Query Answering in Rough Knowledge Bases

Aida Vitória[1], Carlos Viegas Damásio[2], and Jan Małuszyński[3]

[1] Dept. of Science and Technology, Linköping University,
S 601 74 Norrköping, Sweden
aidvi@itn.liu.se
[2] Centro de Inteligência Artificial (CENTRIA),
Dept. Informática, Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
cd@di.fct.unl.pt
[3] Dept. of Computer and Information Science,
Linköping University, S 581 83 Linköping, Sweden
janma@ida.liu.se

Abstract. We propose a logic programming language which makes it possible to define and to reason about *rough sets*. In particular we show how to test for *rough inclusion* and *rough equality*. This extension to our previous work [7] is motivated by the need of these concepts in practical applications.

Key words: Rough sets, logic programming, stable models, uncertain reasoning.

1 Introduction

A rough set [4] is an approximation of a subset of the universe of discourse. It is usually defined by a finite decision table, whose rows correspond to some objects of the universe. We present a natural extension of this formalism that gives direct support for integration of background (e.g. expert) knowledge, reasoning, and possibility to combine different rough sets for defining new ones. We introduce a language for defining rough sets and its compilation to expressive logic programming language. The idea is similar to our previous work [7] but extends it substantially. Our new language is more expressive, and more suitable for defining concepts in terms of the notions of rough set theory, as illustrated by the examples in the paper. The precise meaning of the definitions is provided by translation to extended logic programs, under the *paraconsistent stable model semantics* [5, 6]. In this way we link rough sets with paraconsistent logic.

A program may have several (paraconsistent) stable models (or no model at all). Each of the models describes a family of rough sets and can be seen as a possible scenario of rational beliefs supported by the knowledge represented by the program. The price we pay for the extra expressiveness is the increase in time complexity: to know whether a literal belongs to a stable model is a NP-complete problem. However, efficient implementations exist of the stable model semantics (see e.g. [2, 3]) and these systems can be readily used for querying rough sets defined in our language.

2 Preliminaries

2.1 Rough sets

We deal with a universe U of objects with *attributes*. An attribute a can be seen as a partial function $a : U \to V_a$, where V_a is called the *value domain* of a. Thus, every object is associated with a tuple of attributes. We assume that this tuple is the only way of referring to the object: different objects with same attribute values are *indiscernible*. A subset S of U can only be characterized by two sets of tuples: S^+, including the tuples of the objects in S, and S^-, including the tuples of the objects in the complement of S. Notice that S^+ and S^- neither need to be disjoint nor they have to cover the universe.

A *rough set* (or *rough relation*) S is a pair (S^+, S^-) such that $S^+, S^- \subseteq \prod_{1 \leq i \leq n} V_{a_i}$, for some non empty set of attributes $\{a_1, \cdots, a_n\}$.

The indiscernibility classes S^+ (called the *upper approximation* and also represented as $\overline{S}$) include all elements that possibly belong to S. Similarly, the classes S^- include all elements that possibly do not to belong to S. The *lower approximation* of S defined as $(S^+ - S^-)$ (and represented as $\underline{S}$) includes only the non conflicting elements that belong to S. The *rough complement* of a rough set $S = (S^+, S^-)$ is the rough set $\neg S = (S^-, S^+)$.

We want to stress that in our work a rough set is not defined in terms of objects of the universe, but instead in terms of the tuples that describe each equivalence class (i.e. class of indiscernible objects) to which the objects belong. Moreover, the terms "rough set" and "rough relation" are used interchangeably.

2.2 Extended Logic Programs

This section surveys well-known concepts of logic programming needed in the sequel. For more details the reader is referred to [1, 5, 6]. We start by presenting the syntax of *extended logic programs* and defining briefly the *paraconsistent stable model semantics* of extended logic programs [5, 6], the target logic programming language of the transformations discussed in section 3.2. We resort only to the disjunctive free fragment of the languages described in [5, 6].

