Modular Implementation of Individual Reasoning in PROTODL — the Extensible Description Logic Management System.

Alex Borgida
Daniel Kudenko
Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903
{borgida,kudenko}cs.rutgers.edu

December 1994

Abstract

This is the second report in a series on the PROTODL system, which is an *extensible* knowledge representation and reasoning system based on Description Logics (DLs). We have motivated elsewhere [Borgida&Brachman92, Borgida92] the utility of being able to add new concept constructors to a DL, and, while in previous papers we have concentrated on subsumption reasoning, in this paper we consider reasoning about individuals.

We present the modular implementation of a Description Logic-based KBMS which performs inferences about individuals in such a way that the addition of each new concept constructors is achieved by introducing a series of functions (and possibly modifying some old ones). Considerable emphasis has been placed on the efficient handling of *incremental* updates. This is accomplished by combining the primitive procedures in different ways in order to obtain variants of the standard procedures for infering concept (non)membership – variants that take into account the fact that the previous state of the KB was consistent, and that we know what specific kind of update has been performed, and when dependency links have been set.

1 Introduction

Information Systems can be viewed as mechanisms that support users in developing and maintaining *models* of their application domain. Following Levesque's functional approach to knowledge servers [9], we have developed [5] a view of Information Systems as abstract data types on which one can perform two *kinds* of operations: TELLs and ASKs, using a variety of languages. TELLs are used to build or modify the domain model

$$\text{TELL:} \quad \mathcal{L}_{\text{Tell}} \times \text{KB} \rightarrow \text{KB}$$

while ASKs retrieve information

$$\text{ASK:} \quad \mathcal{L}_{\text{Query}} \times \text{KB} \rightarrow \mathcal{L}_{\text{Answer}}$$

The proper specification of a KB management system and its behavior therefore requires the definition of four kinds of things:

- $\mathcal{L}_{\text{Tell}}$: the language(s) for describing what we know about the world;
- $\mathcal{L}_{\text{Query}}$: the language(s) for describing questions that we wish answered;
- $\mathcal{L}_{\text{Answer}}$: the language(s) in which answers will be phrased;
- Query answering: how answers to queries are related to what has been told to the KB.

Our interest is in Description Languages/Logics (DLs) — a special family of languages that can be used for interacting with an IS in all the roles mentioned above.

```
(and
   C-FUNCTION ,
   at-least(1, args) ,
   all(stmts, C-STATEMENT)  )
```

Figure 1: A description

```
(and
   C-FUNCTION ,
   at-least(2,args) ,
   all(args,all(has-type, ARRAY-TYPE)) ,
   all(stmts, C-STATEMENT)  )
```

Figure 2: Another description

1.1 Description Languages and their Logic

DLs are used to describe situations using various kinds of *individuals*, related by *relationships*, and grouped into *sets/concepts*[1]. This intuition is, of course, shared with a great many other formalisms such as semantic networks, semantic data models, object-oriented languages, etc.

The fundamental observation underlying DLs is that there is a benefit to be gained if languages for talking about these kinds of things are *compositional*, and *structured*. As an example, Figure 1 contains a description of some C functions. Its intended reading would be *"a C-function, which has one or more* `args` *(arguments), and all of its* `stmts` *(statements) are instances of C-statement."* In this description, `args` and `stmts` are binary relations (called roles or set-valued attributes), while `C-FUNCTION` and `C-STATEMENT` are concept names defined elsewhere.

We believe it is instructive to view DLs as special languages obtained by term composition. All DLs therefore have (at least) two sorts of terms: concepts (intuitively, denoting collections of individuals), and roles (intuitively denoting relationships between individuals). Therefore the syntax of DLs consists of rules for creating composite terms from atomic symbols (identifiers of various sorts) and *term constructors*.

The above example was in fact valid according to the syntax of the following simple language, *DL0*:

< conceptDesc> := **thing** | **nothing**
| **prim**(< atomic-concept-id>) *;; concepts without* "iff" *conditions*
| **and**(< conceptDesc>$^+$) *;; concept intersection/conjunction*
| **all**(< roleDesc> ,< conceptDesc>) *;; restricting the range of roles*
| **some**(< roleDesc> ,< conceptDesc>) *;; typed existential quantifier for role fillers*

| **at-least**(< pos-int> , < multiroleDesc>) *;; lower bound on number of fillers*
| **at-most**(< non-neg-int> , < multiroleDesc>) *;; upper bound on the number of fillers*
| **fills**(< roleDesc> , < individual-id>) *;; a particular filler is present*

< roleDesc> := < multiroleDesc > *;; binary relationships*
| < functionalRoleDesc > *;; functional relations*
< functionalRoleDesc > := < attrib-id>
| **compose**(< attrib-id> ,< attrib-id>$^+$) *;; composition of binary relations*

The description in Figure 2 would also be syntactically valid in this term language. It denotes C-functions having at least 2 arguments, all of whose type is some kind of array.

A description, as presented above, is just a variable-free first order term. To give it meaning, we need rules of manipulation which allow us to infer/deduce new descriptions from old ones or relationships between descriptions.

[1] In our examples, we follow the convention that concepts are written in capital letters, relationships in lower-case, and individual's names have their first letters in capitals.

Since concept and role descriptions can easily be thought of as unary and binary predicates, it is natural to consider an entailment relationship between them based on the notion of subset for their denotation (e.g., we would want **at-least**(2,args) to entail **at-least**(1,args)).

The technical term for this is the *subsumption relation* between descriptions: if C and D are of sort "concept", then we want B to subsume C (written as C $\Longrightarrow$ B) when every individual denoted by C is in the denotation of B; similarly for descriptions denoting relationships. Given a description language, we therefore always start by looking for the subsumption relation between concepts ($\Longrightarrow$). The specification of the meaning of *DL0* would therefore indicate, among others, that if the restriction on the role is stronger, then the whole term is more specialized. This could be presented as a rule of inference for example:

$$\frac{\vdash C \Longrightarrow B}{\vdash \mathbf{all}_{(r,C)} \Longrightarrow \mathbf{all}_{(r,B)}}$$

Given appropriate rules for reasoning with *DL0*, we would then expect the description in Figure 1 to subsume the description in Figure 2.

