Towards Large Scale Reasoning on the Semantic Web

Balázs Kádár, Gergely Lukácsy and Péter Szeredi

Budapest University of Technology and Economics
Department of Computer Science and Information Theory
1117 Budapest, Magyar tudósok körútja 2., Hungary
balazs@kadar.biz,{lukacsy,szeredi}@cs.bme.hu

Abstract. Traditional algorithms for description logic (DL) instance retrieval are inefficient for large amounts of underlying data. As description logic is becoming popular in areas such as the Semantic Web, it is very important to have systems that can reason efficiently over large data sets. In this paper we present the DLog description logic reasoner specifically designed for such scenarios.
The DLog approach transforms description logic axioms using the $\mathcal{SHIQ}$ DL language into a Prolog program. This transformation is done without any knowledge of the particular individuals: they are accessed dynamically during the normal Prolog execution of the generated program. This allows us to store the individuals in a database instead of memory, which results in better scalability and helps using description logic ontologies directly on top of existing information sources.
In this paper we focus on the description of the DLog application itself. We present the architecture of DLog and describe its interfaces. These make it possible to use ABoxes stored in databases and to communicate with the Protégé ontology editor, as a server application. We also evaluate the performance of the DLog database extension.
Keywords: large data sets, description logic, reasoning, logic programming, databases

1 Introduction

Description Logics (DLs) allow us to represent *knowledge bases* consisting of terminological axioms (the *TBox*) and assertional knowledge (the *ABox*).

Description Logics are becoming widespread as more and more systems start using semantics for various reasons. As an example, in the Semantic Web idea, DLs are intended to provide the mathematical background needed for more intelligent query answering. Here the knowledge is captured in the form of expressive ontologies, described in the Web Ontology Language (OWL) [1]. This language is mostly based on the $\mathcal{SHIQ}$ description logic, and it is intended to be the standard knowledge representation format of the Web.

However, we have tremendous amounts of information on the Web which calls for reasoners that are able to *efficiently* handle such abundance of data.

Moreover, as these data cannot be stored directly in memory, we need solutions for querying description logic concepts in an environment where the ABox is stored in a *database*.

We found that most existing description logic reasoners are not suitable for this task, as these are not capable of handling ABoxes stored externally. This is not a simple technical problem: most existing algorithms for querying DL concepts need to examine the whole ABox to answer a query. This results in scalability problems and undermines the point of using databases. Because of this, we started to investigate techniques which allow the separation of the inference algorithm from the data storage.

We have developed a solution, where the inference algorithm is divided into two phases. First we create a *query-plan* in Prolog from the actual DL knowledge base, without accessing the underlying data set. Subsequently, this query-plan can be run on real data, to obtain the required results. The implementation of these ideas is incorporated in the DLog reasoning system, available at `http://dlog-reasoner.sourceforge.net`.

In this paper we focus on the architecture of the DLog system, as well as on its external interfaces. We discuss the interface used for accessing databases, which allows description logic reasoning on top of existing information sources. We also describe the Protégé [2] interface that makes it possible to use DLog as the back-end reasoner of this popular ontology editor. Details on the theoretical side of DLog can be found in [3] and in [4].

This paper is structured as follows. Section 2 summarises related work. In Section 3 we give a general introduction to the DLog approach and present the architecture and implementation details of the system. The database and Protégé interfaces are described in Sections 4 and 5, respectively. Section 6 evaluates the performance of the database extension of DLog w.r.t. the version which stores the ABox as Prolog facts. Finally, in Section 7, we conclude with the future work and the summary of our results.

2 Related work

Several techniques have emerged for dealing with ABox-reasoning. Traditional ABox-reasoning is based on the *tableau inference* algorithm, which tries to build a model showing that a given concept is satisfiable. To infer that an individual i is an instance of a concept C, an indirect assumption $\neg C(i)$ is added to the ABox, and the tableau-algorithm is applied. If this reports inconsistency, i is proved to be an instance of C. The main drawback of this approach is that it cannot be directly used for high volume instance retrieval, because it would require checking all instances in the ABox, one by one.

