# $E_X$*Claim*: a hybrid language for knowledge representation and reasoning using description logics

**Liviu Badea**
AI Research Department
Research Institute for Informatics
8-10 Averescu Blvd., Bucharest, Romania
e-mail: `badea@roearn.ici.ro`

**Abstract.**
This paper presents $E_X$*Claim*, a hybrid language for knowledge representation and reasoning. Originally developed as an operationalization language for the KADS knowledge based systems development methodology, $E_X$*Claim* has a meta-level architecture: it structures the knowledge on three levels, namely the domain, inference and task level. An extension of a description logic is used for implementing the domain level. The inference and task levels are procedural and support non-determinism (inferences and tasks being backtrackable). This in turn requires a non-monotonic domain level.

Description logics offer a set of inference services (not available in other KR languages) which are extremely useful in knowledge modelling. Such inference services include domain-level deduction, semantic consistency verification and automatic classification of concepts. Note that most of the existing KBS development tools and environments do not provide any facilities for model consistency or completeness checking. We argue that such validation and verification facilities are extremely important in assisting a knowledge engineer in developing models. In fact, these might be the main facilities a user would expect from a computer-aided KBS development tool.

The final goal of this research is to use $E_X$*Claim* for developing real-world applications and to demonstrate the usefulness of the knowledge level simulation/execution facilities offered by $E_X$*Claim* in KBS development.

## 1 Introduction

This introductory section describes the present state in knowledge based systems development tools and then gives a brief description of our approach.

Knowledge based systems (KBS) are typically large and complex software systems aiming at solving difficult problems in knowledge-intensive domains. *Knowledge engineering* in general and KBS development in particular are notoriously difficult not only because of the sheer size of the problem description, but also because they typically involve complex ontologies, which are usually not easily representable in a single knowledge representation formalism. The lack of well-established algorithms in such complex domains leads to *computational problems* due mainly to the large search spaces involved. The process of *knowledge acquisition* is also more difficult than in other cases, but viewing it as a process of knowledge modelling rather than a process of knowledge transfer (from the expert to the machine) helps to alleviate some of these problems.

In order to assist the knowledge engineer in developing KBSs, a large number of KBS development tools have been built since the eighties. Two main tendencies were followed in the early years.

On one hand, a great number of expert system "shells" were put forward. Systems like KEE, ART, Knowledge Craft, Nexpert Object etc. were successfully used in building a large number of expert systems. These "shells", however, had an important drawback: they used a given symbol-level representation (for instance a frame-based system augmented with rules, daemons, message passing, etc.), which is usually not appropriate for describing reusable knowledge-level models.

An alternative approach to building KBS development tools was inspired by the traditional software engineering (SE) tools. SE tools are nevertheless inappropriate as KBS tools since the domain knowledge (the ontology) is much more complex in the case of a KBS than in the case of a typical software system.

The remarks above suggest the need for a knowledge-level KBS development tool that would provide at least some of the nice simulation facilities offered by traditional SE tools. Such facilities are much harder to develop in the case of KBSs, since, as already mentioned above, we are dealing with much more complex domain knowledge. An extreme approach would be to use full predicate logic as a domain description language and to support the reasoning involved with a full first order logic theorem prover. This approach (followed in the FML component [1] of the CommonKADS Workbench) can be very inefficient in complex cases. Also, the readability of model specifications may sometimes be quite low, especially when dealing with complex logic formulae.

In order to support knowledge-level knowledge modelling, a series of methodologies and specification languages have been put forward, the most important ones being the KADS methodology [18] in Europe, KIF and Ontolingua in the US.

In order to support the KADS methodology with executable tools, a number of KADS operationalization languages and environments have been developed: Si(ML)$^2$/FML, OMOS, MoMo, KARL, MODEL-K, FORKADS [8] etc. Most of these languages are either very expressive, formally sound but computationally inefficient (sometimes even intractable), or they have a more procedural semantics, being less expressive, but more tractable.

This paper proposes using a class of knowledge representation languages, namely the description logics, for both formalising and executing CommonKADS expertise models. The following advantages of this approach can be mentioned.

First, description logics represent the formalism closest to the KADS domain level. In fact, the design of the KADS domain level was obviously inspired by the KL-ONE like languages [4].

Second, description logics offer a set of inference services (not available in other KR languages) which are extremely useful in knowledge modelling. Such inference services include domain-level deduction, semantic consistency verification and automatic classification of concepts. Note that most of the existing KBS development tools and environments do not provide any help for model consistency/completeness checking. The issue of verifying (KADS) models has not been extensively addressed, mainly due to the computational difficulties involved. However, such validation and verification facilities are extremely important in assisting a knowledge engineer in building models. In fact, these might be among the *very first* facilities a user would expect from a computer-aided KBS development tool.

The $E_x$*Claim* knowledge modelling environment, presented in this paper, is particularly interesting since it preserves its runtime efficiency in spite of the fact that it uses a formal knowledge representation language[1] at the domain level. It may therefore be regarded as a reasonable trade-off between expressiveness, readability and efficiency.

## 1.1 Brief description of $E_x$*Claim*

$E_x$*Claim* (Executable CommonKADS Language for Integrated Modelling) is a knowledge modelling environment (based on the CommonKADS methodology [19]) that adds operationalization features to the CommonKADS Workbench developed in the KADS-II project. $E_x$*Claim* is a logic-based language (with a meta-level architecture and supporting non-determinism) for describing and executing KADS models. The process of knowledge engineering is thus improved by the model simulation/execution facilities offered by the system.

The domain knowledge is extended with a description (terminological) logic which provides fairly sophisticated inference services (such as domain-level deduction, semantic consistency checking, automatic classification of concepts, knowledge structuring and indexing).

The system has the meta-level architecture specific to KADS: the knowledge is structured on three levels, namely the domain, inference and task knowledge. Since the inference level refers the domain only indirectly via inference roles, the models are highly reusable.

$E_x$*Claim* supports non-deterministic reasoning, its domain operations, inferences and tasks being backtrackable. Note that, as opposed to traditional software engineering tools, non-determinism is very useful in KBSs.

$E_x$*Claim* is fully integrated in the CommonKADS Workbench and takes advantage of its nice graphical user interface. The XPCE object oriented environment [24] built on top of SWI-Prolog has been used as the basic implementation platform.