1.2 DL-based KBMS

A KBMS usually contains "generic" information about concepts and regularities (schema, definitions, rules) and specific facts about individuals The former are introduced by TELL-like operations such as DEFINE-CONCEPT, CONSTRAIN-CONCEPT, ADD-RULE, while the latter uses operations such as CREATE-INDIVIDUAL, ASSERT-FACT. The reader is assumed to be familiar with the advantages of using DLs for these tasks, as reviewed, for example, in [2]. In a nut-shell, the subsumption relation can be used to detect incoherence and to organize generic ideas; and if queries are descriptions and subsumption is used as part of query processing then descriptions can be used to assert incomplete information about individuals.

In previous papers [3, 1], we have argued that the expressiveness-intractability trade-off for DL reasoners can be most advantageously resolved by abandoning the search for an "ideal" DL, and by adopting a framework where one starts with a small set of description constructors, and extends them on a per-application (family) basis with new constructors, together with their inferences. These papers showed how the processing of subsumption reasoning for descriptions, when viewed as a process of *normalization*, could be modularized to facilitate the addition of new constructors; this is similar in spirit to the research on compiler technology that has produced an architecture that facilitates the addition of new control structures or data types to a procedural programming language, when compared, say, to the original Fortran compiler. And we have illustrated this approach by reconstructing the CLASP reasoner about plans.

This report performs a similar task for the process of reasoning about individuals in a DL-KBMS. Based on an empirical analysis of the implementation of CLASSIC [4], we have factored out a variety of procedures for each concept constructor, so that the final KBMS can be reconstructed from these components. Considerable emphasis has been placed, and new ideas have been introduced, concerning the *incremental* maintenance of the knowledge base using a "coarse-grained" dependency-maintenance system.

2 Individuals in DL-KBMS

Individual objects are created using an operator such as CREATE-INDIVIDUAL, and one can then make three kinds of assertions about them:

- ASSERT-MEMBER(b,A) asserts that individual b is in the extension of description A. Formally, we will denote such assertions using the formula $b : A$.
- ASSERT-FILL(b,p,v) establishes that b is related to v by role p, denoted by $(b, v) : p$.
- ASSERT-CLOSED(b,p) makes an epistemic statement that all fillers of role p on b are now known. This is needed because, in contrast to relational databases, DL KBMS do not make the closed-world assumption. Such assertions are denoted by $closed(b,p)$.

As explained below, the KBMS might itself infer additional facts of the above three forms. Of course, in querying the KBMS, we want to obtain the same kinds of information

- ASK-BELONGS-TO?(b,A) checks whether individual b is in the extension of description A.
- ASK-FOR-FILLERS(b,p) returns the set of fillers for role p on individual b.

- ASK-CLOSED?(b,p) determines whether p on b has been told or inferred to be closed.

Because of the OWA, membership in (composite) descriptions is a 3-valued function. For example, if all we have been told is that $(b,3):p$, then b is definitely known to be an instance of **atleast**$(1,p)$, and b is definitely known *not* to be an instance of **atmost**$(0,p)$, but b's membership in **atleast**$(2,p)$ or **atmost**$(1,p)$ is open ("may be"). For convenience, instead of the 3-valued ASK operation ASK-BELONGS-TO?(b,D), we will often use two ordinary Boolean functions: *Recognizes?(b,D)*, defined as `AskBelongsTo?(b,D)=='definitely yes'`, and *InconsistetWith?(b,D)*, defined as `AskBelongsTo?(b,D)=='definitely no'`. We will occasionally also use the function *ConsistentWith?(b,D)*, defined as **not** `InconsistentWith?(b,D)`.

Our goal is to provide an implementation skeleton for `Recognizes?` and `ConsistentWith?`, which works on a "structural" paradigm: membership in a concept is reduced to testing membership determined by various component constructors:

```
Recognizes?and(b,C)  ;; b: INDIVIDUAL, C: Normalized Concept
 {for every (stem) concept constructor K in C
       if not Recognizes?K(b,get-K(C)) then return false
  for every role restriction rr of C
        role := getRole(rr)
        fillers := getFillers(b,role)
        clsd? := getClosed(b,role)
        for every (branch) constructor K in rr
                ;; possibly need to iterate over entries, like for SOME
                if not Recognizes?K(b,get-K(rr),fillers,clsd?) then return false
  return true
 }
```

The following are two examples of typical recognition operations for constructors K=**at-most** and K=**all**:

```
Recognizes?atmost(b, max, fillers, clsd)
   { if clsd & card(fillers) <= max then true else false }

Recognizes?all(b, vr, fillers, clsd)
   {    if not clsd then false
        else
             for every f in fillers
               if not Recognizes?(f,vr)
                then return false
             return true
  }
```

As with concept subsumption, PROTODL tries to pre-compute and cache the description-membership information for each individual in the current state of the database. In the database $\{b:A,\ (b,b):p,\ closed(b,p)\}$, the following infinite series of assertions is known to be true b : **all**(p,A), b : **all**$(p,$**all**$(p,A))$, ... Therefore we cannot associate a single maximally complete description with every individual. Instead, we propose to pre-determine only positive membership in named concepts – to *classify* b; i.e., we pre-compute `Recognizes?(b,C)` for concepts C, and leave questions like `ConsistentWith?` (as well as `Recognizes?(b,D)`, for description D) to be answered "on-line".

3 Asserting only primitive membership

If we restrict ASSERT-MEMBER(b,A) to cases when A is a primitive concepts, then a collection of assert statements provides us with a partial model of the world, which means that no additional information about role fillers or atomic concept membership can be deduced, nor can any state of the database be inconsistent. For this reason, we study this simpler, restricted case first.

3.1 Classification

In order to support practical applications, PROTODL is designed to accept database information *incrementally* as a sequence of updates, rather than as a single monolithic "definition" of the individual. After

each operation, the individuals in the KB are re-classified with respect to the existing named concepts, and are re-checked for internal consistency.

