KIT REPORT 105

Retraction of Object Descriptions in BACK

Carsten Kindermann

TECHNISCHE UNIVERSITÄT BERLIN
PROJEKT KIT–BACK, FR 5–12
FRANKLINSTR. 28/29
D–1000 BERLIN 10

cmk@cs.tu-berlin.de

December 1992

Abstract

Terminological representation systems permit the construction of knowledge bases and schemata around the notion of concepts, roles, and instantiating objects. Inferential services they provide include checking for inconsistencies in and classification of concept and object descriptions. To improve their performance, several systems store derived propositions together with user-told data in the knowledge base. In this report we address the problem of retraction of object descriptions for systems that employ such a generative implementation. We adopt a data dependency network approach.

Contents

1 Introduction **1**

2 Retraction in Terminological Representation Systems **2**

3 Processing Retractions of Object Descriptions **6**

3.1 Minimal and Maximal Approaches 6

3.2 A Naive Approach 7

3.3 A Semi-Naive Approach Based on Object Dependencies 8

3.4 Labeled Dependency Links 10

3.5 Recomputing the System Normal Form 13

3.6 Further Improvements 15

4 Dependency Relation Maintenance **16**

4.1 Making Dependency Relation Entries 16

4.2 Deleting Dependency Relation Entries 19

5 Discussion of the Algorithm **21**

6 Implementation in BACK **27**

6.1 Data Structures 27

6.2 Dependency Module 27

6.3 Organizing the Assertional Reasoning 28

6.4 Extended Usage of the Approach 29

7 Related Work **29**

8 Summary **31**

A Semantics **35**

B Role Completion Rules **37**

1 Introduction

The terminological knowledge representation approach based on the ideas of KL-ONE [BS85] has led to the development of a number of systems that support so called terminological logics (also concept languages or term subsumption languages). In these systems, the knowledge base (KB) schema contains the intensional knowledge, and is made up of class descriptions (called concepts) related to each other by binary relations (called roles). Extensional knowledge, on the other hand, is contained in the KB that maintains instances of classes, related to other instances by expressing filler relations for roles. Typical inferences in terminological systems are checking for inconsistencies in concept and object descriptions, and automatic classification, i.e., determining whether one concept $C1$ subsumes another concept $C2$ (written $C2 \sqsubseteq C1$), or whether an object x instantiates a concept (written $x \in C$). Descriptions of eight terminological systems are contained in [Ric91].

There is a growing interest in applying such systems for various purposes, such as information systems, problem solving tools (e.g. for configuration tasks), or object management components in larger software systems (see [PvLK91] for an overview of current activities). One of the key factors in an application environment is acceptable performance, and implemented terminological systems make various attempts to meet this requirement. An optimization technique employed by most systems, e.g., CLASSIC, LOOM, or BACK, is to permanently store in the knowledge base together with user-given data those facts that are derived by the system. The optimization consists in avoiding frequent recomputation of the same facts. On the other hand, this raises the problem of maintaining the derived and stored facts in case the knowledge base is changed. Especially when user-given facts are retracted, the cached inference results must be checked, and in case they are no longer supported by the explicitly given knowledge, must be removed. This is the subject addressed in this report: We are looking for a mechanism to handle changes of a knowledge base in the presence of stored inference results.

The problem of change comes at various levels:

- changes of the knowledge base content
- changes of the knowledge base schema
- enforcing changes of the schema on the knowledge base content

Nebel has analyzed in depth the problem of changing the schema of a knowledge base [Neb90]. Changes of the knowledge base, as far as they are monotonic, have also been addressed before: most terminological systems allow for an incremental construction of the knowledge base; since classical inferences in terminological

logics are monotonic, no previously derived facts are invalidated when new facts are added. While treatment of monotonic changes is described in many papers and reports on implementational aspects of terminological systems (e.g., [QK90] for BACK), no such description is found on non-monotonic changes. Finally, the problem of enforcing schema changes onto the knowledge base has to be solved; for our project a solution is proposed in [Tho92] that basically integrates Nebel's ideas on schema modification with the work presented here.

The development of a solution for the retractability problem is guided by several aspects: First, we assume an application scenario in which a terminological representation system is used as a knowledge base system, and the knowledge base is more often queried than changed (as opposed to a scenario where the system acts as a problem solving component managing a dynamically changing problem space). The solution we are looking for should suit this scenario as well as the specific characteristics of terminological logics—we return to this point when we discuss applicability of standard truth-maintenance systems. For pragmatic reasons we are interested in an approach that easily combines with an existing system (BACK in our case); it should neither lead to a major revision of the way monotonic changes—the regular working mode—are processed, nor should it impose a significant change to the already employed data structures. More specifically, we want to separate data just concerned with retractability from the real knowledge, in order to be able to provide applications with fixed knowledge bases that are not messed up with superfluous maintenance data.

The report is organized as follows:[1] First, we give a brief introduction to terminological logics, and describe the problem of retraction more precisely. In Section 3, we analyze different alternatives for an implementation, and then propose a dedicated data dependency network approach. Section 4 describes the maintenance of the dependency relation. In Section 5, the correctness of our approach w.r.t. the described terminological logic is shown. Section 6 summarizes some experiences gained with the implementation of our proposal; we conclude with an outlook to related work and a summary. An appendix lists the formal semantics of the considered logic, and the complete set of inference rules for defined roles.

2 Retraction in Terminological Representation Systems

Terminological representation systems permit the construction of knowledge bases and schemata around the notion of concepts, roles, and instantiating objects. Inferential services they provide include checking for inconsistencies in and classi-

[1] This report is an extended version of [Kin92].

fication of concept and object descriptions. In this report we base our discussion on a particular terminological logic that is similar to the language supported by the BACK system [PSKQ89, QK90]. We choose an abstract notation as agreed upon during the '91 International Workshop on Terminological Logics, cf. [B$^+$91]. The syntax of the considered language is defined as follows (for the semantics see Appendix A): Let A denote primitive concepts, and P primitive roles. *Concepts* (denoted by the letters C and D) are formed according to the syntax rule

$C, D \longrightarrow$	$\top \mid$	(top concept)
	$\bot \mid$	(bottom concept)
	$A \mid C \mid$	(primitive resp. defined concept)
	$\neg A \mid$	(negated primitive concept)
	$C \sqcap D \mid$	(intersection)
	$\forall R\!:\!C \mid$	(value restriction)
	$\geq nR \mid \leq nR \mid$	(number restriction)
	$R\!:\!x \mid R\!:\!\{x, \ldots, y\}$	(filler expression)

where n is a non-negative integer, R denotes an arbitrary role, and x, y denote objects. *Roles* (denoted by R and S) in turn are formed according to the syntax rule

$R, S \longrightarrow$	$P \mid R \mid$	(primitive resp. defined role)
	$R \sqcap S \mid$	(intersection)
	${}_C\!\mid\! R \mid$	(domain restriction)
	$R \mid_C \mid$	(range restriction)
	$R^{-1} \mid$	(inverse role)
	$R \circ S \mid$	(role composition)
	R^+	(transitive closure of role)

The KB schema consists of a set of concept and role introductions. We distinguish *primitive definitions* that specify necessary conditions, and *defined definitions* that specify necessary and sufficient conditions. Let D be a concept term according to the above syntax rule. Then a concept with name C is introduced as primitive[2] by writing $C :< D$; it is introduced as a defined concept by writing $C := D$. Introductions of roles are made accordingly.

The knowledge base is made up of an object descriptions of the form $x \in C$, where x is an object name, and C is a concept. An alternative notation for $x \in R : y$ is

[2]Primitiveness of a concept definition $C :< D$ can always be normalized away by introducing for the primitive concept a unique *primitive component* $\overline{C}$, and redefining C as $C := D \sqcap \overline{C}$, cf. [Neb90, Sec. 3.2.4]. The same holds for roles.

$(x, y) \in R$; we will use the former whenever we wish to emphasize our point of view that "y fills role R at x" is a local property of x.

A remark on notation: for concepts and roles we distinguish between syntactical introductions ($:<$, $:=$) on the one hand, and the semantical subsumption relation ($\sqsubseteq$) on the other hand; for objects we do not make this distinction, but denote by $x \in C$ both a description as entered by the user and the semantical instantiation relation. The reason for this is that, in the following, the distinction will matter for concepts and roles but not for objects.

Example 1: Let *Plant* and *Product* be primitive concepts introduced below the top concept (*Plant* $:< \top$, *Product* $:< \top$). The definition *ChemicalProduct* $:<$ *Product* introduces *ChemicalProduct* as a primitive subconcept of *Product*. The definition *ChemicalPlant* $:=$ *Plant* $\sqcap \forall$*produces* : *ChemicalProduct* defines a concept *ChemicalPlant* that represents all objects which are *Plant*s and *produce* only *ChemicalProducts*. The definition *producedBy* $:=$ *produces*$^{-1}$ introduces the inverse role of *produces*. With this schema, we can introduce objects into the KB: *plant* #1 $\in$ *ChemicalPlant* describes *plant* #1 as an instance of concept *ChemicalPlant*; *product* #17 $\in$ *Product* $\sqcap$ *producedBy* : *plant* #21 describes *product* #17 as a *Product* that is *producedBy* an object with name *plant* #21. □

Let us now turn to how facts are retracted from a terminological KB. On the interface level, we provide the retraction operation **forget**/1, which retracts from the KB the fact passed as an argument. An important restriction is that we only allow for retraction of facts explicitly told by the user, a restriction made in many terminological systems (see e.g. [PS91, Mac91]). Following Nebel [Neb90], we also use the term *literal retraction*. From the system we expect that, after a successful processing of a **forget** operation, system answers reflect a state as if the retracted description had never existed. Other operations can be based on **forget**, for example an operation to delete an entire object, or an operation **redescribe** that completely replaces an object's old description by a new description.

To realize the literal change approach the data derived by the system has to be kept separate from the objects' literal descriptions as provided by the user. The process to obtain derived information may be thought of as a two step process. From the literal description an internal representation is computed which we call *user normal form*, abbreviated NF_U. With respect to the told information, the NF_U may be completed with information derived using concept and role definitions. This is the case in BACK where the NF_U reflects facts that can be derived locally for an object. The NF_U, however, does not take into account information about other objects, and thus is totally independent of their changes. It only has to be recomputed when the object's literal description itself is changed.

The second step consists of classifying an object into the KB. This involves consistency checking, and determining the concepts instantiated by the object. Classification takes into account the descriptions of other objects, and yields a *completed object normal form* (NF_C). It is the NF_C that may become invalid if a second object is changed.

Example 2: To get a better feeling for the inferences drawn by terminological systems consider the following terminology and assertions: the KB schema is $\{C1 := C \sqcap \forall R : D1\}$, and the KB contains the following user-given facts $\{o1 \in C1, o2 \in D, o3 \in C \sqcap \leq 1R, (o1, o2) \in R, (o3, o2) \in R\}$.

From this KB a number of facts are typically derived. In particular $\forall R : D1$ is propagated to the fillers of $o1$ yielding $o2 \in D1$. For $o3$, in turn, $o3 \in C1$ is abstracted because all of $o3$'s fillers of role R (there is only $o2$) are instances of $D1$. Further, because of their fillers, we can abstract $o1 \in \geq 1R$ and $o3 \in \geq 1R$. In BACK, the last two facts belong to the NF_Us, all other derived facts are part of the NF_Cs of the involved objects. □

Having introduced a sample terminology and knowledge base, let us turn to a first example of a retraction operation.