## 2 The domain level

The $E_x$*Claim* domain level is an extension of a description logic (DL). A brief outline of the DL used at this time in $E_x$*Claim* will be given in the present section. Note however, that due to the modularity of the system, any DL implementation that supports knowledge revision can be used in $E_x$*Claim*.

## 2.1 The description logic

Description logics[2](DLs) are descendants of the famous KL-ONE language [4] and can be viewed as formalizations of the frame-based knowledge representation systems. They are also related to other knowledge representation formalisms like semantic networks and object-oriented languages and thus are very useful for representing taxonomic knowledge.

Terminological knowledge representation systems are hybrid systems which separate the described knowledge in two categories: terminological and assertional knowledge. The terminological knowledge is generic and refers to classes of objects and their relationships (being stored in the so called TBox), while the assertional knowledge describes particular instances, or individuals (which are stored in the ABox).

The terminological language provides a concept description language which uses two kinds of terminological knowledge, namely *concepts* and *roles*.[3] Concepts are unary predicates interpreted as sets of individuals, whereas roles represent binary predicates interpreted as binary relations between individuals.

The main advantages of DLs w.r.t. other knowledge representation formalisms are:

- DLs have a formally defined declarative semantics, i.e. the meaning of the constructions used is not described operationally (for example, by an implementation), but in terms of admissible models (in the sense of model theory in formal logic).
- The inference services offered by a DL can be described by means of its declarative semantics.
- The theoretical analysis of the inference mechanisms is possible. The following properties are analysed:

  1. *Correctness* (facts provable by inference are semantically valid).
  2. *Completeness* (all valid facts are provable).
  3. *Decidability* and *complexity* of the algorithms.

[1] a description logic, which, although less expressive than full first order logic, is more efficient

[2] Also known as terminological logics, or term subsumption languages.

[3] DL roles should not be confused with KADS inference roles.

### 2.1.1 Concept and role constructors. Declarative semantics

The ontological primitives of a DL include concepts, roles, attributes (functional roles) and individuals.[4] In the case of individuals, we make the *"unique names assumption"*, which is quite frequent in the field of databases and knowledge bases.

The *concept constructors* of the description logic used in $E_x$*Claim* are presented in Table 1, whereas the relevant *role and attribute constructors* are presented in Table 2.

| Concept | Symbolic | Semantic interpretation |
|---|---|---|
| *top* | $\top$ | $\Delta^{\mathcal{I}}$ |
| *bottom* | $\bot$ | $\emptyset$ |
| $and(C_1, C_2)$ | $C_1 \wedge C_2$ | $C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$ |
| $or(C_1, C_2)$ | $C_1 \vee C_2$ | $C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$ |
| $not(C)$ | $\neg C$ | $\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$ |
| $all(R, C)$ | $\forall R{:}C$ | $\{x \in \Delta^{\mathcal{I}} \| \forall y.(x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$ |
| | | $\{x \in \Delta^{\mathcal{I}} \| R^{\mathcal{I}}(x) \subset C^{\mathcal{I}}\}$ |
| $exists(R, C)$ | $\exists R{:}C$ | $\{x \in \Delta^{\mathcal{I}} \| \exists y.(x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$ |
| | | $\{x \in \Delta^{\mathcal{I}} \| R^{\mathcal{I}}(x) \cap C^{\mathcal{I}} \neq \emptyset\}$ |
| $atleast(n, R)$ | $\geq_n R$ | $\{x \in \Delta^{\mathcal{I}} \| \|R^{\mathcal{I}}(x)\| \geq n\}$ |
| $atmost(n, R)$ | $\leq_n R$ | $\{x \in \Delta^{\mathcal{I}} \| \|R^{\mathcal{I}}(x)\| \leq n\}$ |
| $exactly(n, R)$ | $=_n R$ | $\{x \in \Delta^{\mathcal{I}} \| \|R^{\mathcal{I}}(x)\| = n\}$ |

**Table 1.** Concept constructors and their semantics

| Role | Symbolic | Semantic interpretation |
|---|---|---|
| $and(R_1, R_2)$ | $R_1 \wedge R_2$ | $R_1^{\mathcal{I}} \cap R_2^{\mathcal{I}}$ |
| $inv(R)$ | $R^{-1}$ | $\{(y,x) \| (x,y) \in R^{\mathcal{I}}\}$ |
| $domrestr(R, C)$ | $C \lfloor R$ | $\{(x,y) \| (x,y) \in R^{\mathcal{I}} \wedge x \in C^{\mathcal{I}}\}$ |
| $restrict(R, C)$ | $R \rfloor C$ | $\{(x,y) \| (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$ |

**Table 2.** Role constructors and their semantics

In description logics, a *terminology* is specified by a set of terminological axioms (definitions) which select the models of terminology out of all possible interpretations. The most important forms of *terminological axioms* are presented in Table 3.

The *assertional axioms* are used to define instances of concepts and roles. Unlike terminological axioms, which are intensional descriptions of concepts and roles, assertional axioms are extensional descriptions. They can take the forms presented in Table 4.

The semantics of a DL is a declarative semantics based on the notions of model and interpretation.

An *interpretation* $\mathcal{I}$ of a knowledge base (made up of a terminological and an assertional component) consists of a set $\Delta^{\mathcal{I}}$ called the *interpretation domain* and a function $\cdot^{\mathcal{I}}$ (the *interpretation function*). The interpretation function maps each

[4] The following notations will be used throughout the paper: $CN$ represent concept names, $RN$–role names, $AN$–attribute names, $IN$–individual names. Also, $C$, $R$, $A$ will designate composite concepts, roles and attributes respectively.

