KIT REPORT 103

How to Compute 1 + 1 ?
A Proposal for the Integration of External Functions and Computed Roles into BACK

Gerd Kortüm

Technische Universität Berlin
Projekt KIT, FR 5–12
Franklinstr. 28/29, W–1000 Berlin 10

e-mail: kortuem@cs.tu-berlin.de

University of Oregon
e-mail: kortuem@cs.uoregon.edu

January 1993

Abstract

Today, terminological knowledge representation systems have reached a level of maturity which makes their use for real world applications feasible. However, the construction of practical knowledge representation management systems (KBMS) requires more than a well-defined representation formalism and reasoning algorithm. Flexibility and system integration are two important issues for any KBMS. In this paper, we propose an extension of the BACK terminological representation system which allows the integration of externally realized procedures into the representation formalism. For that purpose, we introduce a new language construct *External Function* which serves as an interface to arbitrary host-language procedures. External Functions allow us to integrate forms of domain specific knowledge such as arithmetic computations into BACK which can not be expressed efficiently in terminological representation systems. We show how results of computations can be integrated into the reasoning process and how *Computed Roles* can be realized by means of external functions.

Contents

Chapter 1

Introduction

How to compute $1 + 1$? Although simple in general, this tends to be a difficult question for a knowledge representation system. In the past, major emphasis in research on knowledge representation has been on theoretical aspects of representation formalisms. As a result, we know much about formalisms, but less about the construction of practical knowledge representation systems. This situation is changing, however, at least for terminological representation systems in the tradition of the KL-ONE representation language [4, 14] which include BACK [11], CLASSIC [5], and LOOM [6]. Based on a logical representation formalism, these systems represent knowledge in form of intensional descriptions. They are equipped with well-balanced reasoning mechanisms which allow to deduce implicit knowledge or to detect inconsistencies in a knowledge base. At the same time, however, terminological representation systems usually lack simple features which become important for real applications. In real life, the key-factors for success are usually not expressiveness or completeness of the representation formalism, but flexibility, efficiency and ease of use provided by the entire system.

A natural drawback of any terminological system is their limitation when it comes to knowledge which cannot be represented efficiently within their logic-based representation formalism. The adequate handling of rather mathematical objects[1] like numbers and temporal entities including time points and time intervals is a common problem for terminological systems. Since such entities are central to business and financial applications, the value of terminological systems which are not able to handle them adequately is seriously limited. Consider, for example, a company which wants to use a knowledge representation system for managing information about products, costs, prices and orders. A simple operation which might be needed here is to compute the amount of an order by multiplying the number of products by their price. With a database system or programming language, this kind of computation could easily be realized, for terminological representation systems, however, this task means a serious problem.

For that reason, some terminological systems such as K-REP [9] provide access to the implementation language by integrating some of its data types as built-in concepts. Other systems try to increase the expressiveness of their representation language. LOOM [6], for example, provides user definable predicates and computable relations as built-in language constructs which allow to represent some interesting properties of such crude entities like numbers and temporal objects within the representation formalism. Both solutions, however, have some drawbacks. The loose integration of data types in K-REP results in limited inferential capabilities. K-REP is not able to reason about such built-in concepts in the same manner as it is for user-definable concepts. In LOOM the various extension of the representation formalism result in incomplete reasoning algorithms where it is not always obvious which kind of inferences can be drawn.

In this paper, we will propose an extension of the terminological representation system BACK

[1] [1] refer to them as *concrete objects*.

[11, 12] which allows one to perform arbitrary computations (e.g. arithmetic computations on numbers) outside from the representation system and to integrate the results into the reasoning process of BACK. We do so by introducing *external functions* into the representation formalism as new representational primitives besides concepts, roles and objects. External functions allow the user to define functional dependencies between roles as part of concept definitions. For example, one might define a functional dependency between the three roles amount, number and price of some concept as amount = number * price. As a result, whenever fillers are known for number and price, the representation system can determine role-fillers for role amount by evaluating the function *.

The meaning of external functions is not built-in, but is provided by the user in form of a PROLOG procedure, i.e. procedurally. The user has the possibility to define and integrate functions into the representation system whenever it seems necessary. Functions have to be declared by the user before they can be used, similarly to functions in ordinary programming languages. This allows BACK to do type checking for arguments and values of functions, and to guarantee the consistency of the represented knowledge. In that way, we realize a well-defined interface for the integration of external functions into the representation formalism.

The paper is organized as follows. In the next section, we will define the representation formalism $\mathcal{L}_{EF}$ which is a subset of the BACK formalism extended by external functions. We will present examples for the usage of functions and a will give a semantics for the formalism which consists of a declarative and a procedural part. The later part is defined by a so-called value function which describes the evaluation process of external functions. In Section 3, we will present reasoning algorithms for our formalism. Besides the usual problems such as consistency checking and determining subsumption relations between concepts, we will explore the management of functional dependencies in the context of non-monotonic update operations on knowledge bases.

Chapter 2

The Representation Formalism: External Functions

2.1 Syntax

As the starting point for the design of our formalism we use a subset of the BACK representation language as described in [11]. As usual, the formalism allows for the definition of *concepts* and *roles* where concepts are meant to represent subsets of a domain and roles are binary relations. As third kind of entities we have *objects* which stand for single elements of the domain. In analogy to BACK, we allow objects to be used as role-fillers in concepts. A knowledge base consists of a set of *axioms* and *facts*. While axioms define concepts and roles, facts describe objects as instances of concepts or as role fillers of some role.

As a novelty for terminological systems, we introduce *external functions* in our formalism. On the one hand, we allow arbitrary functions to be declared by *function declarations*. On the other hand, we define role-fillers by functional expressions which apply a function to a non-empty list of arguments. The syntax of $\mathcal{L}_{EF}$ is given in Figures 2.2 and 2.3

Example 1: As an example consider the following scenario. A company produces a variety of products and sells them to a number of customers. Customers may receive a discount on the original sales price which depends on the amount they purchase. For each order there is a separate invoice issued. For sake of simplicity, we assume that at each time a customer orders just one kind of product. A representation of this example is shown in Figure 2.1.

The definitions in this example are grouped in three parts. In the first part, three functions are introduced by function declarations. These functions are used in the second part for the definition of concepts. The third part finally, defines several roles by specifying their domain and range.

The concepts in this example define a product as an entity which has exactly one price. An invoice is defined by seven roles which specify the customer who has ordered a product, the number s/he has ordered and the product itself. Furthermore, discount, gross-amount and net-amount are defined by means of the functions in dependency of some roles. □

Although simple, this example illustrates many points:

- A function declaration introduces two names for a function, an internal and an external name. Within the knowledge base, only the internal name is used.
- A function declaration defines types for each argument position and for the result.
- Functions are used in concept definitions to define a functional dependency between the arguments and a role.

