 4.10) or evaluating queries (figure 4.11) is reported in the feedback area. The feedback reflects successful operations via a “Rules compiled” feedback when compiling rules. It also reports the answers in table-form when evaluating queries. If the compilation of rules or evaluation of queries is unsuccessful for some reasons then, the feedback reflects the faults and the possible causes. Faults can
be syntax errors, undefined predicate errors or model failures, discussed in section 4.3.2 and 4.3.3.

When writing queries or rules it is useful to see which rough rules and facts are already asserted in the knowledge base. This can be done with the knowledge base viewer by changing mode to “Knowledge Base” (figure 4.12). The knowledge base viewer displays, in tab form, the rough predicates that are asserted in the knowledge base together with the rules or facts that define them.

Manually written queries can easily contain syntactic errors, and writing them can sometimes be time consuming. To facilitate this, we have added a tool for semi-automatic construction of queries. The user can access this tool by the navigation tab “Construct Queries”. The tool helps the user to build queries on the defined rough relations in the rough knowledge base.
4.4. User interface

![Image of knowledge base viewer]

Figure 4.12: Knowledge base viewer.
Chapter 5

Application Example

The example presented in this chapter is inspired by the methodology and results of Komorowski and Øhrn in [KO99]. They study the problem of prognosis of cardiac events in a group of patients with chest pain.

5.1 Prognostic problem

Study [GEvD96] has shown that the single most important independent predictor for future hard cardiac events (cardiac death or non-fatal myocardial infarction) is an abnormal scintigraphic scan pattern. However, performing such a test is an expensive procedure, and may sometimes be redundant with respect to making a prognosis. It is therefore desirable to identify patients that are in need of a scintigraphic scan and avoid it for patients which can be prognosed without such a test. In this example, we intend to run the data from [KO99] to test our system with a real-life problem. Moreover, we want to show the power and expressiveness of the language [VDM03a, VDM03b] implemented in the system.
The data consists of information for patients, observed in [GEvD96]. A group of 417 patients, represented by 335 rough facts, has been examined and each patient has an associated set of medical information. We thus get an information system with patients being objects and the attributes being information regarding the physical state of each patient in the universe. The attributes are boolean and are described in table 5.1.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>&gt; 70 years old?</td>
</tr>
<tr>
<td>Oldmi</td>
<td>Prior Infarction?</td>
</tr>
<tr>
<td>Hypert</td>
<td>Hypertension?</td>
</tr>
<tr>
<td>Dm</td>
<td>Diabetes?</td>
</tr>
<tr>
<td>Smok</td>
<td>Smoking?</td>
</tr>
<tr>
<td>Chol</td>
<td>Hypercholesterolemia?</td>
</tr>
<tr>
<td>Gender</td>
<td>Male?</td>
</tr>
<tr>
<td>Hfmed</td>
<td>History of dec. cordis?</td>
</tr>
<tr>
<td>Angp</td>
<td>History of angina?</td>
</tr>
<tr>
<td>Apstress</td>
<td>Angina during stress?</td>
</tr>
<tr>
<td>Stt</td>
<td>ST-T changes?</td>
</tr>
<tr>
<td>Scanabn</td>
<td>Abnormal scan?</td>
</tr>
<tr>
<td>Deathmi</td>
<td>Cardiac death or infarction?</td>
</tr>
</tbody>
</table>

Table 5.1: Attribute definitions.

The attributes Age to Scanabn in table 5.1 can be used to make a prognosis of the outcome of the Deathmi attribute. One can thus use the medical information to define the decision system Deathmi, where Deathmi is the decision attribute and the others being conditional attributes. In the following we will shorten the number of attributes in an attribute set for readability reasons, and write \{Age, \ldots, Scanabn\} instead of \{Age,Oldmi,Hypert,Dm,Smok,Chol,Gender,Hfmed,Angp,Apstress,Stt,}

---

1See appendix A for the data used in this example.
2The number of attributes are reduced from the original data in [GEvD96].
As discussed in chapter 3, a decision system yields a set of rough facts. Example facts yielded from the decision table in this example are:

- $\text{deathmi}(1,0,1,0,0,0,1,1,0,0,1) : 1$, stating that one patient with future cardiac death or myocardial infarction is above 70 years of age, has hypertension, history of dec. cordis and history of angina, and has an abnormal scintigraphic scan pattern. The other conditional attribute values are false.

- $\neg\text{deathmi}(0,0,1,0,0,0,0,0,0,0,0) : 11$, stating that 11 patients with a negative outcome of the decision attribute $\text{Deathmi}$ have only the conditional attribute hypertension being true.

### 5.3 Classification of patients

Given the information in the decision system $\text{Deathmi}$, one can make a prognosis of future cardiac death or myocardial infarction for patients using the classify procedure. The data for the new patients to be classified comes without the value of the decision attribute.

The following questions are examples of classification requests asked to the system.

- **Make a prognosis of future cardiac events for a patient above 70 years of age and with a prior infarction. The values of the rest of the conditional attributes are not known.**

  **Rough Query:**
  \[ C = \text{classify}(\text{deathmi}(1,1,\text{Hypert},... ,\text{Scanabn})). \]

  **Answer:** $C = [\text{deathmi} = \text{no}, 0.8529]$
  The answer states that a patient above 70 years of age and with a prior infarction have with approximately 85% certainty not a risk for future cardiac death or myocardial infarction.
5.4 Approximating rough relations via VPRSM

Quantitative measures in the body of rough clauses can be used to build more generalized rough approximations of a relation in the same spirit as in the variable precision rough set model (VPRSM) [Zia93], discussed in [VDM03b].

Quantitative measures in the body of a rough clause work as constraints in a similar way as in constraint logic programming. This means that it is possible to define new rough relations by other ones stating that a certain constraint must be fulfilled. These types of constraints are called precision control parameters in VPRSM. Let us define a new rough relation \textit{DeathmiApprox} using precision control parameters in the body of the clauses. \textit{DeathmiApprox} can for example be defined by the boundary region of \textit{Deathmi} and the constraint stating that the accuracy of \textit{Deathmi} should be above a certain threshold, say 70\% (rough clause 5.3). \textit{¬DeathmiApprox} is then defined by the boundary region of \textit{Deathmi} and the constraint stating that the accuracy of \textit{Deathmi} should be below 30\% (rough clause 5.4). This technique can also be seen as a way to “thin” the boundary region and make the approximations obtained less sensitive to possible noise in
the data. The following rough clauses define \textit{DeathmiApprox}:

\begin{align*}
\text{deathmiApprox}(\text{Age}, \ldots, \text{Scanabn}) & :-[1, \_] \quad (5.1) \\
& \quad \text{deathmi}(\text{Age}, \ldots, \text{Scanabn}). \\
\neg \text{deathmiApprox}(\text{Age}, \ldots, \text{Scanabn}) & :-[1, \_] \quad (5.2) \\
& \quad \neg \text{deathmi}(\text{Age}, \ldots, \text{Scanabn}).
\end{align*}