3.1.1 Dependency Links

Clearly, the classification of individual b may change when new information is asserted about b. Unfortunately, the membership of an individual in a concept may depend on facts asserted about *other* individuals in the database, so that after updating x one might have to theoretically re-check the classification of *all* other individuals. PROTODL was designed to prevent needless re-testing while incurring relatively little "truth maintenance overhead". Following the spirit of the Classic implementation, we have adopted a "coarse-grained" TMS where one individual v may point to another, b, if the classification of the latter may change as a result of additional information being added on the former. For example, if {`C=(`**all** `p D), (b,e):p`} and neither `Recognizes?(D,e)` nor `Inconsistent?(D,e)`, then we add a link of the form: `e --DependsOnMe --> b`. Any update on e will cause b's classification to be re-done. (We do not keep track of the fact that only membership w.r.t. *C* may change. It is all redone.)

Let us consider the problem of setting and maintaining the dependency links. For this purpose, all `Recognize?K(b,D)` operations should return either "true" or a list L of individuals (possibly including b) whose modification might change the result of the membership test to True. The task of the main classification algorithm then includes dealing with dependencies:

- if during classification, `Recognizes?(b,*)` does not return true, but some *answ* instead, then invoke `setMemberDepends(answ-{b},b)`
- if b is re-classified or otherwise altered, then put the individuals depending on it on the re-process queue.

Here are some modified versions of Recognizes?K to deal with dependencies:

```
GENERAL FORMAT:

Recognizes?stemK(ind,KTerm)
  ;;: C-IND x TERM --> true U LIST(inds)
Recognizes?branchK(ind,KTerm,fillers,closed?)
  ;;: C-IND x TERM  x LIST(INDS) x BOOL --> true U LIST(inds)
  ;; Returns either true or a list of DEPENDEE individuals whose update requires
  ;; retesting of membership for ind.
  ;; Dependee = nil  --> Recognizes is not even unknown now (NB: but not converse!!!)
  ;; Dependee list may include ind (needed for nested testing)



Recognizes?atoms(b, a-list)
  {;; Dependee = b
        if subsetp(get-Atoms-ind(b), a-list) then true else (list b)
  }
Recognizes?oneof(b, i-list)
  {;; Dependee = nil
        if memberp(b , i-list) then true else nil
  }
Recognizes?atleast( b,k, fillers, clsd)
  {;; Dependee = Nil U b
     if  card(fillers) >= k then return true
     else if clsd?
          then return nil ;; there is no way that b could be
                          ;; incremented to make it satisfy the atleast
          else return (list b)
  }
Recognizes?atmost(b, k, fillers, clsd)
  {;; Dependee = Nil U b
        if clsd  then if card(fillers) <= k then return true
                      else return nil
        else return (list b)
```

```
  }
Recognizes?all(b, D, fillers, clsd)
  {;; Dependee = b U <one returned nested list>
        if not clsd
        then return (list b) ;; b itself must change (become
                             ;; closed on this role)
        else
          for every f in fillers
             answ := Recognizes?(f,D)
             if notTrue(answ) then return  answ
           return true
  }
Recognizes?some(b, D, fillers, clsd)
  {;; Dependee: nil  U  ind  U   Flatten(LIST(<nested list>)
  ;; Every filler may potentially satisfy the condition so all need to be dependees
        deps := if clsd then nil ;; additions to b could not change recognizes-SOME
                else (list b)
        for every f in fillers
                answ:= Recognizes?(f,D)
                if notTrue(answ) then append answ to deps
                else return True
        return deps
  }
Recognizes?and(b,cnc) ;;: C-IND x CNC --> True U LIST(inds)
  {;; Dependee: <one nested answer> -- for conjunction, all of the deps must hold before
  ;;  the ind needs to be re-examined. So return any one
  for every  K in cnc
       {answ := Recognizes?K(b,get-K(cnc))
        if notTrue(answ)  then return answ }
  for every rr of cnc
        role := getRole(rr)
        fillers := getFillers(b,role)
        clsd? := getClosed(b,role)
        for every K of rr
               {answ := Recognizes?K(b,getK(rr),fillers,clsd?)
                if notTrue(answ) then return answ}
  return true
  }
```

So if

```
     C=(some p (some q (all r D))) and

       b --p-- a1 --q-- e1 --r-- f1
                       e1 --r-- f2
                a1 --q-- e2 --r-- f3
       b --p-- a2 --q-- e3
```

with all roles closed except on e3, and no class membership information, then after the question whether b is to be classified under C, the dependencies would be from (f1 *or* f2) *and* f3 to b, *and* e3 to b.

3.1.2 Classification based on update type

As we have seen, the re-classification of an individual b is triggered by one of the update operations: filling a role, closing a role, or asserting membership in a description. Can we gain some advantage by remembering which of these has actually been done, rather than blindly invoking `recognizes?AND`? In other words, can we gain something by having `Recognizes?afterFill(b,p,v, D) Recognizes?afterClose(b,p,v,D), Recognizes?afterAssert(b,A,D)` ?

Unfortunately, if `Recognizes?AND(b, (and D1 ... Dn))` was false before the update, then all we can assume is that for *some* j `Recognizes?K(b,Dj)` failed, but we do not record for which K and j it was

false. Therefore, we cannot make any assumptions when re-running `Recognizes?AND(b, (and D1 ... Dn))`, so we must start over from the beginning.

We do know that dependency links are set from *other* individuals that may be missing some properties, so the parts of the recognition test that depend only on this individual's properties (e.g., one-of, primitives, at-least, at-most) need not be repeated, if we arranged to test for them *before* the others (such as all-restrictions). We could therefore propose a smaller version of `Recognizes?and`, named *Recheck-Recognizes?and*, where `Recheck-Recognizes?and` is assembled from a subset of the calls made by `Recognizes?and`. This demonstrates the utility of our software decomposition. Unfortunately, we need to be careful: RecheckRecognizes?and(b,C) can be run only for the concept C which originally caused a dependency link to be set, pointing back to b. Therefore, such dependencies would need to record the fact that it was the membership test *in C* that should be rechecked. In keeping with our philosophy of not maintaining very detailed dependency information, we avoid this route here, although the same idea will be picked up later, in a different context.