Without loss of generality, we consider only ground logic programs (i.e. no variables occur in an atom A). Assume that the alphabet $\mathcal{K}$ is a set of propositional variables, called *atoms*. An *objective literal* L is either an atom $A \in \mathcal{K}$ or its explicit negation $\neg A$. The set of all objective literals is $\mathcal{K} \cup \neg\mathcal{K}$, where $\neg\mathcal{K} = \{\neg A : A \in \mathcal{K}\}$. The default negation of a literal L is represented by *not* L (also called default negated literal). A *literal* is either an objective literal L or its default negation *not* L.

Intuitively, an objective literal represents a (positive or negative) evidence, while the default negated literal represents a lack of (respectively, positive or negative) evidence. This makes it possible to represent differently the information that a flight departed without delay obtained from the flight control, from lack of the delay announcement. We allow coexistence of positive and negative evidence; for formalizing this we will use a paraconsistent logic.

A *program clause* is an expression $L_0 \texttt{:-} L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$ where each L_i is an objective literal and $n \geq 0$. The left side of the clause (w.r.t. :-) is called the *head* and the right side is designated as *body* of the clause. Informally, a program clause represents an implication: if every literal in the body is true then the head must also be true[4]. An *integrity constraint* has the form $\texttt{:-} L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$ with $n \geq 1$, and can be seen as a clause with the head being $false$.

An *extended logic program* (ELP) is a set of program clauses and integrity constraints. A definite extended logic program is simply an ELP without integrity constraints and occurrences of default negated literals.

An *interpretation* $\mathcal{I}$ of an extended logic program $\mathcal{P}$ is any subset of $\mathcal{K} \cup \neg\mathcal{K}$. As usual, an interpretation settles the set of *true* literals. If $L \in \mathcal{I}$ then the objective literal L has the truth value *true*, and if $L \notin \mathcal{I}$ then the objective literal L is *false* (i.e. *not* L is *true*).

An interpretation $\mathcal{I}$ *satisfies* a program clause if the corresponding implication holds in $\mathcal{I}$, and *satisfies* an integrity constraint if at least one literal in its body is false. A *model* of an extended logic program $\mathcal{P}$ is any interpretation that satisfies every program clause and integrity constraint of $\mathcal{P}$. Intuitively, an integrity constraint discards all model candidates that make every literal in its body *true*.

A program may have many models, they are ordered by set inclusion. Any definite ELP has a least model, other ELP's may have several minimal models. We want to consider only the models where each objective literal can be justified by some evidence in the program. This intuition is captured by the following definition:

Definition 1. *Let P be an extended logic program and $\mathcal{I}$ an interpretation. The reduct of P with respect to $\mathcal{I}$ is the definite extended logic program $P^{\mathcal{I}}$ such that $L_0 \texttt{:-} L_1, \ldots, L_m$ is in $P^{\mathcal{I}}$ iff there is a program clause of the form $L_0 \texttt{:-} L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n$ from P such that $\{L_{m+1}, \ldots, L_n\} \cap \mathcal{I} = \{\}$. The interpretation $\mathcal{I}$ is a paraconsistent stable model of P iff I is the least model of $P^{\mathcal{I}}$ and $\mathcal{I}$ satisfies all integrity constraints of P.*

Whenever confusion does not arise, we sometimes use the term model instead of paraconsistent stable model.

Finally, we refer that *Smodels* [3] and *dlv* [2] are currently available systems for computing stable models of programs (often with tens of thousands of clauses). Both systems can also handle integrity constraints, and can be easily used to determine paraconsistent stable models of extended logic programs.

3 A Language for Defining Rough Relations

In this section, we present a language for defining and querying rough relations, based on logic programming. It substantially extends our previous work [7] by

[4] If variables are allowed then they should be understood as universally quantified.

allowing lower approximations and boundaries of rough relations in the head and in the body of clauses. We illustrate the need and potential usefulness of such extension by motivating examples. We also give the semantics of the new language. Note that the least model semantics of [7] is no more applicable.