*	:	Number × Number → Number **external** times
disc	:	Number × Number → Number **external** proc1
net	:	Number × Number → Number **external** proc2
Customer	:<	**anything**
Product	:<	**exactly**(1,price)
Invoice	:<	**exactly**(1,product) **and** **exactly**(1,number) **and** **exactly**(1,purchaser) **and** **exactly**(1,discount) **and** discount:disc(number) **and** **exactly**(1,gross-amount) **and** gross-amount:*(product.price,number) **exactly**(1,net-amount) **and** net-amount:net(gross-amount,discount)
price	:<	**domain**(Product) **and range**(Number)
purchaser	:<	**domain**(Invoice) **and range**(Customer)
product	:<	**domain**(Invoice) **and range**(Product)
number	:<	**domain**(Invoice) **and range**(Number)
discount	:<	**domain**(Invoice) **and range**(Number)
gross-amount	:<	**domain**(Invoice) **and range**(Number)
net-amount	:<	**domain**(Invoice) **and range**(Number)

Figure 2.1: Example Knowledge Base

- Arguments may be simple role-names such as number or discount as well as role-chains such as product.price. In addition, but not shown in the example, we allow arbitrary sets of objects as arguments.

- Functions are refered to by the internal name and the arity.

- A prefix notation is used for functions.

Unlike concepts or roles, external functions are not defined in a declarative manner. A knowledge base contains only function declarations, but no definitions for functions. Instead, external functions will be defined procedurally by some host language procedure which name is given by the external name of a function./footnoteSince /BACK is implemented in PROLOG, these host language procedures will actually be PROLOG predicates. Function declarations thus serve as interface which allow to integrate arbitrary external functions into the /BACK representation formalism.

In the following, we will call roles such as number and discount, which occur as arguments of a function, *argument roles*. On the contrary, roles like discount, which fillers are specified by means of functions, are refered to as *computed roles*. Finally, an expression of the form $F(A_1, \ldots, A_n)$ is named *functional expression* and a concept term of the form $R{:}F(A_1, \ldots, A_n)$ *functional term*.

Most interesting concerning functions are the logical consequences which follow from their usage. We illustrate this point with an example. After that, we will give a formal semantics for $\mathcal{L}_{EF}$ which reflects the intended meaning.

Example 2: Assume the following is part of the knowledge base of the company from Example 1 containing facts about products and invoices, formulated with respect to the above defined concepts and roles (let P_i and I_j denote products and invoices, respectively):

⟨*concept-term*⟩	::=	**anything** | **nothing**
	|	⟨*concept*-NAME⟩
	|	⟨*concept-term*⟩**and**⟨*concept-term*⟩
	|	**all**(⟨*role-term*⟩,⟨*concept-term*⟩)
	|	**atleast**(⟨*integer*⟩,⟨*role-term*⟩)
	|	**atmost**(⟨*integer*⟩,⟨*role-term*⟩)
	|	**exactly**(⟨*integer*⟩,⟨*role-term*⟩)
	|	⟨*role*-NAME⟩:⟨*filler*⟩$^+$
	|	⟨*role*-NAME⟩:⟨*function*-NAME⟩(⟨*argument*⟩$^+$)
⟨*role-term*⟩	::=	**anyrole**
	|	⟨*role*-NAME⟩
	|	**domain**(⟨*concept-term*⟩)
	|	**range**(⟨*concept-term*⟩)
	|	⟨*role-term*⟩.⟨*role-term*⟩
⟨*argument*⟩	::=	⟨*role*-NAME⟩
	|	⟨*role-term*⟩.⟨*role-term*⟩
	|	⟨*filler*⟩$^+$
⟨*filler*⟩	::=	⟨*object*-NAME⟩
	|	⟨*number*⟩

Figure 2.2: Concept and Role Terms

$(\mathsf{P}_{41},100) \in \mathsf{price}$
$(\mathsf{P}_{78},500) \in \mathsf{price}$
$(\mathsf{P}_{99},200) \in \mathsf{price}$

$(\mathsf{I}_1,\mathsf{P}_{41}) \in \mathsf{product}$
$(\mathsf{I}_1,10) \in \mathsf{number}$

$(\mathsf{I}_2,\mathsf{P}_{78}) \in \mathsf{product}$
$(\mathsf{I}_2,80) \in \mathsf{number}$

$(\mathsf{I}_3,200) \in \mathsf{number}$

$(\mathsf{I}_4,\mathsf{P}_{99}) \in \mathsf{product}$

With these facts, the price of three products, named P_{41}, P_{78} and P_{99} are given as 100\$, 500\$ and 200\$. Furthermore, the numbers and products of four invoices are listed. Note, that no product has been provided for I_3 and no number for I_4.

The functions disc and net specify the discount one receives and the net amount of an invoice. We take as granted that they have been defined as follows, ignoring the fact that we still don't know how we can define them:

$$disc(x) = \begin{cases} 0 & \text{if} \quad 0 \leq x < 50 \\ 5 & \text{if} \quad 50 \geq x < 100 \\ 10 & \text{if} \quad 100 \leq x \end{cases}$$

$$net(x,y) = x * \frac{(100 - y)}{100}$$

As expected, function * is assumed to be defined as arithmetic multiplication.
With these facts and the definitions of Example 1 at hand, we can determine role-fillers for the computed roles discount, gross-amount and net-amount as follows:

$$
\begin{array}{rcl}
\langle axiom \rangle & ::= & \langle concept\text{-}\textsc{name} \rangle :< \langle concept\text{-}term \rangle \\
& | & \langle role\text{-}\textsc{name} \rangle :< \langle role\text{-}term \rangle \\
\langle fact \rangle & ::= & \langle concept\text{-}\textsc{name} \rangle \in \langle object\text{-}\textsc{name} \rangle \\
& | & (\langle object\text{-}\textsc{name} \rangle, \langle object\text{-}\textsc{name} \rangle) \in \langle role\text{-}\textsc{name} \rangle \\
\langle declaration \rangle & ::= & \langle function\text{-}\textsc{name} \rangle : \langle concept\text{-}\textsc{name} \rangle \{ \times \langle concept\text{-}\textsc{name} \rangle \}^* \\
& & \quad \rightarrow \langle concept\text{-}\textsc{name} \rangle \\
& & \quad \textbf{external} \ \langle external\text{-}\textsc{name} \rangle
\end{array}
$$

Figure 2.3: Knowledge Base Entities

$(\mathsf{I}_1, 0) \in \mathsf{discount}$
$(\mathsf{I}_1, 1000) \in \mathsf{net\text{-}amount}$
$(\mathsf{I}_1, 1000) \in \mathsf{gross\text{-}amount}$

$(\mathsf{I}_2, 0.05) \in \mathsf{discount}$
$(\mathsf{I}_2, 4000) \in \mathsf{net\text{-}amount}$
$(\mathsf{I}_2, 3980) \in \mathsf{gross\text{-}amount}$

$(\mathsf{I}_3, 0.1) \in \mathsf{discount}$

□

Since the number of I_1 is between 0 and 50, the discount of I_1 is 0 in accordance to function disc. The gross-amount of I_1 and I_2 have been computed by $100 * 10 = 1000$ and $500 * 80 = 4000$. Similarly, the discount of I_2 is given as 5% by function disc from gross-amount of I_1.