Rough clause 5.1 (5.2) states that the indiscernibility classes in the lower approximation of \textit{Deathmi} (\neg \textit{Deathmi}) are in the upper approximation of \textit{DeathmiApprox} (\neg \textit{DeathmiApprox}). The decision rules corresponding to these indiscernibility classes have 100\% accuracies and are therefore included in the new rough relation.

\begin{align*}
\text{deathmiApprox}(\text{Age}, \ldots, \text{Scanabn}) & :-[1, \text{sum}] \quad (5.3) \\
& \quad \text{deathmi}(\text{Age}, \ldots, \text{Scanabn}), \\
& \quad \neg \text{deathmi}(\text{Age}, \ldots, \text{Scanabn}), \\
& \quad \text{acc}(\text{deathmi}(\text{Age}, \ldots, \text{Scanabn})) \geq 0.7. \\
\neg \text{deathmiApprox}(\text{Age}, \ldots, \text{Scanabn}) & :-[1, \text{sum}] \quad (5.4) \\
& \quad \text{deathmi}(\text{Age}, \ldots, \text{Scanabn}), \\
& \quad \neg \text{deathmi}(\text{Age}, \ldots, \text{Scanabn}), \\
& \quad \text{acc}(\text{deathmi}(\text{Age}, \ldots, \text{Scanabn})) \leq 0.3.
\end{align*}

Rough clause 5.3 (5.4) says that any indiscernibility class \( t \) in the boundary such that \( \text{acc} (\text{deathmi}(t)) \geq 0.7 \) (\( \text{acc} (\text{deathmi}(t)) \leq 0.3 \)) is considered to be in the lower approximation of \textit{DeathmiApprox}.
(\neg \text{DeathmiApprox}).

\[
\text{deathmiApprox}(\text{Age}, \ldots, \text{Scanabn}) : [-1, \_]
\]

\[
\text{deathmi}(\text{Age}, \ldots, \text{Scanabn}),
\text{acc}(\text{deathmi}(\text{Age}, \ldots, \text{Scanabn})) > 0.3,
\text{acc}(\text{deathmi}(\text{Age}, \ldots, \text{Scanabn})) < 0.7.
\]

\[
\neg \text{deathmiApprox}(\text{Age}, \ldots, \text{Scanabn}) : [-1, \_]
\]

\[
\neg \text{deathmi}(\text{Age}, \ldots, \text{Scanabn}),
\text{acc}(\text{deathmi}(\text{Age}, \ldots, \text{Scanabn})) > 0.3,
\text{acc}(\text{deathmi}(\text{Age}, \ldots, \text{Scanabn})) < 0.7.
\]

Rough clauses 5.5 and 5.6 state that any indiscernibility class \( t \) in the boundary such that \( 0.3 < \text{acc}(\text{deathmi}(t)) < 0.7 \) remains in the boundary.

To see that the previously defined rough clauses in fact approximate the rough relation \textit{Deathmi} one can ask the following query to the system:

\textit{Which indiscernibility classes are both in the lower approximation of } \neg \textit{DeathmiApprox} \textit{and in the upper approximation of Deathmi, and what are the accuracies of the corresponding decision rules of Deathmi and } \neg \textit{Deathmi}? \\

\textbf{Rough Query:}

\[
\neg \text{deathmiApprox}(\text{Age}, \ldots, \text{Scanabn}),
\text{deathmi}(\text{Age}, \ldots, \text{Scanabn}),
\]

\[
K1 = \text{acc}(\neg \text{deathmi}(\text{Age}, \ldots, \text{Scanabn})),
K2 = \text{acc}(\text{deathmi}(\text{Age}, \ldots, \text{Scanabn})).
\]

\textbf{Answer:} See table 5.2. The answer states that two indiscernibility classes of \textit{Deathmi} are members of the concept \neg \textit{DeathmiApprox}. The accuracies of the decision rules of \textit{Deathmi} corresponding to these indiscernibility classes are lower than 0.3 which means that, by rough clause 5.4, the classes are included in the lower approximation of \neg \textit{DeathmiApprox}. 

### 5.5 Avoiding scintigraphic scan

If one wants to identify the patients for which the knowledge of the outcome of the scintigraphic scan pattern is required for making a prognosis of future cardiac death or myocardial infarction, the following methodology [KO99] can be used. The patients where knowledge of Scanabn is strictly required are those for whom excluding conditional attribute Scanabn causes migration into the boundary region from either the lower approximation or the outside region.

First, the set of attributes are reduced to not include Scanabn. The new rough relations $D$ (5.7) and its explicit negation $\neg D$ (5.8) are defined by the original, ignoring the last attribute, by the following rough clauses:

\[
\overline{d}(\text{Age}, \ldots, \text{Stt}) \equiv [1, \_] \quad \text{(5.7)}
\]
\[
\overline{\text{deathmi}}(\text{Age}, \ldots, \text{Stt}, \text{Scanabn}).
\]

\[
\overline{\neg d}(\text{Age}, \ldots, \text{Stt}) \equiv [1, \_] \quad \text{(5.8)}
\]
\[
\overline{\neg \text{deathmi}}(\text{Age}, \ldots, \text{Stt}, \text{Scanabn}).
\]

The set of patients migrating into the boundary region of $D$ from either the lower approximation or the outside region corresponds to the rough
relation \( Migrate \) which is defined by the following rough clauses:

\[
migrate(Age,...,Stt) :- [1,\min] \\
\overline{\Delta}(Age,...,Stt), \\
deathmi(Age,...,Stt,Scanabn). \\
\]

\[
migrate(Age,...,Stt) :- [1,\min] \\
\overline{\Delta}(Age,...,Stt), \\
\neg deathmi(Age,...,Stt,Scanabn). \\
\]

The clauses above state that the set of migrating individuals are those patients that are members of the lower approximation of \( Deathmi \) (5.9) or \( \neg Deathmi \) (5.10) and also members of the boundary region of \( D \). If a patient is captured by either of these rules one cannot make a reliable prognosis of future cardiac events without including the knowledge of the outcome of a scintigraphic scan. The explicit negation of \( Migrate \) is defined by the following rough clauses:

\[
\neg migrate(Age,...,Stt) :- [1,\sum] \\
\overline{\Delta}(Age,...,Stt), \\
\neg deathmi(Age,...,Stt,Scanabn). \\
\]

\[
\neg migrate(Age,...,Stt) :- [1,_] \\
d(Age,...,Stt). \\
\]

\[
\neg migrate(Age,...,Stt) :- [1,_] \\
\neg d(Age,...,Stt). \\
\]

Rough clauses 5.11, 5.12 and 5.13 capture the set of non-migrating patients. Clause 5.11 states that the patients that were members of the boundary region of \( Deathmi \) before dropping the \( Scanabn \) attribute are not migrating patients. For those patients one still cannot make a reliable prognosis of future cardiac events. Clauses 5.12 and 5.13 on the other hand capture
those patients that one still can discern, without knowledge of the \textit{Scanabn}
attribute, from the others and thus make a reliable prognosis of.