| Axiom | | Semantics |
|---|---|---|
| `defconcept(CN,C)` | $CN = C$ | $CN^{\mathcal{I}} = C^{\mathcal{I}}$ |
| `defrole(RN,R)` | $RN = R$ | $RN^{\mathcal{I}} = R^{\mathcal{I}}$ |
| `defattribute(AN,A)` | $AN = A$ | $AN^{\mathcal{I}} = A^{\mathcal{I}}$ |
| `defprimeconcept(CN,C)` | $CN \subset C$ | $CN^{\mathcal{I}} \subset C^{\mathcal{I}}$ |
| `defprimerole(RN,R)` | $RN \subset R$ | $RN^{\mathcal{I}} \subset R^{\mathcal{I}}$ |
| `defprimeattribute(AN,A)` | $AN \subset A$ | $AN^{\mathcal{I}} \subset A^{\mathcal{I}}$ |
| `inclusion(`$C_1$`,`$C_2$`)` | $C_1 \subset C_2$ | $C_1^{\mathcal{I}} \subset C_2^{\mathcal{I}}$ |
| `equal(`$C_1$`,`$C_2$`)` | $C_1 = C_2$ | $C_1^{\mathcal{I}} = C_2^{\mathcal{I}}$ |
| `disjoint([`$CN_1$`,...,`$CN_n$`])` | $\bigwedge_{j=1}^{n} CN_j = \bot$ | $\bigcap_{j=1}^{n} CN_j^{\mathcal{I}} = \emptyset$ |

**Table 3.** Terminological axioms

| Axiom | | Semantics |
|---|---|---|
| `assert_ind(IN,C)` | $IN \in C$ | $IN^{\mathcal{I}} \in C^{\mathcal{I}}$ |
| `assert_ind(`$IN_1$`,`$IN_2$`,R)` | $(IN_1, IN_2) \in R$ or $IN_1\ R\ IN_2$ | $(IN_1^{\mathcal{I}}, IN_2^{\mathcal{I}}) \in R^{\mathcal{I}}$ |

**Table 4.** Assertional axioms

concept name $CN$ to a subset $CN^{\mathcal{I}} \subset \Delta^{\mathcal{I}}$ of the interpretation domain, and each role name $RN$ to a binary relation $RN^{\mathcal{I}}$ on $\Delta^{\mathcal{I}}$ (a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$). Attributes names $AN$ are interpreted as partial functions $AN^{\mathcal{I}} : Dom(AN^{\mathcal{I}}) \rightarrow \Delta^{\mathcal{I}}$, where $Dom(AN^{\mathcal{I}}) \subset \Delta^{\mathcal{I}}$ is the domain of definition of the partial function, whereas the individuals $IN$ are interpreted as elements of the interpretation domain $IN^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. According to the *"unique names assumption"*, different individual names denote different elements of $\Delta^{\mathcal{I}}$:

$$IN_1 \neq IN_2 \Rightarrow IN_1^{\mathcal{I}} \neq IN_2^{\mathcal{I}}.$$

The interpretation function is extended to the set of terms representing general concepts, roles and attributes respectively, as shown in Tables 1 and 2. This type of semantics is usually called *"descriptive semantics"*.

The *terminological axioms* specify "generic" knowledge, independent of particular individuals, while the assertional axioms are meant for describing *individuals* (or *instances*) of the *TBox* concepts and roles. An interpretation $\mathcal{I}$ verifies the terminological (assertional) axioms if and only if it verifies the semantic relationships from Table 3 (Table 4 respectively). An interpretation which verifies all the terminological and assertional axioms of a knowledge base $\langle \mathcal{T}, \mathcal{A} \rangle$ is called a *model* of $\langle \mathcal{T}, \mathcal{A} \rangle$.

The *extension* of a concept $C$ (of a role or attribute $R$) in a model $\mathcal{I}$ is by definition the set in which it is interpreted: $C^{\mathcal{I}}$ ($R^{\mathcal{I}}$ respectively).

In the following, we shall assume that the terminological axioms are general inclusions, the other types of concept definitions being easily reducible to inclusions. In particular, we shall allow multiple definitions of concepts and terminological cycles.[5]

[5] The majority of existing DL systems prohibit the existence of *ter-*

Unlike most implemented logical and database systems[6], DLs use the *open world assumption* (it is not automatically assumed that all the individuals known at one moment are all possible individuals). The *unique names assumption* is also made (individuals bearing different names are supposed to be different).

### 2.1.2 Inference services in description logics

The inference services provided by a DL can be described formally as follows:

1. *Satisfiability testing.* The concept $C$ is satisfiable w.r.t. the knowledge base $\langle \mathcal{T}, \mathcal{A} \rangle$ iff there exists a model $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$ in which $C$ has a non-empty extension: $C^{\mathcal{I}} \neq \emptyset$.
2. *Subsumption testing.* The concept $C$ subsumes $D$ w.r.t. the knowledge base $\langle \mathcal{T}, \mathcal{A} \rangle$, that is `subsumes(C,D)`, if and only if the extension of $C$ includes the extension of $D$ in all models $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$: $D^{\mathcal{I}} \subset C^{\mathcal{I}}$.
3. *Equivalence testing.* Concepts $C$ and $D$ are equivalent w.r.t. the knowledge base $\langle \mathcal{T}, \mathcal{A} \rangle$ iff their extensions are identical in all models $\mathcal{I}$ of $\langle \mathcal{T}, \mathcal{A} \rangle$: $D^{\mathcal{I}} = C^{\mathcal{I}}$.
4. *Classification* of a concept $C$ in $\mathcal{T}$ implies the identification of the most specific subsumers of $C$ and of its most general subsumees in $\mathcal{T}$.
   The hierarchy of concepts ordered according to the subsumption relation is automatically constructed by the classifier.
5. The knowledge base $\langle \mathcal{T}, \mathcal{A} \rangle$ is *consistent* iff it admits a non-empty model $\mathcal{I}$.
6. *Determination of the facts deducible from the knowledge base.*
7. *"Realization"* consists in determining the set of the most specific concepts $C$ in the terminology $\mathcal{T}$ whose instance is some given individual $IN$ (occurring in an assertional axiom): $IN \in C$.
8. *Instance retrieval* consists in retrieving all the instances $IN$ (occurring in the assertional axioms) of a given concept $C$.

In a language including concept negation, all of the above services make use of the knowledge base consistency algorithm. For example, subsumption testing `subsumes(C,D)` reduces to unsatisfiability testing of the concept `and(not(C),D)`. More precisely, we have the following results.