3.1 The syntax and motivating examples

In this section, variables can occur in atoms (i.e. we allow non-ground atoms). Thus, an *atom* A is an expression of the form $p(t_1, \ldots, t_m)$, where p is an m-ary predicate symbol and $t_1, \ldots, t_m$ are variables or constants. As in the previous section, an *objective literal* L is an expression of the form A or $\neg A$, where A is an atom.

Given an objective literal L, expressions of the form $\underline{L}$, $\overline{L}$, or $\underline{\overline{L}}$ are called *rough literals*.

A *rough clause* is a formula of the form H :- $B_1, \ldots, B_n$. , where H and every B_i $(0 \leq i \leq n)$ are rough literals.

A *rough program* $\mathcal{P}$ is a set of rough clauses. Moreover, clauses with an empty body (i.e. $n = 0$) are usually called *facts* and are written as H. , where H is a rough literal.

Intuitively, each predicate p denotes a rough relation P and we use rough literals to represent evidence about tuples. For instance, the facts $\overline{p}(t_1, \ldots, t_n)$. and $\overline{\neg p}(t_1, \ldots, t_n)$. express the information that the tuple $<t_1, \ldots, t_n>$ belongs both to the rough relation P and to its complement $\neg P$ (thus, to the boundary of P, denoted by $\overline{p}(\ldots)$). Obviously, $\overline{p}(\ldots)$ and $\overline{\neg p}(\ldots)$ denote the same set of tuples. The lower (upper) approximation of P is represented by $\underline{p}(\ldots)$ $(\overline{p}(\ldots))$.

A decision table $\mathcal{D}$ can be represented in our language. A row $<c_1, \ldots, c_n>$ of $\mathcal{D}$ corresponding to a positive (negative) example, where each c_i is the value of a conditional attribute, is represented as the fact $\overline{d}(c_1, \ldots, c_n)$. $(\overline{\neg d}(c_1, \ldots, c_n).)$.

Since rough clauses allow lower and upper approximations of a relation as well as boundaries to occur both in the body and head of a clause, it is possible to define separately each of the regions (i.e. lower and upper approximations and boundary) of a rough relation in terms of regions of other rough relations. For instance, we can represent that the boundary of a rough relation Q is contained in the lower approximation of another rough relation P. If predicates q and p denote the rough relations Q and P, respectively, then the rough clause $\underline{p}(X_1, \ldots, X_n)$:- $\overline{\underline{q}}(X_1, \ldots, X_n)$. captures such information.

The following examples motivate the potential usefulness of our language.

Example 1. A relation *Train* has two arguments (attributes) representing time and location, respectively. Two sensors automatically detect presence/absence of an approaching train at a crossing, producing facts like $\overline{\texttt{train}}$`(12:50,Montijo).` automatically added to the knowledge base. A malfunction of a sensor may result in the contradictory fact $\overline{\neg\texttt{train}}$`(12:50,Montijo).` being added, too. Crossing is allowed if for sure no train approaches. This can be described by the following clause involving lower approximation in the body.

$\overline{\texttt{cross}}$`(X,Y) :-` $\underline{\neg\texttt{train}}$`(X,Y).`

Example 2. Statistical data on purchases of certain product during a calendar year is organized as a decision table with the following 4 attributes defining groups of customers:

`Area` - zip code of the area where the customer lives
`Income` - customer's income interval
`Age` - customer's age interval
`Unimulti` - does the customer live in a house or in an apartment?

Two experts classify every group as active or not active depending on the number of transactions during the year. The opinions of the experts may be different, thus the decision table defines a rough relation. The marketing department uses the activity tables `act1` and `act2` from two consecutive years to identify the groups of *growing activity* (`ga`). The tables are represented as facts in our language. The activity of a group may be defined: (1) as *definitely growing*, if the group was possibly inactive in year 1 and definitely active in year 2; (2) as *definitely non growing*, if its activity changed from possibly active to definitely inactive; and (3) as a *boundary*, if the activity was boundary in both years. This can be described by the following clauses:

$$\underline{\texttt{ga}}\texttt{(X, Y, W, Z)} \texttt{ :- } \overline{\neg\texttt{act1}}\texttt{(X, Y, W, Z)}, \underline{\texttt{act2}}\texttt{(X, Y, W, Z).} \quad (1)$$
$$\underline{\neg\texttt{ga}}\texttt{(X, Y, W, Z)} \texttt{ :- } \overline{\texttt{act1}}\texttt{(X, Y, W, Z)}, \underline{\neg\texttt{act2}}\texttt{(X, Y, W, Z).} \quad (2)$$
$$\overline{\underline{\texttt{ga}}}\texttt{(X, Y, W, Z)} \texttt{ :- } \underline{\overline{\texttt{act1}}}\texttt{(X, Y, W, Z)}, \underline{\overline{\texttt{act2}}}\texttt{(X, Y, W, Z).} \quad (3)$$

3.2 The Semantics

In this section, we present a transformation of rough programs into extended logic programs, introduced in section 2.2. In this way, we obtain both a declarative and operational semantics for our language.

The intuition is as follows. Assume that P and Q are the rough relations denoted by predicates p and q, respectively. Then, the literal $p(t_1, \ldots, t_n)$ is a statement that the tuple $<t_1, \ldots, t_n>$ is in P and the literal $\neg p(t_1, \ldots, t_n)$ indicates that tuple $< t_1, \ldots, t_n >$ is not in P. (i.e. belongs to $\neg P$). The default negated literal *not* $p(t_1, \ldots, t_n)$ (*not* $\neg p(t_1, \ldots, t_n)$) states that there is no evidence that the tuple $< t_1, \ldots, t_n >$ is a positive (negative) example of P. Now rough literals can be equivalently expressed by conjunctions of literals of ELPs, as formalized by the following transformation τ_2:

$$\begin{aligned}
\tau_2(\underline{p}(t_1, \ldots, t_n)) &= p(t_1, \ldots, t_n), \mathit{not}\ \neg p(t_1, \ldots, t_n)\ , \\
\tau_2(\underline{\neg p}(t_1, \ldots, t_n)) &= \neg p(t_1, \ldots, t_n), \mathit{not}\ p(t_1, \ldots, t_n)\ , \\
\tau_2(\overline{p}(t_1, \ldots, t_n)) &= p(t_1, \ldots, t_n)\ , \\
\tau_2(\overline{\neg p}(t_1, \ldots, t_n)) &= \neg p(t_1, \ldots, t_n)\ , \\
\tau_2(\underline{p}(t_1, \ldots, t_n)) &= p(t_1, \ldots, t_n), \neg p(t_1, \ldots, t_n)\ , \\
\tau_2(\underline{\overline{p}}(t_1, \ldots, t_n)) &= p(t_1, \ldots, t_n), \neg p(t_1, \ldots, t_n)\ , \\
\tau_2(\overline{\underline{\neg p}}(t_1, \ldots, t_n)) &= \tau_2(\underline{\overline{p}}(t_1, \ldots, t_n))\ , \\
\tau_2((B_1, \ldots, B_n)) &= \tau_2(B_1), \ldots, \tau_2(B_n)\ .
\end{aligned}$$

This transformation can be used to compile rough literals in the bodies of source (i.e. rough) program clauses. However, the translation is not directly

applicable to the heads, since the heads in the target programs can contain neither conjunctions of literals nor default literals. Therefore, it may be necessary to compile a clause in the source program into a clause and an integrity constraint of the target program, as described below. For example, consider a rough clause like $\underline{p}(\ldots)\text{:-}\ \overline{q}(\ldots).$ stating that the boundary of Q is contained in the lower approximation of P. Any element in the boundary of Q should be also considered a positive example of P but there shouldn't be evidence that those tuples are examples of $\neg P$. Moreover, a tuple t belongs to the boundary of Q if and only if it represents both positive and negative evidence of it. Thus, $p(\ldots)\text{:-}\ q(\ldots), \neg q(\ldots).$ and $\text{:-}\ \neg p(\ldots), q(\ldots), \neg q(\ldots).$ capture the same information as the rough clause above. The program clause states that tuples belonging to both Q and $\neg Q$ also belong to P, while the integrity constraint does not allow those tuples to belong to $\neg P$.

