A Simple DL Instance Store

Sean Bechhofer Ian Horrocks Daniele Turi
Information Management Group,
Department of Computer Science,
University of Manchester, UK
<lastname>@cs.man.ac.uk

Abstract

A simple instance store allows descriptions of objects in terms of concept terms from a DL model and retrieval of those objects through concept queries. It is not a full A-box, as relationships between objects are not represented.

1 Introduction

Traditionally, Description Logic systems have been split into a *T-Box* and an *A-Box*. The T-Box handles information about the terminology or schema, providing class descriptions in terms of properties, while the A-Box is concerned with facts about particular instances or individuals of classes. With more expressive DL languages (for example when the $oneOf$ operator is present), this distinction between the T- and A-Box collapses [**?**], but in general the distinction is useful.

With the recent explosion in interest in formal ontologies, driven largely by the development of the Semantic Web [**?**], DLs have emerged as a useful tool for the support of ontological engineering. In particular, the use of T-Box reasoning has enabled the development of editing and support environments that can detect inconsistencies in conceptual models and aid modellers in maintaining and organising concept hierarchies [**?**].

A full DL A-box allows us to make assertions about individuals of the form $i \in C$ and $P(i, j)$ for individuals i and j, arbitrary concept descriptions C and properties (binary relationships) P. We can then query the A-box with queries such as:

Retrieval Which individuals occurring in the assertions are instances of some concept description Q?

Realization Given an individual i, what are the most specific concepts C in the T-box that i is an instance of?

Answering such queries requires full A-box reasoning, a task that is known to be difficult in the presence of expressive languages for the T-box [**?**].

An alternative is a simple *instance store*, that provides weaker functionality than a full A-box. In this case, we allow assertions of the form $i \in C$ for arbitrary concept descriptions, but do not allow explicit assertions of relationships between individuals. This should provide sufficient functionality, however, to support applications where individual objects are to be described and then subsequently retrieved using concept descriptions, e.g. service catalogues [**?**].

A well known technique for implementing an instance store [**?**] is to simply treat individuals as primitive concept definitions and add new primitive concepts for each individual. These concepts are then flagged as being "pseudo-individuals". For example, for an assertion $i \in C$, where C is an arbitrary concept expression, we add a new primitive concept I_i, along with an axiom asserting that $I_i \equiv C$. Retrieval then simply involves involves classifying the query Q and returning all individuals i s.t. I_i lies below Q in the concept hierarchy. The problem is that with realistic knowledge bases likely to run into millions of instances, such an approach is unlikely to scale.

In the presence of the above restrictions, however, we can use a standard database to do much of the work required in implementing an instance store, making use of the optimisations that DBMSs provide in order to support large numbers of instances. Although such an approach relies on known technical results, to the best of our knowledge no such implementation exists.

Note that although we do not allow general assertions of the form $P(a, b)$ for object properties, an instance store *can* support assertions of the form e.g. $R(a, 17)$, for some concrete valued property R.

This paper discusses implementation strategies for an instance store using a combination of relational database and DL reasoner. The instances and their relationship with concepts are stored in the database, while the reasoner deals purely with T-Box functionality. Retrieving individuals is then a combination of query against that database and subsumption and classification requests to the reasoner. We concentrate here primarily on the **retrieval** task, although realization can also be provided.

2 Implementation

We make a number of simplifying assumptions:

A Descriptions of individuals are not updated.

B One description is applied to each individual.

C The ontology is not changed or updated.

2.1 Algorithms

The functionality offered by the individual store is encapsulated in three basic functions as shown in Figure 1. Note that we assume that the descriptions being presented in assertions and queries relate to the ontology which was originally loaded.

```
initialise(Ontology O)
assert(Description D, Individual I)
{Individual} retrieve(Description D)
```

Figure 1: Individual Store API

A general retrieval algorithm is shown in Algorithm 1. This makes use of a number of ancillary functions such as $getChildren$ and $allIndividuals$. The implementation of these functions differs depending on the chosen implementation strategy.