Example 3: We change object $o1$ of the previous example by deleting that $o2$ is filler of role R, i.e. **forget**($o1 \in R : o2$). As a local consequence, $o1 \in \geq 1R$ does not hold any longer. As a non-local consequence, $o2 \in D1$ cannot be derived any more ($o2$'s description again is $o2 \in D$, as introduced by the user). Consequently, $o3 \in C1$ is not true any longer, but $o3 \in C \sqcap \geq 1R \sqcap \leq 1R$ still is. Note that we can only delete a fact that explicitly has been introduced into the KB; if we had called **forget**($o2 \in D1$) instead, this would have been rejected because $o2 \in D1$ was only derived by the system. □

If the inferences illustrated in Example 2 would only be drawn when the system has to answer a query, then retraction of facts would not require any special treatment. For efficiency reasons, however, many terminological systems employ *generative* implementations that—to some degree—store inferred facts *permanently* in the underlying KB (in contrast to *derivative* implementations that recompute derivable facts each time they are required).[3] Caching of inferred data may happen as part of precomputation of inferences at assertion time, or in a lazy evaluation mode whenever a query leads to the deduction of new facts (see e.g. [VM86]). For generative systems, retraction of single facts requires to remove from the knowledge base also those facts that were derived in direct or indirect dependence on the retracted ones.

[3]The terms *generative* and *derivative* are used in the context of deductive databases, cf. [GMN84].

3 Processing Retractions of Object Descriptions

We now propose a general retraction algorithm that meets the expectations formulated above. Its general scheme looks as follows:

(1) For each literally changed object delete its NF_U, mark its NF_C as invalid, and recompute the NF_U.

(2) Determine the objects that might be affected by the literal changes, and mark their NF_Cs as invalid.

(3) Reclassify all objects that have no valid NF_C.

As can be seen easily, we have selected system normal forms as the level of abstraction for dealing with retraction; a kind of *delete-and-recompute* strategy is employed that treats NF_Cs as atomic in the sense that, as soon as they become questionable, they are thrown away entirely, and are recomputed anew. Our major goal for the remainder of the section is to further develop Step (2), i.e., to reduce the number of objects to be processed by (3). No attempt is made, however, to process retractions by manipulating internals of normal forms.

It should be remembered that the inferences of terminological systems, as they are commonly understood, are purely monotonic. Thus, whenever we enter new facts into the KB, all inferences drawn—and stored—up to that point remain valid. This case of *adding* facts to the description of an object is already handled in the normal working mode by making the conjunction of the old description and the new facts, cf. [QK90]. Only when facts are *retracted*, previously derived and subsequently cached facts may have to be removed.

3.1 Minimal and Maximal Approaches

Before going into details, let us consider two extremes between which we will develop our solution. The most pessimistic way to determine affected objects is to follow the *edit/compile* approach, i.e., to reclassify *all* objects whenever the definition of a single object has literally been changed. In this case no extra maintenance of object dependencies is required, yet far too many objects are processed.

Reason maintenance or *truth-maintenance* systems (TMS) constitute the other extreme. As Nebel points out in [Neb90, Sec. 6.6], TMSs are able to precisely identify propositions that become invalid when the maintained knowledge is changed. Being designed to be used in general problem solving contexts, however, they require an overhead, in terms of search procedures and maintained data, that is inappropriate for our purpose [Neb90, p. 186]:

- TMSs support the problem solving task by implementing strategies to explore large search spaces, and to find one, many, or all solutions to a given problem.
- In TMS, derived propositions are perhaps marked as *disbelieved* or *no longer derivable*, but they are not deleted. Consequently, the memory consumed by a TMS is growing monotonically.

When we address the problem of retracting descriptions from a knowledge base, our focus is slightly different: In a normal working mode, the knowledge base is supposed to be static. Changes are made on explicit request. It is not very likely that retracted facts will be reasserted in the same form again, and their consequences are unlikely to be rederived again. Hence it seems more natural to "garbage-collect" and to remove propositions that are no longer derivable. By that the amount of data to be stored is kept small. Lastly, consequences of a change operation should be determined in a more direct way than to invoke expensive search algorithms.

3.2 A Naive Approach

A first step to attain an improvement over the edit/compile approach is to reclassify only objects that are *related* to an object x being changed. In the worst case, all these objects could be affected by a change of x, and would have to be reclassified. We call this approach to handle retraction *naive*.

An object y is *related* to x if it is *adjacent* to x, i.e., for some role R $(x, y) \epsilon R$ holds or $(y, x) \epsilon R$, or if y is *adjacent* to another object that is *related* to x.

An analysis of this approach revealed, however, that most objects are related to each other. For the evaluation we used a knowledge base constructed using the BACK system. The KB has been developed in an application in the business area [DBMP90], and describes the organizational structure of the 'Gruppo Ferruzzi', a large industrial holding. The KB contains 604 objects. Out of these 604 objects 143 objects are not related to any other object. For instance, none of the modeled companies is present in Asia, thus $Asia$ is such an isolated object. All other 461 objects are related to each other! If we look at the application this is not totally surprising: Companies are related to each other by ownership relations, or because they are engaged in the same business areas, or are present in the same countries. If we choose an arbitrary object, say manager m of some company c, m is also related through c to all other companies, and the business areas, countries, stock exchanges etc. they are related to.

The result must be expected to be similar for other applications. Even in standard database applications most objects would be related to each other. Consider, for

example, a database that keeps the customers, suppliers, orders, and purchases of a store. The consequence for retraction processing is that the naive approach is not much better than the edit/compile strategy.

3.3 A Semi-Naive Approach Based on Object Dependencies

The naive approach is improved by considering only those objects as affected by a retraction operation that previously have been influenced by the modified object. During classification of objects we record whenever an object x influences another object y. We say that y *depends on* x—denoted by a relation *DependsOn*(y, x)—if classification of x causes some inferences for a *related* object y, and if that information is not already present at y. We also interpret a tuple *DependsOn*(y, x) as a *dependency link* from x to y. The set of objects that at worst may be affected by a change of x is determined by computing the transitive closure of the *DependsOn* relation starting at x. We call this approach for processing retractions *semi-naive*.

Let us turn to how the *DependsOn* relation is built up. In general, the reasoner of the terminological system must be modified so that it makes entries to the dependency relation in the appropriate situations. We will demonstrate this using the example of BACK. For the current BACK system augmented with defined roles we can identify the following situations where one object influences a second one. x is depending on y if:

- y propagates type C forward to x, and C is not already known as a subsumer of x's NF_C;
- y is one of the fillers participating in the abstraction of $x \in \forall R : C$ (*backward propagation*);
- the filler set of x is completed using information at y according to the inference rules shown in Fig. 1 (completion for transitive, composed, inverse roles, or roles containing a **range** component).

The first two items roughly correspond to the functionality of the released BACK V4 described [QK90]. The treatment of defined roles has been proposed by Quantz [Qua90], and has been integrated into the new system release BACK V5. The corresponding *DependsOn* entries are specified in Fig. 3. Note that Fig. 1 contains only those rules that affect other objects; for the complete set of rules see Appendix B.

In the rules of Fig. 1, and in the *forward propagation* and *backward propagation* rules given above, x, y, and z stand for variables over objects in the KB, C and C_1 are variables over concepts, and R, R_1, and R_2 are variables over roles (while in the following example, x, y, z, $S1$, $S2$, and $S3$ are concrete objects, or roles,

Role Definitions		*Inferences*	*No. in* [Qua90]
Role Composition			
1.	R_1:=$R \circ R_2$	for each z with $(y,z) \in R_2$: add $(x,z) \in R_1$;	(11.1)
2.	R_1:=$R_2 \circ R$	for each z with $(z,x) \in R_2$: add $(z,y) \in R_1$;	(11.2)
Inverse Roles			
3.	R_1:=R^{-1} *or* R:=${R_1}^{-1}$	add $(y,x) \in R_1$;	(11.4)
4.	R:<${R_1}^{-1}$	add $(y,x) \in R_1$;	(11.8)
Transitive Roles			
5.	R:<${R_1}^{+}$ *or* R:=${R_1}^{+}$	for each z with $(z,x) \in R$: add $(z,y) \in R$;	
6.	R:<${R_1}^{+}$ *or* R:=${R_1}^{+}$	for each z with $(y,z) \in R$: add $(x,z) \in R$;	
Range-Restricted Roles			
7.	R_1:=$R\|_C$	if $y \in C_1$ and $C_1 \sqsubseteq C$: add $(x,y) \in R_1$	
8.	R_1:=$R\|_{C_1}$ (with $C \sqsubseteq C_1$)	for each y with $(y,x) \in R$: add $(y,x) \in R_1$;	(12.2)

Figure 1: Inference rules for completing filler sets of defined roles in BACK. The rules are triggered by assertion of $(x,y) \in R$ (Rules 1 to 7), or $x \in C$ (Rule 8) resp. The rules are adapted from [Qua90]; Rules 5, 6, and 7 have been newly added.

respectively).[4] For the application of Rules 1.1 to 1.7 of Fig. 1 it is assumed that a new filler relation $(x,y) \in R$ (for Rule 1.8 $x \in C$) is asserted by the user yielding the specified inferences. An example should illustrate this:

Example 4: Let $S3$ be a role defined as $S3$:=$S1 \circ S2$, and let the KB contain $(y,z) \in S2$. Assume a new fact $(x,y) \in S1$ is asserted by the user. Rule 1.1 is applied, and yields $(x,z) \in S3$.

As a dependency we record *DependsOn*(x,y) since if $(y,z) \in S2$ is deleted from y also x has to be processed again. □

The actual implementation of defined roles differs from the description in [Qua90]. Roughly, deciding on a syntactic basis (i.e., role introductions) for which roles to assert new fillers left out too many cases; the syntactic criteria have been exchanged for semantic criteria (i.e., role subsumption). The difference between [Qua90] and the implementation, however, does not affect our discussion of the

[4]Throughout the report, variables and concrete entities will not be distinguished syntactically; it should always be obvious from the context what is meant.

```
process x:
   IF x's user definition has been changed THEN
      recompute NF_U(x);
   L := depending-objects(x);
   FORALL i ∈ L DO
      reclassify i based on NF_U(i).

depending-objects(x):list of objects:
/* objects transitively depending on x */
   L1 := {x}; L2 := ∅;
   WHILE L1 ≠ ∅
      select x ∈ L1; L1 := L1\{x};
      IF x ∉ L2 THEN BEGIN
         /* avoid running into cycles */
         L2 := L2 ∪ {x};
         L1 := L1 ∪ directly-depending-objects(x)
   END END
   RETURN L2.

directly-depending-objects(x):list of objects:
   RETURN { y | DependsOn(y,x,p) ∧ not(subsumes(p,NF_U(x)))}
```

Figure 2: *Semi-naive* algorithm using labeled dependency links.

retraction approach. It is only that the less complete way of treating defined roles will cause less dependencies. The retraction approach itself remains the same.

