Knowledge Modeling and Reusability in $E_x Claim$

Liviu Badea
AI Research Lab
Research Institute for Informatics
8-10 Averescu Blvd., Bucharest, Romania
e-mail: badea@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 (KBS) 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 general logic programs integrated with the domain level by means of upward and downward reflection rules which describe the automatic domain operations performed whenever arguments of inferences or tasks are accessed. Inferences and tasks support non-deterministic reasoning, which in turn requires a non-monotonic domain level.

Description logics offer a set of inference services (some not available in other knowledge representation languages) which are extremely useful in knowledge modeling. Such inference services include domain-level deduction, semantic consistency verification and automatic classification of concepts. We argue that such validation and verification facilities are important in assisting a knowledge engineer in developing models. These models are *reusable* due to the layered architecture as well as to the possibility of writing generic inferences using a reified membership relation.

1 Introduction

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.

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 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 modeling, a series of methodologies and specification languages have been put forward, the most important ones being the KADS methodology [10], KIF and Ontolingua, etc. A number of KADS operationalization languages and environments have been developed for supporting the KADS methodology with executable tools: $\mathrm{Si(ML)}^2$, OMOS, MoMo, KARL, MODEL-K, FORKADS [5], 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 presents a hybrid architecture aiming at integrating description logics (viewed as domain description languages) and logic programming (used for representing inference and control knowledge) in a declarative knowledge modeling environment. The main byproduct of this declarative approach is the possibility of developing reusable libraries of problem solving methods.

1.1 Brief description of $E_x Claim$

$E_x Claim$ (Executable CommonKADS Language for Integrated Modeling) is a knowledge modeling environment integrating description logics (used for representing domain knowledge) and logic programming (for dealing with inference and control knowledge). Originally developed as an op-

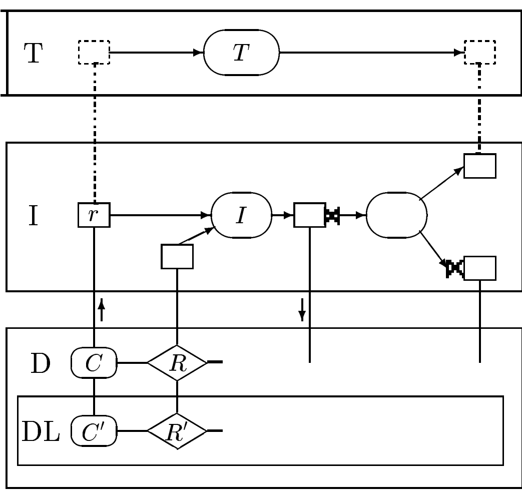

Figure 1: The E_xClaim architecture

erationalization language for the CommonKADS knowledge based systems methodology [2], E_xClaim has a meta-level architecture which structures the knowledge corresponding to a model on three levels, namely the *domain*, *inference* and *task* level. Figure 1 presents a graphical representation of the E_xClaim architecture (which is typical for the KADS expertise model). 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* encodes the domain ontology. The domain knowledge is mainly expressed in a description logic (DL) which provides fairly sophisticated inference services (such as domain-level deduction, semantic consistency checking, automatic classification of concepts, knowledge structuring and indexing).

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).)

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 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*.

2 The domain level

The E_xClaim domain level is an extension of a description logic (DL), with the *concept, relation and attribute constructors* from Table 1 and the *terminological and assertional axioms* shown in Table 2. (The DL implementation used in E_xClaim is the Motel system [6].) Unlike most implemented

Concept	Symbolic	Semantic interpretation
top	$\top$	$\Delta^{\mathcal{I}}$
$bottom$	$\bot$	$\emptyset$
$and(C_1, C_2)$	$C_1 \wedge C_2$	$C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}}$
$or(C_1, C_2)$	$C_1 \vee C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
$not(C)$	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
$all(R, C)$	$\forall R{:}\, C$	$\{x \in \Delta^{\mathcal{I}} \mid \forall y.(x,y) \in R^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
$exists(R, C)$	$\exists R{:}\, C$	$\{x \in \Delta^{\mathcal{I}} \mid \exists y.(x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
$atleast(n, R)$	$\geq_n R$	$\{x \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(x) \rvert \geq n\}$
$atmost(n, R)$	$\leq_n R$	$\{x \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(x) \rvert \leq n\}$
$exactly(n, R)$	$=_n R$	$\{x \in \Delta^{\mathcal{I}} \mid \lvert R^{\mathcal{I}}(x) \rvert = n\}$
Relation	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}	$\{(y,x) \mid (x,y) \in R^{\mathcal{I}}\}$
$domrestr(R, C)$	$C \lfloor R$	$\{(x,y) \mid (x,y) \in R^{\mathcal{I}} \wedge x \in C^{\mathcal{I}}\}$
$restrict(R, C)$	$R \rfloor C$	$\{(x,y) \mid (x,y) \in R^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$