Algorithm 1 $retrieve(Description\ D)$

```
 1: result ← ∅
 2: children ← reasoner.getChildren(D)
 3: for each c ∈ children do
 4:    result ← result ∪ allIndividuals(c)
 5: end for
 6: equivalents ← reasoner.getEquivalents(D)
 7: if equivalents ≠ ∅ then
 8:    for each c ∈ equivalents do
 9:       if c ≠ ⊥ then
10:          result ← result ∪ individuals(c)
11:       end if
12:    end for
13: else
14:    parents ← reasoner.getParents(D)
15:    for each c ∈ parents do
16:       for each i ∈ allIndividuals(c) do
17:          if instanceOf(i, D) then
18:             result ← result ∪ {i}
19:          end if
20:       end for
21:    end for
22: end if
23: return result.
```

An informal description of retrieval is as follows. In order to retrieve all instances of a description, we first find subsumed primitive classes and retrieve all the individuals known to be instances of those classes. If the description is equivalent to a

```
Assertions(individual, description)
Primitives(concept, individual)
```

Figure 2: Database Schema 1

primitive, we also need to return instances of the equivalent, and are then done. If, however, the description is not equivalent to a primitive, we then need to retrieve all the individuals which are instances of immediate parents of the description, and check whether those individuals are, in fact, instances of the given description. If they are, then the relevant individuals should also be returned.

2.2 Database

For each individual, we need to store at least:

1. the concept description that has been applied to the individual;
2. the primitive concepts that the individual is known to be an instance of.

Implementation strategies then boil down to decisions as to how much information we cache in the database, and how much we rely on dynamic use of the reasoner. There are then a number of alternative strategies that we can employ with regards to 2 which will be explored below.

2.3 Basic Strategy

The first approach that we can take stores minimal information in the database, and relies on the reasoner to provide subsumption testing and hierarchy information. The schema is shown in Figure 2.

Algorithms for initialisation, assertion and retrieval are then as shown in Algorithms 2, 3, 4, 5 and 6. The functions $getChildren$, $getParents$, $getEquivalents$, $getDescendants$ and subsumption testing are then all implemented through simple calls to the reasoner.

Algorithm 2 $initialise(Ontology\ O)$

1: Set all database tables to be empty
2: Load the ontology O into the reasoner

2.4 Optimisation 1

The first optimisation we can make is to effectively cache the classification hierarchy in the database. This means that collecting all individuals that are instances of a particular concept can be performed as a database query, rather than through multiple

Algorithm 3 $assert(Description\ D, Individual\ I)$

```
INSERT INTO Assertions VALUES (I, D)
concepts ← reasoner.getEquivalents(D)
if concepts = ∅ then
    concepts ← reasoner.getParents(D)
end if
for each c ∈ concepts do
    if c = ⊥ then
        raise Exception
    end if
    INSERT INTO Primitives VALUES (c, I)
end for
```

Algorithm 4 $individuals(Concept\ c)$

```
return SELECT individual FROM Primitives WHERE concept = c
```

Algorithm 5 $allIndividuals(Concept\ c)$

```
result ← individuals(c)
descendants ← reasoner.getDescendants(c)
for each d ∈ descendants do
    result ← result ∪ individuals(d)
end for
return result
```

Algorithm 6 $instanceOf(Individual\ i, Description\ D)$

```
retrievedDescription ← SELECT description FROM Assertions WHERE individual = i
return reasoner.subsumedBy(retrievedDescription, D)
```

```
Assertions(individual, description)
Primitives(concept, individual)
Subsumption(concept, descendant)
```

Figure 3: Database Schema for Optimisation 1

queries to the reasoner and database. In addition, this has the advantage that the number of explicit union operations are reduced and are instead performed by the DBMS – a task which is likely to have been highly optimised. A naive approach would be to simply add a table containing parent/child pairs that occur in the hierarchy. This is not sufficient, however as it would then require a recursive database queries in order to retrieve the transitive closure (which is required in order to retrieve all the individuals). Instead, we provide a table that caches *all* subsumption relationships between primitive concepts. The extended schema is as shown in Figure 3.