3.4 Labeled Dependency Links

The semi-naive approach reclassifies all objects that directly or indirectly depend on the literally changed object. The number of objects involved may still be large. A further optimization is achieved if dependency links are followed only when they are actually affected by an ongoing retraction operation: Assume that x has been literally changed, and that y has already been identified to be directly or indirectly depending on x. A dependency link from y to some object z is followed only if the data that justified *DependsOn*(z, y) is no longer guaranteed to hold. It is not followed, however, if this data is guaranteed to hold; in that which all dependency links that start at z can be ignored too. To decide whether a dependency link has to be followed requires (i) attaching to dependency links the information which actually influenced the depending object, and (ii) determining when this information becomes *unsafe*, i.e., cannot be guaranteed to hold any longer.

An entry to the dependency relation is now labeled by attaching the information

forward propagation: *DependsOn*$(x, y, y \in \forall R\!:\!C \sqcap R\!:\!x)$
backward propagation and 1.7: *DependsOn*$(x, y, y \in C)$
1.1: *DependsOn*$(x, y, (y, z) \in R_2)$
1.2 and 1.5: *DependsOn*$(z, x, (x, y) \in R)$
1.3 and 1.4: *DependsOn*$(y, x, (x, y) \in R)$
1.6: *DependsOn*$(x, y, (y, z) \in R)$
1.8: *DependsOn*$(y, x, x \in C_1)$

Figure 3: Labeled *DependsOn* entries for the inferences of Sec. 3.3.

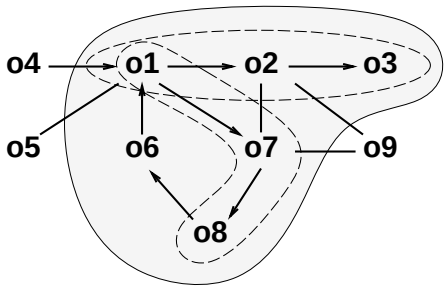
that allowed us to make the entry. Fig. 3 shows the labeled *DependsOn* entries for the inference rules specified in Sec. 3.3. In general, a dependency link *DependsOn*(y, x, p) means that y depends on x, and that if predication p ceases to hold at x the NF_C of y becomes unsafe, i.e., we have to recompute y's NF_C again.

Example 5: Reconsider Example 4. As a labeled dependency we record *DependsOn*$(x, y, (y, z) \in S2)$. □

The data attached to labeled dependency links is exploited when determining the list of objects which depend on a literally changed individual. This is reflected by the function *directly-depending-objects* as shown in Fig. 2. At each object x, for which we have to determine the objects directly depending on it, we follow a dependency link d to y only if the attached predication p is *unsafe*; in this case we also call both dependency d and object y *unsafe*.

A predication p, in turn, is considered *unsafe* if it cannot be confirmed by what is safely known about x. The basis to decide this is x's user normal form $\text{NF}_U(x)$ which is, as we saw, independent of other objects. A labeled *DependsOn* link has to be followed if the attached predication p cannot be derived from $\text{NF}_U(x)$. In other words, predication p is *unsafe* at x if it does not subsume $\text{NF}_U(x)$. Note, that *unsafe* does not mean that the predication is definitely invalid. It only expresses that at that moment we cannot safely decide if it still holds.

Example 6: We extend the KB from Example 2 by the following facts: $(o1, o5) \in R1$, $(o4, o1) \in R3^{-1}$, $(o6, o1) \in R3^{-1}$, $(o7, o1) \in R1$, $(o7, o2) \in R1$, $o8 \in \forall R : D$, $(o8, o6) \in R$, $(o8, o7) \in R1$, $o9 \in \forall R1\!:\!D1 \sqcap R1\!:\!o2$. We ignore the details of the extended terminology—they are not important for what follows. We only mention the existence of role $R2$ defined as $R2 := R1^+$; we will, however, write $R1^+$ directly in the examples. Fig. 4 shows the resulting relations and dependencies. Note, that there is no dependency entry stating that $o2$ depends on $o9$: $o9$'s attempt to forward propagate $D1$ to $o2$ is considered as redundant because



DependsOn$(o1, o4, (o4, o1) \in R3^{-1})$
DependsOn$(o1, o6, (o6, o1) \in R3^{-1})$
DependsOn$(o2, o1, o1 \in \forall R\!:\!D1 \sqcap R\!:\!o2)$
DependsOn$(o3, o2, o2 \in D1)$
DependsOn$(o6, o8, o8 \in \forall R\!:\!D \sqcap R\!:\!o6)$
DependsOn$(o7, o1, (o1, o5) \in R1^{+})$
DependsOn$(o8, o7, (o7, o1) \in R1^{+})$
DependsOn$(o8, o7, (o7, o5) \in R1^{+})$

Figure 4: Consequences of changing $o1$. x–y means "x and y are related to each other", and $x \rightarrow y$ means "x and y are related to each other, and *DependsOn*(y, x)".

the same information had already been successfully propagated by $o1$; we come back to this issue in Sec. 4.

We modify $o1$ by deleting $o1 \in \forall R : D1$ from its description. $\text{NF}_C(o1)$ becomes invalid, and $\text{NF}_U(o1)$ is recomputed. Next the objects transitively depending on $o1$ are determined. According to the entries in *DependsOn*, $o2$ and $o7$ depend directly on $o1$. Let us consider $o2$ first: The predication $o1 \in \forall R : D1 \sqcap R : o2$ of the *DependsOn* entry no longer subsumes $\text{NF}_U(o1)$. Thus, $o2$ actually depends on $o1$, and its NF_C is marked invalid. Next, we examine $o7$. $(o1, o5) \in R1^{+}$ still subsumes $\text{NF}_U(o1)$ (since $(o1, o5) \in R1 \sqsubseteq (o1, o5) \in R1^{+}$). The modification of $o1$ has no influence on $o7$, and we cut off the subgraph of dependency links starting at $o7$. Looking for objects that directly depend on $o2$, we find $o3$. Predication $o2 \in D1$ does not subsume $\text{NF}_U(o2)$ which is $o2 \in D$, thus also $\text{NF}_C(o3)$ becomes invalid. Since there is no outgoing dependency link at $o3$ we are finished; for the retraction at hand, the objects depending on $o1$ are $o2$ and $o3$. □

Example 7: We start from the same situation as in the previous example. This time, instead of $o1 \in \forall R : D1$, we retract $(o1, o5) \in R1$ from $o1$. Again we have to consider $o2$ and $o7$. This time $o2$ is not affected by the retraction ($o1 \in \forall R : D1$ still holds), but $o7$ is because $\text{NF}_U(o1) \not\sqsubseteq (o1, o5) \in R1^{+}$. Similarly, $o8$ depends on $o7$ since $(o7, o5) \in R1^{+}$ does not subsume $o7$'s NF_U. $o6$, however, is not affected by the currently processed retraction operation since $o8 \in \forall R : D$ still is given. The objects depending on $o1$ in this case are $o7$ and $o8$. □

The results of examples 6 and 7 are shown in Fig. 4 which also illustrates the improvements gained by exploiting labeled dependency links. With the two-argument *DependsOn* relation, in both examples all objects within the grey area would have been regarded as depending on $o1$. The naive algorithm would have required reclassification of all objects shown in the figure.

3.5 Recomputing the System Normal Form

So far we discussed how to determine those objects for which the corresponding NF_C became unsafe, i.e., may not reflect any longer the current state of knowledge base facts. Let us now turn to the recomputation of these system normal forms (Step (3) of our overall algorithm), and the problems that may occur.

If we take a closer look at the inference rules employed by terminological systems we see that some of the inference rules are event driven: they are triggered by user input or by data derived through the application of other inference rules, and are applied in a forward-chained manner. Other rules are goal driven: they are triggered by a user query or by a system internal request to prove a goal, and are applied in a backward-chained manner. The setting—which rules are run in which mode—is system dependent. We will use the setting of the BACK system: For the rules considered here this means that all rules except *backward propagation* are applied forward-chained. The *backward propagation* rule is applied backward-chained during the classification of an object, i.e., when its normal form has been completed and the most specific concepts in the taxonomy are soughed of which the object is an instance.

When recomputing the NF_C of object x, two things must be ensured: i) the NF_C must locally be set up in a way that applicable inference rules are actually applied, and, if required, broadcast information to adjacent objects, and ii) whatever had been broadcasted to x by any object through application of such a rule, whether successfully or in vain, must be recollected. This is obtained by the following algorithm:

(3.1) Treat all local fillers of x (according to $\text{NF}_U(x)$) as if they were new, i.e., re-tell them and let this trigger the appropriate rules. Apply the standard inference rules.

(3.2) Determine all objects and roles that have been filled by x before processing of the retraction operation.

(3.3) For every pair of an object y and a role R collected in (3.2) re-broadcast required information to x. For the language considered here, this means: Check if $\text{NF}_C(y)$ is valid. If it is, and if $y \in R : x$ holds:

a) Try to re-establish the inverse filler relation by triggering Rules 3/4 of Fig. 1.

b) Re-broadcast y's value restriction for R by triggering the forward propagation rule.

Else do nothing; either $y \in R : x$ does not hold any longer, then nothing has to be broadcasted to x, or $\text{NF}_C(y)$ is invalid and will be recomputed later, triggering the above rules anyway.[5]

(3.4) Reclassify object x.

Let us add a few comments: (3.1) ensures that all inferences that had been triggered by x in earlier situations are triggered again if still justified by what is known about x as user-told. (3.3) guarantees that everything that had been broadcasted to x before is broadcasted again if still justified by the remote object. This may again allow to trigger inference rules at x. Step (3.3) is necessarily depending on the employed set of inference rules. In Sec. 5 we will see that for our setting this algorithm recomputes the NF_C of an object correctly. For Step (3.2) we assume that we can determine where x has been a filler. This is not a limitation; most systems keep this information anyway, e.g., as a kind of backward reference, or by storing filler relationships as relations (rather than object based). In the worst case the information can be collected by looking up all object NF_Cs.

Finally, during the classification of an object, Step (3.4), the *backward propagation* rule may be triggered. Its purpose is to abstract a *value restriction* $x \in \forall R : D$ on the basis of actual fillers in a closed filler set. This occurs when the object classifier matches object x against a concept C in order to find out whether $x \in C$ is true. When $x \in \forall R : D$ is tested, the test may fail because at least one of the R-fillers of x, call it y, is not known to be an instance of D. It is part of the standard object classifier that x is informed whenever the description of one of its fillers in a closed filler set is changed. So if later y is classified under new concepts, x will get informed, and classification is invoked on x which eventually will then abstract $x \in \forall R : D$.[6]

Example 8: Reconsider Example 6. After the literal change of $o1$, we determined that, for the ongoing retraction operation, $o2$ and $o3$ were transitively depending on $o1$, so their NF_Cs have to be recomputed.

We compute $o2$'s NF_C first. Step (3.1) reprocesses all of $o2$'s fillers. There is only $o7$ which fills role $R1$. As a local consequence, the corresponding transitive role $R1^+$ is filled ($o2 \in R1^+ : o7$). Step (3.2) finds $(o1, R)$ and $(o9, R1)$ as pairs of objects and roles that have been filled by $o2$. From $o1$ no information is broadcasted to $o2$ (either the $\text{NF}_C(o1)$ is still invalid, or we know already that $o1 \in R1 : o2$ does not hold any more). The NF_C of $o9$, however, is valid; $o9$ has not been affected

[5] Because objects with an invalid NF_C are processed on their own, and will eventually trigger the appropriate inference rules, BACK filters those objects out in Step (3.2).