Table 1: Concept/relation constructors

Terminological axiom		Semantics
`defconcept(CN,C)`	$CN = C$	$CN^{\mathcal{I}} = C^{\mathcal{I}}$
`defrelation(RN,R)`	$RN = R$	$RN^{\mathcal{I}} = R^{\mathcal{I}}$
`defattribute(AN,A)`	$AN = A$	$AN^{\mathcal{I}} = A^{\mathcal{I}}$
`defprimeconcept(CN,C)`	$CN \subset C$	$CN^{\mathcal{I}} \subset C^{\mathcal{I}}$
`defprimerelation(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`$_n$`])`	$\bigwedge_{j=1}^{n} CN_j = \bot$	$\bigcap_{j=1}^{n} CN_j^{\mathcal{I}} = \emptyset$
Assertional axiom		Semantics
`assert_ind(IN,C)`	$IN \in C$	$IN^{\mathcal{I}} \in C^{\mathcal{I}}$
`assert_ind(IN`$_1$`,IN`$_2$`,R)`	$(IN_1, IN_2) \in R$	$(IN_1^{\mathcal{I}}, IN_2^{\mathcal{I}}) \in R^{\mathcal{I}}$

Table 2: Terminological and assertional axioms

logical and database systems[1], DLs use the *open world assumption* (it is not automatically assumed that all the individuals known at one moment are all possible individuals).

The *inference services* provided by the DL include satisfiability (consistency) and subsumption testing, automatic classification of concepts in a hierarchy and instance retrieval.

In a language including concept negation, all of the above services make use of the knowledge base consistency algorithm [3]. For example, subsumption testing `subsumes(C,D)` reduces to unsatisfiability testing of the concept `and(not(C),D)`.

2.1 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[2] and we therefore have to *extend* the description logic with such collections of concept instances or relation tuples. This does not affect the completeness of the DL inferences since the terminological and assertional levels of the DL are completely separated (it is impossible to

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

[2] 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).

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 so-called *domain store* element is the following:

$$domain_store\left(simple, \begin{array}{c} concept \\ relation \end{array}, \begin{array}{c} C \\ R \end{array}, \begin{array}{c} IN \\ (IN', IN'') \end{array}\right)$$

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

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

$$DL_description\left(\begin{array}{c} concept \\ relation \end{array}, \begin{array}{c} C \\ R \end{array}, \begin{array}{c} DL_C \\ DL_R \end{array}\right).$$

Here C and R stand for domain level concept/relation names, while DL_C and DL_R represent their associated DL descriptions.

3 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*[3], which can be either inputs or outputs. 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* (the role refers a single DL instance)
- *set* (the role refers a single domain level instance representing a set of DL instances)
- *list* (the role refers a single domain level instance representing a list of DL instances).

Domain links are represented in $E_x Claim$ as:

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

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).

[3]In KADS, the arguments of inferences are called *roles*.

Figure 2: Role mapping types

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 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)

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

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

Note that a "simple" role operation involves a *single*[4] domain store element of the form:

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

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

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 in order to be able to execute the model.

In E_x*Claim*, inferences are *normal logic programs*, i.e. sets of clauses of the form (we are using a Prolog syntax):

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

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 modeling 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, ...]).
```

which performs the following operations:

- unify the input arguments of the inference 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 inference 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.

4 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, tasks have the same syntax as inferences:

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

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

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

5 A simple example

A very simple example of a resource allocation problem will be used to illustrate the facilities of E_x*Claim*. In a university department there is a set of classes to be taught by a set of lecturers. Classes can be either courses or seminars (but not both), while lecturers 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 lecturers is also given. Of course, a lecturer can teach a given class only if he knows it. Also, we require that each class should be taught by a lecturer and that a lecturer cannot teach more than one class (of course, there may be lecturers that don't teach any class at all). The goal of the problem is to find an assignment of lecturers (resources) to classes (requests) such that all the above constraints are verified.

The most straightforward conceptualization of this problem involves defining the following concepts and relations:

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

The relations *teaches* and *knows* link a lecturer 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 lecturer can teach a given class only if he 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 relation composition and concept conjunction) induce the *undecidability* of the DL inference services [9], 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 lecturer and that a lecturer cannot teach more than one class could easily be represented in existing DLs as:

```
defprimeconcept(class, exists(inv(teaches), lecturer)).
defprimeconcept(lecturer, atmost(1, teaches)).
```

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 (lecturers are abstracted as *resources*, while classes are viewed as *requests*). We could also reuse the domain model in a different problem involving lecturers 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 lecturer who is still free (teaches no other class). The corresponding inference structure is depicted in Figure 3. Note that we have used generic (abstract) names for the inference roles denoting lecturers, classes and the relations *teaches* and *knows*. Classes are regarded as requests, whereas lecturers 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`.

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).

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).

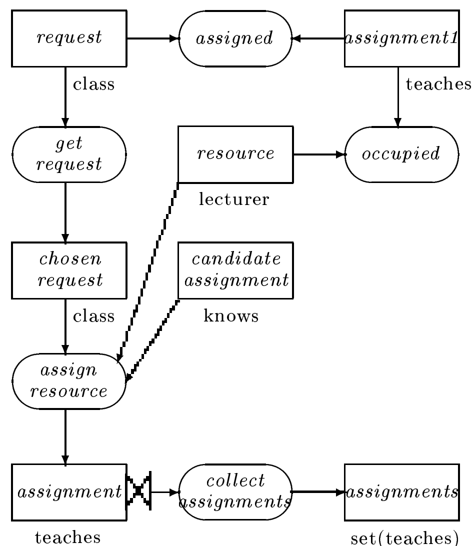

Figure 3: The non-deterministic inference structure for the lecturer allocation problem

Note that the problem-solving process involves computations (deduction) at two different levels: (1) at the domain level (DL deduction) and (2) 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 lecturers (l_1 and l_2), two classes (a *course* c_1 and another class c_2) and in which l_1 and l_2 know both c_1 and c_2.

If we are not told whether l_1 or l_2 are professors or assistants, the system will return two alternative allocations, namely $[(l_1, c_1), (l_2, c_2)]$ and $[(l_1, c_2), (l_2, c_1)]$.

However, if we now specify that l_1 is an *assistant*, only the second assignment will be retained as a consistent one, since an assistant (l_1) cannot teach a course (c_1).

6 Generic inferences and reification

For achieving reusability of the models we need to open the possibility of writing *generic* inferences, i.e. inferences parametrized by the types of their arguments, which will be known only at runtime.

Consider, for example, the following situation. If we want to describe the staff of some research institute, we may want to introduce concepts like *director*, *secretary*, *researcher* and instances like *Tom*, *Joan*, *Mary*, *Peter*, *Fred* etc: $Tom \in director$, $Joan \in secretary$, $Mary \in secretary$, $Peter \in researcher$, $Fred \in researcher$. Note that the concepts *director*, *secretary* and *researcher* represent positions in the research institute. They are therefore not only concepts, but also *instances* of the (meta-level) concept *position*: $director \in position$, $secretary \in position$, $researcher \in position$.

The meta-level concept *position* should not be confused

with the concept *employee*, which is a super-concept of *director*, *secretary* and *researcher* $director \subset employee$, $secretary \subset employee$, $researcher \subset employee$.

Now suppose we would like to write separate inferences for retrieving directors, secretaries and researchers. This would amount to writing three separate pieces of code that are extremely similar. For instance, the inference for retrieving secretaries would be described by

```
domain_link(candidate, simple, concept, secretary).
role_mapping(choose_secretary, candidate, simple, retrieve).
domain_link(chosen_candidate, simple, concept, secretary).
role_mapping(choose_secretary, chosen_candidate, simple, nostore).

choose_secretary([candidate = Cand], [chosen_candidate = Cand]).
```

If we would like to avoid writing three separate pieces of code, we would have to write a *generic* inference that would be parametrized by the *position* of the person we'd like to choose. This can be accomplished by using an input role, called *type*, linked to the concept *position* and which is supposed to specify the position of the person to be chosen.