The discussion above gives a motivation for the formalization of the translation of rough clauses into clauses of an extended logic program. This formalization is defined as the following function τ_1 which refers to the above defined function τ_2.

$$
\begin{array}{ll}
\tau_1(\underline{p}(t_1,\ldots,t_n)\text{:-}\ B.) & = \{p(t_1,\ldots,t_n)\text{:-}\ \tau_2(B).\,,\ \text{:-}\ \neg p(t_1,\ldots,t_n)\,,\tau_2(B).\}\ , \\
\tau_1(\overline{p}(t_1,\ldots,t_n)\text{:-}\ B.) & = \{p(t_1,\ldots,t_n)\text{:-}\ \tau_2(B).\}\ , \\
\tau_1(\underline{\neg p}(t_1,\ldots,t_n)\text{:-}\ B.) & = \{\neg p(t_1,\ldots,t_n)\text{:-}\ \tau_2(B).\,,\ \text{:-}\ p(t_1,\ldots,t_n)\,,\tau_2(B).\}\ , \\
\tau_1(\overline{\neg p}(t_1,\ldots,t_n)\text{:-}\ B.) & = \{\neg p(t_1,\ldots,t_n)\text{:-}\ \tau_2(B).\}\ , \\
\tau_1(\overline{p}(t_1,\ldots,t_n)\text{:-}\ B.) & = \{\neg p(t_1,\ldots,t_n)\text{:-}\ \tau_2(B).\,,p(t_1,\ldots,t_n)\text{:-}\ \tau_2(B).\}\ , \\
\tau_1(\underline{\neg p}(t_1,\ldots,t_n)\text{:-}\ B.) & = \tau_1(\underline{p}(t_1,\ldots,t_n)\text{:-}\ B.)\ .
\end{array}
$$

A rough program $\mathcal{P}$ will be transformed into an extended logic program $\tau_1(\mathcal{P})$ by compiling each rough clause. Moreover, if $\mathcal{M}_{\mathcal{P}'}$ is a paraconsistent stable model of $\mathcal{P}' = \tau_1(\mathcal{P})$ then each predicate symbol q with arity n, occurring in $\mathcal{P}$, denotes the rough relation

$$Q_{\mathcal{M}_{\mathcal{P}'}} = (\{(c_1,\ldots,c_n) \mid q(c_1,\ldots,c_n) \in \mathcal{M}_{\mathcal{P}'}\}, \{(c_1,\ldots,c_n) \mid \neg q(c_1,\ldots,c_n) \in \mathcal{M}_{\mathcal{P}'}\})\ ,$$

in the model $\mathcal{M}_{\mathcal{P}'}$. Recall that $\tau_1(\mathcal{P})$ is an extended logic program and, therefore, may have several paraconsistent stable models (or none). In each model, the predicate q may denote a different rough relation. Consequently, the denotation of a predicate is always with respect to a model.

3.3 Queries

This section proposes a language to query rough programs. This can be achieved by adapting existing systems based on the stable model semantics [2, 3], which is a topic of future work. Here, we only present queries and their expected answers. Since there might exist more than one model, answers are computed w.r.t. one paraconsistent stable model of the program. If a program has a unique paraconsistent stable model, which may often be the case[5], the answers will refer to this model.

[5] For instance, any definite extended logic program or rough program whose clauses do not contain lower approximations in their bodies, has a unique model.