To make tableau-based reasoning more efficient on large data sets, several techniques have been developed in recent years [5]. These are used by the state-of-the-art DL reasoners, such as RacerPro [6] or Pellet [7].

Extreme cases involve serious restrictions on the knowledge base to ensure efficient execution with large amounts of instances. For example, [8] suggests a

solution called the *instance store*, where the ABox is stored externally, and is accessed in a very efficient way. The drawback is that the ABox may contain only axioms of form $C(a)$, i.e. we cannot make role assertions.

Paper [9] discusses how a first order theorem prover such as Vampire can be modified and optimised for reasoning over description logic knowledge bases. This work, however, mostly focuses on TBox reasoning.

In [10], a resolution-based inference algorithm is described, which is not as sensitive to the increase of the ABox size as the tableau-based methods. However, this approach still requires the input of the *whole content* of the ABox before attempting to answer any queries. The KAON2 system [11] implements this method and provides reasoning services over the description logic language $\mathcal{SHIQ}$ by transforming the knowledge base into a disjunctive datalog program.

Although the motivation and goals of KAON2 are similar to ours, unlike KAON2 (1) we use a pure two-phase reasoning approach (i.e. the ABox is accessed only during query answering) and (2) we translate into Prolog which has well-established, efficient and robust implementations.

Article [12] introduces the term Description Logic Programming. This idea uses a direct transformation of $\mathcal{ALC}$ description logic concepts into definite Horn-clauses, and poses some restrictions on the form of the knowledge base, which disallow axioms requiring disjunctive reasoning. As an extension, [13] introduces a fragment of the $\mathcal{SHIQ}$ language that can be transformed into Horn-clauses. This work, however, still poses restrictions on the use of disjunctions.

3 The DLog system

The main idea of the DLog approach is that we transform a $\mathcal{SHIQ}$ knowledge base KB into first-order clauses $\Omega(KB)$ and from these we generate Prolog code [3]. In contrast with [11], all clauses containing function symbols are eliminated during the transformation: the resulting clauses can be resolved further only with ABox clauses. This forms the basis of a pure two phase reasoning framework, where every possible ABox-independent reasoning step is performed before accessing the ABox itself, allowing us to store the content of the ABox in an external database.

Actually, in the general transformation, we use only certain properties of $\Omega(KB)$. These properties are satisfied by a subset of first order clauses that is, in fact, larger than the set of clauses that can be generated from a $\mathcal{SHIQ}$ KB. We call these clauses *DL clauses*. As a consequence of this, our results can be used for DL knowledge bases that are more expressive than $\mathcal{SHIQ}$. This includes the use of certain role constructors, such as union. Furthermore, some parts of the knowledge base can be supplied by the user directly in the form of first order clauses. More details can be found in [3].

As the clauses of a $\mathcal{SHIQ}$ knowledge base KB are normal first-order clauses we can apply the Prolog Technology Theorem Proving (PTTP) technology [14] directly on these. In [3] we have simplified the PTTP techniques for the special

case of DL clauses and we have proved that these modifications are sound and complete for DL clauses.

The simplified PTTP techniques used in DLog include deterministic *ancestor resolution* and *loop elimination*. Both are applicable only to unary predicates, i.e. predicates corresponding to DL concepts.

In the design of the DLog system we focus on modularity. This enables us to easily implement new features and new interfaces. The top level architecture of the system is shown in Figure 1. In this figure, as in subsequent figures of the paper, rectangles with rounded corners represent modules of the DLog system, while data are shown as plain rectangles. In Figure 1 the DLog reasoner is shown within a dashed rectangle.