```
domain_link(candidate, simple, concept, employee).
role_mapping(generic_choose, candidate, simple, retrieve).
domain_link(type, simple, concept, position).
role_mapping(generic_choose, type, simple, retrieve).
domain_link(in, simple, relation, in).
role_mapping(generic_choose, in, simple, retrieve).
domain_link(chosen_candidate, simple, concept, employee).
role_mapping(generic_choose, chosen_candidate, simple, nostore).

generic_choose([candidate = Cand, in = [Cand,Type],
               type = Type, [chosen_candidate = Cand]).
```

Note that we are making use of the built-in relation *in* (*in* links an instance X with a concept C whenever X is an instance of C). It is usually enough to have a single role called *in* (and linked to the predefined relation *in*) since all inferences that need to refer to it can do so. Of course, *in* can be used as input and output role at the same time.[5] Whenever the role *in* is used as an input (output) role, it retrieves (stores) tuples of the form $[X, C]$ whith $X \in C$.

The predefined relation *in* allows therefore a kind of meta-level (generic) inferences which are sometimes very important for writing domain-independent and reusable models.[6] This is achieved by abstracting not only the arguments of inferences, but also their "types". (More theoretical details on reification can be found in [1].)

7 Related approaches and conclusions

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

• 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, as in [2].
• 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. Most KBS development tools do not provide all of these inference services. Also, most of the existing tools provide symbol-level inference services, as opposed to $E_x Claim$, in which knowledge is represented at the knowledge-level (due to its clean integration of the domain-level description logic with the inference and task-level logic programs).
• the description logic used at the domain level can be regarded as a reasonable compromise between expressiveness, readability of formulas and runtime efficiency.
• $E_x Claim$ provides non-deterministic inference and task levels, which rely on a non-monotonic domain level. The lack of non-determinism in a KBS is, in our opinion, an important drawback, since algorithms in KBSs (as opposed to traditional software engineering environments) are complex and usually non-deterministic.
• $E_x Claim$ provides the reified membership relation *in* which can be used to write *generic* inferences. These inferences increase the domain-independence and reusability of models.

Several previous works have dealt with hybrid representation languages combining description logics with logic programming, for example AL-log [4], CARIN [8], F-logic [7]. However, none of these systems allows the KBS developer to specify reusable models, as in $E_x Claim$.

The main goal of this research is the creation of domain-independent executable libraries of problem solving models.

[5] It can be an input for some inferences and an output role for others.

[6] By using a generic inference in our example, we don't have to write any more separate inferences for each type of position (*director*, *secretary* or *researcher*). Not only is it cumbersome to have three identical pieces of code, but these pieces of code would depend on the domain level (the types of positions – *director*, *secretary* and *researcher* are domain-dependent; we cannot change the domain level, for example by introducing a new position, without having to modify the inference level too, since we would have to add a new inference for the new position type. On the other hand, if we are using the generic inference above, we would only have to change the domain link of the role *type*).

Acknowledgments

The research presented in this paper has been partly supported by the European Community project PEKADS (CP93-7599). I am indebted to Doina Ţilivea for developing the graphical user interface of $E_x Claim$, to Jan Wielemaker for support on using the XPCE environment [11] as well as to Ullrich Hustadt and Renate Schmidt for permitting the use of the Motel terminological system [6] in this research.

References

[1] Badea Liviu. *Reifying Concepts in Description Logics.* Proc. IJCAI-97, 142-147.

[2] Breuker J., Van de Velde J. *CommonKADS Library for Expertise Modelling. Reusable Problem Solving Components.* IOS 1994.

[3] Buchheit M., Donini F.M., Schaerf A. *Decidable Reasoning in Terminological Knowledge Representation Systems.* J. of AI Research 1 (1993), 109-138.

[4] Donini F., Lenzerini M., Nardi D., Schaerf A. *A hybrid system integrating datalog and concept languages.* LNAI 549, 1991.

[5] Fensel D., van Harmelen F., *A comparison of languages which operationalize and formalize KADS models of expertise.* Report 280, Universität Karlsruhe, September 1993.

[6] Hustadt U., Nonnengart A., Schmidt R., Timm J. *Motel User Manual.* Max Planck Institute Report MPI-I-92-236, Sept. 1994.

[7] Kifer M., Lausen G., Wu J. *Logical foundations of object-oriented and frame-based languages.* J. of the ACM, May 1995.

[8] Levy A., Rousset M.C. *CARIN: a representation language integrating rules and description logics.* Proc. ECAI'96.

[9] Schmidt-Schauß M. *Subsumption in KL-ONE is undecidable.* Proceedings KR-89, pp. 421-431.

[10] Schreiber G., Wielinga B., Breuker J. *KADS: A Principled Approach to Knowledge-Based System Development.* Academic Press, 1993.

[11] Wielemaker J., Anjewierden A. *Programming in PCE/Prolog.* University of Amsterdam, 1992.