[6] For the details of object classification in BACK confer [QK90]; for a motivation of BACK's strategy of mixed forward-chained and backward-chained inferences see [Kin90].

by the change of $o1$. There exists no inverse role to $R1$, so (3.3a) does not yield anything new. The application of (3.3b) propagates $D1$ to $o2$; since $\text{NF}_C(o2) \not\sqsubseteq D1$ we add $D1$ to the NF_C, and record *DependsOn*$(o2, o9, o9 \in \forall R1 : D1 \sqcap R1 : o2)$.

Reprocessing $\text{NF}_C(o3)$ in Step (3.1) we again abstract that $o3 \in C1$, causing *DependsOn*$(o3, o2, o2 \in D1)$. Since $o3$ was not filler at any other object, nothing else has to be done. □

The details of algorithm (3) are very much oriented towards systems that process objects in a similar way as BACK does, i.e., by running inference rules at assertion-time and caching the results (generative technique). Note, however, that in a *lazy evaluation* model that caches results but runs inference rules only at query-time, Steps (1) and (2) remain unchanged, but (3) can be dropped. The NF_C in that case will be recomputed by and by as queries are asked by the user.

3.6 Further Improvements

Step (3) of the general algorithm, the recomputation of the system normal form of unsafe objects, offers the opportunity for a further optimization which we want to discuss here briefly. The basic idea is to confirm the NF_C of an unsafe object on the basis of safe information, and thereby to let other objects become safe again too. An NF_C can be confirmed when there is some safe information that allows for the derivation of everything that was contained in the NF_C before the processing of a retraction operation started. We illustrate this by an example.

Example 9: In Example 6, $o9$ is safe. It is possible to reestablish $o2 \in D1$ by means of $o9$, which was described as $o9 \in \forall R1 : D1 \sqcap R1 : o2$, and thereby to confirm the entire NF_C of $o2$. In that case, $o2$ does not have to be reclassified; also $\text{NF}_C(o3)$ can again be marked as valid since it was only included into the set of unsafe objects because of $o2$ being unsafe. □

To reestablish an NF_C, as safe information can be used either safe objects—as in the example—or information that has already been established for unsafe objects during the recomputation of their system normal forms.

An object's NF_C can be reconfirmed more effectively if we can determine what information was lost when the object became unsafe. This may be obtained by adding to each *DependsOn* entry a second label that tells what the effect of the dependency has been at the depending object (see also Sec. 4.1). With this information, other objects can be asked more precisely to reconfirm what was lost, rather than letting them provide arbitrary information.

Example 10: In Example 6, an augmented *DependsOn* entry could bear $o2 \in D1$ as a second label, meaning that, when the dependency was established, $o2 \in D1$ has been the consequence of the successful inference. $o2$ becomes unsafe through this *DependsOn* entry, thus $o2 \in D1$ is the information that needs to be reestablished. Consequently, $o9$, the safe object, can be asked directly whether it can reestablish $o2 \in D1$. □

As soon as the NF_C of an object is confirmed, all other objects that were unsafe only because of the reconfirmed object become safe on their own, their NF_Cs again can be marked valid. When setting objects back to a safe status, it must be taken care not to mark as safe objects which must remain unsafe because they depend on more than just the reconfirmed object.

It still remains to be investigated whether the optimization actually pays off. In the end, the attempt to be smart about shortening Step (3) may take longer than straight forward processing of all unsafe objects. Therefore, we currently did not implement this approach in BACK.

4 Dependency Relation Maintenance

Now that we have presented our general retraction approach, we wish to address the details of how dependency information is to be maintained. Basically two issues are to be considered: How many of the potential dependency links are actually stored? When do we delete *DependsOn* entries?

4.1 Making Dependency Relation Entries

Let us start with the creation of dependency links. We have discussed in detail which inference rules will potentially cause which dependency entries if applied successfully.[7] A frequently occurring situation, however, is that an object tries to establish at other objects information already known for them (either because told by the user, or because already established by other objects). In such a situation, we have to decide whether or not we actually want to make an entry to the *DependsOn* relation.

The choice made for the BACK system is to withdraw redundant *DependsOn* entries. This prevents us from later following links where no real dependency exists, and simplifies the maintenance of *DependsOn* entries. A dependency link is withdrawn

[7] For the language considered here, the rules are listed in Sec. 3.3 and in Fig. 1, the according *DependsOn* entries are listed in Fig. 3.

when the information that is propagated along this link is already known at the affected object, i.e., if it is contained in the object's NF_C. More precisely, if x and y are objects, and C is a term, and if an inference rule based on a predication on x attempts to establish $y \in C$, a new dependency between x and y is entered if C does not subsume ("is not contained in") the NF_C ($\text{NF}_C(y) \not\sqsubseteq C$).

Example 11: Assume that in Example 4 the user had explicitly asserted $(x, z) \in S3$ before application of Rule 1.1. No entry would have been made for *DependsOn*, and in fact deleting $(y, z) \in S2$ from y has no influence on x. However, if $(x, z) \in S3$ is later retracted, reclassification of x must reveal the dependencies established in Example 4. □

It must be ensured, however, that a redundant *DependsOn* entry is established whenever the reason for its redundancy vanishes, i.e., when $\text{NF}_C(y) \sqsubseteq C$ is not given any longer. This may happen when the targeted object y, or another object on which y depends, is modified during a retraction operation. In that case we expect that C is indeed enforced anew, and that the previously withdrawn dependency is stored this time. This is realized by Step (3) of our general retraction algorithm as discussed in Sec. 3.5.

Example 12: In Example 6 we saw that a dependency between $o2$ and $o9$ is rejected as redundant. After the literal change of $o1$, however, $o1$'s support for $o2 \in D1$ is not given any longer. When $\text{NF}_C(o2)$ is recomputed (Example 8), $o2 \in D1$ is rederived, and the previously redundant *DependsOn*$(o2, o9, o9 \in \forall R1 : D1 \sqcap R1 : o2)$ is accepted and stored. □

What if we kept the redundant dependencies? First of all there would be a maintenance overhead: more dependencies had to be stored, we had to introduce a *current* marker to prevent following also the redundant links, and we had also to remove more links (see below). But a stronger argument against keeping redundant *DependsOn* entries is that we can not really benefit from them. We cannot because dependency links—as proposed here—do not bear the information that tells us what has been the influence of the applied inference rule. We illustrate this by an example.

Example 13: The knowledge base schema consists of the primitive roles $R1$, $R2$, and $R3$, and a defined role $R4$ introduced as $R4 := R2 \circ R3$. The following facts are asserted by the user in that sequence into the KB: $o1 \in R1 : o2 \sqcap \forall R1 : (R4 : o4)$, $(o2, o3) \in R2$, $(o2, o5) \in R2$, $(o3, o4) \in R3$, $(o3, o6) \in R3$, $(o5, o6) \in R3$, $o7 \in R1 : o2 \sqcap \forall R1 : (R4 : o6)$. As derived facts the KB contains $(o2, o4) \in R4$ (derived by application of the *forward propagation*

rule on $o1$) and $(o2, o6) \epsilon R4$ (derived by application of Rule 1.2, triggered by $(o5, o6) \epsilon R3$). As dependencies we get

$$\textit{DependsOn}(o2, o1, o1 \epsilon R1 : o2 \sqcap \forall R1 : (R4 : o4)), \quad (1)$$

$$*\textit{DependsOn}(o2, o3, o3 \epsilon R3 : o4), \quad (2)$$

$$\textit{DependsOn}(o2, o3, o3 \epsilon R3 : o6), \quad (3)$$

$$*\textit{DependsOn}(o2, o5, o5 \epsilon R3 : o6), \quad (4)$$

$$*\textit{DependsOn}(o2, o7, o7 \epsilon R1 : o2 \sqcap \forall R1 : (R4 : o6)). \quad (5)$$

DependsOn entries marked with $*$ are redundant, those without are the *current* dependencies (the ones we are actually storing). Dependency (5), for example, is redundant because the attempt to propagate $R4 : o6$ along $R1$ to $o2$ did not yield any new information for $o2$; $o2 \epsilon R4 : o6$ had already been established by application of Rule 1.2 (which caused Dependency (3)). Dependency (4) is redundant because it tried to establish the same information (by application of Rule 1.2). The example is depicted in Fig. 5.

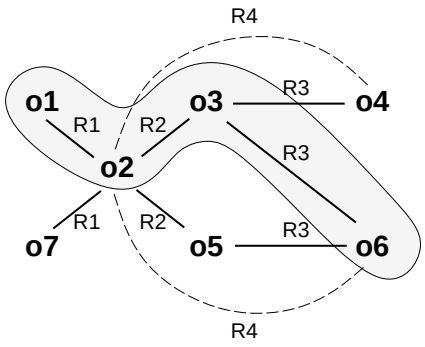


Figure 5: The knowledge base of Example 13. The grey area highlights the objects and relations that establish the derived role relations (represented by the dashed lines).

Assume now that $(o3, o6) \epsilon R3$ is retracted. Taking into account the current dependency entries we correctly determine that, except from the literally changed $o3$, $o2$ is the only affected object. We would not have to recompute $o2$'s NF_C if we still knew that the redundant Dependencies (4) and (5) could reestablish what we lost by the literal change of $o3$, namely $o2 \epsilon R4 : o6$. However, we did not attach that information to the *DependsOn* entries.[8] We could try to derive $o2 \epsilon R4 : o6$

[8]We are not going to discuss further the approach that adds a second label to each *DependsOn* entry, a label that carries the information telling us what had been the influence of the applied inference rule. For such an approach the maintenance of redundant dependencies may have a positive effect. The overall approach, however, would be more complicate.

from the *DependsOn* entries as are. But then, how could we prevent looking at too many dependencies? In the example, (2) looks just as promising as (4) on the first sight. □

For the problems raised in the example we decided to omit redundant *DependsOn* entries, and to recompute the propagated information as needed.

A second kind of redundancy can occur between *subsuming dependency entries.* We say that a dependency entry $d1$ subsumes a dependency entry $d2$, if $d1$ is *DependsOn*$(x1, y1, p1)$, $d2$ is *DependsOn*$(x2, y2, p2)$, $x1 = x2$, $y1 = y2$, and $p1$ subsumes $p2$ ($p2 \sqsubseteq p1$). In this case it is sufficient to keep the more special dependency $d2$ (cf. also [Tho92]). It is easy to see that $x1/x2$ still will be determined as *unsafe* correctly: The withdrawn $d1$ would become unsafe if $\text{NF}_U(y1/y2) \not\sqsubseteq p1$. Since $p2 \sqsubseteq p1$ it immediately follows that $\text{NF}_U(y1/y2) \not\sqsubseteq p2$. In other words, whenever the withdrawn $d1$ would become unsafe, also its subsumee $d2$ becomes unsafe.

4.2 Deleting Dependency Relation Entries

Dependency links are deleted during the processing of a retraction operation: For each object for which the NF_C has to be recomputed, the "incoming" dependency links are deleted. If x is one of these objects—because it has been literally changed, or has been affected by another object's change—the incoming dependency links are those where x stands in the first argument position of the dependency relation, i.e., *DependsOn*$(x, \_, \_)$. This would delete more dependency entries than necessary; many of them will be reestablished during Step (3) of our retraction algorithm. Consequently, an implementation will not directly delete those links, but will mark them only and delete them just when not confirmed by (3). We will neglect this implementational detail in the following, though.

