An Experimental Platform for Approximate Databases

Martin Magnusson, Patrick Doherty, Andrzej Szałas
Department of Computer and Information Science
SE-581 83 Linköping, Sweden
email: {marma,patdo,andsz}@ida.liu.se

March 31, 2005

Abstract

We have implemented an experimental platform for approximate knowledge databases based on a semantics inspired by rough sets called the Rough Knowledge Database (RKDB). The implementation is based upon the use of a standard SQL database to store logical facts, augmented with several query interface layers implemented in JAVA through which extensional and intensional approximate logical formula queries, approximate fixpoint formula queries, and local closed world queries can be evaluated. Designing the database, entering data, and querying can all be accomplished in an accessible way through the Graphical Database Design user interface.

1 Introduction

A standard deductive database (see, e.g., [1]) can store ground atomic formulas to represent factual knowledge and use, for example, Horn-clause logic formulas as deductive rules to infer additional facts. Such databases often make assumptions, such as the assumption that the stored knowledge is precise or complete, that render them less suitable for a context in which these assumptions can not be made. The concept of *approximate databases*, discussed in a number of publications referenced throughout this paper and summarized in book form in [3], relaxes some of these assumptions and might be applicable where standard techniques are not. In order to be able to investigate whether this is the case, we have implemented an approximate database, called the Rough Knowledge Database, that serves as an experimental platform.

This presentation is structured as follows. Section 2 introduces and motivates the concepts that are used in Section 3, which describes the RKDB system architecture and its different parts together with a detailed example. We continue with a fancy screen shot of the user interface in Section 4 before finally concluding in Section 5.

2 Preliminaries

Assume we want to reason about the weather and we want to decide what to wear on our way to work. We might want to express knowledge such as "if it's cold and windy wear the winter coat" and "if it's raining bring the umbrella, and whenever you choose to bring the umbrella the sky should at least be overcast". It is clear that relations such as "windy" are better described as approximate than exact, and that relations such as "cold" don't really have a precise definition at all. Furthermore, the knowledge we wrote down about umbrellas doesn't completely specify when to bring the umbrella and when to leave it at home. Instead it provides sufficient conditions, raining, and necessary conditions, overcast, that only partially define the concept.

2.1 Approximate Relations

To naturally express knowledge such as that described above, we introduce the *approximate relation* as a basic element of our database, contrasting it with the *crisp* relations and formulas used in a regular deductive database. The idea of approximate relations started out from the idea of rough sets, where instead of partitioning objects into those that are in the set and those that are outside the set, one has a third group of objects that *might* be in the set, called the boundary part. This semantics was then generalized to create a semantics of approximate databases. We also extend the logical language used for writing formulas with the following expressions:

Any approximate relation R has

- a *positive part*, denoted by R^+, containing objects known to satisfy the relation,
- a *negative part*, denoted by R^-, containing objects known not to satisfy the relation,
- a *boundary part*, denoted by $R^\pm$, containing objects that are neither known to satisfy the relation nor known not to,
- a *positive-boundary part*, denoted by $R^\oplus$, containing objects in the positive or boundary part,
- a *negative-boundary part*, denoted by $R^\ominus$, containing objects in the negative or boundary part.

Queries to an approximate database are approximate formulas, i.e. logical formulas in which all references to relations are approximate. A query formula will usually contain free variables, in which case the result of the query is a list of substitutions for the free variables that satisfies the formula. When a query formula does not contain free variables, the query result is true, false or unknown.

2.2 Open World Assumption

Suppose we decided to bring the umbrella on our way to work, if we could just remember where we put it. We are wondering if it might not be in the hallway

closet, but we don't know. If we were trying to represent the situation using a regular deductive database we would be forced to decide whether to have a fact $In(Umbrella, Closet)$ in the database or not. Considering that we don't know if it's in there, adding the fact that it *is* seems erroneous. Choosing the latter alternative, we query the database asking if the umbrella is in fact in the closet. Most databases would now use the well-known *closed world assumption* to conclude that since there is no evidence of the umbrella being in the closet, it must not be, which is clearly not a well-founded conclusion in this scenario.