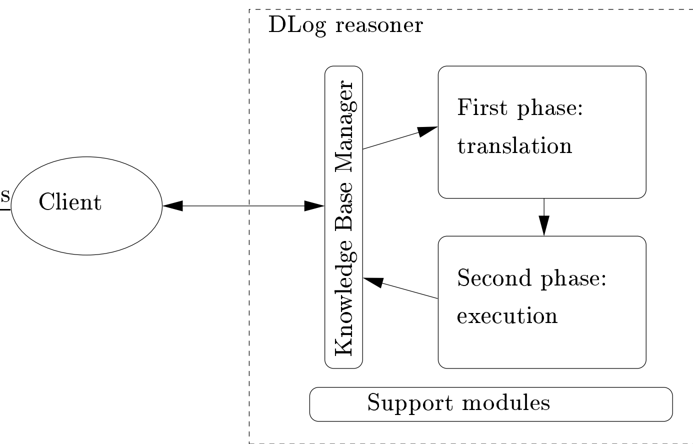

Fig. 1. The top level architecture of the DLog system.

The user (either local or remote) accesses DLog through one of the external interfaces. These interfaces range from a local console to server interfaces like DIG used by the Protégé ontology editor. The knowledge base manager is the central piece of the system. It coordinates the tasks of the other modules, and performs the administration of multiple concurrent knowledge bases. It forwards the request arriving from the interfaces to the reasoner modules.

The *support modules* consist of several tools that are used by most parts of the system. They include a configuration manager module, a logger, an XML reader, a run-time system for the second phase, and several portability tools that allow DLog to run under different Prolog implementations (currently SWI and SICStus).

The first phase, translation, shown in Figure 2, takes a set of description logic axioms as input. These axioms are divided into two parts: the TBox or terminology box stores concept and role inclusion axioms, while the ABox or assertion box contains the factual data. The ABox may be stored (partly or

completely) in external databases. The ABox is processed first, producing the *ABox code* (which is a Prolog module), and the ABox signature, which is required for translating the TBox. The generation of ABox code includes optimisations such as indexing on second argument for roles stored in memory.

Next, the TBox is processed in two steps. First the DL translator module transforms the description logic formulae to a set of DL clauses [15], which are passed on to the TBox translator module that generates the executable *TBox code*. This generated code is equivalent, with respect to instance retrieval, to the input DL knowledge base. The TBox translator module uses various optimisations [3] to obtain more efficient Prolog programs. The ABox and TBox code can be generated directly into memory or may be saved to disk for later (standalone) use.

PSfrag replacements

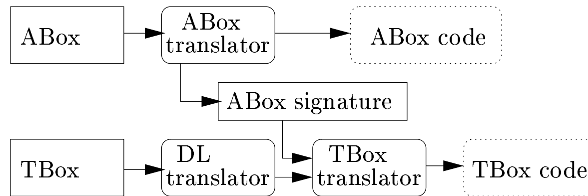

Fig. 2. The first phase: translation.

The second phase, execution, shown in Figure 3, uses the ABox and TBox programs generated in the first phase, to answer queries. There are two ways to execute queries: the generated TBox can be called directly from Prolog as a low-level interface, or the *Query module* provides a high-level interface that provides basic support for composite queries and can aggregate the results. In normal operation the query module is called by the knowledge base manager, which forwards the results to the user interface. As the query module does not depend on the rest of the system, it may be used in standalone operation. The run-time system (shown as RTS in the figure) includes a hash table implemented in C used to speed up the reasoning, and optional collection of statistics.

PSfrag replacements

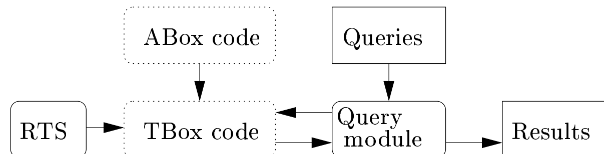

Fig. 3. The second phase: execution.

4 Integrating DLog with databases

As the first phase of reasoning (i.e. the generation of a query plan) only depends on the signature of the data set, and because of the top-down inference of Prolog, DLog can efficiently use databases to store the ABox.

There may be several advantages in using databases to store the ABox. Firstly, this allows reasoning on data sets that cannot fit into memory. Secondly, it makes integrating DLog with existing systems easier, as the reasoner can use the existing databases of other applications. Thirdly, querying some concepts (namely those corresponding to so-called *query predicates*) may be performed using complex database queries, rather than DL reasoning, which is expected to deliver a marked increase in performance.