On a first glance it seems sufficient to delete only *unsafe* dependency entries, those for which the test *DependsOn*$(y, x, p) \wedge (\text{NF}_U(x) \not\sqsubseteq p)$ succeeds during calculation of the transitive closure of depending objects. The basic problem is that we cannot guarantee that Step (3) would process objects in the same order as has been done before. As the following example shows, if we delete only the unsafe dependency links Step (3) will yield redundant entries in general.

Example 14: We return to Example 13. When we retract $(o3, o6) \in R3$, the NF_C has to be recomputed for two objects: the literally changed $o3$ and the depending $o2$. We are interested in the latter. Assume we would delete only *unsafe* dependency links. Here, Dependency (3) becomes unsafe, so we delete it. Recomputation of $\text{NF}_C(o2)$ rederives $(o2, o4) \in R4$ and $(o2, o6) \in R4$. However,

we are not sure that this is done using the same objects, in the same order, as before. On the contrary: while the original order of inference rule applications was controlled by the order of the user input, this time it is controlled by the way NF_C recomputation is performed. In particular, Step (3.1) is performed before (3.3), and $(o2, o4) \in R4$ will be derived by Rule 1.1, yielding Dependency (2).[9] Since also Dependency (1) is still present—it had not been considered as unsafe, thus had not been deleted—we now store two redundant dependencies.

The problem is resolved if we delete all incoming dependencies. When $o2$ is detected as unsafe, we delete the incoming dependencies (3) *and* (1). Recomputation of $\text{NF}_C(o2)$ establishes dependencies (2) and, presumably, (4). □

When recomputing system normal forms, we treat literally changed objects in the same way as depending objects. Since the user normal forms are adjusted prior to the NF_Cs, we can guarantee that user-input wins over system derivations, i.e., a *DependsOn* entry is deleted when made redundant by a user-told fact.

The situation changes, however, if *new* facts are asserted into the knowledge base, i.e., if objects are changed monotonically. The problem then occurs that a dependency relation entry may become redundant, but is not retracted. This is the case if the user asserts a fact that previously was derived on basis of another object. In the normal, monotonic processing mode, an object's NF_C is not invalidated and not recomputed as in the retraction case. Consequently, also incoming dependency links are not deleted. The new fact is simply added to the NF_U and the NF_C in a monotonic fashion; since all other facts remain, the NF_C is also still valid, and recomputation is avoided.

Example 15: Take Example 4, or 5, resp., as a starting point. Later the user tells $(x, z) \in S3$ explicitly. $x \in S3 : z$ is added to $\text{NF}_U(x)$; it is also "added" to $\text{NF}_C(x)$ which remains unchanged, however, since $x \in S3 : z$ had already been established via y (cf. Example 4). The dependency relation *DependsOn*$(x, y, (y, z) \in S2)$ now becomes redundant, but is not deleted. □

The consequence of not eliminating all redundant dependency links is that an object eventually will be considered unsafe when it is not, and that its NF_C will be recomputed unnecessarily. The good news is that as soon as such an object is involved in the processing of a retraction operation, its NF_C is recomputed and redundant incoming dependency links are deleted.

Example 16: In Example 15, if x becomes unsafe, e.g., because y has become unsafe for some reason, the redundant *DependsOn*$(x, y, (y, z) \in S2)$ will be deleted.

[9] If we recompute $\text{NF}_C(o3)$ before $\text{NF}_C(o2)$, $(o2, o4) \in R4$ will be derived by Rule 1.2; again Dependency (2) is recorded.

When recomputing $\mathrm{NF}_C(x)$, it is removed as an incoming dependency link. Then, in Step (3.1), Rule 1.1 will be triggered by $(x, y) \in S1$, but $(x, z) \in S3$ is already known ($\mathrm{NF}_C(x) \sqsubseteq (x, z) \in S3$), and the dependency link is not reestablished. □

5 Discussion of the Algorithm

Before proving correctness of our retraction approach, we have to reconsider the usage of the subsumption test which is employed for determining whether an object is unsafe and whether a dependency entry is redundant. In many terminological systems, subsumption is only implemented incompletely; let us call the corresponding predicate `subsumes`. If $T1$ and $T2$ are terms for which subsumption is tested, the following relationships are possible between the semantic subsumption relation $\sqsubseteq$ and its implementation `subsumes`:

(i) $T2 \sqsubseteq T1$ and `subsumes(`$T1$`,`$T2$`)` succeeds

(ii) $T2 \not\sqsubseteq T1$ and `subsumes(`$T1$`,`$T2$`)` fails

(iii) $T2 \sqsubseteq T1$ but `subsumes(`$T1$`,`$T2$`)` fails

Cases (i) and (ii) correspond to situations where `subsumes` is a sound and complete algorithm for subsumption checking. Case (iii) corresponds to the additional situation that may occur when `subsumes` is sound but incomplete: although semantically $T1$ subsumes $T2$, this is not detected by the predicate `subsumes`.

As we will see below, incompleteness of `subsumes` does not impair our retraction approach: in both cases in which subsumption is involved, determining whether an object is safe and determining whether a dependency entry is redundant, we use the failure of the subsumption test as the discriminating criterion. In other words, if due to its incompleteness `subsumes` fails more often than it should, the consequence is that some objects will unnecessarily be considered as unsafe, and some dependency entries will be made although being redundant.[10]

The first lemma shows that (at least) all necessary dependency relation entries are made, and that not too many of them are deleted. For the proof we have to assume that all inference rules of the system were analyzed and classified into rules with only a local effect and rules with non-local effects, i.e., an influence on other objects. For all rules with non-local effects we assume the specification of the according *DependsOn* entries. For the language considered in this report, we did

[10] Since here the terms which are compared against normal forms are not complicate in structure, we expect that the incompleteness of `subsumes` will not have an effect too often.

this by listing all inference rules (*forward propagation*, *backward propagation*, *role completion* rules as in Appendix B), by identifying the rules with non-local effects (Sec. 3.3), and specifying the *DependsOn* entries (Fig. 3). We omit the easy proof that we identified the right rules, and that the specified *DependsOn* entries are correct.

Lemma 1 *The maintenance of the dependency relation* DependsOn *is correct:*

1. *At assertion time, i.e., when objects are entered into the knowledge base, the correct* DependsOn *entries are made in all situations in which one object adds something new to the system normal form NF_C of another object.*
2. *When monotonic changes are made, no dependency entry is deleted that is still required.*
3. *When non-monotonic changes are made, no dependency entry is deleted that is still required.*

Proof: We start with the first of the above items. Entries to the dependency relation are made whenever the application of a non-local inference rule attempts to establish $x \in C$, and `subsumes(`C`, NF`$_C$`(`x`))` fails. Since `subsumes` is correct, if `subsumes(`C`, NF`$_C$`(`x`))` succeeds, C is already known for x, and no dependency relation entry is required. Due to incompleteness of `subsumes`, situation (iii) of our above discussion may occur leading to redundant entries in the dependency relation.

The second item of the Lemma is trivial: during monotonic changes no dependency entry is deleted at all.

Also the last item of the Lemma is easy: According to Sec. 4.2, a dependency relation entry is deleted only if it has x in its first argument position, and x is an object for which the NF_C has to be recomputed, i.e., x is unsafe. For each unsafe object x the NF_C is marked as invalid and reset to the NF_U (Step (2) of the overall algorithm); since $NF_U(x)$ does not depend on other objects, x should not have incoming dependency links. Comment: During Step (3), when $NF_C(x)$ is recomputed, some of the deleted dependency links will be reestablished, in particular those on which the recomputed NF_C depends. Lemma 3 will show that this is done correctly. □

Lemma 2 *When non-monotonic changes are made, the function* **depending-objects** *of Fig. 2 returns all objects that might be affected by the processed retraction operation.*

Proof: Assume x has already been determined as affected. Then **directly-depending-objects** determines all objects that are directly affected because x is unsafe: If a dependency d=*DependsOn*(y, x, p) exists, because of Lemma 1, x might have had an influence on y. If the test `subsumes`(p, NF$_U(x)$) fails y is considered as affected; case (iii) of our discussion on incompleteness of `subsumes` may apply, in which case y is unnecessarily considered as affected. If the test succeeds, the predication p that justified the inference rule and led to d still holds, thus also the result of the rule's application still holds, and y does not have to be reprocessed.—Other objects, those with no incoming dependency link from x, are not directly affected by the fact that x is unsafe, because according to Lemma 1 there would be a dependency link if x had ever influenced one of them.

Starting from a literally changed, thus unsafe, object, **depending-objects** computes the transitive closure of **directly-depending-objects**. Therefore, all affected objects are returned. □

For the proof of the following lemma, we assume that an arbitrary term C (e.g., $R:y$) was valid at the state before the NF$_C$ was invalidated. The following proves that algorithm (3) recomputes $x \in C$ in case this is still supported by what is known about x and adjacent objects, and that all the necessary dependency links are made. We prove this by considering all possible cases for recomputing $x \in C$. Note again that we do not claim that the inference rules in Fig. 1, or Appendix B, resp., are complete; we just have to ensure that all inferences that are drawn in the normal assertion mode are equally drawn during the retraction mode.

Lemma 3 *Algorithm (3) processes an unsafe object correctly in the sense that it recomputes the object's system normal form such that it contains the same information as if the object was asserted anew.*

Proof: Let s_0 be the state before the retraction process starts. Let s_n, $n > 0$, be a state during the recomputation of system normal forms. The different s_n denote different states of the NF$_C$ recomputation: for i, j with $j > i$, s_j is a state after s_i. Consequently, NF$_C^{s0}$ denotes a valid system normal form before the retraction process, and NF$_C^{sn}$ denotes a system normal form under recomputation.

We assume that $x \in C$ was valid in NF$_U^{s0}(x)$, C being an arbitrary term. We prove correctness of (3) by considering all possible cases to recompute $x \in C$ including those where $x \in C$ definitely can not be recomputed.

1. $x \in C$ is contained in NF$_U^{s1}(x)$; then it is trivially also in NF$_C^{s1}(x)$.
2. $x \in C$ is contained in NF$_C^{sj}(x)$ because it follows by application of local inference rules—not considered here—from NF$_C^{si}(x)$, $j > i > 0$.

3. *Inverse Roles:* C is of the form $R:y$; R participates in a definition of the form R_1:=R^{-1} or R:=$R_1{}^{-1}$ or R_1:<R^{-1}. Let s_k be the current state, i.e., we are working on $\text{NF}_C^{s_k}(x)$:

   (a) $y \in R_1:x$ is already valid in $\text{NF}_C^{s_i}(y)$, $0 < i < k$. Then because of (3.2) and (3.3a), $R:y$ is added to $\text{NF}_C^{s_k}(x)$, *DependsOn*$(x, y, R_1:x)$ is added to the dependency relation.

   (b) $y \in R_1:x$ is valid in $\text{NF}_C^{s_i}(y)$, $0 < k < i$, i.e., it is only established later during the reconstruction of $\text{NF}_C(y)$. But then, at state s_i, Rules 3 and 4 of Fig. 1 are applied yielding $x \in R:y$ and *DependsOn*$(x, y, R_1:x)$

   (c) $y \in R_1:x$ is not valid in $\text{NF}_C^{s_i}(y)$ for any $i > 0$. Then $x \in R:y$ can not be established through inverse roles.

4. *Role Composition:* C is of the form $R:y$; R is defined as R:=$R_1 \circ R_2$. Let s_k be the current state, i.e., we are working on $\text{NF}_C^{s_k}(x)$.