Using the approximate database we choose not to add any facts regarding the whereabouts of the umbrella, but this time when querying about its location we will find that it is unknown. This is the *open world assumption* (which makes no assumptions at all really). Since it's unknown if the umbrella is in the closet, we decide to look, but there is no sign of the umbrella. We can't be entirely sure it's not in there since it might be hidden inside something else, or lie in a dark corner where it's impossible to see, but we would still like to conclude, *assuming we looked thoroughly*, that it's not there so that we can get on with the searching elsewhere. If the umbrella is nowhere else to be found, we might want to go back and remove our previous assumption that we searched the closet thoroughly enough, and find that without the assumption it is still unknown if the umbrella is in the closet. This method of reasoning is a form of *local closed world* reasoning supported by the approximate database as will be detailed in Section 3.4.

3 The Rough Knowledge Database

It is important to facilitate experimentation to provide a better feel for the utility and applicability of the ideas and techniques behind approximate databases, and such experience will likely only come from a real system. We have consequently spent some considerable effort on an actual implementation called the Rough Knowledge Database (RKDB). Grounded in a standard SQL database, several extension layers each provide an extended query language building upon the layer below. Figure 1 displays an architectural overview of the system that is described in more detail below, starting from the bottom abstraction layer, moving upwards.

3.1 SQL Database

Even though the database can be said to form the basis of the system it can easily be replaced, choosing from a long list of SQL databases such as PostgreSQL or Microsoft Access. Whatever database chosen will use regular, crisp, database relations to store approximate relations, but need not know the details of the representation. Deciding the exact format is the concern of the next abstraction layer, which still leaves the SQL database the task of optimizing and executing SQL queries passed down from layers above. Note that for some applications a standard SQL database might not be the most efficient means of storing and retrieving data. In a robotic system secondary memory footprint may need to be minimized or a high frequency of low complexity queries might call for an implementation where data is stored in primary memory, in which case such an implementation can be plugged into the system, bypassing the SQL interface.

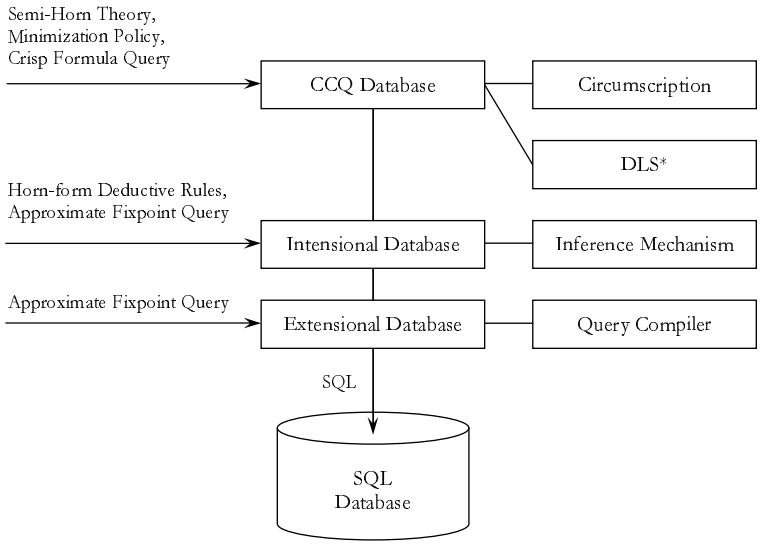

Figure 1: An overview of the RKDB architecture.

3.2 Extensional Database

The Extensional Database layer (EDB) provides a mapping from approximate relations to a specific representation scheme based on regular database relations. In particular, the positive (R^+) and negative (R^-) parts of relations are stored explicitly in tables while the boundary, positive boundary, and negative boundary are only stored implicitly but can still be generated through more complex queries to the database. When all relation arguments are assumed to have finite domains, all parts of relations consist of a finite number of tuples, and the division between explicit and implicit storage becomes a potential implementation choice point to which the best answer will depend on the final application. We believe it wise, in the general case, to save space by storing known information explicitly and unknown information implicitly since we are likely to know just a little and be ignorant of a great deal more.