A predicate is a *query predicate* [3], if it is non-recursive, it does not invoke its negation, and is not invoked from within its negation. Here, a predicate P_0 is said to invoke a predicate P_n, $n \geq 1$, if there are $n-1$ intermediate predicates $P_1 \dots P_{n-1}$, such that P_i is directly invoked by P_{i-1}, i.e. it occurs in a clause body the head of which is P_{i-1}, for $i = 1, \dots, n$.

Query predicates require neither loop elimination, nor ancestor resolution during execution. The name "query predicate" reflects that fact that such predicates can be transformed to complex database queries (provided that all concepts and roles required are stored in a single database). This can increase the performance as the database engine can optimise the query using statistical and structural knowledge of the database in question.

We designed the database interface to be as simple as possible. The databases are accessed via the ODBC driver of SWI-Prolog; as a consequence DLog can interface with most modern database systems. We wanted a way to specify database access using existing tools and interfaces – such as Protégé and the DIG interface it utilises – even if those do not, at the moment, provide a way to specify database usage. To access a database, several pieces of information are needed: the name of the database, a user name, a password, a description of which table to use for given concepts and roles, etc. Because of the aforementioned requirements we decided to use ABox assertions to carry this meta-information. ABox assertions are description logic constructs that are readily available in DL systems and interfaces, such as OWL and DIG.

In order to specify the database access for concepts and roles we introduce new roles (object properties), attributes (datatype properties) and individuals defined in the namespace `http://www.cs.bme.hu/dlogDB`.

The ODBC interface prescribes that database connections are to be identified by a *Data Source Name* (DSN). In DLog we introduce an individual to represent a given database connection. Roles and concepts are also represented by individuals. An arbitrary name can be used for such an individual.

The meta data provided is used to connect to the database, and, for each concept and role, an additional clause is generated, which, by executing an appropriate database query, lists appropriate individuals (or pairs of individuals). This allows concepts and roles to be stored partially in databases and partially in memory. This may be very useful when developing ontologies.

4.1 Specifying the Database Interface

Database connections are represented by individuals that have the string attribute `hasDSN` defined. The value of this attribute is the name of the data source (DSN). As all other names in this section, this name is defined in the namespace `http://www.cs.bme.hu/dlogDB`. Additional string attributes, namely `hasUserName` and `hasPassword`, may be used to specify the user name and the password for the given connection, if required.

The object property `hasConnection` links an individual representing a role or a concept with the database connection to be used for accessing it. This makes it possible to use one data source for one concept, and a different one for another. The instance on the left hand side is the individual representing the role or concept, while the instance on the right hand side is the individual representing the connection.

Two methods are provided to specify how to get the data from the database. One is to specify a query that is to be directly executed on the database. This method, named the *simple interface*, is provided because of its simplicity: it can be applied to databases without any modification. However it has two drawbacks:

- it makes transforming query predicates to database queries very difficult; and
- it performs badly for instance check queries.

The latter is a large setback as most of the queries are instance checks, assuming the the *projection* optimisation of [3] is used.

Therefore the second, preferred, way is to provide the name of a table or of a view and the name of the column(s) of this table. This approach, called the *complex interface* may require the creation of new views in the database, but provides much greater flexibility and better performance.

The SQL query in the simple interface is defined using the string attribute `hasQuery`. The individual represents the role or concept and the attribute value is the query string. For individuals representing roles the query must return two columns, and for those used for concepts it must return one column that contains the individual name.

If the complex interface is used, the name of the table or view to use is specified by the string attribute `hasTable`. The name of the column listing the individuals of a concept is given using the string attribute `hasColumn`. For roles, the attributes `hasLHS` and `hasRHS` are used for the left and the right hand side, respectively.

Because, in Protégé, individuals cannot be specified as instances of a negated concept, we provide some additional attributes: `hasNegQuery`, `hasNegTable` and `hasNegColumn`. These are used to specify the database access of negated concepts, in a way similar to their respective positive pairs. By providing an attribute `hasNegQuery` for a name representing the concept C we specify a query listing the individuals of $\neg C$. Obviously, both `hasQuery` and `hasNegQuery` can appear as attributes of the same individual.