### 2.1.3 Reduction of the inference services to KB consistency testing

All the inference services offered by a DL can be reduced (in linear time) to the knowledge base consistency test[7] in the following way:

1. The concept $C$ is satisfiable w.r.t. $\langle \mathcal{T}, \mathcal{A} \rangle$ iff the knowledge base $\langle \mathcal{T}, \mathcal{A} \cup \{y \in C\} \rangle$ is consistent ($y$ being a new instance name).
2. $C$ is subsumed by $D$ ($C \subset D$) w.r.t. $\langle \mathcal{T}, \mathcal{A} \rangle$ iff $C \wedge \neg D$ is not satisfiable w.r.t. $\langle \mathcal{T}, \mathcal{A} \rangle$.
3. The individual $x$ is an instance of the concept $C$ (i.e. $x \in C$) iff the knowledge base $\langle \mathcal{T}, \mathcal{A} \cup \{x \in \neg C\} \rangle$ is inconsistent.

The knowledge base consistency test is usually performed using a tableaux based calculus and will not be described here. For more details, see e.g. [10, 9].

The *universal* terminological language described in [14] has an undecidable subsumption problem. For this reason, the languages used in practice usually implement only a subset of the above-mentioned constructors, in order to ensure the decidability of subsumption testing (which is essential for this class of languages).

As could be expected, there is a tight interdependence between the expressiveness of the language (which depends on the concept/role constructors used) and the complexity of the subsumption and satisfiability testing algorithms.

As far as expressiveness is concerned, the class of languages with polynomial (subsumption and satisfiability testing) algorithms is too restrictive, since only extremely simple domains can be represented in them.

On the other hand, the complete subsumption and satisfiability testing algorithms for more expressive languages are usually NP-complete, co-NP-complete, PSPACE-complete, EXPTIME-complete or worse. However, one should constantly bear in mind the fact that such high complexities correspond to the worst cases, while the algorithms may behave well in practical cases. $\mathcal{KRIS}$ [2] and recently CRACK [5] and $\mathcal{R}eg\mathcal{AL}$ [3] are the only operational systems with complete algorithms.

In conclusion, DLs are not merely yet another approach to the domain knowledge representation, but also an attempt at formalising the representation systems used so-far. They provide powerful and, what is even more important, complete inference services (as opposed to semantic networks, which have a procedural semantics and in which the incomplete inference services are described by an implementation and not declaratively as in the case of DLs).

## 2.2 The domain level extension

In order to be usable in real-life applications, our KR&R language will have to be able to describe *collections* of objects (such as sets or lists of instances/tuples). However, existing (implemented) DL systems usually lack constructors for sets or lists of objects[8] and we therefore have to *extend* the description logic with such collections of concept instances or role tuples. The inherent *incompleteness* due to the fact that such collections are not taken into account in DL inferences is of no practical consequence since:

- unlike most logic and database systems, description logics use the *open world assumption* (it is not automatically assumed that an individual that cannot be proven to be the instance of a given concept $C$ is the instance of its negation $\neg C$).

*minological cycles*, i.e. they do not allow a concept name to appear (neither directly, nor indirectly) in its own definition. Also, a concept or role name is usually allowed to occur only once in the left hand side of a definition. The absence of terminological cycles drastically reduces the expressivity of the language; without them we are unable to represent recursively defined data structures such as lists or trees.

[6] which usually use a form of closed world semantics

[7] This is possible only if the language includes concept negation.

[8] Some description logics provide the $one\_of(IN_1, \ldots, IN_n)$ construct which denotes the concept whose extension is given by the set of instances $\{IN_1, \ldots, IN_n\}$. However, what we need is a concept construct whose *instances* denote sets or lists of other instances (or tuples).

- the terminological and assertional levels of the DL are completely separated (it is impossible to have a DL instance that represents a collection of other DL instances).

This domain extension leads to a *hybrid* domain level in which *simple* instances are represented in the DL, while *collections* (sets or lists) are stored in the extension.

The internal representation of a *domain store* element is the following:

$$domain\_store\left(simple, \begin{matrix} concept \\ relation \end{matrix}, \begin{matrix} C \\ R \end{matrix}, \begin{matrix} IN \\ (IN', IN'') \end{matrix}\right)$$

$$domain\_store\left(set/list, \begin{matrix} concept \\ relation \end{matrix}, \begin{matrix} C \\ R \end{matrix}, \begin{matrix} [IN_1, IN_2, \ldots] \\ [(IN_1', IN_1''), \ldots] \end{matrix}\right).$$

(Since there exists the possibility of confusing the DL-roles with the inference roles, we shall refer in the following to DL-roles as relations.)

A domain-level concept or (binary) relation can have an associated DL description, represented relationally as

$$DL\_description\left(\begin{matrix} concept \\ relation \end{matrix}, \begin{matrix} C \\ R \end{matrix}, \begin{matrix} DL\_C \\ DL\_R \end{matrix}\right).$$

Here $C$ and $R$ stand for domain level concept/relation names, while $DL\_C$ and $DL\_R$ represent their associated DL descriptions.

The following primitives should be used for asserting domain store instances:

$$assert\_domain\_store\left(simple, \begin{matrix} concept \\ relation \end{matrix}, \begin{matrix} C \\ R \end{matrix}, \begin{matrix} IN \\ (IN', IN'') \end{matrix}\right)$$

$$assert\_domain\_store\left(set/list, \begin{matrix} concept \\ relation \end{matrix}, \begin{matrix} C \\ R \end{matrix}, \begin{matrix} [IN_1, IN_2, \ldots] \\ [(IN_1', IN_1''), \ldots] \end{matrix}\right).$$

## 3 The meta-level architecture

*$E_x$Claim* has a meta-level architecture specific to KADS which structures the knowledge corresponding to a model on three levels:

- the domain level
- the inference level
- the task level.

Figure 1 presents a graphical representation of the *$E_x$Claim* architecture (which is typical for the KADS expertise model [22]).

Although the decomposition of a given model in the three knowledge levels may not be unique, it is usually relatively easy to map an informal description of the model onto this three-level architecture.

The *domain level* mainly encodes the domain ontology. The corresponding process of conceptualisation consists in delimiting the relevant concepts and relations between the concepts of the domain.