As mentioned previously, logic is consistently used as the query language in all parts of the RKDB. Both the fact that the logical query language may refer explicitly to the boundary parts that are not explicitly stored and the fact that any, arbitrarily complex, logical formula may be used as a query contribute to the necessity of some kind of evaluation mechanism. To meet this need we have developed a query compilation mechanism that will recursively transform any logical formula query into a, sometimes very elaborate, SQL query, the only exception being fixpoint formulas, which need to be iteratively evaluated until a fixpoint is reached before the result can be returned. By delaying the actual evaluation of any part of the query until it reaches the SQL database we can benefit from the potential performance increase resulting from SQL query optimization techniques employed by our particular database of choice.

3.3 Intensional Database

The Intensional Database layer (IDB) uses stored rules to infer additional information from the facts in the EDB. The intensional rules are approximate implications with a single literal head, but differ from Horn-clause type rules in that the head literals can contain negations. To deduce new facts the rules are translated into approximate formula fixpoint queries that must then be checked for consistency since both new positive and new negative information might be produced.

Performing the necessary inferences for a specific query is the task of the inference mechanism, which currently uses the naive method of repeatedly applying all IDB-rules until no new facts can be inferred, evaluating the query in this new context, and finally withdrawing the generated facts to restore the original database state. This method can clearly be improved upon using the rich set of techniques developed for efficiently evaluating intensional database queries (see, e.g., Chapter 13 of [1]).

3.4 Contextually Closed Query Database

The Contextually Closed Query database layer (CCQ) provides the functionality that renders locally closed world reasoning feasible. This is accomplished through the use of *circumscription* in the context of a *closure policy* provided by the user, exemplified below.

3.4.1 Circumscription

A contextually closed query consists of a logical formula query together with a crisp logical theory, describing the relations in the query, and a closure policy defining what relations are affected by the closure in one of the following ways:

- A relation may be *fixed*, in which case it will not be modified by the closure.
- A relation may be *minimized*, in which case tuples may be moved from the boundary part into the negative part of the relation in order to minimize its extension.
- A relation may be *varied*, in which case tuples may be moved from the boundary part into either the positive or negative part as an effect of the minimization of other relations.

3.4.2 A Surveillance Mission

Consider a scenario involving an unmanned autonomous helicopter that makes use of contextually closed queries during a surveillance mission. A black car has been reported stolen and the task of the robotic helicopter is to locate the car by investigating areas in which the car is suspected to be located. To represent this scenario we make use of the relations $In(x, y)$, $Color(x, z)$, $SuspectIn(y)$, and $Investigate(x, y)$ to express that car x is in region y, the color of car x is z, the stolen car is suspected to be in region y, and the helicopter should search for car x in region y respectively. Using these relations we construct a crisp logical theory (1) expressing the behaviour we wish the helicopter to exhibit.

All black cars that are in a suspect region should be investigated. Although, if the car is known to have some color other than black it is not necessary to look for it in any region. Finally, when we know a car is not in a region, there is no point going there looking for it.

$$\begin{array}{l}\forall x, y.[In(x, y) \wedge SuspectIn(y) \wedge Color(x, Black) \rightarrow Investigate(x, y)] \wedge \\ \forall x, y, z.[Color(x, z) \wedge z \neq Black \rightarrow \neg Investigate(x, y)] \wedge \\ \forall x, y.[\neg In(x, y) \rightarrow \neg Investigate(x, y)]\end{array} \quad (1)$$

Additionally an approximate intensional rule is added to the IDB expressing the fact that if we know the region a car is in, it can not simultaneously be in some other region.

$$\forall x, y_1, y_2.[In^+(x, y_1) \wedge y_1 \neq y_2 \rightarrow In^-(x, y_2)] \quad (2)$$

Continuing the example, we construct a specific scenario by adding facts to the approximate knowledge base. Given three cars, C_1, C_2 and C_3, three regions, R_1, R_2 and R_3, and two colors, $Black$ and Red, we add the facts expressed in (3). A black car C_1 is known to be in region R_1, the car C_2 is red but we do not know in which region it is, and nothing is known about the third car C_3. Furthermore, the stolen car is believed to be located somewhere in region R_1 or R_2.