3.2 Checking inconsistency

Although currently we do not record non-membership links, in the next section we will need to detect the inconsistency of individuals with their descriptions. Therefore we will take a look, even in this limited context, at the incremental determination of non-membership/inconsistency. The same kind of considerations apply as in the case of `Recognizes?`. But in anticipation of its future use, we will have `InconsistentW?` indicate true by signaling an exception, rather than returning true; therefore, when it returns control normally, it means that `InconsistentW` returns false, but it actually needs to return a list of individuals whose changed properties may cause the inconsistency to turn true in the future.

3.2.1 Dependency links

Observe that in the database $\{(b,e) : p, (b,f) : p\}$, if neither e nor f are inconsistent with `D`, then b is not yet known to be inconsistent with `C` $\equiv$ `(all p D)`, but we would need to have dependencies pointing back from *both* e and f to b, because a change to either may give us enough information to deduce non-membership in `C`. In contrast, we would only need a dependency from either one of them to b, if all we wanted is to be alerted when b becomes a member of `C`. For this reason, we will distinguish two different relationships: `x --member-dependson-me--> y` and `x --inconsist-dependson-me--> y`. In the above case, we add membership dependencies from e or f to b, and inconsistency dependencies from both e and f to b.

Therefore `InconsistentW?` signals an exception only if it is sure to be inconsistent; otherwise, it returns a list of objects (possibly including itself) on which inconsistency may depend, based on future updates. If the list is nil, the individual is sure to be consistent, but not vice-versa.

```
InconsistentW?and(b,cnc) ;;: C-IND x CNC -> LIST(LIST(<ret'd deps>)) + signals INCONS
                        ;; propagates nested  INCONS signal or returns dependees
                        ;; (including possibly b) whose change may lead to
                        ;; inconsistency in the future (hence, if nil,then
                        ;; b is surely consistent with D
 ;; DEPENDEES: a collection of many lists
 {deps := nil;
  for every constructor K in cnc
               deps+= InconsistentW?K(b,get-K(cnc))
  for every rr of D
        role := getRole(rr)
        fillers := getFillers(b,role)
        clsd? := getClosed(b,role)
        for every constructor  K in rr
               deps += InconsistentW?K(b,get-K(cnc,fillers,clsd?)
  return deps
  }
```

The following are two example functions for detecting and signalling inconsistency as an exception, and returning dependencies otherwise.

```
InconsistentW?all(b,D, fillers, clsd) ;; Dependees: (Nil U b) +  FLAT-
TEN(LIST(<returned deps>))
```

```
 { deps := if clsd then nil else (list b)
   for every f in fillers
        deps += InconsistentW?(f,D)
   return deps
 }

InconsistentW?some(b,D, fillers, clsd)
  {;; Dependees:  b U  Nil U <returned deps>
    if not clsd then return (list b)
    else
        for every f in fillers
          (catch INCONS
                {answ:= InconsistentW?(f,D)
                  return answ})
           )
   signal INCONS ;; because  all tries in the loop above signalled INCONS
  }
```

3.2.2 Inconsistency checking based on update type

If, before an update, b was not yet a known non-member of (**and** D1 ... Dn) (i.e., if InconsistentW?and(b, (**and** D1 ... Dn)) was false), then InconsistentW?K(b, Dj) must have been false for *every* j. When re-considering the non-membership of b after the update, we can use this as an assumption in order to decrease the testing needed. In particular, we can consider three variants of InconsistentW?: *InconsistentWFilling?(b,p,v, D), InconsistentWClosing?(b,p,D)* and *InconsistentWAssert?(b,A,D)* .

InconsistentWFilling?and(b,p,v,D) usually makes calls to InconsistentWFilling?K(b,p,v,KD) for K-components KD of D related to role p, since consistency with most stem constructors, and role constructors on other roles are not affected. (But see the **same-as** constructor in the last section.)

InconsistentWClosing?(b,p,D) is similar.

InconsistentWAssert?(b,A,D) in this case is trivial since we can only assert atomic descriptions. (If we had disjoint primitives, it might have to deal with them here.)

For example,

```
InconsistentWFilling?and(b,p,newf,D) ;;: C-IND  x ROLE x IND x CNC -> Dependees
                                    ;;  + signals INCONS
  ;; Dependees in this case are only the *additional* ones due to new filler
  ;; DEPS: LIST(LIST(<returned deps>))
  ;; ASSUMES:  newf already added to b.p fillers
{       deps := nil
        rr = getRR(p,D)
        fillers := getFillers(b,p)
        deps += InconsistentWFilling?atmost(b,newf,fillers, get-Atmost(rr))
        deps += InconsistentWFilling?all(b,newf,fillers, get-All(rr))
        return deps
 }

InconsistentWFilling?atmost(b,newf,fillers,k) ;;for efficiency,
                ;;it is occasionally useful to pass in additional information
                ;; like the set of previous fillers
  {
        if count(fillers) > k then signal INCONS
        else {b} /* can't be closed since it is being filled */
  }

InconsistentWFilling?all(b,newf,fillers,vr)
  {;; Deps: <returned deps>
        deps := InconsistentW?and(newf,vr)
        return deps
```

```
  }

 ;; ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

InconsistentWClosing?and(b,p,D) ;;: C-IND x role x CNC ->
                                ;;        LIST(<returned deps>)) + signal INCONS
  {     ;; Dependees are only the *additional* ones due to this closing
        ;; Deps:  LIST(<returned deps>)
        deps := nil
        rr = getRR(p,D)
        fillers := getfillers(b,p)
        deps += InconsistentWClosing?atleast(b,fillers,get-Atleast(rr))
        for every vr in get-Some(rr)
                        deps += InconsistentWClosing?some(b,fillers,vr)
        return deps
  }
InconsistentWClosing?atleast(b,fillers,k)
  ;; Deps: b
        if count(fillers) < k then signal INCONS
        else return {b};

InconsistentWClosing?some(b,fillers,vr)
  {;; Deps: <returned deps>
        for f in fillers
                (catch INCONS
                        {deps := InconsistentW?(f,vr)
                         return deps ;;if deps/=nil, we could be more
                                ;; careful,and see if it is surely
                                ;;consistent with some other filler, but it
                        }       ;; is not worth it
                )
        signal INCONS ;; only if every iteration also signalled it
  }
```