   (a) $x \in R_1:z$ is already valid in $\text{NF}_C^{s_i}(x)$, $0 < i \leq k$, for an arbitrary z. Because of (3.1), Rule 1 of Fig. 1 is triggered.

      i. if $z \in R_2 : y$ is valid in $\text{NF}_C^{s_j}(z)$, $i < j \leq k$, then Rule 1 can be applied successfully, and yields $x \in R : y$ and *DependsOn*$(x, z, z \in R_2:y)$.

      ii. if $z \in R_2:y$ is valid in $\text{NF}_C^{s_j}(z)$, $0 < k < j$, i.e., it is only established later during the reconstruction of $\text{NF}_C(z)$, then Rule 1 can not be applied successfully at s_k. At state s_j, however, Rule 2 of Fig. 1 is triggered, and its successful application yields $x \in R : y$ and *DependsOn*$(x, z, z \in R_2:y)$.

      iii. $z \in R_2:y$ is not valid in $\text{NF}_C^{s_j}(z)$ for any $j > 0$. Then $x \in R:y$ can not be established through role composition.

   (b) $x \in R_1:z$ is valid in $\text{NF}_C^{s_i}(x)$, $0 < k < i$, i.e., it is only established later. during the reconstruction of $\text{NF}_C(x)$. But at state s_i, Rule 1 of Fig. 1 is triggered, leading to the same situations as in cases 4(a)i through 4(a)iii.

   (c) $x \in R_1:z$ is not valid in $\text{NF}_C^{s_i}(x)$ for any $i > 0$. Then $x \in R:y$ can not be established through role composition.

   Note that the following situation, although being related to role composition, is actually covered by case 3: C is of the form $R:y$, but R is the inverse role of S, which in turn is defined as S:=$S_1 \circ S_2$. At s_0 there might have been a constellation with $y \in S_1:z$ and $z \in S_2:x$, which have caused $y \in S:x$, and consequently $x \in R:y$.

5. *Transitive Closure of Roles:* C is of the form $R:y$; R is defined as R:=$R_1{}^+$ or R:<$R_1{}^+$. Let s_k be the current state, i.e., we are working on $\text{NF}_C^{s_k}(x)$.

   (a) $x \epsilon R:z$ is already valid in $\text{NF}_C^{s_i}(x)$, $0 < i \leq k$, for an arbitrary z. Because of (3.1), Rule 6 of Fig. 1 is triggered.

      i. if $z \epsilon R:y$ is valid in $\text{NF}_C^{s_j}(z)$, $i < j \leq k$, then Rule 6 can be applied successfully, and yields $x \epsilon R:y$ and *DependsOn*$(x, z, z \epsilon R:y)$.

      ii. if $z \epsilon R:y$ is valid in $\text{NF}_C^{s_j}(z)$, $0 < k < j$, i.e., it is only established later during the reconstruction of $\text{NF}_C(z)$, then Rule 6 can not be applied successfully at s_k. At state s_j, however, Rule 5 of Fig. 1 is triggered, and its successful application yields $x \epsilon R:y$ and *DependsOn*$(x, z, z \epsilon R:y)$.

      iii. $z \epsilon R:y$ is not valid in $\text{NF}_C^{s_j}(z)$ for any $j > 0$. Then $x \epsilon R:y$ can not be established through transitive roles.

   (b) $x \epsilon R:z$ is valid in $\text{NF}_C^{s_i}(x)$, $0 < k < i$, i.e., it is only established later during the reconstruction of $\text{NF}_C(x)$. But at state s_i, Rule 6 of Fig. 1 is triggered, leading to the same situations as in cases 5(a)i through 5(a)iii.

   (c) $x \epsilon R:z$ is not valid in $\text{NF}_C^{s_i}(x)$ for any $i > 0$. Then $x \epsilon R:y$ can not be established through transitive roles.

6. *Range Restricted Roles:* C is of the form $R:y$; R is defined as R:=$R_1|_{C_1}$. Let s_k be the current state, i.e., we are working on $\text{NF}_C^{s_k}(x)$.

   (a) $x \epsilon R_1:y$ is already valid in $\text{NF}_C^{s_i}(x)$, $0 < i \leq k$. Because of (3.1), Rule 7 of Fig. 1 is triggered.

      i. if $y \epsilon C_1$ is valid in $\text{NF}_C^{s_j}(y)$, $i < j \leq k$, Rule 7 can be applied successfully, and yields $x \epsilon R:y$ and *DependsOn*$(x, y, y \epsilon C_1)$.

      ii. if $y \epsilon C_1$ is valid in $\text{NF}_C^{s_j}(y)$, $0 < k < j$, i.e., it is only established later, then Rule 8 is triggered at s_j, and its successful application yields $x \epsilon R:y$ and *DependsOn*$(x, y, y \epsilon C_1)$.

      iii. $y \epsilon C_1$ is not valid in $\text{NF}_C^{s_j}(y)$ for any $j > 0$. Then $x \epsilon R:y$ can not be established through range restricted roles.

   (b) $x \epsilon R_1:y$ is valid in $\text{NF}_C^{s_i}(x)$, $0 < k < i$, i.e., it is only established later. At state s_i, Rule 7 of Fig. 1 is triggered, leading to the same situations as in cases 6(a)i through 6(a)iii.

   (c) $x \epsilon R_1:y$ is not valid in $\text{NF}_C^{s_i}(x)$ for any $i > 0$. Then $x \epsilon R:y$ can not be established through range-restricted roles.

7. *Forward Propagation:* Let s_k be the current state, i.e., we are working on $\text{NF}_C^{s_k}(x)$. $y \epsilon R:x$ was true before retraction for some y and R, as has been determined by (3.2).

(a) $y \in R : x \sqcap \forall R : C$ is valid in $\text{NF}_C^{s_i}(y)$, $0 < i \leq k$. Because of (3.3b), C is added to $\text{NF}_C^{s_k}(x)$, *DependsOn*$(x, y, y \in R : x \sqcap \forall R : C)$ is added to the dependency relation.

(b) $y \in R : x \sqcap \forall R : C$ is valid in $\text{NF}_C^{s_i}(y)$, $0 < k < i$, i.e., it is only established later during the reconstruction of $\text{NF}_C(y)$. But at state s_i, the *forward propagation* rule is applied yielding $x \in C$ and *DependsOn*$(x, y, y \in R : x \sqcap \forall R : C)$

(c) $y \in R : x \sqcap \forall R : C$ is not valid in $\text{NF}_C^{s_i}(y)$ for any $i > 0$. Then $x \in C$ can not be established through forward propagation.

8. *Backward Propagation:* C is of the form $\forall R : D$. Let s_k be the current state, i.e., we are working on $\text{NF}_C^{s_k}(x)$.

(a) $x \in R : \{y_1, \ldots, y_n\} \sqcap \leq nR$ is already valid in $\text{NF}_C^{s_i}(x)$, $0 < i \leq k$. During (3.4), the *backward propagation* rule is eventually applied.

i. if $y_h \in D$ is valid in $\text{NF}_C^{s_j}(y_h)$, $i < j \leq k$, for every $y_h \in \{y_1, \ldots, y_n\}$ the *backward propagation* rule is applied successfully, and yields $x \in \forall R : D$ and *DependsOn*$(x, y_h, y_h \in D)$ for every y_h.

ii. there is a later state s_j, $0 < k < j$, in which $y_h \in D$ is valid in $\text{NF}_C^{s_j}(y_h)$ for every $y_h \in \{y_1, \ldots, y_n\}$; as part of classification x is informed about the change of also the last of the y_h; (3.4) is applied again on x, leading to 8(a)i.

iii. there is no state s_j, $0 < k < j$, in which $y_h \in D$ is valid in $\text{NF}_C^{s_j}(y_h)$ for every $y_h \in \{y_1, \ldots, y_n\}$; then $x \in \forall R : D$ can not be established through backward propagation.

(b) $x \in R : \{y_1, \ldots, y_n\} \sqcap \leq nR$ is valid in $\text{NF}_C^{s_i}(x)$, $0 < k < i$, i.e., it is only established later. As a consequence, (3.4) will be applied again on x, leading to the same situations as in cases 8(a)i through 8(a)iii.

(c) $x \in R : \{y_1, \ldots, y_n\} \sqcap \leq nR$ is not valid in $\text{NF}_C^{s_i}(x)$ for any $i > 0$. Then $x \in \forall R : D$ can not be established through backward propagation.

Since C and x are arbitrarily chosen, this proves that (3) correctly recomputes the system normal forms of all objects that had been collected as unsafe. □

We now come to the concluding theorem, which states correctness of our retraction approach.

Theorem 1 *The retraction procedure described in this report correctly processes all objects for which the literal change of another object may have a direct or indirect influence, meaning that, after the retraction operation, all objects in the knowledge base are in a state as if the retracted facts had never existed.*

Proof: In Lemma 1 we have proven that the *DependsOn* relation is built up and maintained correctly. In Lemma 2 we have proven that, based on the *DependsOn* relation, all questionable objects are included into the set of unsafe objects, and that objects not in this set actually have not to be touched. Finally, in Lemma 3 we have proven that all objects considered as unsafe are correctly processed, such that their normal forms are recomputed correctly, and classification is properly invoked. Thus, the theorem is proven. □

6 Implementation in BACK

The approach presented in this report has been implemented successfully in the BACK system Version 5. In this section we want to summarize some of the experiences we made, and which steps were necessary to integrate the approach into an existing reasoning system.[11]

6.1 Data Structures

In previous implementations of BACK, all information concerning an object was merged into a single normal form. For the implementation of the retraction functionality we first of all have separated the user definitions of objects from their normal forms; this enables us to check whether a call of the interface predicate **forget**/1 actually retracts user-told facts. In a second step we have distinguished several normal forms to reflect the different states of reasoning: the NF_U, which reflects application of local inferences on the user definitions, the NF_C, which reflects also application of global inferences on the NF_U, and the NF_I, which is needed in a further reasoning mode (see below).

Aside from this separation of data, BACK's data structures are not further affected by the retraction approach. Especially we have reached one of our initial goals: to keep the normal form data structures independent of data that has to be maintained solely for the purpose of handling retractions.

6.2 Dependency Module

All data structures and functionality concerned with object dependencies are encapsulated in a separate *dependency module* (cf. also [Tho92]). The dependency module provides the functionality to create single *DependsOn* entries, to add and remove them from the *DependsOn* relation, and to determine unsafe objects. The

[11] The implementation of BACK Version 4, the basis for Version 5, is described in [QK90].

latter is an implementation of function **depending-objects**/1 of Fig. 2. For the computation of the transitive closure of direct dependencies it uses an adapted version of the Warshall algorithm—in the Prolog version given in [OKe90]—into which the call of **directly-depending-objects**/1 has been unfolded.

The dependency relation is a slightly optimized representation of the relation we have used in this report. In order to support retraction also for facts inferred by application of *implication links*, see below, two distinct dependency relations are maintained, one for NF_C dependencies and one for NF_I dependencies.

First experiments suggest not to store all dependencies explicitly in the *DependsOn* relation. In BACK, this is especially true for dependencies introduced through defined roles. At the moment, BACK introduces for each role an inverse role; the reasons lie in the processing of defined roles. As a consequence, each assertion of a role relation $(x, y) \in R$ causes $(y, x) \in R^{-1}$. Since the latter is rarely contained also in $\text{NF}_U(y)$, this yields a *DependsOn* entry which only duplicates the known information. In such a case, it is more advisable to keep the dependency between y and x implicitly; the functional interface than should provide the appropriate abstraction.