$$\begin{array}{l}In^+(C_1, R_1) \wedge \\ Color^+(C_1, Black) \wedge Color^+(C_2, Red) \wedge \\ SuspectIn^+(R_1) \wedge SuspectIn^+(R_2)\end{array} \quad (3)$$

The knowledge base does not contain any information about which cars and what regions are interesting for the autonomous helicopter given its mission, but it is now possible to locally close the world model during a database query by specifying a closure policy of minimization and variation of relations in order to obtain new information about these relations. Determining which relations to vary, which relations to minimize and which to leave unchanged in the general case is a very interesting research topic, but for our example scenario we choose to minimize the number of suspected regions, in order to avoid searching regions that we have no specific reason to believe the stolen car to be in, while varying what cars and regions the helicopter should investigate to obtain information about possible actions to take. Consequently we construct the policy of minimizing $SuspectIn$ while varying $Investigate$ and fixing the remaining relations In and $Color$.

Using predicate circumscription to apply the closure policy to the logical theory T, as in [2], we will generate a second-order circumscription axiom and try to reduce it to an equivalent first-order formula T' using the DLS* algorithm described in Section 3.4.3. If the reduction is successful it is possible to extract syntactic definitions of the new minimized and varied relations by constructing a second-order formula for each, existentially quantifying the relation R in front of the circumscribed theory T' containing it.

$$\exists R.[T'(R)] \quad (4)$$

By creating a formula such as (4) for each relation we can constructively find the syntactic characterizations through application of DLS* and use them to query the knowledge base. Returning to the helicopter scenario we obtain syntactic definitions for $SuspectIn$ and $Investigate$ as shown in (5).

$$\begin{array}{rl} SuspectIn: & SuspectIn^{+}(y) \\ \neg SuspectIn: & SuspectIn^{\ominus}(y) \\ Investigate: & Investigate^{+}(x,y) \vee \\ & In^{+}(x,y) \wedge SuspectIn^{+}(y) \wedge Color^{+}(x, Black) \\ \neg Investigate: & Investigate^{-}(x,y) \vee In^{-}(x,y) \vee \\ & \exists z.[Color^{+}(x,z) \wedge z \neq Black] \end{array} \tag{5}$$

Observe that the definitions contain unbound variables, and presenting them to the knowledge base as queries will produce exactly the tuples satisfying the new relation definitions. To evaluate a complex query containing the minimized or varied relations it suffices to replace those occurrences with their syntactic definitions and passing the query to the intensional database layer. Evaluating the definitions in our examples produces the tuples seen in (6), including new tuples produced by the IDB rule.

$$\begin{array}{rl} In(x,y): & \langle C_1, R_1 \rangle \\ \neg In(x,y): & \langle C_1, R_2 \rangle, \langle C_1, R_3 \rangle \\ SuspectIn(y): & \langle R_1 \rangle, \langle R_2 \rangle \\ \neg SuspectIn(y): & \langle R_3 \rangle \\ Investigate(x,y): & \langle C_1, R_1 \rangle \\ \neg Investigate(x,y): & \langle C_1, R_2 \rangle, \langle C_1, R_3 \rangle, \langle C_2, R_1 \rangle, \\ & \langle C_2, R_2 \rangle, \langle C_2, R_3 \rangle \end{array} \tag{6}$$

Although the IDB rule excluded the possibility of C_1 being anywhere else than in R_1, it remains unknown which regions the other cars are in. Minimizing $SuspectIn$ writes R_3 off the list of suspected regions since there is no reason to believe otherwise, while varying $Investigate$ prompts the helicopter to search for C_1 in region R_1 since we know it is a black car located in a region which we suspect the stolen car to be in. In addition, the helicopter robot concludes that it is not necessary to look for C_1 anywhere else, using the IDB rule and the part of the theory stating that it should not investigate a region, looking for a car it knows is not there. Car C_2 can be in any of the regions but there is no point looking for it as it has the color Red, different from $Black$. Finally, it remains unknown, even after applying the closure policy, if searching for the car C_3 in any of the regions is necessary.