Whereas fillers for all computed roles of I_3 and I_4 could be computed, we are not able to determine fillers for all roles of I_3 and I_4. The reason is that fillers of some argument roles involved at I_3 and I_4 are unknown what makes it impossible to evaluate any of the corresponding functions. For example, no product and therefore no price is known for I_3. Consequently, no value for function net can be determined at object I_3. We can state generally that no computation of function values can be performed as long as there is at least one argument role with unknown fillers.

2.2 Semantics

We now give a semantics for our formalism which reflects the intended logical consequences as described.

The semantics consists of two parts. The first one is a model-theoretic semantics which gives a declarative definition of the semantical account. This part is designed along the tradition of terminological representation formalisms. The second part deals with the evaluation of external functions. It defines an evaluation process for functions in form of an evaluation function which integrates results of an external evaluation process into the representation formalism. In this respect, the evaluation function is a significant supplement to the denotational semantics.

We use the following symbols to denote single terms and entities: C(concept-term), R(role-term), A(argument), O(object), F(function, internal name), X(function, external name).
The sets of all terms or all entities of a certain kind are denoted as follows: $\mathcal{C}$(concept-terms), $\mathcal{R}$(role-terms) and $\mathcal{A}$(arguments).
For the sets of all names of a certain sort we use the following symbols: $\mathcal{CN}$(concept-names), $\mathcal{RN}$(role-names), $\mathcal{O}$(object-names), $\mathcal{F}$ (function names, intern), $\mathcal{X}$ (function names, extern).

The model-theoretic semantics for $\mathcal{L}_{EF}$ is given as follows.

Definition 1: (Semantics)
The semantics of $\mathcal{L}_{EF}$ is given by the tripel $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}}, \text{VALUE} \rangle$, called *interpretation*. The interpretation consists of the domain $\Delta^{\mathcal{I}}$ which is an arbitrary set, an interpretation function $\cdot^{\mathcal{I}}$ and a value function VALUE. The interpretation function assigns to each concept $C \in \mathcal{C}$ a subset $C^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$, to each role $R \in \mathcal{R}$ a binary relation $R^{\mathcal{I}}$ and to each object $O \in \mathcal{O}$ an element $O^{\mathcal{I}}$ of $\Delta^{\mathcal{I}}$. The value function VALUE is a trinary function on functions, sets of arguments and objects ($\mathcal{F} \times \mathcal{A}^* \times \mathcal{O}$) which assigns to each tripel a set of objects $\text{VALUE}(F, A^*, O) \subseteq \mathcal{O}$.

The interpretation of complex concept and role terms is given by the following equations:

$$
\begin{aligned}
\textbf{anything}^{\mathcal{I}} &= \Delta^{\mathcal{I}} \\
\textbf{nothing}^{\mathcal{I}} &= \emptyset \\
(C_1 \textbf{ and } C_2)^{\mathcal{I}} &= C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}} \\
\textbf{all}(R, C)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} : \langle x, y \rangle \in R^{\mathcal{I}} \Rightarrow y \in C^{\mathcal{I}}\} \\
\textbf{atleast}(m, R)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} : |\{y : \langle x, y \rangle \in R^{\mathcal{I}}\}| \geq m\} \\
\textbf{atmost}(m, R)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} : |\{y : \langle x, y \rangle \in R^{\mathcal{I}}\}| \leq m\} \\
\textbf{exactly}(m, R)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} : |\{y : \langle x, y \rangle \in R^{\mathcal{I}}\}| = m\} \\
(R : O_1, \ldots, O_n)^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} : R^{\mathcal{I}}(x) = \{O_1^{\mathcal{I}}, \ldots, O_n^{\mathcal{I}}\}\} \\
(R : F(A_1, \ldots, A_n))^{\mathcal{I}} &= \{x \in \Delta^{\mathcal{I}} : R^{\mathcal{I}}(x) = \{O^{\mathcal{I}} | O \in \text{VALUE}(F, \{A_1, \ldots, A_n\}, x^{\mathcal{I}^{-1}})\}\} \\
\textbf{anyrole}^{\mathcal{I}} &= \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \\
\textbf{domain}(C))^{\mathcal{I}} &= C^{\mathcal{I}} \times \Delta^{\mathcal{I}} \\
\textbf{range}(C)^{\mathcal{I}} &= \Delta^{\mathcal{I}} \times C^{\mathcal{I}} \\
(R_1.R_2)^{\mathcal{I}} &= R_1^{\mathcal{I}} \circ R_2^{\mathcal{I}}
\end{aligned}
$$

An interpretation $\mathcal{I}$ is a *model* of a knowledge base if

$$
\begin{aligned}
C^{\mathcal{I}} &\subseteq \mathcal{D}^{\mathcal{I}} \quad \text{for all axioms} \quad C \text{¡}\mathcal{D} \\
R_1^{\mathcal{I}} &\subseteq R_2^{\mathcal{I}} \quad \text{for all axioms} \quad R_1 \text{¡} R_2 \\
O^{\mathcal{I}} &\in \mathcal{D}^{\mathcal{I}} \quad \text{for all facts} \quad O \in C \\
\langle O_1^{\mathcal{I}}, O_2^{\mathcal{I}} \rangle &\in R^{\mathcal{I}} \quad \text{for all facts} \quad (O_1, O_2) \in R
\end{aligned}
$$

□

The semantics for $\mathcal{L}_{EF}$ has been defined as usual except for the introduction of a value function for the definition of the denotation of functional terms. The value function is not fixed, but is provided by the user in form of external procedures which compute the value function. In that way, we realize an open user-definable semantics for external functions with a well-defined interface to the model-theoretic semantics of the rest of the formalismn.

In the following section, we will explore the nature of the value function in more detail. Before, we give some useful definitions.