As previously discussed, we want to know for which patients the scintigraphic scan test is needed for a reliable prognosis. The following query is given to the system:

\textit{For which patients will it be useful to request the expensive scintigraphic scan and what is the percentage of patients for whom the test is needed?}

\textbf{Rough Query:} \texttt{migrate(Age,...,Stt),}  
\texttt{K1 = strength(migrate(Age,...,Stt)),}  
\texttt{K2 = strength(¬migrate(Age,...,Stt)).}

\textbf{Answer:} (see table 5.3) The strengths of the decision rules corresponding to \textit{Migrate} and \textit{¬Migrate} are used for computing the quotient of patients in the universe which are in the indiscernibility classes of \textit{Migrate}. The percentage of patients for whom the scintigraphic test is needed is computed by the sum

\[
\Phi = 100 \cdot \sum_{t \in \text{Migrate}} (\text{strength(migrate}(t)) + \text{strength(¬migrate}(t)))).
\]

The answer indicates that if only the migrating patients, given by table 5.3, undergo the expensive test scintigraphic scan, one may expect to avoid the test for approximately 98\% of all the patients. Notice that if \(\Phi\) would get too close to 100, then this would indicate that not that much would be gained by not requesting the scintigraphic scan for all patients. For some patients we may not be able to make a reliable prognosis of future cardiac events. Non-migrating patients that are members of the boundary region of \(D\) cannot be reliably prognosed. For these patients it may still be needed to perform the scintigraphic scan test if that is the opinion of a medical expert.
### 5.5. Avoiding scintigraphic scan

<table>
<thead>
<tr>
<th>K1</th>
<th>K2</th>
<th>Age</th>
<th>Oldmi</th>
<th>Hypert</th>
<th>Dm</th>
<th>Smok</th>
<th>Chol</th>
<th>Gender</th>
<th>Hfmed</th>
<th>Angp</th>
<th>Apstress</th>
<th>Stt</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0024</td>
<td>0.0073</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.0024</td>
<td>0.0049</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0.0049</td>
<td>0.0000</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$\Phi \approx 2\%$

Table 5.3: Migrating patients
Chapter 6

Discussion

We have implemented a knowledge base system for rough set theory within the logic programming framework (discussed in chapters 3 and 4). The kernel of the system is implemented in Prolog. The kernel provides methods for defining rough sets and querying rough knowledge. To this kernel, we have added a Java implementation that serves as a user front-end. The front-end uses servlets to interact with a user through a web page. From the web page, a user can request compilation of rough clauses and evaluation of rough queries. These requests are propagated by the front-end to the Prolog kernel that fulfills the requests.

The combination of rough set theory and logic programming is a novel approach so the implementation cannot be compared to any similar system. We can, however, discuss the pros and cons of our implementation choices. We will resort our discussion to mainly address the Prolog implementation.

6.1 Portability

The kernel of the system is implemented in XSB Prolog. The very most of the Prolog code is ISO Prolog compliant. However, we have used some predicates that are specific to XSB Prolog. These are exception handling methods used for detecting undefined predicates in user requests, and meth-
ods for socket communication with the Java front-end. As we want to have
the rough knowledge base system available on the World Wide Web, this
restricts the choice of Prolog system. XSB Prolog is an open source project
that is licensed by the GNU Lesser General Public License (LGPL)\(^1\), and
is thus preferably used rather than a commercial Prolog system. SWI Pro-
log\(^2\) is another example of a LGPL licensed Prolog system that more or
less adapts the ISO Prolog standard. The system is portable to any other
ISO Prolog compliant Prolog system with minor changes. The exception
handling and socket communication methods will have to be changed to
the ISO Prolog standard.

As the Prolog implementation handles the rough knowledge bases and
provides for them methods for extending and querying rough knowledge,
this kernel can be used as a stand-alone program. This implies the possi-
bility of integrating the rough knowledge base system in other applications.
One may, for example, want to integrate the system in applications dealing
with various problems within the field of artificial intelligence.

\section{6.2 Preserving consistency}

Rough clauses that define the lower approximations of rough relations are
rewritten into internal Prolog code together with a model failure rule (see
chapter 4). A model failure rule represents an integrity constraint. The
body of such a rule is the actual constraint and the head is a model failure
atom. As discussed in chapter 4, the system will check if there are any
model failure atoms that become true whenever a new rough clause or
rough fact is added to the rough knowledge base. If the head of a model
failure rule is true this would mean that the body is true, i.e. the integrity
constraint is true. If an integrity constraint is true in the knowledge base
then the knowledge base becomes inconsistent. The system thus needs
methods for preserving the consistency. To check if there are any true
model failure atoms every time new rough knowledge is added to the rough
knowledge base is not the most efficient solution. One may suggest that

\footnotesize
\(^1\)The software is free for use and free to change,
see \url{http://www.opensource.org/licenses/}.
\(^2\)\url{http://www.swi-prolog.org/}
the evaluation of model failure rules in the rough knowledge base is done only at some times, not after every assertion, for example at the end of every request from the user. This will, however, mean that it is harder to find out which clause or fact that violated the integrity constraint, a thing that should preferably be reported to the user.

There are no libraries provided by XSB Prolog that handle integrity constraints. There are however different ways of implementing integrity constraint checking in a Prolog system. We have considered the simplest solution that is easiest to implement, i.e. checking for model failures after every assertion. Another more efficient strategy is to build a system that manages dependency graphs [JB01] for the rough knowledge. With dependency graphs the system will only check for inconsistencies if there are any model failure rules that are explicitly or implicitly dependent on the rough knowledge to be asserted. A model failure rule is dependent on a certain rough knowledge if the body of the rule includes a rough literal representing the rough knowledge or any other literal that is implicitly defined by that rough knowledge. This strategy will probably improve the evaluation time of integrity constraints in the rough knowledge bases. However, as the construction of dependency graphs will require more complex operations when adding rough knowledge to the knowledge base, this strategy will probably imply an increase in compilation time for rough knowledge when there are no integrity constraints occurring in the rough knowledge base.

6.3 Performance measurements

To measure how well the rough knowledge base system performs, we will give some examples of compilation of clauses and evaluation of queries. All computation times presented in this section has been measured using the built-in XSB Prolog predicate cputime/1. The Java front-end has not been used during these measurements.