Now, assume the robotic helicopter takes action, flying over region R_1 looking for C_1, and that it finds the car but it is not the stolen car we are looking for. It updates the knowledge base by removing R_1 from the list of suspected regions and adding the fact that it did not find C_3, expressed by $In^{-}(C_3,R_1)$. Using the same syntactic definitions of relations, we reevaluate the queries in light of these new facts.

$$
\begin{array}{rl}
In(x,y): & \langle C_1, R_1\rangle \\
\neg In(x,y): & \langle C_1, R_2\rangle, \langle C_1, R_3\rangle, \langle C_3, R_1\rangle \\
SuspectIn(y): & \langle R_2\rangle \\
\neg SuspectIn(y): & \langle R_1\rangle, \langle R_3\rangle \\
Investigate(x,y): & \\
\neg Investigate(x,y): & \langle C_1, R_2\rangle, \langle C_1, R_3\rangle, \langle C_2, R_1\rangle, \\
 & \langle C_2, R_2\rangle, \langle C_2, R_3\rangle, \langle C_3, R_1\rangle
\end{array} \tag{7}
$$

The In tuples in (7) changed to incorporate the fact that C_3 has not yet been found, and the R_1 tuple in the $SuspectIn$ relation has moved to reflect the fact that no stolen car was found there, but the varied $Investigate$ relation has changed too. The helicopter has already searched region R_1 for C_1, and it concludes that it is no longer necessary to investigate whether C_3 is in R_1, but it is still unknown if the helicopter should look for C_3 in one of the other regions.

Notice that without changing the definitions, the query results have changed to reflect the new knowledge situation. This will stay true until we modify the closure policy or the logical theory describing the mission, in which case the definitions must be recalculated. As long as the policy and theory stay the same, we can cache the calculated definitions, improving efficiency.

In its current state of uncertainty, the autonomous helicopter might either explain the two remaining possibilities to a mission operator, asking for new information or advice on which action to take, or continue on itself, e.g. by systematically searching for C_3, first in region R_2 and then in R_3. Assuming the latter alternative, and that the stolen car is in fact located in one of the regions, the helicopter will find it and successfully complete the mission.

3.4.3 DLS*

Utilizing a fixpoint generalization of an equivalence between second-order logic formulas of a certain form and first-order formula counterparts, generated through a specific syntactic transformation, the DLS* algorithm can reduce any second-order formula from the class of semi-Horn formulas into an equivalent first-order formula in polynomial time [4]. Other formulas might also be reduced and although there is no guarantee in the general case (indeed there could not be since the problem is not computable), DLS* can provide a completeness proof for a well defined subclass of first-order logic.

An earlier implementation in a constraint logic programming language is available online at [6], but we chose to re-implement the algorithm in JAVA to add some extensions and make the algorithm an integrated part of the RKDB. This new version will also be made available online after a testing phase.

3.4.4 Simplification

Forcing a logical theory through the DLS* algorithm after first circumscribing it can have undesirable effects on its complexity. Although encouraging results in [5] show that, at least in the semi-Horn case, the size of the syntactic characterizations of varied and minimized relations are linear in relation to the size of the CCQ query, in practice there is often both a need and opportunity for simplifications. A number of equivalence-preserving simplifications are applied