To specify that the individual `concept` represents the concept C, one simply has to make `concept` an instance of C. The DLog system will check each concept occurring in the ABox if it contains an instance which is in the namespace `http://www.cs.bme.hu/dlogDB`. If such an instance is found, it is interpreted as a "handle" to a database which is to produce (additional) instances for the given concept.

Similarly, to specify that an individual `role` represents the role R, we require that the user includes the triple `{role,` R`, indiv}` in the ABox. Here `indiv` is an arbitrary individual. Again DLog will look for an instance in the namespace `http://www.cs.bme.hu/dlogDB` within the domain (i.e. the left hand side) of each role, and use it to construct a database access for the given role.

The database interface is currently in the alpha test phase. We believe that our approach for this task, discussed above, is an intermediate solution. Ultimately the standard interfaces, such as DIG, should be extended to allow storing (parts of) the ABox in databases. However, we hope that our work contributes to implementing this ultimate goal.

4.2 Examples of Using the Database Interface

We now present two examples for interfacing with databases, one for the simple, and one for the complex interface.

The examples contain ABox assertions, which are displayed as RDF triples in `{subject, predicate, object}` format. String values are shown between quotes. The namespace `http://www.cs.bme.hu/dlogDB#` is represented by the `dlog:` prefix.

Figure 4 shows the use of the simplified interface for the ABox of the *Iocaste* example. This classical example involves the concept describing a person having a patricide child, who, in turn, has a non-patricide child. The ABox axioms, which are now to be stored in a database, describe the `hasChild` relation between pairs of individuals (traditionally containing `(Iocaste, Oedipus)`, `(Iocaste, Polyneikes)`, `(Oedipus, Polyneikes)` and `(Polyneikes, Thersandros)`). The ABox also specifies which individuals are patricide and which are non-patricide (traditionally `Oedipus` is known to belong to the former, while `Thersandros` to the latter).

We have chosen the namespace represented by the `io:` prefix for the names in this ontology. The database connection is named `iodb`, and the corresponding DSN is specified as `"iocaste"` (line 1). This connection is accessed without specifying a user name or a password. Accordingly, `iodb` has no attributes other than `dlog:hasDSN`.

Both the role `hasChild` and the concept `Patricide` are taken from this database. The role `hasChild` is represented by the instance `dlog:riohasChild`. We chose this name as a mnemonic for a *r*ole from the namespace *io*, called *hasChild*, but any other name could have been used. Line 2 tells the system that this individual represents the role `io:hasChild`. Here, the right hand side of the role is of no interest, so we chose to have the same individual as on the left hand side. Line 6 tells that the individual `dlog:cioPatricide` is an instance of

```
1  {dlog:iodb, dlog:hasDSN, "iocaste"}
2  {dlog:riohasChild, io:hasChild, dlog:riohasChild}
3  {dlog:riohasChild, dlog:hasConnection, dlog:iodb}
4  {dlog:riohasChild, dlog:hasQuery,
5                     "SELECT parent, child FROM hasChild"}
6  {dlog:cioPatricide, rdf:type, io:Patricide}
7  {dlog:cioPatricide, dlog:hasConnection, dlog:iodb}
8  {dlog:cioPatricide, dlog:hasQuery,
9                      "SELECT name FROM people WHERE patricide"}
10 {dlog:cioPatricide, dlog:hasNegQuery,
11                     "SELECT name FROM people WHERE NOT patricide"}
```

Fig. 4. An example of the simplified database interface.

the concept `io:Patricide`[1]. This individual, which thus represents the concept `io:Patricide`, has two queries associated with it: one for `io:Patricide` (line 8) and one for its negation (line 10).

The simplified interface allows complex queries, such as the one for Patricide which has a `WHERE` clause. This way the existing table `people` can be used without modification. However, this approach makes it very difficult to transform any possible query predicates in the TBox to direct database queries, and instance check queries run with a poor performance.