The full `InconsistentW?K` procedures mentioned earlier can be recovered by simulating the addition of all the fillers (and closing).

```
InconsistentW?atmost(b,fillers,clsd,k)
   {  if clsd then InconsistentWClosing?atmost(b, k, fillers)
     else return {b}
   }

InconsistentW?all(b,fillers,clsd,D)
 { deps := if clsd then nil else (list b)
   for every f in fillers
        interim-deps:=InconsistentWFilling(b,D,f,fillers-{f})
        append interim-deps to deps
   return deps
 }

InconsistentW?atleast(b, fillers, clsd, k)
   { if clsd then InconsistentWClosing?atleast(b, fillers, k)
     else return {b}
   }

InconsistentW?some(b, fillers, clsd, D)
   { if not clsd then return (list b)
    else
        InconsistentWClosing(b,fillers,D)
```

}

As for recognition, when returning along a dependency link to check the consistency of some individual, we can eliminate them, since they will be re-introduced if needed.

4 Asserting membership in general descriptions

Suppose we now allow ASSERT-MEMBER to deal not just with atomic/primitive concepts but also general, composite ones; in particular, we allow $b : D$ for arbitrary descriptions D. Such assertions will be called "intensional" facts, in distinction with the earlier, "extensional" facts. For convenience, we will gather all assertions of the form $b : D_i$ into a single assertion $b :$ (**and** D_1...), and we will denote the description (**and** D_1...) by $descr(b)$.

In this case, the KB is no longer a partial interpretation but a theory, which could have zero or more models, and we must consider several new aspects:

1. If the resulting theory does not have any models, we need to signal the user and reject the update that has lead to the inconsistency. This can arise because the KB explicitly asserts that v belongs to a description D for which `inconsistentW(b,D)` returns true.
   e.g., KB0={ b:(**at-least** 1 p) , closed(b,p) }
2. In order to make sure that our questions concerning fillers and atomic membership are correctly answered, we need to see if any additional "extensional" facts may be deduced. This would happen because such facts would be true in all the possible models of the KB theory. For example
   { b:(**all** p C) , (b,f):p } $\models$ f:C
3. There are situations which cannot be translated into extensional facts in the KB. For example, from { b:(**some** p C) , (b,f):p, (b,e):p, closed(b,p) } one cannot deduce either `f:C` nor `e:C`, though one of them must be true. For this reason, during recognition we need to take into account `descr(b)` as well.

We address briefly each of these issues in the following subsections.

4.1 Consistency checking with `descr(b)`

For (1), we must add a consistency check to every update. To begin with, whenever the intensional description of an individual is changed, we need to make sure that `descr(b)` is not **nothing**. The way to do this is to normalize `descr(b)` using the standard description procedures (see [3]). Second, we need to invoke `InconsistentW?(b, descr(b))` as part of an update, in order to check that no contradiction arises. As before, we can specialize the kind of procedure to call depending on the update, having `InconsistentW?filling, InconsistentW?closing, InconsistentW?asserting`.

Moreover, we can use a procedure `RecheckConsistentW?`, which assumes that it was called as a result of a dependency trigger having been set off. By arranging that consistency tests with respect to $descr(b)$ be ordered so that ones that cannot set dependencies (e.g., tests for **at-least**, **at-most**,...) appear before the ones that can set dependencies, then the former can be eliminated in `RecheckConsistentW?`.

```
RecheckConsistentW?AND(b)
 {deps := nil;
  ;; no need to check consistency with prims, oneof -- tests yes, because
    ;; they can depend on properties of fillers.
  for every rr of descr(b)
        role := getRole(rr)
        fillers := getFillers(b,role)
        clsd? := getClosed(b,role)
        ;; no need to check consistency with atmost,atleast,
        deps += InconsistentW?all(b,get-All(cnc),fillers,clsd?)
        for every D in get-Some(cnc) do
                  deps += InconsistentW?some(b,D,fillers,clsd?)
dd  return deps
```

4.2 Inferring new facts

For (2), we need to develop procedures, `InferFromAND(b,D)`, for finding the implications of having some description; this calls various procedures, `InferFrom-K(b,term)`, to infer from component descriptions. In order for this to work properly, normalization of descriptions should have the additional property that `InferFrom-AND(b, (and C1 C2 ...))` should have the same effect as `InferFrom-AND(b, C1))`; `InferFrom-AND(b, C2))`; .... The result of inferences are additional tell operations, which are posted on a queue, Infer-Q. (We use ops Infer-assert, Infer-fills, Infer-closed to put things on the queue.)

Because of the above additivity, we can also break up `InferFrom-K` into calls to `InferFrom-filling-K` and `InferFrom-closing-K`, which are independently needed for incremental updates.

```
InferFrom-filling-AND (b,p,f,D) ;;: C-IND x ROLE x IND x CNC
   ;; EFFECT: posts other operations on the infer-q
   ;; ASSUMES: f has already been added to  b.p
   rr := get-Rr(D,p)
   InferFrom-filling-ALL(b,p,f,get-All(rr))
   InferFrom-filling-ATMOST(b,p,get-Fillers(b,p),get-Atmost(rr))

InferFrom-filling-ALL (b,p,f,vr)
        post-assert(f,vr)

InferFrom-filling-ATMOST (b,p,fillers,k)
   ;; remember, the new filler is in b.p already so we want *all* fillers
       if  card(fillers) = k
        then post-closed(b,p) ;;?Should we just call the tell-closed procedure here?


InferFrom-closing-AND (b,p,D);; C-IND x ROLE x IND x CNC
   ;; EFFECT: posts inferences
   ;;ASSUMES: b.p has already been marked as closed ??
   rr := get-Rr(D,p)
   InferFrom-closing-SOME(b,p,get-Some(rr))

InferFrom-closing-SOME(b,p,VR);; C-IND x ROLE x LIST(NC)
  {if all but one filler f is known to be inconsistent with a VR
      and not recognizes?(f,VR)
    then post-assert(f,VR)
   }

InferFrom-asserting-AND(b,CNC)
   for every rr of CNC
        role := get-Role(rr)
        fillers := get-Fillers(b,role)
        clsd := get-Closed(b,role)
        InferFrom-asserting-All(b,fillers,clsd,get-All(rr))
        InferFrom-asserting-Atmost(b,fillers,clsd,get-Atmost(rr))
        for every vr in get-Some(rr)
            InferFrom-asserting-SOME(b,fillers,clsd,vr)
  }
InferFrom-asserting-ALL(b,fillers,clsd,vr)
    for f in fillers do
        InferFrom-filling-ALL(b,p,f,vr)

InferFrom-asserting-ATMOST(b,fillers,clsd,k)
   InferFrom-filling-ATMOST(b,p,fillers,k)

InferFrom-asserting-SOME(b,fillers,clsd,vr)
   if clsd then InferFrom-closing-SOME(b,p,fillers,vr)
```