Definition 2: A knowledge base is *consistent* if it has a model.

Definition 3: A concept C is *coherent* if $C^{\mathcal{I}} \neq \emptyset$ for all models.

Definition 4: A concept C_1 *subsumes* C_2 ($C_1 \succeq C_2$) if $C_1^{\mathcal{I}} \supseteq C_2^{\mathcal{I}}$ for all models.

Chapter 3

The Evaluation Function value

The evaluation function VALUE defines what is meant by the value of an external function. The evaluation function defines how external functions are evaluated in order to determine their functional value. Since external functions are defined by PROLOG procedures, determining the value requires an evaluation process part of which is calling these procedures and returning the function value. This evaluation process consists of two steps:

- In the first step, the arguments of a function, which are given as symbols and structures of the representation formalism, are transformed into structures on which the external procedures operate.
- In the second step, the Prolog procedure assigned to a function via declaration is called by the terminological system and the result value, if any, is passed back to the system.

Accordingly, we define the value function as follows.

Definition 5: (Value Function)
The value function VALUE is defined as follows:

$$\text{VALUE}: \mathcal{F} \times \mathcal{A}^* \times \mathcal{O} \rightarrow \mathcal{O}^*$$

$$\text{VALUE}(F, \{A_1, \ldots, A_n\}, O) = \text{CALL}(X_F, \{\text{TRANS}(A_1, O), \ldots, \text{TRANS}(A_n, O)\})$$

X_F is the name of the external procedure assigned to F via declaration and TRANS and CALL are functions which we will define below. □

The evaluation function VALUE serves as a syntactical frame for the evaluation process. There does not exist any a priori condition concerning its definition. The question for their definition is exclusively a question of practicability. In the following, we will present one specific solution to this problem which has been chosen to allow for a simple integration of the evaluation process in the representation system and to make it as easy as possible for the user to actually implement external functions. First, we will describe the transformation process of arguments which is defined by the function TRANS. Second, we present the calling process which is defined by the function CALL.

3.1 Parameter Passing

One central question concerning the realization of external procedures is on which kind of entities they should operate. Here, we have at least two possibilities:

- Arguments are passed to procedures as they occur in a knowledge base, i.e. as symbols and structures of the representation language.[1] Reconsider example 1 where fillers of the computed role discount are defined by the term discount:disc(number). In that case, we could pass the role-name number as a Prolog string or atom to the corresponding external procedure. The drawback of this solution is, however, that the external procedure has to know what to do with the role-name. Consequently, someone who wants to implement an external procedure has to have knowledge about the meaning of the representational primitives and how to use them properly. As a prerequisite, a kind of programming interface has to be provided by the terminological system in order to allow access to internal data structures. Although this might be useful for an experienced programmer, the user, who just wants to define a simple function such as disc from example 1, is surly not interested in handling maybe complex internal data structures for that task.

- As a consequence, we would like to vote for a solution which transforms arguments before Prolog procedures are called. For example, roles could be transformed into names of role-fillers and object-names into themself.

Correspondingly, we define the transformation of arguments as follows.

Definition 6: (Argument Transformation)
The transformation of arguments is given by a function TRANS: $\mathcal{O} \times \mathcal{A} \rightarrow \mathcal{O}^*$ which is defined as follows:

$$\text{TRANS}(A, O) = \begin{cases} A & \text{if} \quad A \in \mathcal{O} \\ \{x | (O, x) \in A\} & \text{if} \quad A \in \mathcal{RN} \end{cases}$$

The result of the transformation is called *value of the argument.* □

¿From now on, we use the term "parameter" whenever we refer to entities which are passed to external procedures and which are the result of the transformation of arguments. On the other side, we use the term "argument" for knowledge base entities which occur in the argument position of a functional expression. Remember that parameters are sets of objects.

So far, we have left open the definition of TRANS applied to role chains. We have done so, because unlike for simple roles there is no naturally given single set of objects which could be identified as argument value of a role-chain. Instead, we get one set of objects for each possible 'way' through a role chain as can be seen in the following example:

Example 3: Suppose a company represents information about employees and departments in the following way where roles have been defined as expected:

ave : Number → Number **external** average

Employee	:<	**all**(has-salary,Number)
Department	:<	**all**(employees,Employee) **and**
		all(average-salary,Number) **and**
		average-salary:ave(employees.has-salary)

We assume that the following facts are contained in the knowledge base (Let D be a department and E_i employees):

[1] This is the solution chosen in CLASSIC for test-predicates (see [13]).

(D,E_1) $\in$ employees
(D,E_2) $\in$ employees
(D,E_3) $\in$ employees
(D,E_4) $\in$ employees
(D,E_5) $\in$ employees

(E_1,5000) $\in$ has-salary
(E_2,4500) $\in$ has-salary
(E_3,6000) $\in$ has-salary
(E_4,5000) $\in$ has-salary
(E_5,4500) $\in$ has-salary

□

If the goal is to compute the average salary of all employees for each department, we could pass the value of the role-chain employees.has-salary in two different ways to the Prolog procedure 'average':

1. All objects are put together in one set. As a result, the parameter would be the set (i.e. a list if we use the Prolog notation) $\{5000, 4500, 6000, 5000, 4500\}$. In that case, procedure 'average' could easily compute the average salary as 5000.

2. Role-chains are transformed into sets of sets of objects grouping the objects according to the structure of role chains. Here, the parameter passed to the external procedure average is

   { {E_1, 5000}
   {E_2, 4500}
   {E_3, 6000}
   {E_4, 5000}
   {E_5, 4500} }

   In that case the procedure 'average' has to be adopted to the data structure in order to break it up properly.

   Although the additional information provided is a disadvantage if we just want to compute the average salary of a department, there are some cases where this additional information becomes important. Suppose the company wants to identify for each department the best earning employee. Here, the information which employee earns how much is significant.

3.2 Calling External Procedures

By now, we have made clear what kind of entities parameters of external procedures are. Before we can go on and can give examples of procedures for the functions introduced so far, we have to clarify the calling process itself. In order to make sure that both, the user who implements external procedures and the terminological system, agree on what is meant by the value of a function, we give a set of conditions which control the calling process as precisely as possible.

Let us first introduce a new term.

Definition 7: (Type)
Let R:$F(A_1, \ldots, A_n)$ be a functional term and $F : C_1 \times \ldots \times C_n \rightarrow C$ **external** X the corresponding function declaration. The type of argument A_i, denoted by TYPE(A_i), is defined to be the concept C_i. The type of function F, denoted by TYPE(F), is defined to be concept C.