We now present a second example. The TBox of this example, taken from [4], is shown below.

```
1  ∃hasFriend.Alcoholic  ⊑  ¬Alcoholic
2  ∃hasParent.¬Alcoholic  ⊑  ¬Alcoholic
```

Line 1 describes that those who have a friend who is alcoholic are non-alcoholic (as they see a bad example), while line 2 states that those who have a non-alcoholic parent are non-alcoholic (as they see a good example). In the classic form the ABox contains role assertions for the `hasParent` and `hasFriend` relations only, and no concept assertions about anyone being alcoholic or non-alcoholic. In spite of this, in the presence of certain role instance patterns, one can infer some people to be non-alcoholic, using case analysis.

For example, consider the following pattern: Jack is Joe's parent and also his friend. Now, if we assume that Jack is alcoholic, then the axiom in line 1 implies that Joe is not alcoholic. On the other hand, if Jack is not alcoholic, it follows from line 2 that Joe is not alcoholic, either. Thus these two role assertions imply that Joe has to be non-alcoholic. Other patterns, where Joe can be inferred to be non-alcoholic, are the following: Joe is a friend of himself; Joe is a friend of an ancestor; and Joe's two ancestors are in the `hasFriend` relationship.

[1] Note that the prefix `rdf`, used in the predicate position of the triple in line 6, refers to the RDF namespace: `http://www.w3.org/1999/02/22-rdf-syntax-ns#`.

In Figure 5 we present a database access specification for the above example, using the complex interface. Here, the database `alcoholic` is accessed with the user name `"drunkard"` and the password `"palinka"` (lines 1–3). We assume that a new view, called `"hasParentView"`, was defined in the database to hide the complex query for the role `hasParent`, cf. lines 4–6. The columns of this view, `child` and `parent` (lines 7–8), contain the data for the role `hasParent`. From this information DLog can create a query for instance retrieval (`"SELECT child, parent FROM hasParentView"`), and three other query patterns for the cases when at least one of the individuals is known (e.g. `"SELECT child FROM hasParentView WHERE parent = ?"`). This approach allows for the generation of complex database queries for the query predicates.

```
1  {dlog:alcdb, dlog:hasDSN, "alcoholic"}
2  {dlog:alcdb, dlog:hasUserName, "drunkard"}
3  {dlog:alcdb, dlog:hasPassword, "palinka"}
4  {dlog:ralchasParent, alc:hasParent, dlog:ralchasParent}
5  {dlog:ralchasParent, dlog:hasConnection, dlog:alcdb}
6  {dlog:ralchasParent, dlog:hasTable, "hasParentView"}
7  {dlog:ralchasParent, dlog:hasLHS, "child"}
8  {dlog:ralchasParent, dlog:hasRHS, "parent"}
9  {dlog:ralchasFriend, alc:hasFriend, dlog:ralchasFriend}
10 {dlog:ralchasFriend, dlog:hasConnection, dlog:alcdb}
11 {dlog:ralchasFriend, dlog:hasTable, "friends"}
12 {dlog:ralchasFriend, dlog:hasLHS, "friend1"}
13 {dlog:ralchasFriend, dlog:hasRHS, "friend2"}
14 {dlog:calcAlcoholic, rdf:type, alc:Alcoholic}
15 {dlog:calcAlcoholic, dlog:hasConnection, dlog:alcdb}
16 {dlog:calcAlcoholic, dlog:hasTable, "alcoholicView"}
17 {dlog:calcAlcoholic, dlog:hasColumn, "name"}
18 {dlog:calcAlcoholic, dlog:hasNegTable, "nonalcoholicView"}
19 {dlog:calcAlcoholic, dlog:hasNegColumn, "name"}
```

Fig. 5. An example of the complex database interface.

In Figure 5, lines 10–13 specify the database access for the role `hasFriend`, while lines 14–19 allow for accessing individuals belonging to the concept `alcoholic` and its negation through appropriate database views.