4.3 Using descr(b) in recognition.

Finally, in order to use descriptions in recognition, at the very least the first step in Recognizes?K(b,term) should be a call to Subsumes?K(descr(b),term). (This could be put as part of the Recognizes?and algorithm.)

```
Recognizes?and(b,conc) ;;: C-IND x C-ND -> IND-list
  ;; Dependee: <nested answer>
        b-desc := get-Descr(b)

        unless subsume-atom(get-Atoms-cnc(conc),get-Atoms-cnc(b-desc))
           return {b} ;;it might get more atoms

  for every rr of conc
        role := getRole(rr)
        fillers := getFillers(b,role)
        clsd? := getClosed(b,role)
        ind-rr := get-Rr(b-desc)

        unless subsume-atleast(get-Atleast(rr),ind-rr)
                answ:= recognizes?atleast(ind-rr,get-Atleast(rr),fillers,clsd)
                if notTrue(answ) then return answ
        unless subsume-atmost(get-Atmost(rr),ind-rr)
                answ:= recognizes?atmost(ind-rr,get-Atmost(rr),fillers,clsd)
                if notTrue(answ) then return answ
        unless subsume-all(get-All(rr),ind-rr)
                answ:= recognizes?all(ind-rr,get-All(rr),fillers,clsd)
                if notTrue(answ) then return answ
        for every vr in get-Some(rr)
                unless subsume-some(vr,ind-rr)
                        answ:= recognizes?some(ind-rr,vr,fillers,clsd)
                        if notTrue(answ) then return answ
  return true
```

5 The top level

We will use 4 global data structures to keep track of work to be done yet. Although there is no particular reason to treat them as queues, we will add the suffix "Q" to their names:

DependsOnQ — groups of dependencies will be added to this Q; for optimization, these could be broken apart, and duplicates eliminated, before values are removed from it.

ClassifyQ — list of individuals that may have to be reclassified; duplicates can also be eliminated.

Consistent?Q — list of individuals whose consistency with their description needs to be rechecked.

InferQ — list of update operations to be performed on individuals.

Classification is carried out by a procedure IND-CLASSIFY, whose actions are relatively similar to those of concept classification. The following is just the most straightforward implementation:

```
IND-CLASSIFY(ind) ;;: IND -> NoChange U LIST(ind)
  ;; ASSUMES: ind or one of its role fillers has been modified or reclassified
  ;; EFFECT: the answer is 'NoChange iff there is no change in classification;
  ;; otherwise, the SET (no duplics) of inds whose modification might cause further
  ;;  reclassification, is returned as dependees.
  dependees:=nil
  unchanged := true
  loop through sibling concepts C of ind's parents <which may change inside loop!>
        answ := recognizes?(ind,C)
        if isTrue(answ) then unchanged := false  &  update parents(ind)
        else dependees += answ
  if unchanged then return 'NoChange else return   removeDuplicates(dependees)
```

Most importantly, we need to show the implementations of the three KBMS operations: ASSERT-FILL, ASSERT-CLOSED and ASSERT-MEMBER. The top-level procedures for these are responsible for orchestrating the calls to other procedures and especially for managing the queues that are being maintained. So-called "inner-versions" perform the more limited task of reasoning about just one individual.

```
ASSERT-FILL(ind, role, filler) ; membership and dependency links reset
  IF redundant THEN signal REDUND-EXN
  add-filler-ind(ind,role,filler) ;  put in the filler;
  DependsOnQ := nil
  ClassifyQ := (list ind) U getMemberDependsOnMe(ind)
  Infer-Q := nil
  Consistent?Q := nil
  ind-descr := get-Descriptor(ind)
  ;;check consistency of ind and post other checks
           answ:= i-consistent-with-filling-AND(ind,  ind-descr, role, filler)
                [INCONS => ...]
           DependsOnQ += ('CONSIST, ind, answ)
           Consistent?Q += getConsistentDependsOnMe(ind)
   ;;post other inferences
         InferFrom-filling-AND(ind,role,filler,ind-descr)
   ;;perform inferences, which will be INNER-TELLS
          while nonEmpty(inferQ) do
                 perform-next(inferQ)  [INCONS ==> ...]
   ;; reclassify ind and other modified objects
          while y:= getNext(ClassifyQ) do
                ;; (getNext skips a value if it also appears later in Q)
                answ := IND-CLASSIFY(y)
                if changed?(answ)
                        then {DependsOnQ += ('MEMB,  y, (answ/dependees - {y})
                              ClassifyQ += getMembDependsOnMe(y)
                              Consistent?Q += getConsDependsOnMe(y)
                              }
   ;; check consistency of other objects
       WHILE z:= getNext(Consistent?Q) DO
          {answ =RecheckConsistentW?AND(z) ;;Consistent?Q only gets things
                                           ;;from dependency links!
           DependsOnQ+= ('CONSIST, z,answ)
          }

  Install(DependsOnQ) ;; this function regroups the lists, reduces redundancy,..
}


INNER-TELL-FILLER(ind,role,filler) ;;C-IND x ROLE x IND ->may raise INCONS
  ;; Like ASSERT-FILL, except that it does not worry about processing
  ;; the queues. It may post new things to the queues in fact.
  ;; ACCESSES GLOBAL: DependsOnQ/write, ClassifyQ/write, InferQ/write
  IF redundant THEN signal REDUND-EXN
  add-filler-ind(ind,role,filler) ;  put in the filler;
  ind-descr := get-Descr(ind)
  ;;check consistency of ind and post other checks
           answ:= consistent-with-filling-AND(ind, role, filler, ind-descr)
                [INCONS => ...]
           DependsOnQ += ('CONSIST,ind, answ)
           Consistent?Q += getConsistentDependsOnMe(ind)
   ;;post other inferences
         InferFrom-filling-AND(ind,role,filler,ind-descr)
  ClassifyQ += (list ind) U getMemberDependsOnMe(ind)
```

The pairs of procedures `assert-closed, inner-tell-closed` and `assert-member, inner-tell-member` can also be easily described.