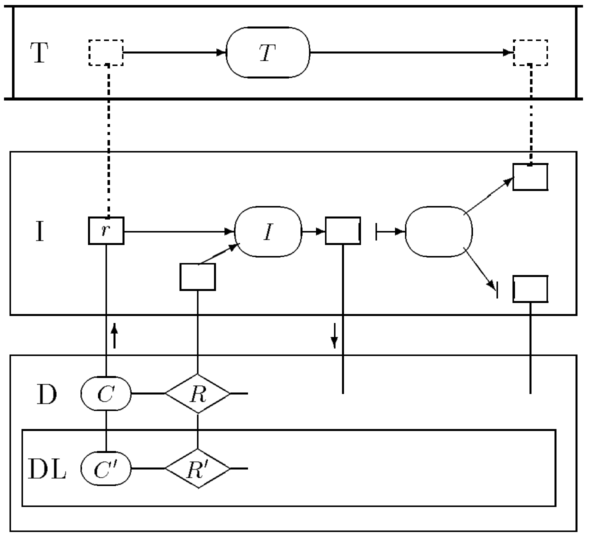

**Figure 1.** The *$E_x$Claim* architecture

Often, the conceptualisation of a given situation is not unique and usually decisively influences the efficiency (and sometimes even the possibility) of problem solving in the given domain. Changing the conceptualisation may sometimes make the expression of certain types of knowledge impossible. That is why finding an appropriate conceptualisation for a given problem (in other words, its relevant representation) can be at least as difficult as solving the problem, since we may regard the problem-solving process as a search (in the space of all possible conceptualisations) for a conceptualisation in which the original problem reduces to an immediately solvable problem.

The *inference level* consists of a set of primitive problem solving actions, whose internal functioning is irrelevant from the point of view of the conceptual problem solving model. Inferences (represented graphically as ovals) have a set of input and output *roles* (depicted in diagrams as rectangles), which denote, roughly speaking, the arguments of the inference.

Inference roles represent a kind of meta-level abstraction of domain level objects (concepts, relations, etc). In order to enhance the flexibility of the mapping between inference roles and domain level objects, the following types of inference role *domain links* have been introduced:

- *simple* (refers a single DL instance)
- *set* (refers a single domain level instance representing a set of DL instances)
- *list* (refers a single domain level instance representing a list of DL instances)

Domain links are represented in *$E_x$Claim* as:

$$domain\_link\left(InferenceRole, \begin{matrix} simple \\ set \\ list \end{matrix}, \begin{matrix} concept \\ relation \end{matrix}, \begin{matrix} C \\ R \end{matrix}\right).$$

Although the execution of inferences induces domain level operations, inferences do not manipulate domain level objects directly. Since they refer to domain object only *indirectly via* *inference roles*, (partial) models in which the domain-level has been stripped off (removed) can be easily reused in a different domain. Reusability is thus a key feature of KADS expertise models as it enables the construction of domain-independent libraries of models.

The inference structures represent the data-flow of a given model. The control of the various inferences is accomplished at the *task level*. Tasks can be either

- *primitive* (corresponding to an inference),
- *composite* (built from other subtasks), or
- *transfer* tasks (which interact with the environment).

## 4 The inference level

Inferences are primitive problem solving actions which perform elementary problem solving operations (i.e. operations whose internal functioning is irrelevant from the point of view of the conceptual model).

Inferences operate on *inference roles*, which can be either inputs or outputs.

*Input roles* implement the *upward reflection* rules of the meta-level architecture, i.e. they are responsible, broadly speaking, for retrieving domain level instances. More precisely, input roles can perform the following types of *domain operations*:

- *retrieve* (retrieve an instance of the domain level object linked to the input role, but do not remove the instance afterwards)
- *noretrieve* (no instances are retrieved from the domain, as if no domain operation was performed; the value of the role is set in the call of the inference rather than retrieved from the domain)
- *delete* (retrieves a domain level instance and subsequently removes it; the domain level description logic must provide facilities for knowledge revision in order to support this operation).

*Output roles* implement the *downward reflection* rules of the meta-level architecture since they are responsible mainly for storing object instances in the domain level. More precisely, output roles can perform the following types of domain operations:

- *store* ("instances" of the given output role are asserted in the domain)
- *nostore* (the "instances" of the output role are not reflected in the domain; instead, their value is passed to the caller of the inference).

Inferences perform *automatic domain operations* on their input/output roles. Since no direct domain reference is made in inferences (or tasks), these levels of the model are domain-independent and thus reusable (the code of the inference body can remain exactly the same even after changing the domain level).

The automatic domain operations in inferences can be regarded as a more evolved form of parameter passing in an inference call. For instance, the operation types "noretrieve" and "nostore" perform no actual domain operations and rely on the explicit parameter passing mechanism in the call of the inference.

On the other hand, the operation types "delete" and "store" perform domain operations and should be backtrackable if we intend to provide a non-deterministic computation model. This in turn requires the *non-monotonicity* of the domain level and the existence of *knowledge revision* facilities in the corresponding description logic.

The *backtrackability* of domain operations requires that whenever the inference (that performed the corresponding domain operation) fails, the state of the domain store and the description logic is restored to the state before the call of the failing inference. The same happens when new solutions are sought for by backtracking.

In order to further enhance the flexibility of the inference level primitives, two types of *role mappings* have been provided (see also Figure 2):

- *simple* (refers to a single domain store element associated with the inference role)
- *set* (refers to the set of all simple domain store elements associated with the inference role)

r
I
simple
r
I
set

**Figure 2.** Role mapping types

The $E_x Claim$ representation of role mappings and the associated role operations is:

$$role\_mapping\left(Inference, \begin{array}{c} InputRole \\ OutputRole \end{array}, simple/set, \right.$$
$$\left. \begin{array}{c} noretrieve/retrieve/delete \\ nostore/store \end{array}\right).$$

Note that a "simple" role operation involves a *single*[9] domain store element of the form:

$$domain\_store\left(\begin{array}{c} simple \\ set \\ list \end{array}, \begin{array}{c} concept \\ relation \end{array}, \begin{array}{c} C \\ R \end{array}, Instance\right).$$

From the conceptual point of view, inferences are primitive problem solving actions and their internal structure as well as their functioning need not be further detailed. However, if we are aiming at an operational system, the knowledge engineer would have to provide the code of the inference bodies in order to be able to execute the model.

For reasons of simplicity (and also since $E_x Claim$ itself is implemented in Prolog), inference bodies are written in Prolog according to the following parameter passing convention:

```
inference_name([input_role_i = Value_i, ...],
               [output_role_j = Value_j, ...]) :-
    prolog_code.
```

[9] irrespective of the domain link type, which can be: simple, set or list.