Definition 8: (CALL)
The function CALL is defined as follows:

$$\text{CALL} : \mathcal{X} \times (\mathcal{O}^*)^* \rightarrow \mathcal{O}^*$$

On input (X, $\{\{O_{11},\ldots,O_{1m}\}$, ..., $\{O_{n1}$, ..., $O_{nk}\}\}$), function CALL calls the external Prolog procedure with name X. The result of CALL is a set of objects returned by procedure X as specified below if the following conditions are fulfilled:

(U1) X is a Prolog atom;

(U2) all object-names O_{ij} are Prolog atoms;

(U3) there exists a Prolog procedure with name X and arity $n+1$;

(S1) the i-th parameter ($i = 1..n$) is bound to the Prolog list containing objects O_{ij};

(S2) all objects contained in the list of (S1) are instances of TYPE(A_i) where A_i is the i-th argument

(S3) the $(n+1)$-th parameter is an unbound variable;

(U4) a procedure call respecting (S1) and (S2) never fails;

(U5) a procedure call respecting (S1) and (S2) always terminates;

(U6) on exit of procedure X, the $(n+1)$-th parameter is bound to a list of atoms;

(S4) all atoms contained in the list of (U6) are objects which are instances of TYPE(F).

The value of F is the list of object as mentioned in condition (S4). If any of these nine conditions is not fulfilled, function CALL returns the empty set. □

The definition of CALL states six conditions which must be fulfilled by the user (denoted U1-U6) and four conditions (S1-S4) which must be fulfilled by the terminological system. They are an important part of the semantic definition of the formalism $\mathcal{L}_{EF}$, since they control the semantical gap which necessarily has been left open in the model-theoretic semantics.

The most important conditions are (S2) and (S4). They require the system to do type checking before and after calling a Prolog procedure. On the one hand, each object which is passed to the external procedure must be an instance of the type declared for the corresponding argument position. On the other side, all objects returned as value must be instances of the result type of a function. However, as we will see in section 4 we can realize type checking more elegantly on the conceptual level.

Some remarks:

Note that so far nothing has been said about the point on that a procedure is called by the terminological system. In particular, the user cannot be sure that the procedure is invoked at all.

The evaluation mechanism as described does not allow to distinguish between an unsuccessful evaluation due to an unfulfilled condition and the empty set as result. If needed, one could change the definitions to cause the terminological system to signal an inconsistency whenever the evaluation process fails.

3.3 External Procedures: Examples

External functions are intended to give the user the possibility to extend the reasoning capabilities of the terminological system by introducing new functions just like new concepts or roles can

```
proc1( AMOUNT , DISCOUNT ) :-
          (   AMOUNT < 50, !, DISCOUNT = 0
          ;   AMOUNT <= 100, !, DISCOUNT = 5
          ;   AMOUNT > 100, !, DISCOUNT = 10
          ).

proc2( GROSSAMOUNT, DISCOUNT, NETAMOUNT ) :-
          H is (100 - DISCOUNT) / 100,
          NETAMOUNT is (GROSSAMOUNT * H).

times( NUMBER1, NUMBER2, RESULT ) :-
          RESULT is (NUMBER1 * NUMBER2).
```

Figure 3.1: PROLOG code for the procedures proc1, proc2 and times

be introduced into a knowledge base at any time. The realization of such an extension consists of two steps. First, a declaration has to be be formulated specifying the internal and external name as well as the type of each argument. This has been shown in the previous examples. Second, an external procedure has to be written which can be invoked by the system whenever the result value of a function has to be determined. The second step is facilitated enormously by the design of external functions:

- No knowledge about internal data structures is required since arguments are evaluated into extensional sets of object-names.
- Type checking is done by the system automatically before and after a procedure is called. As a result, the user is freed from writing error code which would be necessary if arbitrary data structures could be passed to a procedure.

Both points together allow to restrict the code of external procedures to almost purely definitional knowledge and prevents the user from doing a lot of awful work.

Prolog code for procedures proc1, proc2 and times from example 1 are shown in figure 3.1. As can be seen, the knowledge required to write external procedures is restricted to some working knowledge of the programming language Prolog. Nothing has to be known about internal data structures of the terminological representation system.

Chapter 4

Reasoning Algorithms

In this section, we will present reasoning algorithms for the formalism $\mathcal{L}_{EF}$. Our goal is not to develop new algorithms from the scratch, but to use the algorithms realized in BACK as far as possible.

In general, inferences drawn by a terminological system can be described by a set of reasoning tasks such as classification, recognition or retrieval. In almost all implemented systems, however, these reasoning tasks are combined to interface operations for the interaction with a knowledge base. The BACK system, for example, provides an operation for adding axioms and facts to a knowledge base (TELL) and an operation for querying knowledge bases (ASK). Furthermore, a FORGET operation allows one to retract facts which have been inserted via TELL.

For the description of our algorithms we take as granted that knowledge base operations are provided for an representation formalism without external functions as shown in figure 4.1. We will refer to these operations by BACKTELL and BACKFORGET.

4.1 Coherence of Functional Terms

One of the most important tasks of a terminological system is to check the consistency of a knowledge base. In general, for representation formalisms with a carefully restricted expressiveness, there exist complete algorithms for consistency checking. For the formalism $\mathcal{L}_{EF}$, however, there does not exist such an algorithm, since we have a procedural semantics for functional terms which does not provide a complete intensional description of the objects which are instances of a functional term. Thus, for a given concept which is defined by means of a functional term, we cannot completely rule out the possibility that it denotes the empty set, i.e. that there cannot exist an instance of that concept. However, for a given object we can always verify if it is an instance of that concept or not. This is due to the fact that in this case we do not have to consider all possible instances of a concept, but only one specific object.

