1. Requirements

This paper continues our explorations [1, 3, 2] into the implementation of extensible DL reasoners that try to maintain a model-like data-structure, rather than using a refutation technique\(^1\).

A DL-KBMS\(^2\) typically manages a set of named concept definitions ("classes"), including maintaining an up-to-date taxonomy of them organized by specificity according to subsumption. In practice, the DL-KBMS must also manage information about individuals, which accumulate information in the form of (i) associated descriptions (Anna has all(heightInCM,min(0.90))), and (ii) fillers for roles, describing the relationships between individuals (Anna has Mimi as sister).

For this purpose, the KBMS offers operations CREATEIND, TELLCLASS, TELLFILLER and TELLCLOSED. (The latter is needed to deal with the open-world assumption: TELLCLOSED(Anna,sister) says there are no more sisters to come. There are of course corresponding retrieval operations: ASKCLASS, ASKFILLER, ASKCLOSED. More commonly, users wish to obtain the list of all individuals known to be fillers for a role: GETALLFILLERS(Ind,role), and the lowest class(es) in the taxonomy to which an individual belongs: ASKPARENTS(Ind).

In order to detect inconsistencies as early as possible, and as a general principle, our system performs "eager reasoning", computing and caching the answers to the last three kinds of questions for each individual.

A crucial property of individual processing is that the acquisition of information about individuals is incremental. Whenever any new fact is asserted (either explicitly by a user or as a result of some inference) some of the following kinds of inferences may result:

- Additional facts (descriptions or filler information) can be inferred about the individual (e.g., learning that a role has no fillers implies the individual satisfies every all restriction).
- New facts can be inferred about other individuals (e.g., after TELLFILLER(Anna,sibling,Mimi) we can infer that Mimi is an instance of FEMALE, if we already know that Anna has all(siblings,FEMALE)).
- An individual may become "inconsistent" either because the conjunction of the descriptions asserted about it are incoherent (this is detected by concept-level reasoning, not discussed here) or because of conflict between the descriptor and filler information (e.g., more fillers added to an attribute than permitted by an at-most restriction).
- As a result of a change in an individual, it or some other individual may need to be reclassified with respect to the taxonomy of classes.

2. Software Architecture for Individual Reasoning

The data structure for every individual includes both a concept (to be called Descriptor), obtained by conjoining and normalizing the descriptions successively asserted about the individual; and individual attribute-filler information recording the specific values assigned so far, and whether the role is closed or not.

This allows more efficient handling of filler information (possibly by indexing), since in many applications this kind of information may be orders of magnitude larger than concept-level information.

The ProtoDL architecture uses three functions:

\texttt{Recognizes?}(ind,term),
\texttt{ConsistentW?}(ind,term) and
\texttt{InferFrom}(ind,term)

in order to implement the various tell operations, as well as the one that performs classification. As with concept processing [1], we use the and constructor to drive the processing, and modularize the implementation so each kind of concept constructor $K$ has its own set of procedures: $\texttt{Recognizes?}[K](\text{ind,term})$

\texttt{ConsistentW?}[K](ind,term) and \texttt{InferFrom}[K](ind,term)\(^3\). For examples, see Figure 1.

---

\(^1\)Hence, the major weakness of such reasoners is dealing with disjunctive situations requiring case-by-case reasoning.

\(^2\)We assume the reader is familiar with the terminology of DLs and concept processing.

\(^3\)In the case of constructors $K$ that involve a role, we have found it useful to automatically pass as parameters the fillers for
function InferFrom\[(\text{and})\](b:IND,C:CONCEPT)
{for every term \(K(T)\) in \(C\) \{InferFrom\[(K)\](b,K(T))\}}
for every role \(r\) in \(C\),
\{retrieve the fillers \(F\) and closed information \(\text{clsd}\?)\} on \(b\) for \(r\);
for every term \(K(T)\) on \(r\) in \(C\) \{InferFrom\[(K)\](b,fillers,clsd?,K(T))\}\}

function Recognizes\[(\text{all})\](b:IND,fillers:SET(IND),clsd?:BOOL,\text{all}(vr:CONCEPT)) returns BOOL
{if not clsd? then return false ; more fillers might be told later
else for every \(f\) in fillers \{if not Recognizes\[(\text{and})\](f,vr) then return false\}}
return true\}