Therefore, for each inference, the user will have to write a (set of) clause(s) like above. The heads of such clauses have two arguments representing the lists of input and output role bindings. A *role binding* is a term of the form `role_name = RoleValue` (`RoleValue` can be a variable or a (partially) instantiated Prolog term). The order of the role bindings in the binding lists is irrelevant.

An inference body can contain calls to other inferences or tasks, but this is not recommended as a good modelling approach (since inferences should be thought of as primitive executable objects).

In $E_x$*Claim*, inferences are executed using the following primitive:

```
exec_inference(inference_name,
               [input_role_i = InputValue_i, ...],
               [output_role_j = OutputValue_j, ...]).
```

The following operations are performed in `exec_inference`:

- unify the input arguments of the body with those of the call
- perform domain operations for the input roles (noretrieve, retrieve, delete)
- execute the inference body
- unify the output arguments of the body with those of the call
- perform domain operations for the output roles (nostore, store).

All the above steps of `exec_inference` are backtrackable. As already mentioned, backtracking to a domain operation may involve domain level knowledge revision too.

## 5 The task level

The task level embodies the *control knowledge* of a model. Tasks do not perform domain operations since they are viewed as composite executable objects (only the primitive executable objects, i.e. the inferences, are allowed to perform domain operations).

Since no domain operations are associated to task roles, tasks are, from an operational point of view, like inferences with "*noretrieve*" input roles and "*nostore*" output roles.

One and the same role can be an inference role and a task role at the same time. (For example, the input role of a composite task can also be the input role of a component inference or subtask. The actual domain operations are performed when the inference is executed.)

Parameter passing in tasks is done explicitly in the call of the task. From the programmer's point of view, task bodies have the same syntax as inference bodies:

```
task_name([input_role_i = InputValue_i, ...],
          [output_role_i = OutputValue_j, ...]) :-
    prolog_code.
```

Task bodies can, of course, contain calls to other inferences and subtasks.

Executing a task with

```
exec_task(task_name,
          [input_role_i = InputValue_i, ...],
          [output_role_j = OutputValue_j, ...])
```

amounts to

- unifying the input arguments of the body with those of the call
- executing the body
- unifying the output arguments of the body with those of the call.

All the above execution steps of `exec_task` are backtrackable.

Figure 3 depicts the basic $E_x$*Claim* architecture viewed from a functional viewpoint.

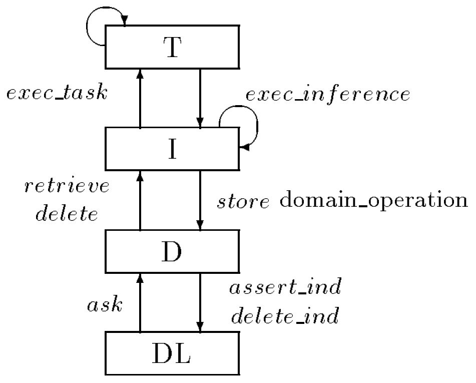

**Figure 3.** The $E_x$*Claim* architecture from a functional viewpoint

## 6 A simple example

A very simple example of a resource allocation problem will be used to illustrate the facilities of $E_x$*Claim*.

Consider the following allocation problem. In a university department there is a set of classes to be taught by a set of teachers. Classes can be either courses or seminars (but not both), while teachers are either professors or assistants (but not both). Let us further assume that assistants are allowed to teach only seminars and that the list of classes familiar to (known by) the various teachers is also given.

Of course, a teacher can teach a given class only if he knows it. Also, we require that each class should be taught by a teacher and that a teacher cannot teach more than one class (of course, there may be teachers that don't teach any class at all).

The goal of the problem is to find an assignment of teachers (resources) to classes (requests) such that all the above constraints are verified.

The most straightforward conceptualisation of this problem involves defining the following concepts:

```
defconcept(teacher, or(prof, assistant)).
defprimeconcept(prof, teacher).
defprimeconcept(assistant, all(teaches, seminar)).
disjoint([prof, assistant]).
defconcept(class, or(course, seminar)).
defprimeconcept(course, class).
defprimeconcept(seminar, class).
```

```
disjoint([course, seminar]).
defprimerole(knows).
defprimerole(teaches).
```

The relations *teaches* and *knows* link a teacher with the course he teaches or knows respectively.

Given the relation *knows*, one must find the relation *teaches* subject to all the problem constraints. Some of these constraints are easily expressible in the description logic (like the ones presented above). Other constraints may not be expressible in the DL and we may have to take them into account at the inference level. For instance, the constraint mentioning that "a teacher can teach a given class only if be knows it" cannot be represented in the DL unless the particular DL we are using allows the famous role-value map constructor:

```
equal(subset(teaches, knows), top).
```

However, since role-value maps (together with role composition and concept conjunction) induce the *undecidability* of the DL inference services [17], they are usually not provided in implemented DL systems with complete algorithms. Therefore, we will have to encode this constraint at the higher levels of the model (inference and/or task level).

On the other hand, the constraints that each class should be taught by a teacher and that a teacher cannot teach more than one class could easily be represented in existing DLs as:

$$\begin{aligned} class &\subset exists(inv(teaches), top) \\ teacher &\subset atmost(1, teaches). \end{aligned}$$

In fact, if all the problem constraints could be represented in the description logic, we could use the DL inference services to solve our problem without additional support from the inference or task level (DL inference services are usually reducible to the knowledge base consistency test, which typically works by constructing models of the KB. The model constructed while proving the KB consistency can then be used to extract the solution of the problem).

However, not all constraints are expressible in a given DL, so that the additional levels are really necessary. Also, we may wish to exert a tighter control on the problem solving process and thus inference and task levels are again needed (relying entirely on the description logic inference services may turn out to be too expensive from a computational point of view).

Last, if we are trying to develop reusable models, having separate domain, inference and task levels turns out to be again very useful. For instance, stripping off the domain level from our simple allocation example leads to a reusable problem-solving model for general resource allocation problems (teachers are abstracted as *resources*, while classes are viewed as *requests*). We could also reuse the domain model in a different problem involving teachers and classes.

After having completely described the domain level of our simple model, we proceed to the construction of the inference level. An extremely simple non-deterministic approach will be followed.