The following examples will not give a complete reference of the performance of the system, but will serve as an insight on how different variations of compilation orders of rough knowledge and how different constructions of rough queries will affect the system performance.
6.3.1 Compilation of rough clauses

The following example serves as a discussion on how integrity constraints will affect the system performance, and how possible performance flaws can be avoided by varying the order in which the rough knowledge is added to the rough knowledge base. Consider the \textit{Deathmi} decision system, discussed in chapter 5 and specified in appendix A. There are 418 individuals in the universe represented by 335 rough facts that correspond to the rough relations \textit{Deathmi} and \textit{¬Deathmi}. The arity of each rough fact is twelve, i.e. \textit{Deathmi} has twelve conditional attributes. In chapter 5 we discussed how it is possible to approximate rough relations via the variable precision rough set model. Consider two of the six rough clauses from chapter 5 that define the rough relation \textit{DeathmiApprox}:

\begin{align*}
\text{deathmiApprox}(Age,...,Scanabn) &:=[1,\text{sum}] \\
&\text{deathmi}(Age,...,Scanabn), \\
&\text{¬deathmi}(Age,...,Scanabn), \\
&\text{acc}(\text{deathmi}(Age,...,Scanabn)) \geq 0.7. \\
\text{¬deathmiApprox}(Age,...,Scanabn) &:=[1,\text{sum}] \\
&\text{deathmi}(Age,...,Scanabn), \\
&\text{¬deathmi}(Age,...,Scanabn), \\
&\text{acc}(\text{deathmi}(Age,...,Scanabn)) \leq 0.3.
\end{align*}

Rough clauses 6.1 and 6.2 will generate the following model failure rules, respectively:

\begin{verbatim}
model_failure(deathmiApprox,12) :-
  ¬deathmiApprox\textsuperscript{\pi}(T),
  deathmi(T,_),
  ¬deathmi(T,_) ,
  %% Code that restricts the accuracy of 
  %% deathmiApprox to be greater than 
  %% or equal to 0.7
\end{verbatim}
Discussion

model_failure(¬deathmiApproxA,12) :-
    deathmiApprox^\pi(T),
    deathmi(T,^),
    ¬deathmi(T,^),
    % Code that restricts the accuracy of
    % ¬deathmiApprox to be less than
    % or equal to 0.3

As we discussed in section 6.2, the model failure rules are evaluated after every assertion of new rough knowledge in the knowledge base. If no rough clauses defining the lower approximations of rough relations have been added to the knowledge base then there are no model failure rules in the rough knowledge base. If this is the case, then no time is needed for evaluating integrity constraints and we may thus expect less computation time. When compiling DeathmiApproxA together with Deathmi in a single request, the computation time may differ if we vary the order of the rough relations in the compilation request. If Deathmi is defined before DeathmiApproxA then there are no asserted model failure rules in the rough knowledge base that have to be evaluated when asserting the rough facts in Deathmi. On the other hand, if DeathmiApproxA is defined before Deathmi then one may expect the worst case in computation time. With that order, the system has to evaluate the model failure rules for every single fact in Deathmi. The computation time for checking these rules depends on the time to evaluate the bodies of the above model failure rules. Hence, as the number of facts increases the computation time for checking model failures may also increase.

In the following table (6.1), we display the best, worst, and average computation times in seconds, for ten computations of the following single compilation requests:

1. compilation of Deathmi

2. compilation of Deathmi together with DeathmiApproxA at the end

3. compilation of DeathmiApproxA together with Deathmi at the end
### 6.3. Performance measurements

<table>
<thead>
<tr>
<th>Request</th>
<th>Comp. time (sec)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Worst</td>
<td>Average</td>
</tr>
<tr>
<td>Deathmi</td>
<td>0.8200</td>
<td>0.9400</td>
<td>0.8652</td>
</tr>
<tr>
<td>Deathmi + DeathmiApprox</td>
<td>2.6700</td>
<td>2.7500</td>
<td>2.7183</td>
</tr>
<tr>
<td>DeathmiApprox + Deathmi</td>
<td>140.1499</td>
<td>140.9999</td>
<td>140.5057</td>
</tr>
</tbody>
</table>

Table 6.1: The best, worst, and average computation time for different compilation requests.

Table 6.1 shows that the computation time for Deathmi is rather low (average 0.8652 seconds). When we add the definitions of DeathmiApprox at the end of this requests the computation time is approximately three times longer (average 2.7183 seconds). When we compile the two rough relations in reverse order, the computation time increases to average 140.5 seconds. This example shows that one has to remember that the order of compilation can significantly change the compilation time, when using the constraint checking methods as it is implemented in our system.

Except for the last request above we may conclude that the average computation times for the compilation of these rough relations are satisfiably low. However, we may not draw any conclusions on the general performance of the system. Many more examples requesting the compilation of complex rough relations will be needed to be able to draw any general conclusions of the performance.

### 6.3.2 Evaluation of rough queries

The previous section provided measurements on the computation time when compiling two examples of rough relations. Having the relations Deathmi and DeathmiApprox defined in the rough knowledge base we may measure the evaluation time of rough queries on these relations.

To give an insight on the performance of evaluating rough queries we will alternate the queries on the two rough relations, regarding the number of single queries in a compound query and the rough regions on which the queries refer to. However, with the great multitude of ways for combin-
ing rough queries on the relations Deathmi and DeathmiApprox the small
numbers of queries that we provide will not give a complete performance
reference, but will serve as examples of the system performance.

Evaluation requests of numerous variants of ground rough queries have
been given to the rough knowledge base system. The evaluation time for
these ground queries were very low. The great part of the queries were
evaluated on less than 0.01 seconds.

The rough relations that we consider in this example vary regarding
the number of positive and negative examples that define them. We
may expect less computation time when evaluating rough queries request-
ing information in rough relations defined by few rough facts than when
evaluating rough queries on rough relations defined by many rough facts.
However, this depends on the efficiency of the unification procedure used by
the Prolog system and the implementation of the Prolog system. We may
also expect longer computation time when compiling quantitative measures
than when requesting indiscernibility classes in rough regions, as the com-
putation of quantitative measures will require more complex operations.
To validate these expectations, the following non-ground rough queries are
given to the rough knowledge base system:

1. upper(deathmi(Age,B,Hypert,Dm,Smok,Chol,Gender,Hfmed,
   Angp,Apstress,Stt,Scanabn)).

2. upper(∼deathmi(Age,B,Hypert,Dm,Smok,Chol,Gender,Hfmed,
   Angp,Apstress,Stt,Scanabn)).