5 Integrating DLog with Protégé

Protégé [2] is an open source ontology editor that supports the Web Ontology Language (OWL) [1], and can connect to reasoners via the HTTP-based DIG interface [16]. The DLog server implements the DIG interface and can be used to execute instance retrieval queries issued from the graphical interface of Protégé.

The DIG interface specifies communication via HTTP, and uses XML data format. For the implementation we used the HTTP server provided with SWI-Prolog. In implementing the interface we faced difficulties caused by some ambiguities of the DIG specifications, despite there being an (exact) XML schema definition. Another difficulty was that Protégé does not strictly follow the definition of the interface. For example it uses a `clearKB` command that is not even defined in version 1.1 of DIG. In DIG 1.0, which supported only a single database, this command was defined, but Protégé uses the new version that supports multiple concurrent knowledge bases. We strove for an implementation as generic and complying to the interface definition as possible while, also being compatible with Protégé.

For parsing XML we use the SGML module of SWI-Prolog, which can be operated in an XML compatibility mode, allowing namespaces. As this is not a direct XML parser, it has some difficulties when used in XML mode. For example even with the strictest settings and treating all warnings as errors, it accepts input files that are not even well-formed XML. Because of this, and in hope of better performance, we are planning to switch to Apache Xerces-C++. With Xerces we plan to use SAX parsing, instead of DOM, with the hope of lower memory usage and faster parsing.

The data are extracted from the XML DOM using Definite Clause Grammars (DCG).

Figure 6 shows the results of a query issued from Protégé, as answered by the DLog server.

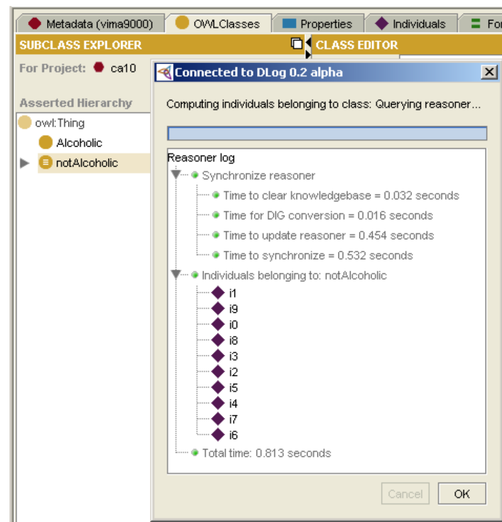

Fig. 6. Screenshot of query results in Protégé answered by DLog.

The integration of Protégé and the database interface is in progress. A serious difficulty is that if the results of a query contain individuals that are not defined in Protégé (i.e. individuals present only in databases) Protégé silently drops these individuals from the list of query results.

6 Evaluation

This section contains a preliminary performance test of the database interface.

We tried the database interface on a large version of the Iocaste problem which contains 5058 pairs in the `hasChild` relation, 855 instances that are known to be patricide, and 314 that are known to be non-patricide.

The execution results are summarised in Table 1. The load time means the time it takes to load the file which contains the axioms, including the XML parsing. The translation time is the time it takes to generate the TBox and ABox code from the axioms, while execution time is the run-time of the query.

Table 1. Comparing the in-memory and database version of a large Iocaste test.

(seconds)	load	translate	execute	total
in-memory	0.88	0.53	0.02	1.43
database	0.05	0.02	0.36	0.43

When the ABox is stored in memory, the translation takes 1.41 seconds, and the execution takes only 0.02 seconds. Note that these figures were obtained with the indexing optimisation turned off. When this optimisation is turned on, the number of generated ABox clauses is doubled, and translation time increases accordingly.

The database variant of the example enumerates all the instances of the queried concept in 0.36 seconds. This, compared to the original 0.02 seconds is much slower. However, the time we spent at compile-time was altogether 0.07 seconds, resulting in a total execution time of 0.43 seconds. To sum up, in terms of total query execution time, more than a three-fold decrease was achieved, using the database interface.