The introduction of declarations for external functions gives us the opportunity to check the validity of its arguments. Concerning a function declaration $F : C_1 \times \ldots \times C_n \rightarrow C$ **external** X and a functional term $R{:}F(A_1, \ldots, A_n)$ we can identify the following type conditions (let $\textsc{range}(R)$ denote the range concept of role R):

1. $A_i \in \textsc{type}(A_i)$ for all $A_i \in \mathcal{O}$
2. $\textsc{range}(A_i) \preceq \textsc{type}(A_i)$ for all $A_i \in \mathcal{R}$
3. $\textsc{type}(F) \preceq \textsc{range}(R)$
4. $\textsc{value}(F, \{A_1, \ldots, A_n\}, O) \in \textsc{type}(F)$ for all $O \in \mathcal{O}$

⟨*kb-operation*⟩	::=	⟨*tell*⟩ | ⟨*forget*⟩ | ⟨*ask*⟩
⟨*tell*⟩	::=	**tell**(⟨*axiom*⟩)
	|	**tell**(⟨*fact*⟩)
	|	**tell**(⟨*declaration*⟩)
⟨*forget*⟩	::=	**forget**(⟨*fact*⟩)
⟨*ask*⟩	::=	**subsumes**(⟨*concept*-NAME⟩,⟨*concept*-NAME⟩)
	|	**holds**(⟨*fact*⟩)

Figure 4.1: Knowledge Base Operations

Condition 1 states that an object which occurs as an argument has to be an instance of the argument type. Similarly, condition 2 formulates that the range of an argument role must be subsumed by the argument type. These conditions ensure that all parameters passed to the external procedure are instances the argument type and therefore define a type checking for arguments. Conditions 3 and 4 express similar type constraints for the result value of a function. Whereas conditions 1 to 3 can be checked before evaluation of an external term, condition 4 can only be verified after the value of a functional term has been determined. As result, we define two properties of functional terms as follows:

Definition 9: (Applicability of Functional Terms)
A functional term which satisfies conditions 1 to 3 is called *applicable*.

Definition 10: (Validity of Functional Terms)
A functional term which satisfies condition 1 to 4 is called *valid*.

Obviously, the notions of applicability and validity are weaker than the notion of coherence.

In the following, we will call the process of checking the validity of external terms *type checking*. As can be seen, type checking for functional terms is reducible to the subsumption problem in linear time.

As a conclusion, let us mention an important difference between axioms and facts on the one hand and declarations on the other hand. Whereas information from declarations is only used for type checking, i.e. to make sure that certain conditions hold for arguments of a functional term, information from axioms and facts is used for inferences and is therefore propagated throughout the knowledge base. If, for example, an object is used as a role-filler of a certain role, it can be deduced that this object is an instance of the range concept of the role. This information is then added to the description of the object. On the contrary, an object as argument of a functional term is only checked against the corresponding type, but this type is not added to its description.

4.2 Subsumption

The subsumption problem is strongly connected to the consistency problem. Consequently, for determining subsumption relations between concepts we have similar problems as for consistency checking. Any algorithm for subsumption in our formalism is necessarily incomplete due to a missing intensional description of instances of functional terms. As a small step, we can identify the following subsumption relations concerning functional terms by reformulating conditions 2 and 3 from Section 3.1:

- $\mathbf{all}(R,\text{TYPE}(F)) \succeq R{:}F(A_1,\ldots,A_n)$
- $\mathbf{all}(A_i,\text{TYPE}(A_i)) \succeq R{:}F(A_1,\ldots,A_n)$ for all $i = 1..n$

- $R{:}F(A_1, \ldots, A_n) \succeq \textbf{atleast}(1, A_i)$ for all $i = 1..n$.

Conditions 1 and 4 which states type constraints for objects does not have an effect on subsumption.

We propose that the procedures which check consistency and determine subsumption relations should be adopted in order to work in accordance with the identified rules. In the remainder, we will assume that the knowledge base operations BACKTELL and BACKFORGET have been changed accordingly.

Although one might consider the incompleteness of our algorithms as a serious problem, we think it is not. The reason why we want to introduce external functions into BACK lies in the possibility to do simple computations on the assertional level as shown in Section 1. For most real applications of the BACK knowledge representation system there is no need for conceptual reasoning based on external functions.

4.3 Managing Dependencies

Functional terms define dependencies between the arguments and its value. If an argument is changed in some way, for example by using the FORGET-operation, the function value might change as well. Consequently, whenever new information is known about an entity used as an argument somewhere in a knowledge base or information is deleted, the value of the corresponding function has to be reevaluated. In some cases, rechecking of validity might be enough. In general, we cannot expect that a monotonic update operation on an argument leads to a monotonic update operation of the corresponding value. Reconsider example 3 where the concept Department has been defined including a computed role average-salary which gives the average salary of all employees working in the department. Whenever the salary of at least one employee is changed or a new employee is added to an department or an employee is deleted from an department, the average salary has to be recomputed and stored as the new role-filler of role average-salary. This operation is clearly non-monotonic.

In order to record dependencies of the various kinds we use following definitions:

Definition 11: (Role Dependency)
Let $C :< \ldots R{:}F(A_1,\ldots,A_n)\ldots$ be a concept defined by means of a functional term. We say *role R at concept C depends on Role R_i by function F* iff VALUE($F(A_1, \ldots, A_n)$,O) might change for any $O \in C$ if the extension of $R_i(C)$ changes. We write *DependsOnRole*(R, R_i, F, C). □

Definition 12: (Object Dependency)
Let $C :< \ldots R{:}F(A_1,\ldots,A_n)\ldots$ be a concept defined by means of a functional term. We say *role R at concept C depends on Object O_i by function F* iff VALUE($F(A_1, \ldots, A_n)$, O) might change if the description of O is changed. We write *DependsOnObject*(R, O, F, C). □

Definition 13: (Ordering Relation for Concepts)
We say *role R_1 is before role R_2 at concept C* iff role-fillers of R_1 have to be known before role-fillers of R_2 can be determined for all instances of C. We write *Before*(R_1, R_2, C). □

Definition 14: (Ordering Relation for Objects)
We say *role R_1 is before role R_2 at object O* iff role-fillers of R_1 for O have to be known before role-fillers of R_2 for O can be determined. We write *Before*(R_1, R_2, O). □

With these dependencies at hand, we can identify all functions which might be affected by an update operation and which thus have to be reevaluated.

We now give algorithms which compute the above introduced dependencies. The first algorithm computes all conceptual dependencies.

Algorithm 1: (CONCEPT-DEPENDENCY)
The procedure CONCEPT-DEPENDENCY takes a concept definition ($C :< \mathcal{D}$) as input and computes all functional dependencies implied by the definition according to the following rules. The rules are applied recursively, until no further rule is applicable:

(C1) $\mathcal{D}$ contains a functional subterm $R_1{:}F(A_1, \ldots, R_2, \ldots, A_n)$ where role R_2 is used as argument of function F
$\longrightarrow$ *DependsOnRole*(R_1, R_2, F, C)

(C2) $\mathcal{D}$ contains a functional subterm $R_1{:}F(A_1, \ldots, O, \ldots, A_n)$ where object O is used as argument of function F
$\longrightarrow$ *DependsOnObject*(R_1, O, F, C)

(C3) $\mathcal{D}$ contains a functional subterm $R_1{:}F(A_1, \ldots, RC, \ldots, A_n)$ where a role-chain $RC = R_1..R_i..R_n$ is used as argument
$\longrightarrow$ *DependsOnRole*(R, R_i, F, C)