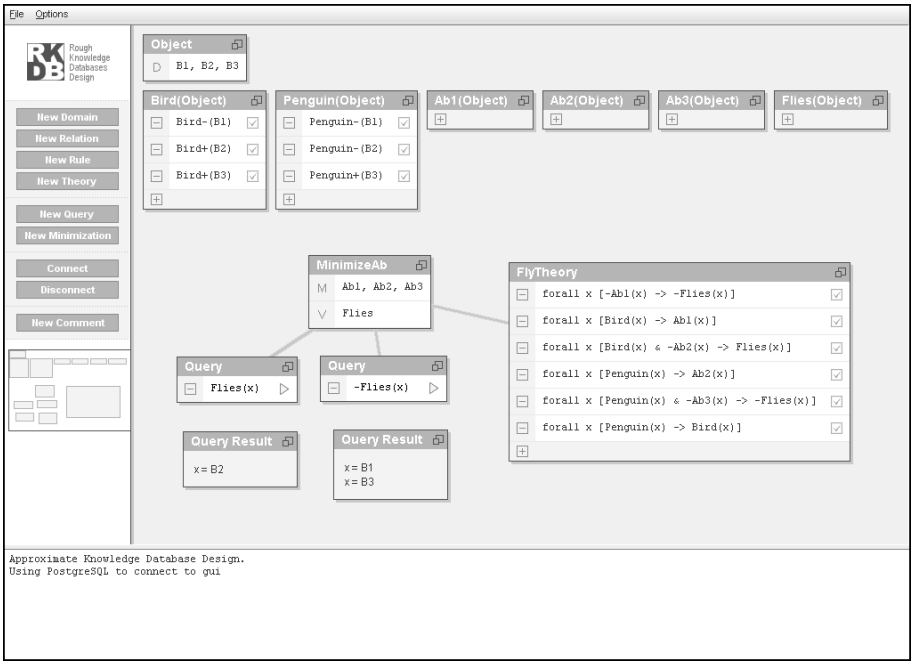

Figure 2: A screen shot of the Graphical Database Design user interface.

to each contextually closed query, e.g. taking advantage of the unique names assumption in the RKDB.

3.5 Logic Parser

A parser for database input was generated using the JavaCC parser generator. Our syntax supports the representation of approximate formulas in addition to regular first-order and fixpoint formulas, but also knowledge base-specific constructs such as relation definitions, extensional facts, intensional rules, and logical theories together with closure policies, making it possible to specify entire use scenarios in a single text file. This latter functionality is complementary to the use of the RKDB as a service in a larger system, in which case one would make direct use of the RKDB interface to gain access to methods for, among other things, adding and retracting facts or rules.

4 Graphical Database Design

Even if not a necessary functional part of a system, a graphical user interface can often encourage and simplify experimentation. Considering that we are building an experimental platform for a collection of new deductive database techniques not yet extensively explored, such qualities seem beneficial. This is the idea behind the Graphical Database Design tool, shown in Figure 2, for the RKDB that provides an environment where knowledge bases, complete with relation definitions, facts, rules, and theories, can be created, changed, or destroyed.

The interface builds upon a window system, where each relation, theory, policy, or query, has its own window. The windows can then be connected to link a query with a policy and a theory.

5 Conclusions

We have created the foundation of an experimental environment in which the ideas and techniques in [3] can be investigated and explored. The system called the Rough Knowledge Database is implemented in JAVA and consists of a layered architecture based on a plug-in SQL database. The RKDB may be used either as a service through an interface, or stand-alone through file input or a graphical user interface, the Graphical Database Design tool. It is our hope that this implementation will be of considerable support to anyone wishing to explore the use of approximate databases for knowledge representation.

Acknowledgements

This work is partially funded by the Wallenberg Foundation under the WITAS UAV Project and the NFFP03-539 COMPAS Project.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Patrick Doherty, Jarosław Kachniarz, and Andrzej Szałas. Using contextually closed queries for local closed-world reasoning in rough knowledge databases. In Sankar K. Pal, Lech Polkowski, and Anrzej Skowron, editors, *Rough-Neuro Computing: Techniques for Computing with Words*, Cognitive Technologies, pages 219–250. Springer–Verlag, 2003.

[3] Patrick Doherty, Witold Łukaszewicz, Andrzej Skowron, and Andrzej Szałas. *Knowledge Engineering: A Rough Set Approach*. Studies in Fuziness and Soft Computing. Springer Physica Verlag, 2005. To appear.

[4] Patrick Doherty, Witold Łukaszewicz, and Andrzej Szałas. General domain circumscription and its effective reductions. *Fundamenta Informaticae*, 36(1):23–55, 1998.

[5] Patrick Doherty, Witold Łukaszewicz, and Andrzej Szałas. Declarative PTIME queries for relational databases using quantifier elimination. *Journal of Logic and Computation*, 9(5):739–761, 1999.

[6] Joakim Gustafsson. The DLS algorithm, 1996. `http://www.ida.liu.se/labs/kplab/projects/dls/`.