Definition 2. *A* rough query *is a pair* $(Q, \mathcal{P})$*, where* $\mathcal{P}$ *is a rough program and* Q *is defined by the following abstract syntax rule*

$$\begin{aligned} Q \longrightarrow\ & \overline{L} \mid \underline{L} \mid \underline{\overline{L}} \mid \\ & \underline{L_1} \subseteq \underline{L_2} \mid \overline{L_1} \subseteq \overline{L_2} \mid L_1 \underline{\sqsubseteq} L_2 \mid \\ & \underline{L_1} = \underline{L_2} \mid \overline{L_1} = \overline{L_2} \mid L_1 \approx L_2 \ . \end{aligned}$$

where L*,* L_1*, and* L_2 *are objective literals.*

The first three cases for a rough query ($\overline{L}$, $\underline{L}$, and $\underline{\overline{L}}$) have already been introduced in our previous work [7]. For instance, with the query $(\underline{\overline{q}}(c_1, c_2), \mathcal{P})$ we want to know whether the tuple $<c_1, c_2>$ belongs to the boundary region of the rough relation denoted by q in some model of $\mathcal{P}$. Due to lack of space, we omit here any further details about translation of this kind of queries.

In some applications (see e.g. [8]) it is necessary to check rough inclusion or rough equality of given rough relations. Our query language has the respective queries and we now discuss how they could be answered. The idea is to translate them to a set of integrity constraints that are added to the compiled program ($\tau_1(\mathcal{P})$). Hence, a new extended logic program $\mathcal{P}'$ is obtained in this way. Then, the query is answered positively (i.e. the test succeeds) if $\mathcal{P}'$ has at least one paraconsistent stable model. Otherwise, the query is answered negatively (i.e. the test fails). Thus, we reduce the answering problem for this kind of queries to the problem of checking the existence of paraconsistent stable models of an *ELP* where certain properties, expressed by the integrity constraints, hold.

Given an objective literal L, we consider that $\neg\neg L$ and L have the same meaning. Moreover, assume that the objective literals L_1 and L_2 denote rough relations Q_1 and Q_2, respectively. We start by considering the queries $(\overline{L_1} \subseteq \overline{L_2}, \mathcal{P})$ and $(\underline{L_1} \subseteq \underline{L_2}, \mathcal{P})$ and define a function τ_3 that transforms these queries into a set of integrity constraints. We remind the reader that candidate models that make each literal in the body of the constraints *true* are rejected.

$$\begin{aligned} & \tau_3(\overline{L_1} \subseteq \overline{L_2}) = \{\texttt{:-}\ L_1, \mathit{not}\ L_2.\}\ , \\ & \tau_3(\underline{L_1} \subseteq \underline{L_2}) = \{\texttt{:-}\ L_1, \mathit{not}\ \neg L_1, \mathit{not}\ L_2.\ , \texttt{:-}\ L_1, \mathit{not}\ \neg L_1, \neg L_2.\}\ . \end{aligned}$$

We recall the notions of rough inclusion and rough equality [4]. Rough relation Q_1 is *roughly included* in rough relation Q_2, denoted as $Q_1 \underline{\sqsubseteq} Q_2$, if and only if $\underline{Q_1} \subseteq \underline{Q_2}$ and $\overline{Q_1} \subseteq \overline{Q_2}$. The rough sets Q_1 and Q_2 are *roughly equal*, denoted as $Q_1 \approx Q_2$, if and only if $\overline{Q_1} = \overline{Q_2}$ and $\underline{Q_1} = \underline{Q_2}$.

Given a rough program $\mathcal{P}$, we have that the answer to the query

(i) $(\overline{L_1} \subseteq \overline{L_2}, \mathcal{P})$ is `yes` *iff* the *ELP* $\mathcal{P}'_1 = \tau_1(\mathcal{P}) \cup \tau_3(\overline{L_1} \subseteq \overline{L_2})$ has a model.

(ii) $(\underline{L_1} \subseteq \underline{L_2}, \mathcal{P})$ is `yes` *iff* the *ELP* $\mathcal{P}'_2 = \tau_1(\mathcal{P}) \cup \tau_3(\underline{L_1} \subseteq \underline{L_2})$ has a model.

(iii) $(L_1 \underline{\sqsubseteq} L_2, \mathcal{P})$ is `yes` *iff* the *ELP* $\mathcal{P}' = \mathcal{P}'_1 \cup \mathcal{P}'_2$ has a model.