Assume that a partial assignment (of the *teaches* relation) has been constructed up to this point and that we are currently attempting to extend this partial assignment with a new tuple for *teaches* chosen from the tuples of *knows* and linking a class that has not already been assigned and a teacher who is still free (teaches no other class). The corresponding inference structure is depicted in figure 4. Note that we have used generic (abstract) names for the inference roles denoting teachers, classes and the relations *teaches* and *knows*. Classes are regarded as requests, whereas teachers are the resources to be allocated to these requests. The tuples of *teaches* are thought of as `assignments`, whereas the tuples of *knows* are just `candidate_assignments`.

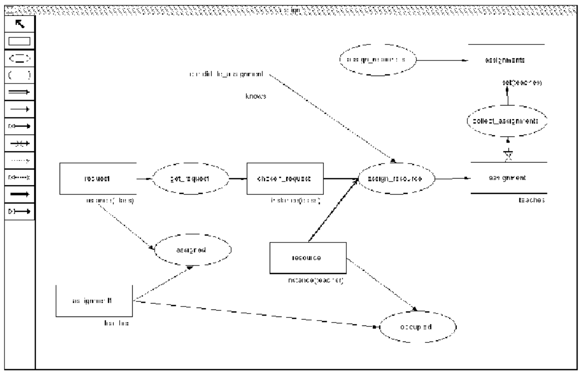

**Figure 4.** The non-deterministic inference structure for the "teachers" problem

The inference `get_request` chooses a `request` that has not been assigned yet. This `chosen_request` is passed on to `assign_resource`, which tries to retrieve a `candidate_assignment` for this request. If it succeeds, the `assignment` is stored in the domain level. The whole process is repeated (at the task level) until there are no more unassigned requests (case in which it terminates with success) or until a failure occurs (case in which the system automatically backtracks to a previous state). Backtracking involves not only the inference and task levels, but also the domain level since the DL has to be restored to its previous state (before the call).

Note that inference and task bodies are extremely simple since we are heavily relying on the automatic domain operations performed by inferences. We are also relying on the powerful description logic inference mechanisms (mainly when doing domain store retrieval but also when checking for global consistency after a solution has been found).

Note that the problem-solving process involves computations (deduction) at two different levels:

- at the domain level (DL deduction)
- at the inference and task levels (execution of inferences and tasks).

This observation shows that not only the description of the problem, but also the problem-solving process itself is distributed at different levels. We feel that this separation of computations (performed while solving the problem) is extremely useful and natural, leading to a higher reusability of the models. For instance, a change in the domain model does not require modifications at the inference or task levels. Let us illustrate this with an example.

Consider a problem instance in which there are only two teachers ($p_1$ and $p_2$) and two classes (a *course* $c_1$ and another class $c_2$):

```
assert_domain_store(simple, concept, teacher, p1).
assert_domain_store(simple, concept, teacher, p2).
assert_domain_store(simple, concept, course, c1).
assert_domain_store(simple, concept, class, c2).
assert_domain_store(simple, relation, knows, [p1,c1]).
assert_domain_store(simple, relation, knows, [p1,c2]).
assert_domain_store(simple, relation, knows, [p2,c1]).
assert_domain_store(simple, relation, knows, [p2,c2]).
```

If we are not told whether $p_1$ or $p_2$ are professors or assistants, the system will return two alternative allocations, namely $[(p_1,c_1),(p_2,c_2)]$ and $[(p_1,c_2),(p_2,c_1)]$.

However, if we now specify that $p_1$ is an *assistant*, only the second assignment will be retained as a consistent one, since an assistant $(p_1)$ cannot teach a course $(c_1)$.

## 7 Subtleties

### 7.1 Eager versus lazy inference mechanisms

As already pointed out, description logics provide non-trivial domain level inference mechanisms. For reasons of efficiency, however, these mechanisms can be either

- *eager*: are performed automatically in certain key situations, for example instance assertions or queries (*ask* operations)
- *lazy*: are activated only upon explicit invocation, for example global consistency check.

Eager deduction mechanisms are usually faster (*ask* does a limited amount of theorem proving, but does not check global consistency), whereas lazy ones are computationally hard and should be used only when absolutely necessary.

The global KB consistency check is a typical example of a lazy deduction mechanism. In order to ensure the KB consistency after each assertion, the consistency test should be invoked after each such operation (in $E_x$*Claim* there is an option that turns these checks on). However, this test is extremely time expensive and should be avoided whenever possible by explicitly testing for consistency in the critical points only (the $E_x$*Claim* `check_consistency` option has to be turned off).

### 7.2 Interaction between domain links and role mappings

In the following, we intend to clarify the difference between a situation (a) in which a role with a domain link of type "set" has a "simple" role mapping to a certain inference $I$ and a situation (b) in which a role with a "simple" domain link has a mapping of type "set" (to a certain inference $I$).

The structure in Figure 5.a collects/asserts a *single* domain store element of the form:

```
domain_store(set, concept, C, [x_1, ...])
```

whereas the one in figure 5.b collects/asserts *all* simple instances of $C$: $[x_1, x_2, \ldots]$, with

```
domain_store(simple, concept, C, x_i)
```

for all $i$.

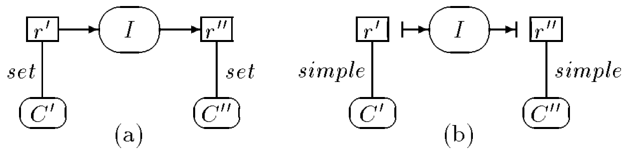

**Figure 5.** Interaction between domain links and role mappings

### 7.3 Knowledge revision

Domain instances are deleted only if they were explicitly asserted and not if they are just deducible from explicitly asserted facts.

Consider for example the following concept definition

$$father = man \wedge \exists child\colon \top$$

and the instance assertions

$$\begin{array}{rcl} peter & \in & man \\ (peter, john) & \in & child. \end{array}$$

The above knowledge base entails $john \in father$.

Now consider a role $r$ linked to the domain concept *father*. If a role operation of type "delete" is attempted on $r$, the instance $john \in father$ will be retrieved, but since $john \in father$ was not explicitly asserted, it will not be deleted (the domain operation succeeds though).

## 8 Conclusions

The following advantages of the approach presented in this paper can be mentioned:

- easing the process of knowledge engineering in KBS development.
- the meta-level architecture of the system enables the development of reusable domain-independent problem solving models (PSMs) and of application-independent ontologies.
- the possibility of developing domain-independent executable libraries of PSMs.
- supporting the process of KBS validation by using the inference services offered by the domain-level language:
  - semantic consistency checking
  - domain level deduction
  - automatic concept classification, knowledge structuring and indexing.