6 Adding a new constructor: SAME-AS

Let us consider the extensions needed for a rather complex new concept constructor – **sameas**, which is considered to be a stem constructor. The semantics of (**same-as** (f1 ... fn) (g1 ... gm)) is that it denotes individuals for which the two attribute chains f1...fn and g1...gm have the same known filler.

To begin with, we need a recognition procedure for this constructor. This returns as membership dependencies one or two individuals along the two attribute paths that were in the way of finding the final values.

```
Recognizes?sameas(b, eqlty)
  {;; Dependee = Nil U <last elt on an incomplete path from b>
    e1 := follow path (first eqlty) from b
    e2 := follow path (second eqlty) from b
    if one of the paths does not end,
          then return (list last-individual-on-path)
    else if e1=e2 then true else nil
  }
```

This will be called from inside `Recognizes?and` as follows:

```
      for every eqlt in get-Sameas(conc)
              unless subsume-sameas(eqlt,b-desc)
                {answ := recognizes?sameas(b,eqlt)
                if notTrue(answ) then return answ}
```

Once we allow arbitrary assertions to be associated with b in descr(b), we need to take these into account if we are to maintain complete reasoning. For example,

{ (b,f):p, (b,f):q, f:(**same-as** r s) } $\models$ b:(**same-as** [p r] [q s])

Similar problems will arise due to the presence of constructions of the form (**all** f_1(**all** ...(**fills** f_n e))). Since these may in general require combinatorial checking, we will omit it in this case, and treat it as another example of incomplete reasoning – one of the *raison d'etre* of PROTODL :-)

In order to check consistency, we need to consider inconsistencies that may arise because of the three kinds of updates. Because **sameas** involves individuals that are not role fillers for this specific object, an inconsistency checking procedure cannot be easily assembled by simulating updates on one object, so we start with a general inconsistency checking procedure:

```
InconsistentW?sameas(b, eqlty) ;;: Dependees: Nil U <ind at end of an incomplete path>
  {  e1 := follow path (first eqlty) from b
    e2 := follow path (second eqlty) from b
    if either one of the paths does not end,
          then return {last individual on ONE path}
    if e1==e2 then  return nil else signal INCONS
  }
```

Since the roles in an equality are attributes, these get closed automatically when a filler is provided, so there is usually no explicit closing of the role that can affect **same-as.** (If we allowed partial functions for attributes, there might be something to do.) So the only case to consider is `InconsistentWFilling?sameas`, which will have to be used in `InconsistentWFilling?and(b,p,.,.)` for every role p:

```
InconsistentWFilling?sameas(b,p,f,eqlty)
 {if p=(car (first eqlty)) | p = (car (second eqlty))
  then InconsistentW?sameas(b,eqlty)
```

For `RecheckConsistentW?and`, we need to include a call to `InconsistentW?sameas` since this can set consistency dependencies.

Equalities can lead to inferences when one of the paths is completely known, and all but the last attribute on the second path is known.

{ a:sameas([p1 p2 p3], [q1]), (a,b):p1, (b,c):p2, (a,d):q1 } $\models$ (c,d):q3

This inference is captured in

```
InferFrom-asserting-SAMEAS(b, eql)
  {  e1 := follow path (first eqlty) from b
   e2 := follow path (second eqlty) from b
   if exactly one of the paths (say the second one) does not end,
      and it is only missing the last attribute p
   then post-fills(e2,p,e1)
  }
```

Furthermore, in order to activate this procedure, it seems we will need to introduce inference dependencies, since the inferenences for the other constructors are triggered by the immediate fillers of an object's role, but this one may be triggered by the filler of some individual further along the path.

6.1 Performance and alternatives

The above is a prime example of a situation where a performance penalty is paid for separating out consistency checking, recognition and inference: each of these procedures attempts to traverse the individuals along the two paths of the equality in an effort to reach the ends.

If there are relatively few individuals with **same-as** conditions attached, or if the paths are usually only one or two attributes long (the usual case for uses of **same-as** for parameter binding), this price is probably negligeable.

If there are many such individuals, and the paths in the equalities are long, then we can introduce caching to speed up the code: associate with every equality eqlty and individual b two pairs (e1,r1) and (e2,r2) representing the ends e_i reached on each path, and the remainder of the paths r_i left to be traversed. (So if the path is completely known, $r = nil$.)

A final alternative is to add the code for InferFromSameas into the `InconsistentW?sameas` procedure, so the latter performs all the deductions involving **same-as**.

Note also that if b's p2 filler was not yet known, then we would have wanted a infer-dependency pointing from b to a. Similarly, if d was not yet a known filler for a's q1 role, then `InferFrom-filling-SAMEAS` should check for the inference, when the appropriate attribute is filled.

```
InferFrom-filling-SAMEAS (b,p,f,eql)
  {
   if p = first(first(eql)) | p=first(second(eql))
    then do the same as InferFrom-asserting-SAMEAS but with path
         computed taking into account b--p-->f.
  }
```

which would be called from `Inferfromfilling-AND`.

7 Summary

In the Krypton system [6], reasoning about individuals was carried out using a first order theorem prover, which made calls to the subsumption reasoner of a DL in order to take into account the semantics of predicates. In contrast, DL-KBMS build special services for reasoning with individuals. With a sufficiently expressive DL, it is possible to convert all information about an individual into a description and then inferences about class membership become just subsumption tests.