Figure 1: InferFrom\[(\text{and})\] and Recognizes\[(\text{all})\]

2.1 Dependency Links
Whenever information about \(Ind\) changes, the result of a function such as Recognizes\[(\text{all})\] or ConsistentW\[?\] might change not just for \(Ind\) but for almost any other individual. To find these, we will maintain dependency links. Rather than use a full Truth-Maintenance System, we follow the approach taken by Peter Patel-Schneider in the CLASSIC implementation, where one individual \(v\) may point to another, \(b\), if the result of a decision about the latter \(b\) may change as a result of additional information being added to the former \(v\). For example, if a filler \(v\) is added to attribute \(\text{pet}\) of \(b\), and there is a class with definition \(\text{all}(\text{pet,DOG})\), then if it is not known whether \(v\) is or is not a \(\text{DOG}\), then there will be a dependency link from \(v\) to \(b\) indicating that \(b\) should be re-classified when \(v\) is modified (since the modification might have decided the \(\text{DOG}\)-hood of \(v\)).

We will keep different kinds of dependency links for each of the three kinds of procedures (Recognizes\[(\text{all})\], ConsistentW\[?\], InferFrom), and in turn, each call to these needs to produce a list of individuals whose modification might change the result of function call (e.g., the membership test having been undecided changes to it being True). The task of the main algorithms then includes dealing with dependencies by setting appropriate links, and if some individual \(b\) has newly inferred facts about it, then the individuals depending on it need to be put on a queue for re-processing. (See section 2.3)

2.2 Inferences from Incremental Updates
Since updates to individuals are incremental in nature, in order to increase efficiency we want to take into account the fact that the KBMS had already performed all the inferences on this individual up to, but not including, the current update. For example, if some individual \(b\) has been asserted to be an instance of \(\text{all}(\text{pet,DOG})\), and already has \(\text{pet}\)-fillers \(e,f\), then if the current update is \(\text{TELLFILLER}(b,\text{pet},g)\), then we only need to propagate the information that \(\text{pet}\)-fillers are \(\text{DOG}\)s to \(g\), since the others would have been processed earlier. We will therefore have the following three variants of InferFrom\[(K)\]:

\[
\begin{align*}
\text{InferFromFilling}[K] \\
\text{InferFromClosing}[K] \\
\text{InferFromAsserting}[K],
\end{align*}
\]
and similar variants of ConsistentW\[?\][K].

For example, ConsistentW\-Filling\[(\text{all})\](\(ind\),new filler,\(\text{all}(r,vr)\)) invokes ConsistentW\-Asserting\[(\text{and})\](new filler,\(vr\)) to check that the new filler does not contradict the role restriction; on the other hand, ConsistentW\-Closing\[(\text{all})\](\(ind\),\(r\),\(\text{all}(r,vr)\)) just returns true, or better yet, disappears as a result of macro expansion; and ConsistentW\-Asserting\[(\text{all})\](\(ind\),\(\text{all}(r,vr)\)) verifies that all \(r\)-fillers \(y\) on \(ind\) are consistent with \(vr\) by calling ConsistentW\-Asserting\[(\text{and})\]\((y,\text{ann})\).

We note that the above approach is based on an implicit assumption (born out by empirical evidence with attempted extensions) that inferences about individuals from the conjunction of \(C\) and \(D\), can be usually
function TELLCLOSED(ind, role)
    {if redundant information then signal RedundantException;
        add the information that role is closed to inds data structure;
        initialize the blackboard lists to empty;
    }
    ind-descr := Descriptor(ind)
    /* deal with consistency issues involving ind */
    ConsistentW-Closing?[and](ind, ind-descr, role)
        [handle InconsistentException by rolling back update and printing error msg]
        /* this procedure will post dependencies if the answer is not a definite YES */
        ToRecheckConsistency += getConsistentDependsOnMe(ind)
        /* this puts up for reconsideration objects dependent on ind */
        /* check for inferences */
        InferFromClosing[ind](ind, ind-descr, role, filler);
        /* this may post new updates and/or dependencies */
        ToRedoInferFrom += getInferDependsOnMe(ind)
    /* reclassify at least this individual */
    Reclassify(ind);
    ToReclassify += getRecognizeDependsOnMe(ind)
    /* Process tasks left on the blackboard */
    ProcessBlackBoard;
    }  [handle InconsistentException by rolling back update and printing error msg]

Figure 2: Pseudo-code for TELLCLOSED

obtained from the union of the inferences from C and those from D. (This is not always the case, so we must view these decompositions as heuristic.)