  Such inference services did not exist in other KADS operationalization environments, except maybe Si(ML)$^2$/FML. However, deduction in a DL is faster than in full first order logic; on the other hand, expressivity is lower.
- the description logic used at the domain level can be regarded as a compromise between expressiveness and efficiency. The readability of DL formulas is also reasonably high.
- $E_x$*Claim* provides non-deterministic inference and task levels, which rely on a non-monotonic domain level.

  The lack of non-determinism is, in our opinion, an important drawback for KBSs. Algorithms in KBSs, as opposed to traditional software engineering environments, are complex and usually non-deterministic. If only deterministic structures are allowed, then one has to simulate the non-determinism (this can be done in many different ways and can be domain-dependent since it may depend on the domain constraints). But the model should be as abstract and unique as possible, in order to be reusable.

The following main objectives will be pursued in our future research:

- Demonstrate the usefulness of the knowledge level simulation/execution facilities (offered by $E_x$*Claim*) in KBS development.
- Develop real-world knowledge based systems (KBS) applications using the $E_x$*Claim* knowledge modelling environment.

We also plan to improve the implementation of $E_x$*Claim* by developing user-friendly animation tools for the simulation of KADS models as well as by assisting the KBS developer with context sensitive help facilities (KADS is known to be a complex methodology, so that providing such help facilities would be extremely useful).

Facilities for automatic code generation starting from KADS models will also be provided.

We expect that the feedback obtained while developing applications will be essential for shaping the final version of $E_x$*Claim*.

Knowledge based system (KBS) development is a very complex process. Any tools providing some automatic assistance in this process would significantly improve its efficiency as well as the reliability of the end product (the KBS). The present work tries to take some steps forward in this direction.

## Acknowledgements

The research presented in this paper has been partly supported by the European Community project PEKADS (CP93-7599). Thanks are due to Doina Ţilivea and Ştefan Trăuşan for developing the graphical user interface of $E_x$*Claim* and to Jan Wielemaker for support on using the XPCE environment. I would also like to thank Ullrich Hustadt and Renate Schmidt for permitting the use of the Motel terminological system [11] in this research.

## REFERENCES

[1] ABEN M., BALDER J., VAN HARMELEN F. *Support for the formalization and validation of KADS expertise models.* Report KADS-II/M2/UvA/DM2.6a/1.0, University of Amsterdam, 1994.

[2] BAADER F., HOLLUNDER B. *KRIS: Knowledge Representation and Inference System – System Description.* DFKI TM-90-03.

[3] BADEA LIVIU. *A unified architecture for knowledge representation and reasoning based on terminological logics.* International Workshop on Description Logics DL-95, Roma 1995.

[4] BRACHMAN R.J., SCHMOLZE J.G. *An Overview of the KL-ONE Knowledge Representation System.* Cognitive Science 9 (2) 1985.

[5] BRESCIANI P., FRANCONI E., TESSARIS S. *Implementing and testing expressive description logics.* International Workshop on Description Logics DL-95, Roma 1995.

[6] BUCHHEIT M., DONINI F.M., SCHAERF A. *Decidable Reasoning in Terminological Knowledge Representation Systems.* DFKI RR-93-10.

[7] DE GIACOMO G., LENZERINI M. *What's in an Aggregate: Foundations for Description Logics with Tuples and Sets.* Proc. IJCAI-95, pp. 801-807.

[8] FENSEL D., VAN HARMELEN F., *A comparison of languages which operationalize and formalize KADS models of expertise.* Report 280, Universität Karlsruhe, September 1993.

[9] HOLLUNDER B., NUTT W. *Subsumption Algorithms for Concept Languages.* DFKI RR-90-04.

[10] HOLLUNDER B. *Hybrid Inferences in KL-ONE-based Knowledge Representation Systems* DFKI Research Report RR-90-06.

[11] HUSTADT U., NONNENGART A., SCHMIDT R., TIMM J. *Motel User Manual.* Max Planck Institute Report MPI-I-92-236, September 1994.

[12] PATEL SCHNEIDER E.A. *Term Subsumption in Knowledge Representation.* AI Magazine, Summer 1990, p. 16.

[13] PATEL-SCHNEIDER P.F. *Undecidability of Subsumption in NIKL.* AI 39 (1989), pp. 263-272.

[14] SCHILD KLAUS. *Undecidability of Subsumption in U.* KIT Report Technische Universität Berlin, October 1988.

[15] SCHILD KLAUS. *A correspondence theory for terminological logics: preliminary report.* IJCAI-91.

[16] SCHMIDT-SCHAUSS M., SMOLKA G. *Attributive concept descriptions with complements.* Artificial Intelligence 48 (1), pp. 1-26, 1991.

[17] SCHMIDT-SCHAUSS M. *Subsumption in KL-ONE is undecidable.* Proceedings KR-89, pp. 421-431.

[18] SCHREIBER G., WIELINGA B., BREUKER J. *KADS: A Principled Approach to Knowledge-Based System Development.* Academic Press, 1993.

[19] SCHREIBER G., WIELINGA B., AKKERMANS H., VAN DE VELDE W., DE HOOG R. *CommonKADS. A comprehensive methodology for KBS development.* Report KADS-II/M1/PP/UvA/70/2.0, University of Amsterdam, 1994.

[20] SOWA J.F. (ED) *Principles of Semantic Networks.* Morgan Kaufmann 1991.

[21] WIELINGA B.J., SCHREIBER A.TH., BREUKER J.A. *KADS: A Modeling Approach to Knowledge Engineering.* Knowledge Acquisition 4 (1), special issue "The KADS approach to knowledge engineering", also Report KADS II/T1.1, University of Amsterdam, 1992.

[22] WIELINGA B.J. (ED.) *Expertise Model Definition Document.* Report KADS II/M2/UvA/026/5.0, University of Amsterdam, 1992.

[23] WIELEMAKER J. *SWI-Prolog 1.9 Reference Manual.* University of Amsterdam, 1994.

[24] WIELEMAKER J., ANJEWIERDEN A. *Programming in PCE/Prolog.* University of Amsterdam, 1992.