(C4) *DependsOnRole*(R_1, R_2, F, C_1) and $C_2 \preceq C_1$
$\longrightarrow$ *DependsOnRole*(R_1, R_2, F, C_2)

(C5) *DependsOnObject*(R, O, F, C_1) and $C_2 \preceq C_1$
$\longrightarrow$ *DependsOnObject*(R, O, F, C_2)

(C6) *DependsOnRole*(R_1, R_2, F, C) and R_2 is a computed role
$\longrightarrow$ *Before*$(R_1, R_2, C\ )$

(C7) *Before*$(R_1, R_2, C\ )$ and *Before*$(R_2, R_3, C\ )$
$\longrightarrow$ *Before*$(R_1, R_3, C\ )$

□

The first two rules cover the simplest functional dependencies where a role directly depends on the arguments of the corresponding function. Rule (C3) does similar for role-chains. Rules (C4) and (C5) realize downward inheritance of dependencies. Rules (C6) simply transforms a functional dependency into an ordering relation between computed roles. Rule (C7) finally, computes the transitive closure of the ordering relation.

Cyclic dependencies occurring in a knowledge base form a serious problem. A cyclic dependency exists if role-fillers of one role have to be known in order to determine role-fillers of another role which, again, depends on the fillers of the first role. A simple example is given by $C :< R_1{:}F_1(R_2)$ **and** $R_2{:}F_2(R_1)$. Clearly, algorithm 1 loops if cyclic dependencies between roles occur. In general, we require the dependency relation to be cycle-free and assume that the algorithm has been extended accordingly.

The next algorithm we present computes dependency relations concerning objects under the assumption that conceptual dependencies already have been computed:

Algorithm 2: (OBJECT-DEPENDENCY)
The procedure OBJECT-DEPENDENCY takes an object description $O \in C$ as input and computes the functional dependencies which hold for object O. The computation is done according to the following rules (after the recognition process):

(O1) *DependsOnRole*(R_1, R_2, F, C) and $O \in C$
$\longrightarrow$ *DependsOnRole*(R_1, R_2, F, O)

(O2) *DependsOnObject*(R_1, O_1, F, C) and $O_2 \in C$
$\longrightarrow$ *DependsOnObject*(R, O_1, F, O_2)

(O3) *Before*(R_1, R_2, C) and $O \in C$
$\longrightarrow$ *Before*(R_1, R_2, O)

(O4) *DependsOnRole*(R_1, R_2, F, O_2) and $O_1 \in$ TRANS(R_2, O_2)
$\longrightarrow$ *DependsOnObject*(R_1, O_1, F, O_2)

□

Rules (O1) to (O3) simply transfer all dependency information from concepts to their instances. Rule (O4) covers the case that an object is not directly used as argument but is computed during the transformation of arguments into parameters. As parameter, it clearly affects the value of a function.

We now present algorithms which use the dependencies whenever a new fact is introduced into a knowledge base or is retracted from it. In the first case, new information is deduced by evaluating functions for which new information is known about one of its arguments. In the second case, already computed function values are retracted which are no longer valid since the description of one of its arguments has changed. The first algorithm is invoked whenever an object is described as instance of a concept.

Algorithm 3: (PROPAGATE-INSTANCE)
The function PROPAGATE-INSTANCE is defined as:

$$\text{PROPAGATE-INSTANCE}: \mathcal{O} \times \mathcal{CN} \rightarrow \{true, false\}$$

It takes a fact $O \in C$ as input, stores O as instance of C and propagates this information to all objects with functions whose value depends on O. The result of PROPAGATE-INSTANCE is *false*, if an inconsistency has been detected during propagation, otherwise *true*.

function PROPAGATE-INSTANCE($O_1 \in C$) **is**
$M_1 := \{(O_2, R_1, F) | DependsOnRole(R_1, R_2, F, O_2) \wedge C \preceq \textbf{range}(R_2)\}$
for all $(O_2, R_1, F) \in M_1$ **do**
M_2:= VALUE(F,O_2)
for all $O_3 \in M_2$ **do**
B := PROPAGATE-FILLER((O_2,O_3) $\in$ R_1)
if B = *false* **then return** *false*
end
end
B:=BACKTELL($O_1 \in C$)
return B

□

The algorithm PROPAGATE-FILLER used in algorithm PROPAGATE-INSTANCE is presented in the following. It is invoked whenever a new role-filler is defined for a role. Since this might affect the result of the transformation of the role if used as an argument, we have to reevaluate the corresponding functions.

Algorithm 4: (PROPAGATE-FILLER)
The function PROPAGATE-FILLER is defined as:

$$\text{PROPAGATE-FILLER}: \mathcal{O} \times \mathcal{O} \times \mathcal{RN} \rightarrow \{true, false\}$$

PROPAGATE-FILLER takes a relational fact $(O_1,O_2) \in R$ as input, stores the fact in the knowledge base and propagates this information recursively to all objects which have a function depending on O_2. The result of PROPAGATE-FILLER is *false*, if an inconsistency has been detected during propagation, otherwise *true*.

function PROPAGATE-FILLER($(O_1,O_2) \in R$) **is**
$M_1 := \{(R_1, F) | DependsOnRole(R_1, R, F, O_1)\}$
for all $(R_1, F) \in M_1$ **do**
    M_2:= VALUE(F,O_1)
    for all $O_3 \in M_2$ **do**
        B := PROPAGATE-FILLER($(O_1,O_3) \in R_1$)
        if B = *false* **then return** *false*
    end
end
B:=BACKTELL($(O_1, O_2) \in R$)
return B

□

The next algorithm presented retracts information which is no longer valid whenever a role-filler is deleted from a role.

Algorithm 5: (FORGET-FILLER)
The procedure FORGET-FILLER is defined on $\mathcal{O} \times \mathcal{O} \times \mathcal{RN}$. It takes a relational fact $(O_1,O_2) \in R$ as input, deletes O_2 as role-filler of role R at object O_1 ($(O_1, O_2) \notin R$)) and propagates this information to all objects where O_2 has been used to compute a role-filler. This role-filler is then recursively deleted.

procedure FORGET-FILLER($(O_1,O_2) \in R$) **is**
BACKFORGET($(O_1,O_2) \in R$)
$M_1 := \{R_1 | DependsOnRole(R_1, R, F, O_1)\}$
for all $R_1 \in M_1$ **do**
    M_2:= $\{O_3 | (O_1, O_3) \in R_1\}$
    for all $O_3 \in M_2$ **do**
        B := FORGET-FILLER($(O_1,O_3) \in R_1$)
    end
end

□

Algorithm 6: (FORGET-OBJECT)
The procedure FORGET-OBJECT is defined on the set $\mathcal{O}$ of all objects. On input O, it deletes O from the KB and propagates this information to all objects which have used O to determine some of their role-fillers. These role-fillers are then recursively deleted.

procedure FORGET-OBJECT(O) **is**
BACKFORGET(O)
$M_1 := \{(O_2, R, F) | DependsOnObject(R, O, F, O_2)\}$

$M_2 := \{(O_2, O_3, R) | (O_2, R) \in M_1 \wedge (O_2, O_3) \in R\}$
for all $(O_2, O_3, R) \in M_2$ **do**
FORGET-FILLER(O_2,O_3,R)
end

□

4.4 Knowledge Base Operations

As a conclusion, we will now put together all algorithms presented so far and gives algorithms for the TELL- and FORGET-operations for our formalism.