2.3 Using the functions

The top-level KBMS update operations orchestrate the invocation of the various functions, but in the case of individuals must also be concerned with repeatedly (i) setting appropriate dependency links, (ii) gathering into lists other objects whose dependency links have been “ticked” by the latest update, (iii) invoking appropriate functions to continue the reasoning on these objects.

For this purpose, we have ended up using a black-board architecture that maintains 5 different sets of objects: (i) ToRecheckConsistency, ToRedoInferFrom and ToReclassify contain objects reached from ind through respective dependency links, as well as, in the last case, any update to the object; (ii) ToPerformUpdate, containing facts to be asserted about individuals as a result of inference procedures; (iii) ToAddDependency, keeping track of dependencies found to be needed by the three operations, as previously described.

Because, for example, an update may cause a propagation inference, which is another update, we have no linear order in which these lists can be processed. Theoretically, we may therefore have a loop where a "demon" removes an arbitrary object from its list, invokes the required procedure, and then repeat the process. The only requirement is that dependencies on the ToAddDependencies list be also considered as having been installed, whenever chasing objects that may have been affected by an update.

Several policies can help the performance of the system: (1) In all cases, it is helpful to eliminate duplicates from the lists. (2) Locality of context may improve performance; therefore grouping together operations to be performed on one object is advisable (e.g., gather together all the dependencies from object y on ToAddDependency). (3) If we expect inconsistencies to arise infrequently, then this list can be processed at the end.

The pseudo-code for TELLCLOSED is presented in Figure 2.

Notice that we signal inconsistency by raising an exception rather than returning false in ConsistentW?. The advantage of this is that exceptions are propagated up the dynamic function call hierarchy, interrupting
function Recognizes?[sum](b, sum(p, f))
    {if closed?(b, p) and closed?(b, f) and
        getFiller(b, f) == (∑_{e ∈ getFillers(b, p)} e)
    then return true else return false }

function ConsistentWithFilling[sum](b, q, v, sum(p, f))
    if ((q == f) and closed?(b, p) and (v != (∑_{e ∈ getFillers(p)} e)))
    then signal InconsistentException else return true

function InferFromFilling[sum](b, q, v, sum(p, f))
    if (q == f) then {
        n = get[at-most](b, p);
        E = getFillers(b, p);
        if (cardinality(E) == (n-1))
            then ToPerformUpdates += " TELLFiller(b, p, v- ∑(E)) ";
    }
    if (q == p) and closed?(b, f) then {
        a = getFiller(b, f);
        n = get[at-most](b, p);
        E = getFillers(b, p);
        if (cardinality(E) == (n-2))
            then ToPerformUpdates += " TELLFiller(b, p, a-v- ∑(E)) ";
    }

Figure 3: Some procedures for sum

functions until a handler is encountered — usually at an external or internal TELL operation; this issues appropriate error messages, resets any local changes made, and then re-raises/propagates the exception further up, until the top level is reached.

2.4 An example of adding a new constructor: sum

Let us consider the extensions needed for a constructor, sum, that would be useful in several applications, such as databases. We will introduce it in a number of increasingly more complex variants.

To begin with, sum(roleId, attributeId), as in sum(lengths, totalLength), denotes individuals b such that the sum of their roleId fillers (which are required to be real numbers), equals to the value of the attributeId. For now, let us assume that the other constructors in the language are and, at-least, at-most, and all.