3. K = strength(deathmi(Age,B,Hypert,Dm,Smok,Chol,Gender,
   Hfmed,Angp,Apstress,Stt,Scanabn)).

4. K = strength(∼deathmi(Age,B,Hypert,Dm,Smok,Chol,Gender,
   Hfmed,Angp,Apstress,Stt,Scanabn)).

5. K = strength(∼deathmi(Age,B,Hypert,Dm,Smok,Chol,Gender,
   Hfmed,Angp,Apstress,Stt,Scanabn)),
   boundary(deathmi(Age,B,Hypert,Dm,Smok,Chol,Gender,Hfmed,
   Angp,Apstress,Stt,Scanabn)).
6. upper(\(~\text{deathmi}(A, B, \text{Hypert}, D, \text{Smok}, Chol, Gender, Hfmed, Angp, Apstress, Stt, Scanabn)\))
acc(\(~\text{deathmi}(A, B, \text{Hypert}, D, \text{Smok}, Chol, Gender, Hfmed, Angp, Apstress, Stt, Scanabn)\)) > 0.1
strength(\(~\text{deathmi}(A, B, \text{Hypert}, D, \text{Smok}, Chol, Gender, Hfmed, Angp, Apstress, Stt, Scanabn)\)) > 0.1
supp(\(~\text{deathmi}(A, B, \text{Hypert}, D, \text{Smok}, Chol, Gender, Hfmed, Angp, Apstress, Stt, Scanabn)\)) > 1
cov(\(~\text{deathmi}(A, B, \text{Hypert}, D, \text{Smok}, Chol, Gender, Hfmed, Angp, Apstress, Stt, Scanabn)\)) > 0.

7. upper(\text{deathmiApprox}(A, B, \text{Hypert}, D, \text{Smok}, Chol, Gender, Hfmed, Angp, Apstress, Stt, Scanabn)).

8. upper(\(~\text{deathmiApprox}(A, B, \text{Hypert}, D, \text{Smok}, Chol, Gender, Hfmed, Angp, Apstress, Stt, Scanabn)\)).

9. \(K = \text{acc}(\text{deathmiApprox}(A, B, \text{Hypert}, D, \text{Smok}, Chol, Gender, Hfmed, Angp, Apstress, Stt, Scanabn))\).

10. \(K = \text{acc}(\(~\text{deathmiApprox}(A, B, \text{Hypert}, D, \text{Smok}, Chol, Gender, Hfmed, Angp, Apstress, Stt, Scanabn)\))\).

The best, worst, and average computation time for ten evaluations of the above queries are displayed in table 6.2. The queries are sorted by regards of average computation time.

The overall performance for non-ground rough queries on the rough relations is fairly good, considering the computation time from a user’s perspective. The average computation time for the evaluated queries differs from 0.0503 seconds to 2.2273 seconds (table 6.2). One can see differences in computation time between evaluating queries for the relations \text{Deathmi} and \neg \text{Deathmi}. The reason for this is that the number of rough facts describing \neg \text{Deathmi} is approximately six time greater than the number of rough facts describing \text{Deathmi}. Moreover, evaluating queries on the implicitly defined rough relation \text{DeathmiApprox} is more time consuming than evaluating queries on the explicitly defined rough relation \text{Deathmi}. The reason for this is that the evaluation trees for the unification procedure are generally more shallow when evaluating queries on an explicitly defined rough
Table 6.2: The best, worst and average computation time for the evaluation of non-ground rough queries. The rough queries are sorted by regards of average computation time.

relation than when evaluating queries on an implicitly defined rough relation. To evaluate the tenth rough query is the most time consuming. The computation time for the evaluation of this query is twice longer than the computation time for the rough query with the next-longest average computation time. The tenth query evaluates the accuracies of decision rules in $\neg DeathmiApprox$. The rough clauses that define this relation also integrate accuracies in the bodies of the clauses as quantitative measure constraints. Hence, evaluating the accuracy of a decision rule in $\neg DeathmiApprox$ will require the evaluation of multiple accuracies, which will have a negative effect on the computation time.

Our expectation that the evaluation of quantitative measures are more time consuming than the instantiations of indiscernibility classes in rough relations has been shown correct in the previous example. It would of course be nice to try more complex rough queries on more complex rough relations to get a more general overview of the performance of the system.
6.4 Conclusions

We have implemented a knowledge base system for rough set theory integrated within the logic programming framework. The combination of rough set theory with logic programming is a novel approach. The language supports definition of new rough sets combined by rough regions of other defined rough sets via rough clauses. The implementation served as a prototype system for the theory presented in [VDM03a, VDM03b].

We have added a user front-end to the system. This front-end is implemented in Java together with a web page that provides a graphical user interface. Having the system accessible on the World Wide Web makes it easier for researchers to access the system and makes it possible to provide the system with focus on usability. Numerous improvements can however be done regarding the usability of the system. Examples of such improvements are integration of tools that simplify the construction of requests that are to be given to the rough knowledge base system. We have implemented a tool for semi-automatic construction of rough queries. A tool that helps the user to construct rough clauses is another example of an improvement that would benefit usability, if added to the interface.

The rough knowledge base system has been tested on a medium real-life example and has shown fair performances. The expressivity of the language makes it easy to request complex information from rough sets via rough queries. Quantitative measures can be incorporated in the body of a rough clause as constraints. This makes it possible to define new rough sets in the spirit of the variable precision rough set model.

6.5 Future work

The rough knowledge base system is a prototype implementation. This first version has shown fair performances for the great part of the provided functionalities on some examples. We would like to try the system with more examples to evaluate its performance and to get indications on how to further improve it with new functionalities and optimizations.

The implemented rough language does not allow recursive clauses. One may consider allowing recursive clauses in a future version of the system.
The semantics of a language allowing recursive rough clauses is discussed in [VDM03a].

In a future version, the framework for integrity constraints in Prolog has to be optimized for better performances. It would also be nice to incorporate methods for the computation of reducts. This can be accomplished by the integration of the ROSETTA system [OK97], a popular software for rough set theory. The Java front-end can be extended to focus more on usability.

On the web page that gives the users access to the system, the users can extend the rough knowledge base with rough facts and rough clauses. These definitions can be either imported from local files, URLs, or from manually written definitions on the web page. At present, the decision tables can only be imported as rough facts. It would greatly benefit usability if parsers for other decision table formats would be integrated in the system. This would mean that, for example, decision tables exported from ROSETTA could be imported in the system without having first to rewrite them to our language. Moreover, manually written rough statements can easily contain syntactic errors. A tool for the construction of rough clauses and rough facts would avoid this problem, if being provided on the web page. Finally, methods for modification the rough knowledge base, such as retracting certain rough knowledge from the knowledge base, are other missing functionalities that would be nice to incorporate and that would improve usability even more.
6.5. Future work
Appendix A