6.3 Organizing the Assertional Reasoning

In the new version of BACK, application of inference rules on objects is organized by a central processing control program, which is called *scheduler*. Its purpose is to determine the order in which objects are processed, and to maintain request handling between objects: objects are not manipulating each other directly, but through requests which they pass via the scheduler. When an object x is processed, and a non-local inference rule is applied, the consequence of this rule, e.g., $y \in C$, is not directly enforced on y. Instead, x issues a request of the form `tell(`$y \in C$`, `x`, `d`)` which expresses that x has inferred $y \in C$, and that dependency d must be stored in the *DependsOn* relation if $y \in C$ adds new information to y. The scheduler collects such requests; when y is processed, x's request is passed to y, which decides upon newness of $y \in C$, and, if necessary, adds d to the dependency relation.

The functional interface of the module implementing objects had to be augmented with a function to recompute object normal forms; especially the backward application of the inference rules for forward propagation and inverse roles, Step (3.3), have been added newly.

6.4 Extended Usage of the Approach

We already have mentioned *implication links*. Implication links are a simple rule mechanism supported by some terminological systems. In BACK, their form is $C1 => C2$, where $C1$ and $C2$ are arbitrary concept terms. They are basically applied on the object level as soon as the NF_Cs of all objects are stable: when an object x is recognized as an instance of $C1$, the above rule is fired causing $x \epsilon C2$.[12] All facts inferred in this way go into the NF_I of an object, the object normal form depending on the application of implication links.

One reason to distinguish NF_C and NF_I is to offer to the user the choice whether an answer to a query such as "is x an instance of C?" shall take into account implication links. A second reason is that, during the processing of a retraction operation, an object's NF_C may remain safe, and only its NF_I has to be recomputed. For extending the retraction approach to handle also the implication link cases, it was only necessary to have a separate *DependsOn* relation for this mode, and to direct all dependencies into this relation as soon as implication links are applied.

The approach described here is also employed to enforce changes of the KB schema, i.e., concept and role redefinition, onto the KB. A description of this work is found in [Tho92].

7 Related Work

The work described in this report adopts a monotonic data dependency network management approach (see [CRM80, Chap. 16] or [Neb90, Sec. 6.6.1]). A data dependency network (DDN) basically consists of *nodes*, which denote believed propositions, *justifications*, which denote sets of propositions used in derivations, and the links between them. Justifications *support* nodes, and nodes *participate* in justifications. A DDN is monotonic if adding new information only causes additions to, and never retractions from, the database, and retraction of database facts only occurs after a call to a dedicated retraction predicate; this is the case for the kind of systems we are considering here.

The major distinction of our approach to classical DDNs stems from different granularities of believed nodes and justifications: In a traditional DDN approach, the set of believed nodes would consist of atomized parts of our (system) normal forms, i.e., atomic facts like $x \epsilon C$, plus the terminological inference rules as axioms, while in our approach only a normal form as a whole is considered as a believed node. In DDNs, therefore, justifications support atomic propositions,

[12] For a description of *implication links* in BACK cf. [QK90]; BACK basically follows the approach described in [OK88].

while in our case they are supporting entire normal forms. Furthermore, in a DDN approach, also inference rules participate in justifications, while our approach abstracts from the applied inference rules. As a consequence, we keep less data related to dependency management, but may have to do more recomputation when a retraction operation is processed.

In the literature, more support can be found for our reservation about employing standard truth-maintenance systems for implementing retractability of knowledge base facts (see Sec. 3.1). Van Marcke develops a consistency maintenance system called FPPD to support management of cached inference results [VM86]. FPPD distinguishes primitive values, which are given explicitly, and computed values, which depend on the execution of functions and rely on the validity of the functions' argument values. If primitive values are changed denials are propagated to all depending values; the latter are recomputed the next time they are queried (*lazy evaluation*). There are several differences to our approach: In BACK, fillers of defined roles—which can be compared to computed values—can be both user-told or system-derived. In addition, BACK deals with abstraction and propagation of type information. In FPPD, dependencies are established between atomic propositions rather than entire object descriptions; on the other hand, justifications of dependencies are not maintained, which results in dependencies similar to our unlabeled version of *DependsOn*.—Euzenat shows in [Euz90] that even in the presence of non-monotonic inferences a dedicated inference cache system can be superior to a standard TMS. He compares the two approaches for the object-based SHIRKA knowledge base management system. Inferences in SHIRKA are based on the activation of objects' methods rather than on structural properties as in BACK. Euzenat concludes that TMSs, being designed for maintenance of formula validity, are not well suited for dealing with attribute values in object-based models, and require too much storage space for being employed for large object bases.

In the context of terminological systems, an early solution to the retraction problem was provided by KL-TWO [Vil85]. It consisted of a purely terminological component (not supporting object management), and, as its assertional component, a version of RUP [McA82], a reason-maintenance system supporting propositional logic. Although an elegant solution with respect to the retraction problem, the two components did not match very well (cf. [Neb90, p. 64] for details).

The BACK system is more closely related to the systems CLASSIC of Bell Labs, and LOOM of USC/ISI (see the descriptions in [Ric91]), in that the components that deal with the object level have been especially designed to match the supported terminological logics. With respect to retractability of object descriptions there are some similarities:[13] As BACK, both systems permit only retraction of explicitly told

[13]There is no published material on the retraction mechanisms in CLASSIC and LOOM; the details mentioned here are from personal communications [Mac91, PS91].

information, and both systems are implemented using a generative technique.[14]

Differences are caused by the expressivity of the supported languages. LOOM aims to integrate general artificial intelligence programming techniques into its framework; as a consequence, a DDN like approach had to be replaced by truth-maintenance of arbitrary first-order logic predicates [Mac91]. CLASSIC, on the other hand, supports a more restricted language than the one discussed in this report; essentially, it does not provide defined roles. Retraction can be handled more easily since the possible sources for dependencies between objects are limited. This leads to different choices for the details of dependency maintenance [PS91]: CLASSIC stores at each object what information has been broadcasted to, and what has been received from, other objects. This corresponds to making *DependsOn* entries labeled both with their justifications and consequences, and to retaining redundant dependencies. CLASSIC also seems to encorporate an optimization similar to the one discussed in Sec. 3.6.

8 Summary

We have presented a framework to deal with retraction in the special kind of monotonic knowledge representation systems developed under the paradigm of terminological logics. The presented solution is applicable if a generative implementation technique is employed, i.e., if a system, to optimize subsequent retrieval, caches inferred data permanently in its knowledge base, independent on whether the system precomputes most inferences at assertion time, or whether it uses a lazy evaluation mode.

At the user interface level, retraction operations have been limited to literal descriptions, to information explicitly told by the user. Alternatives for their implementation have been analyzed, and a data dependency network mechanism has been adopted. A processing model has been described that maintains three types of information for each object: a set of user-given descriptions, and a system internal representation as normal forms, split up into a user normal form, that is independent of other objects, and a completed system normal form NF_C, that may depend on other objects. System normal forms have been selected as the appropriate level of abstraction for dealing with retraction; they are treated as atomic in the sense that as soon as they become questionable they are thrown away and are recomputed anew. Dependencies between objects are maintained to decide when this has to be done. A single dependency is established by the reasoning component whenever an application of an inference rule at one object causes a change

[14] For LOOM also exist newer experiments with derivative implementation techniques, cf. [MB92].

of the NF_C of another object; redundant dependencies are withdrawn. To each dependency its justification is attached in order to be able to reduce the number of objects considered as unsafe.

The proposed approach has been illustrated for a sublanguage of the BACK system which is distinguished from other terminological languages by the possibility to express knowledge by means of defined roles. The proposal has been implemented successfully for the full language of BACK V5 which includes further language constructs and an additional reasoning mode for a simple rule-like mechanism (*implication links*).

Acknowledgements

This work was supported by the Commission of the European Communities and is part of the ESPRIT project 5210 AIMS which involves the following participants: Datamont (I), Non Standard Logic (F), Technische Universität Berlin (D), Deutsches Herzzentrum Berlin (D), Onera-Cert (F), Quinary (I), Universidad del Pais Vasco (E).

I would like to thank Martin Fischer, Bob MacGregor, Bernhard Nebel, Peter Patel-Schneider, Christof Peltason, Joachim Quantz, and Albrecht Schmiedel for comments on earlier versions of this report, and/or discussions on the subject in general. Jan Thomsen deserves special thanks for his contribution to the implementation and evaluation of the approach.

References

[B+91] Franz Baader et al. Terminological Knowledge Representation: A Proposal for a Terminological Logic. In Bernhard Nebel, Christof Peltason, and Kai von Luck, editors, *International Workshop on Terminological Logics*, KIT Report 89, pages 120–128, Technische Universität Berlin, Germany, August 1991.

[BS85] Ronald J. Brachman and James G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, April 1985.

[CRM80] Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott. *Artificial Intelligence Programming*. Erlbaum, Hillsdale, N.J., 1980.

[DBMP90] Maria Damiani, Sandro Bottarelli, Manlio Migliorati, and Christof Peltason. Terminological Information Management in ADKMS. In

ESPRIT '90 Conference Proceedings, pages 163–176. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[Euz90] Jérôme Euzenat. Inference cache consistency in large object knowledge bases. Rapport interne, Laboratoire ARTEMIS, Grenoble, France, September 1990.

[GMN84] Hervé Gallaire, Jack Minker, and Jean-Marie Nicolas. Logic and databases: A deductive approach. *ACM Computing Surveys*, 16(2):153–185, 1984.

[Kin90] Carsten Kindermann. Class Instances in a Terminological Framework – An Experience Report. In H. Marburger, editor, *GWAI-90. 14th German Workshop on Artificial Intelligence*, pages 48–57. Springer-Verlag, Berlin, Germany, September 1990.

[Kin92] Carsten Kindermann. Retraction in Terminological Knowledge Bases. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 420–424. Wiley, August 1992.

[Mac91] Robert MacGregor. Personal Communication. November 1991.

[MB92] Robert M. MacGregor and David Brill. Recognition algorithms for the loom classifier. In *Proceedings of the 11th National Conference of the American Association for Artificial Intelligence*, San Jose, Cal., 1992.

[McA82] David A. McAllester. Reasoning utility package user's manual. AI Memo 667, AI Laboratory, Massachusetts Institute of Technology, Cambridge, Mass., April 1982.

[Neb90] Bernhard Nebel. *Reasoning and Revision in Hybrid Representation Systems*, volume 422 of *Lecture Notes in Artificail Intelligence*. Springer-Verlag, Berlin, Germany, 1990.

[OK88] Bernd Owsnicki-Klewe. Configuration as a consistency maintenance task. In W. Hoeppner, editor, *GWAI-88. 12th German Workshop on Artificial Intelligence*, pages 77–87. Springer-Verlag, Berlin, Germany, 1988.

[OKe90] Richard A. O'Keefe. *The Craft of Prolog*. Logic Programming Series. MIT Press, Cambridge, Mass., 1990.

[PS91] Peter Patel-Schneider. Personal Communication. November 1991.