Initialisation then requires us to populate the `Subsumption` table as shown in Algorithm 7. We can now retrieve all individuals that are instances of a concept with

Algorithm 7 $initialise_1(Ontology\ O)$

1: Set all database tables to be empty
2: Load the ontology O into the reasoner
3: **for each** $c \in reasoner.allPrimitiveConcepts$ **do**
4: `INSERT INTO Subsumption VALUES` (c, c)
5: **for each** $d \in reasoner.getDescendants(c)$ **do**
6: `INSERT INTO Subsumption VALUES` (c, d)
7: **end for**
8: **end for**

a single query as in Algorithm 8.

Algorithm 8 $allIndividuals_1(Concept\ c)$

1: return
```
SELECT Primitives.individual
FROM Primitives, Subsumption
WHERE Subsumption.descendant = Primitives.concept
AND Subsumption.concept = c
```

2.5 Optimisation 2

An alternative optimisation is to effectively cache the transitive closure in the `Primitives` table. This requires a slight change to the schema as shown in Figure 4. In this case, no preprocessing stage is required to compute the transitive closures. We alter the assertion process as shown in Algorithm 9.

```
Assertions(individual, description)
Primitives(concept, individual, mostSpecific)
```

Figure 4: Database Schema for Optimisation 2

Algorithm 9 $assert_2(Description\ D, Individual\ I)$

```
1: INSERT INTO Assertions VALUES (I, D)
2: concepts ← reasoner.getEquivalents(D)
3: if concepts = ∅ then
4:    concepts ← reasoner.getParents(D)
5: end if
6: for each c ∈ concepts do
7:    if c = ⊥ then
8:       raise Exception
9:    end if
10:   INSERT INTO Primitives VALUES (c, I, true)
11:   for each d ∈ reasoner.getAncestors(c) do
12:      INSERT INTO AllPrimitives VALUES (d, I, false)
13:   end for
14: end for
```

Individuals can again be retrieved through a single query, as shown in Algorithm 11, while retrieval of direct individuals is as shown in Algorithm 10.

Algorithm 10 $allIndividuals_2(Concept\ c)$

```
1: return SELECT individual
   FROM Primitives
   WHERE concept = c
```

2.6 Further Optimisations

There are additional optimisations that we could try.

- We can probably replace steps 3-5 Algorithm 1 with a single database query once we've got hold of all the children of the description.
- Similarly, steps 15-21 can probably be improved on.
- Storing the asserted descriptions in a normal form and caching the results of their classification.

Algorithm 11 $individuals_2(Concept\ c)$

1: return `SELECT individual`
`FROM Primitives`
`WHERE concept =` c `AND mostSpecific =` $true$

3 Comparisons & Evaluation

Could do with some evaluation of the various approaches. Table 1 provides a rough characterisation of the approaches.

Table 1: Comparison

Strategy	Pros	Cons
Basic	Conceptually simple	Lots of unioning going on. Communication with the reasoner
Optimisation 1	Less reasoner communication	Preprocessing step to calculate the transitive closure. Transitive closure table may be large – $O(n^2)$ in the size of the ontology).
Optimisation 2	Single lookup.	Primitives table may get very big – something like $O(mn^2)$ where m is the number of individuals, n is the size of the ontology. For e.g. 10^6 individuals and $5,000$ concepts, this could be prohibitively large.

Would be nice to include some discussion of the GO experiment.

4 Relaxing Assumptions

For assumption **A**, if we wish to change the assertion(s) that apply to an individual I, we can simply remove all tuples containing the individual I and then reassert with the new information.

Restriction **B** can be handled similarly. If a assertion $i \in D$ is added to an individual where $i \in C$ has already been asserted, we simple remove this (as in **A**) and replace with $i \in C \sqcap D$. This is, of course, not an optimal solution, but would work.

Relaxation of **C** may be harder to deal with, as new information may force a reclassification of primitive concepts. Optimisations 1 and 2 would probably require updates to the transitive closure information recorded in the tables if the ontology was changed.