The Deathmi Decision System

1 upper(deathmi(1,0,1,0,0,0,0,1,1,0,0,1)) : 1.
2 upper(deathmi(0,1,0,0,1,1,1,1,0,0,0,1)) : 1.
3 upper(deathmi(0,1,0,0,1,0,0,1,1,0,1,1)) : 1.
4 upper(deathmi(1,0,0,0,0,0,1,1,0,0,0,1)) : 1.
5 upper(deathmi(0,0,1,0,0,0,1,1,0,0,1,1)) : 1.
6 upper(deathmi(0,1,0,1,0,0,0,0,1,1,0,1)) : 1.
7 upper(deathmi(0,0,0,0,0,0,1,1,1,1,2,0)) : 1.
8 upper(deathmi(0,1,1,0,1,1,1,1,1,0,0,1)) : 1.
9 upper(deathmi(1,0,0,0,0,0,1,1,0,0,1,1)) : 1.
10 upper(deathmi(0,1,0,1,0,1,0,1,1,0,1,1)) : 1.
11 upper(deathmi(1,1,0,0,0,0,1,1,1,1,1,1)) : 1.
12 upper(deathmi(0,1,0,0,1,0,1,1,1,0,0,1)) : 1.
13 upper(deathmi(0,1,0,1,0,0,0,1,1,1,2,1)) : 1.
14 upper(deathmi(1,1,0,1,0,0,1,1,1,0,0,1)) : 1.
15 upper(deathmi(0,1,0,0,1,1,0,1,1,1,0,1,1)) : 1.
16 upper(deathmi(1,1,1,1,0,0,0,0,0,1,0,1)) : 1.
17 upper(deathmi(0,1,1,0,0,0,1,0,0,0,0,1)) : 1.
18 upper(deathmi(1,0,1,0,0,0,1,1,1,0,0,1)) : 1.
19 upper(deathmi(1,0,1,0,0,0,1,0,0,0,0,1)) : 1.
20 upper(deathmi(0,1,0,0,1,1,1,1,1,0,0,1)) : 1.
21 upper(deathmi(0,1,1,0,0,0,1,1,1,0,0,1)) : 1.
22 upper(deathmi(1,0,1,0,0,0,1,0,1,1,0,1)) : 1.
23 upper(deathmi(0,1,1,0,0,0,1,0,0,1,1,1)) : 1.
24 upper(deathmi(0,0,0,0,0,0,0,0,0,0,0,1)) : 1.
25 upper(deathmi(0,1,1,0,1,0,1,1,0,1,1,1)) : 1.
26 upper(deathmi(0,0,1,0,0,0,1,0,1,1,0,1)) : 1.
27 upper(deathmi(0,1,1,1,0,0,1,1,0,1,1,0,1)) : 1.
upper(deathmi(1,0,0,0,0,0,0,0,1,0,1)) : 1.
upper(deathmi(1,1,1,0,1,1,0,0,0,0,1)) : 1.
upper(deathmi(0,1,0,0,0,0,0,1,1,1,1)) : 1.
upper(deathmi(0,1,0,0,0,0,1,0,0,0,0)) : 1.
upper(deathmi(0,0,1,0,0,0,1,0,0,1,0)) : 1.
upper(deathmi(0,1,0,0,0,0,1,0,1,0,0)) : 1.
upper(deathmi(0,1,1,0,1,0,1,1,1,1,1)) : 1.
upper(deathmi(0,1,1,0,0,0,1,1,0,1,1)) : 1.
upper(deathmi(0,1,1,0,0,0,1,0,0,0,0)) : 1.
upper(deathmi(0,1,0,0,0,0,1,1,0,1,1)) : 1.
upper(deathmi(0,1,0,0,0,0,1,1,1,0,1)) : 1.
upper(deathmi(0,1,0,0,0,0,0,0,0,0,0)) : 1.
upper(~deathmi(0,0,0,0,0,0,0,0,0,0,0)) : 7.
upper(~deathmi(0,1,0,0,0,0,0,0,0,0,0)) : 1.
upper(~deathmi(0,0,0,0,0,0,0,0,0,0,0)) : 4.
upper(~deathmi(0,1,0,0,0,0,0,0,0,0,0)) : 5.
upper(~deathmi(0,1,0,0,0,0,0,0,0,0,0)) : 2.
upper(~deathmi(0,1,0,0,0,0,0,0,0,0,0)) : 3.
upper(~deathmi(0,1,0,0,0,0,0,0,0,0,0)) : 4.
upper(~deathmi(0,1,0,0,0,0,0,0,0,0,0)) : 4.
upper(~deathmi(0,1,0,0,0,0,0,0,0,0,0)) : 1.
upper(~deathmi(0,1,0,0,0,0,0,0,0,0,0)) : 1.

80
upper (~deathmi(0,1,0,1,1,1,0,0,0,0,0,1)) : 1.
upper (~deathmi(0,1,0,0,0,1,1,0,0,0,0,1)) : 1.
upper (~deathmi(0,1,0,1,0,0,1,0,1,1,0,1)) : 2.
upper (~deathmi(0,1,0,0,0,1,1,1,1,0,1,0)) : 1.
upper (~deathmi(1,1,0,0,0,0,1,1,1,1,0,1)) : 1.
upper (~deathmi(0,0,1,1,0,0,1,1,1,0,0,1)) : 1.
upper (~deathmi(0,1,0,0,0,0,1,1,1,0,0,0)) : 1.
upper (~deathmi(0,0,0,0,0,1,1,1,0,0,0,1)) : 1.
upper (~deathmi(1,1,0,0,0,0,1,0,0,2,0,1)) : 1.
upper (~deathmi(0,1,0,0,0,0,1,0,0,0,0,0)) : 1.
upper (~deathmi(0,1,0,0,0,1,1,0,1,1,0,0)) : 1.
upper (~deathmi(0,1,0,0,0,0,1,0,0,0,0,1)) : 1.
upper (~deathmi(0,0,0,0,0,1,1,0,0,0,0,1)) : 1.
upper (~deathmi(0,0,0,0,0,1,0,1,0,0,0,1)) : 1.
upper (~deathmi(0,1,0,0,0,0,1,0,0,0,0,0)) : 1.
upper (~deathmi(0,0,0,0,0,0,0,0,0,2,0,0)) : 3.
upper (~deathmi(0,0,0,0,0,0,0,1,1,0,0,0)) : 1.
upper (~deathmi(0,0,0,0,0,0,0,1,0,0,0,0)) : 1.
upper (~deathmi(0,0,0,0,0,0,1,1,1,0,0,0)) : 1.
upper (~deathmi(0,0,0,0,0,0,1,0,1,0,0,1)) : 1.
upper (~deathmi(0,0,0,0,0,0,1,0,0,0,1,1)) : 1.
upper (~deathmi(0,0,0,0,0,0,0,0,0,2,0,1)) : 3.
upper (~deathmi(0,1,0,0,0,0,0,1,1,0,0,0)) : 1.
upper (~deathmi(0,0,0,0,1,0,0,0,0,0,0,0)) : 2.
upper (~deathmi(0,1,0,0,1,1,0,1,2,1,1)) : 1.
upper (~deathmi(1,0,1,0,0,0,0,0,0,1,0)) : 1.
upper (~deathmi(0,0,1,0,0,1,1,1,2,0,1)) : 1.
upper (~deathmi(0,0,0,0,0,1,0,1,1,0,1)) : 1.
upper (~deathmi(1,0,0,0,0,1,1,1,0,1,0)) : 1.
upper (~deathmi(0,0,1,0,0,0,0,0,0,1,0)) : 2.
upper (~deathmi(0,0,1,0,0,1,1,0,1,1,1,0)) : 1.
upper (~deathmi(0,1,0,0,0,0,1,0,1,1,1,1)) : 1.
upper (~deathmi(0,0,0,0,0,0,1,0,0,0,0,1)) : 1.
upper (~deathmi(1,0,0,0,0,1,0,1,0,0,1,1)) : 1.
upper (~deathmi(1,0,1,0,0,1,1,0,1,0,0,0)) : 1.
upper (~deathmi(0,1,0,0,0,0,0,0,0,1,0,1)) : 1.
upper (~deathmi(0,1,0,0,0,1,0,1,0,0,1,1)) : 1.
upper (~deathmi(0,0,1,0,1,0,1,0,0,1,0,1)) : 1.
upper (~deathmi(0,0,0,0,0,0,0,0,0,0,0,1)) : 1.
upper (~deathmi(1,0,0,0,0,0,0,1,0,1,0,0)) : 1.
upper (~deathmi(0,1,1,0,0,1,0,1,0,0,1,1)) : 1.
upper (~deathmi(0,1,0,0,0,0,0,0,0,1,0,1)) : 1.
upper (~deathmi(0,1,0,0,1,1,1,0,0,0,0,1)) : 1.
upper (~deathmi(1,1,0,0,0,1,1,0,0,0,0,1)) : 1.
upper (~deathmi(0,0,1,1,0,1,0,1,1,1,1,1)) : 1.
upper (~deathmi(0,1,1,0,0,0,1,0,1,0,0,1)) : 1.
upper (~deathmi(0,0,0,0,0,0,0,0,0,0,0,0)) : 1.
The Deathmi Decision System