[PSKQ89] Christof Peltason, Albrecht Schmiedel, Carsten Kindermann, and Joachim Quantz. The BACK System Revisited. KIT Report 75, Technische Universität Berlin, Germany, September 1989.

[PvLK91] Christof Peltason, Kai von Luck, and Carsten Kindermann. Proceedings of the Terminological Logic Users Workshop. KIT Report 95, Technische Universität Berlin, Germany, December 1991.

[QK90] Joachim Quantz and Carsten Kindermann. Implementation of the BACK System Version 4. KIT Report 78, Department of Computer Science, Technische Universität Berlin, Berlin, Germany, September 1990.

[Qua90] Joachim Quantz. Modeling and Reasoning with Defined Roles in BACK. KIT Report 84, Technische Universität Berlin, Germany, November 1990.

[Ric91] Charles Rich. Special issue on implemented knowledge representation and reasoning systems. *SIGART Bulletin*, 2(3), June 1991.

[Tho92] Jan Thomsen. Schema evolution and its consequences for a terminological knowledge base. Unpublished Draft, 1992. (A report on the subject is expected for summer 1993.)

[Vil85] Marc B. Vilain. The restricted language architecture of a hybrid representation system. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 547–551, Los Angeles, Cal., August 1985.

[VM86] Kris Van Marcke. A parallel algorithm for consistency maintenance in knowledge representation. In *Proceedings of the 7th European Conference on Artificial Intelligence*, pages 278–290, Brighton, England, July 1986.

A Semantics

The following defines the semantics of the terminological logic introduced in Sec. 2. A *term* (denoted by T,T_1,T_2) is either a concept (denoted by C, D) or a role (denoted by R, S); a *formula* is $T_1 \preceq T_2$ or $x \in C$. A *terminology* $\mathcal{T}$ is a set of $\preceq$-formulae which are derived from term introductions: each primitive introduction $T_1 {:<} T_2$ corresponds to the formula $T_1 \preceq T_2$, each defined introduction $T_1 {:=} T_2$ corresponds to the two formulae $T_1 \preceq T_2$ and $T_2 \preceq T_1$. The usual restrictions are imposed on terminologies: on the left hand side of introductions only names are allowed, and any name should appear only once as a left hand side of an introduction.

A structure $\mathcal{M}$ is defined as a pair $\langle \Delta, \mathcal{I} \rangle$ where Δ is a set called *domain* of $\mathcal{M}$. $\mathcal{I}$ is an *interpretation function* mapping object names injectively to elements of Δ, concepts into subsets of Δ, and roles into subsets of $\Delta \times \Delta$, such that the following equations are satisfied:

$$\begin{aligned}
\mathcal{I}[\top] &= \Delta && (6)\\
\mathcal{I}[\bot] &= \emptyset && (7)\\
\mathcal{I}[\neg T] &= \Delta \backslash \mathcal{I}[T] && (8)\\
\mathcal{I}[T_1 \sqcap T_2] &= \mathcal{I}[T_1] \cap \mathcal{I}[T_2] && (9)\\
\mathcal{I}[\forall R{:}C] &= \{x \in \Delta \mid \forall \langle x, y \rangle \in \mathcal{I}[R] : y \in \mathcal{I}[C]\} && (10)\\
\mathcal{I}[\geq n R] &= \{x \in \Delta \mid |\{y \in \Delta \mid \langle x, y \rangle \in \mathcal{I}[R]\}| \geq n\} && (11)\\
\mathcal{I}[\leq n R] &= \{x \in \Delta \mid |\{y \in \Delta \mid \langle x, y \rangle \in \mathcal{I}[R]\}| \leq n\} && (12)\\
\mathcal{I}[R{:}y] &= \{x \in \Delta \mid \langle x, y \rangle \in \mathcal{I}[R]\} && (13)\\
\mathcal{I}[R{:}Y] &= \{x \in \Delta \mid Y \subseteq \{y \in \Delta \mid \langle x, y \rangle \in \mathcal{I}[R]\}\} && (14)\\
\mathcal{I}[_C|R] &= \mathcal{I}[R] \cap (\mathcal{I}[C] \times \Delta) && (15)\\
\mathcal{I}[R|_C] &= \mathcal{I}[R] \cap (\Delta \times \mathcal{I}[C]) && (16)\\
\mathcal{I}[R^{-1}] &= \{\langle x, y \rangle \in \Delta \times \Delta \mid \langle y, x \rangle \in \mathcal{I}[R]\} && (17)\\
\mathcal{I}[R \circ S] &= \mathcal{I}[R] \circ \mathcal{I}[S] && (18)\\
\mathcal{I}[R^+] &= (\mathcal{I}[R])^+ && (19)
\end{aligned}$$

A formula is satisfied in a structure $\mathcal{M}$, if:

$$\begin{aligned}
\mathcal{M} \models T_1 \preceq T_2 \quad \textit{iff} \quad \mathcal{I}[T_1] \subseteq \mathcal{I}[T_2] && (20)\\
\mathcal{M} \models X \in C \quad \textit{iff} \quad \mathcal{I}[X] \in \mathcal{I}[C] && (21)
\end{aligned}$$

A structure $\mathcal{M}$ is a model of a formula ϕ iff $\mathcal{M} \models \phi$; it is a model of a set of formulae Φ iff it is a model of all formulae in Φ. A formula ϕ follows from a set of formulae, $\Phi \models \phi$, iff each structure which is a model for Φ is also a model for ϕ.

The semantic relation $\preceq$ introduced above denotes subsumption in its general form; since in the considered language, concept terms can refer to objects, subsumption of terms may also depend on descriptions of objects. For example, from the set $\{C_1 \preceq R : x \sqcap \leq 1R, x \in C_2\}$ follows $C_1 \preceq \forall R : C_2$. In BACK, we have chosen to provide a limited subsumption relation $\sqsubseteq$, which does not take into account object descriptions. A limited subsumption relation is defined with respect to a terminology:

$$T_1 \sqsubseteq T_2 \quad \textit{iff} \quad \mathcal{T} \models T_1 \preceq T_2 \tag{22}$$

Throughout the report, only the limited subsumption relation $\sqsubseteq$ was used.

B Role Completion Rules

In the following we give the complete set of inference rules for the completion of object descriptions in connection with defined roles. The rules have originally been developed by Quantz [Qua90]. We have adopted a different notation that combines into a single notation Quantz's sets of roles needed for object classification [Qua90, Fig. 10] and the according inference rules [Qua90, Figs. 11,12]. We have maintained the syntactical approach, which is less complete than what has actually been implemented in BACK (cf. p. 9), but have added a few rules to make the syntactical approach more complete in itself; the new roles are those having no entry in the column that refers to [Qua90].

In both tables, x, y, and z stand for variables over objects in the KB, C and C_1 are variables over concepts, and R, R_1, and R_2 are variables over roles.

The first table shows the inference rules which are triggered as soon as $x \,\epsilon\, C$ becomes explicitly known in the knowledge base (either because told by the user or because cached by the system as the consequence of a deduction):

Role Definitions	*Inferences*	*No. in* [Qua90]	*No. in* Fig. 1
Domain-Restricted Roles			
$R_1{:=}_{C_1}\vert R_2$	for each y with $(x,y)\,\epsilon\, R_2$:	(12.1)	
(with $C \sqsubseteq C_1$)	add $(x,y)\,\epsilon\, R_1$;		
Range-Restricted Roles			
$R_1{:=}R_2\vert_{C_1}$	for each y with $(y,x)\,\epsilon\, R_2$:	(12.2)	8
(with $C \sqsubseteq C_1$)	add $(y,x)\,\epsilon\, R_1$;		
Roles with Minimum Restriction at C			
$R{:<}{R_1}^+$ *or* $R{:=}{R_1}^+$	if $C \sqsubseteq (\geq n\,R)$ and $n \geq 1$	(12.3a)	
or $R{:=}R_1 \circ R_2$	add $x\,\epsilon\, \geq 1 R_1$		
or $R{:<}R_1 \circ R_2$			
$R_1{:=}R^+$	if $C \sqsubseteq (\geq n\,R)$ and	(12.3b)	
or $R_1{:=}R \sqcap R_2$	$\neg\exists m.(m > n \wedge C \sqsubseteq (\geq m R))$		
or $R_1{:=}R_2 \sqcap R$	add $x\,\epsilon\, \geq n R_1$		
or $R_1{:<}R \sqcap R_2$			
or $R_1{:<}R_2 \sqcap R$			
Roles with Maximum Restriction at C			
$R{:<}{R_1}^+$ *or* $R{:=}{R_1}^+$	if $C \sqsubseteq (\leq n\,R)$ and	(12.4a)/	
or $R_1{:=}R \sqcap R_2$	$\neg\exists m.(m < n \wedge C \sqsubseteq (\leq m R))$	(12.4b)	
or $R_1{:=}R_2 \sqcap R$	add $x\,\epsilon\, \leq n R_1$		
$R_1{:=}R \circ R_2$	if $C \sqsubseteq (\leq 0 R)$ add $x\,\epsilon\, \leq 0 R_1$	(12.4c)	

The second table shows the inference rules which are triggered as soon as $(x, y) \epsilon R$ becomes explicitly known in the knowledge base (either because told by the user or because cached by the system as the consequence of a deduction):

Role Definitions	*Inferences*	*No. in* [Qua90]	*No. in* Fig. 1
Role Composition			
R_1:=$R \circ R_2$	for each z with $(y, z) \epsilon R_2$: add $(x, z) \epsilon R_1$;	(11.1)	1
R_1:=$R_2 \circ R$	for each z with $(z, x) \epsilon R_2$: add $(z, y) \epsilon R_1$;	(11.2)	2
R:=$R_1 \circ R_2$ *or* R:<$R_1 \circ R_2$	add $x \epsilon \geq 1 R_1$	(11.6)	
Inverse Roles			
R_1:=R^{-1} *or* R:=${R_1}^{-1}$	add $(y, x) \epsilon R_1$;	(11.4)	3
R:<${R_1}^{-1}$	add $(y, x) \epsilon R_1$;	(11.8)	4
Role Conjunction			
R_1:=$R \sqcap R_2$ *or* R_1:=$R_2 \sqcap R$	if $(x, y) \epsilon R_2$ add $(x, y) \epsilon R_1$	(11.5)	
R:=$R_1 \sqcap R_2$ *or* R:<$R_1 \sqcap R_2$	add $(x, y) \epsilon R_1$, add $(x, y) \epsilon R_2$	(11.9)	
Transitive Roles			
R_1:=${R}^{+}$	add $(x, y) \epsilon R_1$	(11.3a)	
R:<${R_1}^{+}$ *or* R:=${R_1}^{+}$	add $x \epsilon \geq 1 R_1$	(11.7)	
R:<${R_1}^{+}$ *or* R:=${R_1}^{+}$	for each z with $(z, x) \epsilon R$: add $(z, y) \epsilon R$;	≈(11.3b)	5
R:<${R_1}^{+}$ *or* R:=${R_1}^{+}$	for each z with $(y, z) \epsilon R$: add $(x, z) \epsilon R$;		6
Domain-Restricted Roles			
R_1:=$_C \vert R$	if $x \epsilon C_1$ and $C_1 \sqsubseteq C$: add $(x, y) \epsilon R_1$		
Range-Restricted Roles			
R_1:=$R \vert_C$	if $y \epsilon C_1$ and $C_1 \sqsubseteq C$: add $(x, y) \epsilon R_1$		7