Algorithm 7: (TELL)

procedure TELL($C :< \mathcal{D}$) **is**
1. Check validity of concept term $\mathcal{D}$
2. compute the functional dependencies using algorithm CONCEPT-DEPENDENCY
3. insert the axiom into the knowledge base using BACKTELL($C :< \mathcal{D}$)

procedure TELL($R_1 :< R_2$) **is**
1. BACKTELL($R_1 :< R_2$)

procedure TELL($O \in C$) **is**
1. compute the functional dependencies using algorithm OBJECT-DEPENDENCY
2. insert the fact into the knowledge base using PROPAGATE-INSTANCE($O \in C$)

procedure TELL((O_1,O_2) $\in R$) **is**
1. insert the fact into the knowledge base using PROPAGATE-FILLER((O_1,O_2) $\in R$)

□

Algorithm 8: (FORGET)

procedure FORGET($O \in C$) **is**
1. delete the fact from the knowledge base using FORGET-OBJECT($O \in C$)

procedure FORGET((O_1,O_2) $\in R$) **is**
1. delete the fact from the knowledge base using FORGET-FILLER((O_1,O_2) $\in R$)

□

Chapter 5

Conclusion

In this paper, we have shown a solution to a problem of which almost all terminological representation systems suffer. By using a declarative logic-based approach there seem to be always some sort of knowledge which cannot adequately be represented in a given representation formalism either because of efficiency or expressiveness reasons. Our proposed extension of the BACK formalism consists therefore of an interface which allows one in a well-defined manner to add external procedures and to integrate their results in the reasoning process. We see the most important advantages of our approach as follows.

- The proposed integration of external functions in the BACK formalism allows the efficient realization of computed-roles. By accessing computational resources provided by the implementation language of BACK, we can now handle entities like numbers, strings, time points and time intervals more adequately than before.
- By realizing external functions as add-on language constructs, we get an easily extensible representation system. A set of built-in functions always tends to be insufficient for a given application or results in unnecessary system overhead.
- The use of function declarations allows a significant amount of reasoning to be done by BACK. On the one hand, type-checking allows to verify the validity of arguments and of the computed functional value. On the other hand, additional subsumption rules realize some form of intensional reasoning about external functions.
- The proposed extension of the BACK formalism, that is the evaluation process as defined by the evaluation function CALL, is easy to integrate into the existing BACK system. As shown in Section 4, existing algorithms for knowledge base operations such as TELL and ASK can be reused in order to implement similar algorithms for the extended formalism. No major changes are needed.
- The choosen interface between the representation system and PROLOG procedures makes the writing of these procedures extremely easy. Although a procedural way of representing knowledge, the PROLOG procedures allow an almost declarative style for the definition of external functions. This is especially important, since the user is not required to know anything about system internal data structures. External functions can thus not only be implemented by the designers of BACK, but by everybody who has some knowledge about PROLOG.

Our approach is clearly limited, since we have restricted our attention to functions, that is relations between concepts, opposed to concepts itself. We belief that a similar approach is feasible for the realization of a more advanced version of test-concepts, a way of defining concepts

procedurally which is used in CLASSIC. Obviously, the here proposed extensions of the BACK representation formalism are not purely declarative and might therefore not fully satisfying if one seeks a pure declarative solution. However, as argued at the beginning, the key for success or failure of terminological representation systems lies not necessarily in logical properties of the used formalisms and algorithms, but in architectural features of the entire system.

References

[1] F. Baader, P. Hanschke. *A Schema for Integrating Concrete Domains into Concept Languages.* DFKI Research Report RR-91-10, DFKI, Postfach 2080, W-6750 Kaiserslautern, Germany. Proceedings of IJCAI '91.

[2] F. Baader et al, *Terminological Knowledge Representation: A Proposal for a Terminological Logic,* in B. Nebel, C. Peltason, and K. von Luck (eds.), *International Workshop on Terminological Logics*, KIT Report 89, Technische Universität Berlin, pp. 120–128, Aug. 1991.

[3] R. Backofen and L. Euler, *Towards the Integration of Functions, Relations and Types in an AI Programming Language* in DFKI Research Report, Kaiserslautern, 1991.

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

[5] R. J. Brachman et al, *Living with Classic: When and How to Use a Kl-One-like Language* in: John Sowa (Ed.) *Principles of Semantic Networks*, 401-456, 1991.

[6] R. MacGregor, R. Bates. *The* LOOM *Knowledge Representation Language.* Report ISI/Representation System-87-188, USC/Information Science Institute, Marina del Ray, 1987.

[7] B. Hollunder, W. Nutt. *Subsumption Algorithms for Concept Languages.* DFKI Research Report RR-90-04, DFKI, Postfach 2080, W-6750 Kaiserslautern, Germany.

[8] C. Kindermann. *Retraction of Object Descriptions in* BACK, Proc. ECAI 92

[9] E. Mays et al, *Organizing Knowledge in a Complex Financial Domain* IEEE Expert, Vol. 2(3), 1987.

[10] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*, Lecture Notes in Artificial Intelligence, LNAI 422, Springer Verlag, 1990.

[11] Ch. Peltason, A. Schmiedel, J. Quantz and C. Kindermann, *The* BACK*-System Revisited* KIT Report 75, TU Berlin, Sep. 1989.

[12] J. Quantz and C. Kindermann, *Implementation of the* BACK *System Version 4,* KIT Report 78, TU Berlin, Dec. 1990.

[13] L. A. Resnick et al, CLASSIC*Description and Reference Manual For the COMMON LISP Implementation Version 1.1, The CLASSIC Group, AT&T Bell Labs*, January 1991.

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

[15] A. Schmiedel, *The Lower End of* BACK*: Attached Data and Procedures,* Internal working paper, TU Berlin, May 1991.