upper(\(\text{deathmi}(0,0,1,0,0,0,1,0,1,0,1,0)\)) : 1.
upper(\(\text{deathmi}(1,1,0,0,0,0,1,0,0,1,0,1)\)) : 1.
upper(\(\text{deathmi}(0,0,1,0,0,0,1,0,0,0,0,1)\)) : 1.
upper(\(\text{deathmi}(1,0,1,0,0,1,0,0,1,0,0,1)\)) : 1.
upper(\(\text{deathmi}(0,0,0,0,0,0,1,0,0,1,0,0)\)) : 1.
upper(\(\text{deathmi}(0,1,1,0,1,0,1,0,0,1,0,1)\)) : 1.
upper(\(\text{deathmi}(0,1,1,0,0,0,0,1,0,0,0,0)\)) : 2.
upper(\(\text{deathmi}(0,1,1,0,0,1,0,0,1,0,0,1)\)) : 1.
upper(\(\text{deathmi}(0,0,1,0,0,0,1,0,1,0,0,1)\)) : 1.
upper(\(\text{deathmi}(0,0,0,0,1,0,1,0,0,1,0,1)\)) : 1.
upper(\(\text{deathmi}(0,0,1,0,0,0,0,0,1,1,0,1)\)) : 1.
upper(\(\text{deathmi}(0,1,1,0,0,1,0,0,1,0,0,0)\)) : 1.
upper(\(\text{deathmi}(0,1,1,0,0,0,0,0,0,0,0,0)\)) : 1.
upper(\(\text{deathmi}(0,1,1,0,0,0,0,0,0,0,0,1)\)) : 1.
upper(\(\text{deathmi}(0,0,1,0,0,0,0,0,0,0,0,0)\)) : 2.
upper(\(\text{deathmi}(1,0,0,0,1,0,0,0,0,0,0,0)\)) : 1.
upper(\(\text{deathmi}(1,0,0,0,1,0,0,0,0,0,0,0)\)) : 1.
upper(\(\text{deathmi}(1,0,1,0,0,0,1,0,0,0,0,0)\)) : 1.
upper(\(\text{deathmi}(0,0,0,0,1,0,1,0,0,1,0,1)\)) : 1.
upper(\(\text{deathmi}(0,1,0,0,1,1,0,1,0,1,0,1)\)) : 1.
upper(\(\text{deathmi}(0,0,1,0,0,0,0,0,0,1,1,1)\)) : 1.
upper(\(\text{deathmi}(1,1,1,0,0,0,1,0,0,0,0,0)\)) : 1.
upper(\texttt{~deathmi(0,1,0,1,0,0,1,0,0,2,0,1)}) : 1.
upper(\texttt{~deathmi(1,1,0,1,0,0,0,1,0,1,0,1)}) : 1.
upper(\texttt{~deathmi(0,0,1,0,0,0,1,0,0,2,0,1)}) : 1.
upper(\texttt{~deathmi(0,1,0,0,1,1,0,0,0,2,0,1)}) : 1.
upper(\texttt{~deathmi(0,0,0,0,1,0,0,0,2,0,1)}) : 2.
upper(\texttt{~deathmi(0,1,0,0,0,1,1,0,0,1,1,1)}) : 1.
upper(\texttt{~deathmi(0,1,0,1,0,1,0,1,1,0,1,0)}) : 1.
upper(\texttt{~deathmi(0,1,0,0,0,1,1,1,0,2,1,1)}) : 1.
upper(\texttt{~deathmi(0,0,0,0,0,0,1,0,0,2,0,0)}) : 3.
upper(\texttt{~deathmi(0,1,0,1,1,1,1,1,0,1,1,1)}) : 1.
upper(\texttt{~deathmi(0,1,1,0,0,0,1,1,1,2,1,1)}) : 1.
upper(\texttt{~deathmi(0,1,1,0,0,0,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(\texttt{~deathmi(1,1,1,0,0,0,1,1,1,2,0,0,0)}) : 1.
upper(¬deathmi(0,0,1,0,0,1,1,1,0,2,0,0)) : 1.
upper(¬deathmi(0,1,0,1,1,1,1,0,0,1,0,1)) : 1.
upper(¬deathmi(0,0,0,0,0,1,0,0,0,2,1,0)) : 1.
upper(¬deathmi(0,1,1,1,0,0,1,0,0,1,0,1)) : 1.
upper(¬deathmi(1,0,1,0,0,0,1,0,2,0,0,0)) : 1.
upper(¬deathmi(1,0,1,0,0,0,1,0,2,0,1,1)) : 1.
upper(¬deathmi(0,0,0,1,0,1,1,1,1,1,2,1,1)) : 1.
upper(¬deathmi(0,1,1,1,0,0,1,0,0,2,0,1,0)) : 1.
upper(¬deathmi(0,1,1,1,0,0,1,0,0,1,1,2,0,1)) : 1.
upper(¬deathmi(1,1,0,1,0,0,0,0,0,2,1,0,0)) : 1.
upper(¬deathmi(0,1,0,0,1,0,1,0,0,2,1,0,1)) : 1.
upper(¬deathmi(0,0,0,0,0,1,0,0,0,2,0,0,0)) : 1.
upper(¬deathmi(0,0,0,1,1,0,1,0,0,1,0,1,0)) : 1.
Bibliography