(iv) $(\overline{L_1} = \overline{L_2}, \mathcal{P})$ is `yes` *iff* the *ELP* $\mathcal{P}'_3 = \tau_1(\mathcal{P}) \cup \tau_3(\overline{L_1} \subseteq \overline{L_2}) \cup \tau_3(\overline{L_2} \subseteq \overline{L_1})$ has a model.

(v) $(\underline{L_1} = \underline{L_2}, \mathcal{P})$ is `yes` *iff* the *ELP* $\mathcal{P}'_4 = \tau_1(\mathcal{P}) \cup \tau_3(\underline{L_1} \subseteq \underline{L_2}) \cup \tau_3(\underline{L_2} \subseteq \underline{L_1})$ has a model.

(vi) $(L_1 \approx L_2, \mathcal{P})$ is `yes` *iff* the *ELP* $\mathcal{P}' = \mathcal{P}'_3 \cup \mathcal{P}'_4$ has a model.

4 Conclusions and Future Work

We introduced a language for representing vague knowledge in the framework of the rough set theory and a query language. We defined a natural translation of our language to extended logic programming under the paraconsistent stable model semantics. This opens for re-use of the existing logic programming systems based on the stable model semantics for answering rough set queries.

Our language uses common notions of rough set theory: lower and upper approximations, and boundaries for implicit definitions of rough sets. The usual technique of defining rough sets by decision tables is embedded as a special case. The language is flexible enough to allow separate definitions of each region of a rough set and substantially extends our previous proposal in [7]. Since a program $\mathcal{P}$ in our language may have different models, the rough sets defined by $\mathcal{P}$ may not be unique. This fact can be useful in some applications.

The proposed query language is very expressive. It makes it possible not only to search for elements in particular regions of rough relations but also to test for rough inclusion and rough equality, which is essential in some applications.

Continuation of this work will include development of a system based on the presented ideas, integrating them with techniques commonly used in rough sets such as reducts and quantitative techniques , and testing it on real life examples.

References

1. K. Apt and R. Bol. Logic programming and negation: A survey. In *Journal of Logic Programming*, volume 19/20, pages 9–72. Elsevier, May/July 1994.
2. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system `dlv`: Progress report, comparisons and benchmarks. In A. G. Cohn, L. Schubert, and S. C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 406–417, San Francisco, California, 1998. Morgan Kaufmann.
3. I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In M. Maher, editor, *Proc. of the Joint International Conference and Symposium on Logic Programming*, pages 289–303, Bonn, Germany, 1996. MIT Press.
4. Z. Pawlak. *Rough sets. Theoretical Aspects of Reasoning about Data*. Kluwer Academic Publishers, Dordrecht, 1991.
5. D. Pearce. Answer sets and constructive logic, II: Extended logic programs and related non-monotonic formalisms. In L. Pereira and A. Nerode, editors, *Logic Programming and Nonmonotonic Reasoning - proceedings of the second international workshop*, pages 457–475. MIT Press, 1993.
6. C. Sakama and K. Inoue. Paraconsistent Stable Semantics for Extended Disjunctive Programs. *Journal of Logic and Computation*, 5(3):265–285, 1995.
7. A. Vitória and J. Małuszyński. A logic programming farmework for rough sets. In J. Alpigini, J. Peters, A. Skowron, and N. Zhong, editors, *Proc. of the 3rd International Conference on Rough Sets and Current Trends in Computing, RSCTC'02*, number 2475 in LNCS/LNAI, pages 205–212. Springer-Verlag, 2002.
8. W. Ziarko and X. Fei. VPRSM approach to WEB searching. In J. Alpigini, J. Peters, A. Skowron, and N. Zhong, editors, *Proc. of the 3rd International Conference on Rough Sets and Current Trends in Computing, RSCTC'02*, number 2475 in LNCS/LNAI, pages 514–521. Springer-Verlag, 2002.