From the above data it may seem that using a database for storing the ABox, which fits into memory, is beneficial only because of the reduced compile-time. However, we believe that in the case of large data sets and complex queries (especially if these contain concepts giving rise to query predicates) execution time can also be better than that of the in-memory variant.

Detailed evaluation of the DLog System can be found in [3].

7 Summary and future work

In this paper we have shown the architecture of the DLog system, discussed a database interface for representing large ABoxes, and reported on the integration of DLog with the Protégé ontology editor.

The database interface is especially useful if the data set cannot fit in memory or if it is shared with other systems. Using databases can greatly reduce compile time and, with advanced optimisations, it may provide efficiency similar to that of the in-memory version.

Future improvements include the optimisation of query predicates, by transforming them to database queries, and better integration of Protégé and the database interface. Our plans also include the implementation of a query module to handle composite queries, and the support for additional interface formats, such as OWL, or the KRSS notation used by e.g. the RacerPro engine.

Acknowledgements

The authors are grateful to the anonymous reviewers for their comments on the earlier version of the paper, and especially for recommending the *Billion Triples Challenge* for evaluation.

References

1. Bechhofer, S.: OWL web ontology language reference. W3C recommendation (February 2004)
2. Noy, N., Fergerson, R., Musen, M.: The knowledge model of Protege-2000: Combining interoperability and flexibility. http://citeseer.nj.nec.com/noy01knowledge.html (2000)
3. Lukácsy, G., Szeredi, P.: Efficient description logic reasoning in Prolog: the DLog system. Technical report, Budapest University of Technology and Economics (January 2008) Conditionally accepted for publication in Theory and Practice of Logic Programming.
4. Lukácsy, G., Szeredi, P., Kádár, B.: Prolog based description logic reasoning. (December 2008) To appear in ICLP 2008.
5. Haarslev, V., Möller, R.: Optimization techniques for retrieving resources described in OWL/RDF documents: First results. In: Ninth International Conference on the Principles of Knowledge Representation and Reasoning, KR 2004, Whistler, BC, Canada, June 2-5. (2004) 163–173
6. Haarslev, V., Möller, R., van der Straeten, R., Wessel, M.: Extended Query Facilities for Racer and an Application to Software-Engineering Problems. In: Proceedings of the 2004 International Workshop on Description Logics (DL-2004), Whistler, BC, Canada, June 6-8. (2004) 148–157
7. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semant. **5**(2) (2007) 51–53
8. Horrocks, I., Li, L., Turi, D., Bechhofer, S.: The Instance Store: DL reasoning with large numbers of individuals. In: Proceedings of DL2004, British Columbia, Canada. (2004)

9. Horrocks, I., Voronkov, A.: Reasoning support for expressive ontology languages using a theorem prover. In: FoIKS. Volume 3861 of Lecture Notes in Computer Science., Springer (2006) 201–218
10. Hustadt, U., Motik, B., Sattler, U.: Reasoning for Description Logics around SHIQ in a resolution framework. Technical report, FZI, Karlsruhe (2004)
11. Motik, B.: Reasoning in Description Logics using Resolution and Deductive Databases. PhD thesis, Univesität Karlsruhe (TH), Karlsruhe, Germany (January 2006)
12. Grosof, B.N., Horrocks, I., Volz, R., Decker, S.: Description logic programs: Combining logic programs with description logic. In: Proc. of the Twelfth International World Wide Web Conference (WWW 2003), ACM (2003) 48–57
13. Hustadt, U., Motik, B., Sattler, U.: Data complexity of reasoning in very expressive description logics. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI 2005), International Joint Conferences on Artificial Intelligence (2005) 466–471
14. Stickel, M.E.: A Prolog technology theorem prover: a new exposition and implementation in Prolog. Theoretical Computer Science **104**(1) (1992) 109–128
15. Zombori, Zs.: Efficient two-phase data reasoning for description logics. In: Proceedings of the International Federation for Information Processing Technical Committee on Artificial Intelligence (TC12), Milan, Italy (September 2008) Accepted conference paper.
16. Bechhofer, S.: The DIG description logic interface. http://dig.cs.manchester.ac.uk/ (2006)