Index

classify ..................... 49
compile_file/2 ........... 40
rkb_scan/4 .................. 44

accuracy ..................... 16, 48
definition ..................... 16
arity ........................... 24
atom ........................... 24

body ............................ 25
boundary region ............. 10

certainty factor ............. 33
classification procedure ..... 32
classify ......................... 32
closed-world reasoning ...... 23
compilation of rough clauses.40–41
conjunction ................... 24
consistency .................... 9
constant ........................ 24
coverage ....................... 16, 48
definition ....................... 16

crisp set ........................ 10

DCG ............................ 41, 45–53
acc/4 .......................... 52
body/5 ......................... 47
boundary/4 ..................... 50
compound_query/4 41, 48
cov/4 .......................... 52
fact/5 .......................... 45
lower/4 ........................ 50
lower_clause/6 .............. 46
strength/4 ..................... 51
supp/4 .......................... 51
upper/4 ........................ 50
upper_clause/5 .............. 46

DCG rule ....................... 45
decision ........................ 7
negative ....................... 11
positive ....................... 11
decision attribute .......... 7
decision class ................. 8
definition ...................... 8
decision rule .................. 14
approximative ............... 17
exact ............................ 17
decision synthesis .......... 17–18
decision system ............. 7
consistent ................................. 9
inconsistent ............................. 9
decision table ............................. 7
default negated literal ................. 24
default negation ......................... 23
definite clause grammar .............. 41, 45
discernibility function ................. 13
definition ............................... 13
discernibility matrix .................... 13
definition ............................... 13
ELP ....................................... 25
equivalence class ........................ 8
equivalence relation ..................... 8
evaluation of rough queries ............ 40–41
explicit negation ......................... 23
extended logic program ............... 23, 25
fact ........................................ 25
general decision ......................... 9
definition ............................... 9
general predicate rules ................. 45
ground
atom ....................................... 24
clause ..................................... 25
head ........................................ 25
inconsistency ............................ 9
indiscernibility ......................... 8–9
indiscernibility relation ............... 8
definition ............................... 8
reflexive .................................. 8
symmetric ................................ 8
transitive ................................ 8
indiscernible ....................... 8
information system ..................... 6–8
definition ............................... 6
information table ........................ 6
integrity constraint ...................... 25
integrity constraint violation ........... 40
ISO Prolog standard .................... 35
knowledge base ......................... 21
knowledge base system ................. 21
logic program ......................... 23–25
syntax ...................................... 24
logical connective ....................... 24
lower approximation .................... 10
model failure ............................ 40, 46
objective literal ......................... 24
open-world reasoning ................... 23
paraconsistent semantics ............... 23
predicate symbol ....................... 24
program clause ......................... 24
Prolog implementation ................. 40
Prolog server ............................ 40
quantitative measure .................... 15–16
assignment ............................ 32
constraint ............................... 30
syntax ..................................... 29
reduct ..................................... 12–14
definition ............................... 13
request type ............................. 40
RKB ........................................ 36
RKBS ..................................... 36
RKBServer ............................... 38
<table>
<thead>
<tr>
<th>Index Item</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RKBServerThread</td>
<td>38</td>
</tr>
<tr>
<td>RKBServlet</td>
<td>37</td>
</tr>
<tr>
<td>RKBToXSBClient</td>
<td>38</td>
</tr>
<tr>
<td>rough clause</td>
<td>27</td>
</tr>
<tr>
<td>rough complement</td>
<td>23</td>
</tr>
<tr>
<td>rough fact</td>
<td>26</td>
</tr>
<tr>
<td>rough knowledge</td>
<td>22</td>
</tr>
<tr>
<td>rough knowledge base</td>
<td>21–22</td>
</tr>
<tr>
<td>rough language</td>
<td>28</td>
</tr>
<tr>
<td>rough literal</td>
<td>29</td>
</tr>
<tr>
<td>rough membership function</td>
<td>11</td>
</tr>
<tr>
<td>rough program</td>
<td>31</td>
</tr>
<tr>
<td>rough relation</td>
<td>22</td>
</tr>
<tr>
<td>rough set</td>
<td>10, 22</td>
</tr>
<tr>
<td>definition</td>
<td>11, 22</td>
</tr>
<tr>
<td>rough set approximation</td>
<td>10</td>
</tr>
<tr>
<td>scanner</td>
<td>40, 44</td>
</tr>
<tr>
<td>server</td>
<td>40</td>
</tr>
<tr>
<td>set approximation</td>
<td>9–12</td>
</tr>
<tr>
<td>strength</td>
<td>16, 48</td>
</tr>
<tr>
<td>definition</td>
<td>16</td>
</tr>
<tr>
<td>support</td>
<td>15, 48</td>
</tr>
<tr>
<td>definition</td>
<td>16</td>
</tr>
<tr>
<td>support-combining function</td>
<td>30</td>
</tr>
<tr>
<td>syntax error</td>
<td>40</td>
</tr>
<tr>
<td>system overview</td>
<td>36–38</td>
</tr>
<tr>
<td>term</td>
<td>24</td>
</tr>
<tr>
<td>trust</td>
<td>30</td>
</tr>
<tr>
<td>undefined predicate error</td>
<td>40</td>
</tr>
<tr>
<td>upper approximation</td>
<td>10</td>
</tr>
<tr>
<td>user interface</td>
<td>53–55</td>
</tr>
<tr>
<td>variable</td>
<td>24</td>
</tr>
<tr>
<td>XSB process</td>
<td>38</td>
</tr>
</tbody>
</table>
This thesis presents the implementation of a knowledge base system for rough sets [Paw92] within the logic programming framework. The combination of rough set theory with logic programming is a novel approach. The presented implementation serves as a prototype system for the ideas presented in [VDM03a, VDM03b]. The system is available at “http://www.ida.liu.se/rkbs”.

The presented language for describing knowledge in the rough knowledge base caters for implicit definition of rough sets by combining different regions (e.g. upper approximation, lower approximation, boundary) of other defined rough sets. The rough knowledge base system also provides methods for querying the knowledge base and methods for computing quantitative measures.

We test the implemented system on a medium sized application example to illustrate the usefulness of the system and the incorporated language. We also provide performance measurements of the system.

Nyckelord
Rough set theory, rough sets, logic programming, knowledge bases, artificial intelligence, uncertain reasoning, incomplete reasoning, quantitative measures.
Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är krävande för upphovsmannens litterära eller konstnärliga anseende eller egenart. För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement. For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Robin Andersson
Linköping, 11th June 2004