Some DL-KBMS, including CLASSIC and PROTODL, have chosen not to reduce individual reasoning to subsumption reasoning. This is based on several observations. First, with a weaker DL, it is not always possible [8] to find a "most specific description" that can be used for the classification inference. Second, it is an empirical observation that in our applications the number of individuals and role-fillers is orders of magnitude larger than the number of distinct concepts/descriptions appearing in the KB. Among others, most individuals do not have associated incomplete information, so that reasoning can be carried out mostly as model checking (as in standard data base processing), which is much faster than subsumption reasoning. Also, even in terms of space, it is much more effective to store the individuals with pointers to role fillers and super-concepts, rather than creating a separate normalized description for each individual.

The third important assumption of this work is that although concept definitions might be relatively stable in a KB, the information about individuals accumulates incrementally over time, so that a significant part of the job is to reduce the amount of work needed to *re-check* the inferences that might

be applicable after an update. (As an analogy, suppose that concept definitions might evolve over time; then, in addition to techniques for efficiently classifying concepts [1], a relevant research question would be to consider concept *re-classification*).

The approach presented here, and implemented in Common Lisp, is based on the "structural subsumption" paradigm [7], where a description D is normalized to a conjunction of "independent" descriptions D_j, each built with one of the primitive concept constructors K. Then reasoning is carried out by special procedures for each kind of constructor K, under the "direction" of procedures for the constructor **and**.

We have identified three categories of procedures/inferences: those dealing with (1) recognizing membership in description, (2) consistency with descriptions, and (3) inferring new fillers or information about other individuals.

- `Recognizes?K`(b,D) returns "true" iff b is a known instance of description D, built with constructor K; otherwise, it returns a list of individuals, such that changes to any of them trigger a re-examination of the question whether b is an instance of D. This function is normally used to (re)classify affected individuals after updates.
- `InconsistentW?K`(b,D) signals exception INCONS iff b cannot be an instance of description D in the current KB; it is normally used after every update to verify that an individual b is still consistent with its associated description $descr(b)$. Inconsistency checking also returns a list of dependencies — objects whose modification may result in an inconsistency about the present object being detected.

  It is possible to specialize (in)consistency rechecking based on the specific primitive update that has taken place — `consistentWfilling?K, consistentWclosing?K, consistentWAsserting?K` — which is quite useful since each of these usually perform only a subset of the actions in `InconsistentW?K`.
- Similarly, for every constructor we require procedures `InferFrom-K, InferFromFilling-K, InferFromClosing-K` and possibly `InferFromAsserting-K`. For simplicity, these procedures simply post additional ASSERT operations to a blackboard, whenever new facts to be inferred are discovered.

When an update occurs on an individual, the above three kinds of procedures are run on it. Just as importantly, dependency links from the updated individual to other individuals are followed (and removed, though possibly only temporarily), so that we have a queue of individuals that need to be *re-examined* for either membership, consistency, or inferences[2]. The removal and "treatment" of individuals in these queues is one primary task of the top-level ASSERT operations on individuals. A second task is putting in the dependency links whenever membership or consistency test results are inconclusive.

We have also observed that it is often possible to order the consistency tests for the various constructor in such a way that those that cannot set dependencies are done first, so that when *re*-checking consistency triggered by a dependency link, these tests can be avoided. This attests to the utility of our modular decomposition.

We note that there are two important features that are present in the CLASSIC system but are currently missing from PROTODL: (i) supporting *retraction* of told facts, and (ii) supporting forward-chaining rules. These have been omitted mostly because we do not have any new insights to add to this aspect of the reasoning, and did not find the time to implement them. Readers interested in experimenting with extended DLs are invited to contact the first author to obtain a copy of the PROTODL implementation described in this paper and in [3].

Acknowledgment

Our ideas for the architecture of PROTODL's implementation are evidently influenced by the implementation of the CLASSIC system at AT&T Bell Laboratories. We are indebted to the members of the CLASSIC group, especially Peter Patel Schneider and Lori Alperin Resnick, as well as to Ronald Brachman, with whom we started this whole adventure.

This research was supported in part by funds from the National Science Foundation, under grant NSF-IRI 91-19310.

[2] Until now, we have not found sufficient use for inference queues, and in the one case where it was needed – the **same-as** constructor — we have overloaded consistency checking to also take care of inferences.

References

[1] Borgida, A. "Towards the systematic development of terminological reasoners: CLASP reconstructed", *Proc. Conf. on Principles of Knowledge Representation (KR'92)*, Boston, MA, October 1992.

[2] A. Borgida, "Description Logics are not Just for the `Flightless Birds`", Technical Report DCS-TR-295, June 1992, Dept. of Computer Science, Rutgers University.

[3] Borgida, A., and Brachman, R., " Customizable Classification Inference in the ProtoDL Description Management System", *Proc. Conf. Information and Knowledge Management*, Baltimore, MD, November 1992, pp.482–490.

[4] Borgida, A., Brachman, R. J., McGuinness, D. L., and Resnick, L. A. "CLASSIC: A Structural Data Model for Objects," *Proc. 1989 ACM SIGMOD International Conference on Management of Data*, June, 1989, pp. 59–67.

[5] A. Borgida, and P. Devanbu, "Knowledge Base Management Systems using Description Logics and their role in Software Information Systems"*Information Processing 92 (Vol.3)*, pp.171–181, Elsevier Science Publishers, 1992.

[6] Brachman, R. J., Fikes, R. E., and Levesque, H. J., "Krypton: A functional approach to knowledge representation", *IEEE Computer*, *Special Issue on Knowledge Representation* **16** (10) (1983) 67–73.

[7] Cohen, W., Borgida, A., and Hirsh H., "Computing Least Common Subsumers in Description Logics", *Proc. of AAAI'92*, San Jose, CA., May 1992.

[8] Donini, F.M. and A. Era, "Most specific concepts for knowledge bases with incomplete information", *Proc. CIKM-92*, Baltimor, MD, pp.545-551, November 1992.

[9] Levesque, H. "Foundations of a Functional Approach to Knowledge Representation", *Artificial Intelligence 23*(2), 1984, pp. 155–212.