The Recognizes? and ConsistentW? procedures for this constructor are based on the following inference rule, which expresses the denotational semantics of the constructor:

\[
\begin{align*}
    \frac{\text{closed}(p) \quad \text{fillers}(b, p) = E \quad \text{fillers}(b, f) = a}{b \rightarrow \text{sum}(p, f)} \quad a = (∑_{e ∈ E} e)
\end{align*}
\]

Several procedures dealing with sum are presented in Figure 3.²

An interesting case occurs for inferences when all but one of the fillers of p are known, as well as the value of f:

\[
\begin{align*}
    b \rightarrow \text{sum}(p, f) \quad b \rightarrow \text{at-most}(m, p) \quad E = \text{fillers}(b, p) \quad a = \text{filler}(b, f) \quad c = a - (∑_{e ∈ E} e), \quad |E| = m - 1
\end{align*}
\]

This leads to two cases where one can infer a new value: when the second-to-last value is added to p, or when the value of f becomes known (see Figure 3).

²These assume that filling an attribute closes it immediately, but that when a role is filled, a separate TELLCLOSED operation is needed, even if it be inferred from the current fills.
Suppose we enhance the logic to represent intervals of real numbers, by adding constructors \(\text{min}\) and \(\text{max}\): e.g., \(\text{and}(\text{min}(2.0), \text{max}(4.0))\) represents reals from 2 to 4. Then the consistency checking and inference procedures need to be extended to account for the following rule of inference:

\[
\begin{align*}
\begin{array}{c}
\frac{b \rightarrow \text{and}(\text{sum}(p,f), \text{all}(p, \text{min}(n))), \text{at-least}(k))}{\frac{E = \text{getFillers}(b,p)}{n = (\sum_{e \in E} e) + m \times (k - |E|)}}
\end{array}
\end{align*}
\]

Finally, we would probably want to extend \(\text{sum}\) so that we can refer to paths of roles and attributes, not just single attributes, when summing up; for example, we would like to say \(\text{sum}(\text{subsidiaries.income}, \text{totalIncome})\). To discuss the novelty involved here, consider the simple case where the syntax is \(\text{sum}(\text{roleId.attributeId1, attributeId2})\). For example, when checking consistency using \(\text{ConsistentWithFilling}[\text{sum}](b,p,v,\text{sum}(p,g,f))\), we might find that \(v\) does not yet have a \(g\)-filler, so we can't be sure yet that \(b\) is consistent with \(\text{sum}(p,g,f)\): it depends on changes to \(v\). Therefore \(\text{ConsistentWithFilling}[\text{sum}](b,p,v,\text{sum}(p,g,f))\) would add \(\boxed{b \text{ depends on } v \text{ for consistency}}\) to the \(\text{ToAddDependency}\) list.

3. Conclusions

We have presented the software architecture of individual reasoning in a DL-KBMS system, using generic terms that support extensibility or adaptation to any particular set of constructor. The main ideas involved were

- the use of 3 building-block procedures for recognizing, checking consistency, and inferring new facts;
- the refinement of each of these procedures into 3 variants that take into account the specific update that triggers their invocation (i.e., a filler being added, a role being closed or a description being added);
- having each of these procedures keep track of objects on which their results depend, assuming that updates are monotonically increasing\(^5\);
- using a black-board architecture, containing lists to which various tasks are posted for asynchronous processing.

The closest related work to these ideas is the treatment of test-concepts in the Classic DL: test functions are essentially combinations of Recognize? and ConsistentW? (and are thus three-valued), and they are also supposed to return dependency links. The novelty here are \(\text{InferFrom}\), the \emph{systematic}, \emph{uniform} treatment of all constructors (made possible by the \(\text{and}\) constructor, and allowing built-ins like \(\text{all}\) to be modified when new additions are made), and especially the refinement of the procedures and dependency links to deal with the different kinds of updates, thus promoting greater efficiency in many cases.

Other research concerning individual reasoning in DLs has been carried out usually in the same framework as subsumption reasoning [4].

Acknowledgments: I am especially indebted to Peter Patel-Schneider, and Daniel Kudenko, who implemented earlier versions of some of these ideas.

References


\(^5\)If the KBMS were to support retraction of told information, then we would also have to keep track of a second kind of dependency for this purpose.