Type Extensibility of a Knowledge Representation System with Powersets

Cécile Capponi

Projet SHERPA, INRIA Rhône-Alpes
655 ave. de l'Europe — 38330 Montbonnot — France

Abstract. This paper deals with KRSs (knowledge representation systems) close to frame systems including description logic and object-based systems. The main relation that leads to inferences is *subsumption* which is usually computed over knowledge descriptions that refer to data types. Although subsumption between knowledge terms is well-defined, its implementation on external data types depends upon the host language which is used for the actual implementation of data types. As a consequence, no KRS is able to integrate a new (external) data type such that its values can be safely involved in subsumption and further inferences. This is the problem addressed in this paper. The solution we propose relies on the design of a polymorphic type system connected to both the KRS and the host language. It is designed so that it can extend the KRS with any type implementation available in the host language (standard atomic types, library, package, etc.). Meanwhile, the values of the new type get safely involved in the KRS reasoning processes. The presented type system prevents the incompleteness of subsumption due to its incomplete processing on external data.

1 Introduction

Knowledge representation systems (KRSs) aim at helping people in an application domain to both organize and describe the knowledge related to this domain. Most of all, these KRSs require the development of expressive representation languages. Description logic systems (DLSs, also called terminological languages, or KL-ONE family [3]) are close to object-based languages, because they allow one to identify and describe knowledge as either groups or individuals.

The problem we address is to provide an object-based KRS (OBKRS) with an extensible collection of computable data types which can be used as parts of knowledge descriptions, while ensuring the type consistency of knowledge bases. In a KRS, knowledge terms are dynamically typed by subsets of values — namely powersets — built from the whole set of values a data type captures. Faced with these powersets, the extensibility of a OBKRS type collection gets different of what classical extensible type systems use to consider.

The paper is organized as follows: in section 2 we introduce the main features of OBKRSs, which are used further on in section 3 to identify the ways data types are processed within knowledge bases. The presented results are important to sketch the type system satisfying our purpose. In the section 4, we present the main organization of the METEO type system we designed for OBKRSs. We then present how METEO may be easily and dynamically used to extend the type collection of the KRS. Finally, in section 5, we compare our study with previous results in the area of DLSs.

2 Basics of Object-Based KRSs

Object-based knowledge representation systems (OBKRSs) stem from frame-based systems, except that they distinguish between frames that denote individuals (instances) and frames that denote sets (class).

Individual elements of the application domain are represented in the knowledge base by *instances*. Individual elements of the application domain can be classified into meaningful groups of somehow similar elements. That widespread organization principle is translated in an object-based knowledge base through the classification of instances into *classes*. Thus a class represents a set of individual elements it describes by means of *attributes*. These attributes are then valued in the description of instances. Attributes basically represent relevant characteristics of individuals. The description of an attribute in a class is the association of a domain (set of values or objects) with the attribute name. Such a domain is specified through *descriptors*.

```
Class today-play-6/49
  Attributes:
    numbers: set integer; card [6;9];
             among [1;49];
    player: in Person;
    outlay: in integer; interval [1;+∞];
    date: in Date; value (12,03,1996);


Instance Pl#2A1 is-a today-play-6/49
  Attributes:
    numbers =  {1 4 9 23 24 41 45}
    player =  Person#621AA4
    outlay =  5
    date =  (12,03,1996)
```

The description on the top is this of a class that is intended to group all the plays of a 6/49 lottery draw of a given day. The description on the bottom is this of an instance of the class (`is-a` relationship). *Person* is a class of the knowledge base intended to represent human individuals, *Person#621AA4* is an instance of *Person*, `Date` is a data type; `interval`, `in`, `value`, `card` and `among` are descriptors. The descriptors associated to *numbers* in the class description together define all sets of a cardinality ranging from 6 to 9, whose elements are integer values ranging from 1 to 49.

Terms of the KRL are given a denotational semantics. The interpretation domain is $\mathcal{U}$; it contains all type values and instances, which are unit elements, as well as lists or sets constructed from unit elements. Let us consider a class attribute $\langle a : d\rangle$, where a is the identifier of the attribute and d is its domain: $d = \delta_1; \cdots; \delta_n$; (each δ_i is a descriptor). The denotation of the attribute in $\mathcal{U}$ is: $\|\langle a : d\rangle\| = \cap_{i\in[1;n]}\|\delta_i\|$ (see [4] for descriptor syntax and semantics). The description of an instance is a conjunction of valued attributes: it is written $I = \{\langle a_i = v_i\rangle\}_{i\in[1;m]}$. Similarly, the description of a class is written $C = \{\langle a_j : d_j\rangle\}_{j\in[1;n]}$. The denotation of the class C in $\mathcal{U}$ is: $\|C\| = \{I = \{\langle a_i = v_i\rangle\}_{i\in[1;m]} \in$

$\mathcal{U} \mid n \leq m$ and $\forall j \in [1;n], v_j \in \|d_j\|\}$.

Two general and fundamental relations exist among terms. They deal with a specific way of organizing knowledge that is usual in many application domains.

First, *attachment* is a binary relation, written $\in_\alpha$, defined between an instance and a class. It signifies that the individual the instance represents belongs to the group which the class denotes in the application domain. Attachment is partly defined on descriptions of involved terms. The restrictions on descriptions are necessary conditions for a whole attachment relationship to be agreed upon in the knowledge base. Let $I = \{\langle a_i; v_i \rangle\}_{i \in [1;m]}$ be an instance and $C = \{\langle a_i; d_i \rangle\}_{i \in [1;n]}$ be a class. Then the attachment between two term descriptions satisfies: $i \in_\alpha C \Longrightarrow m \leq n$ and $\forall i \in [1;m] : v_i \in d_i$. For example, this condition is satisfied conjointly by the instance *Pl#2A1* and the class *today-play-6/49*. Thus the instance *Pl#2A1* can safely be attached to the class *today-play-6/49* according to both descriptions.

Second, *specialization* is a binary relation, written $\leq_\sigma$, defined between two classes. It signifies the inclusion of the group represented by the most specialized class in the group represented by the less specialized class. As for attachment, specialization leads to necessary restrictions on involved term descriptions. Let $C_j = \{\langle a_i : d_{ji} \rangle\}_{i \in [1;n_j]}$, $j = 1, 2$, be two classes. Then specialization between class descriptions satisfies: $C_1 \leq_\sigma C_2 \Longrightarrow n_2 \leq n_1$ and $\forall i \in [1;n_2] : d_{1i} \subseteq d_{2i}$.

In DLSs, both specialization and attachment are merged towards a unique relation, namely subsumption [14]. In the following, we may refer to subsumption in order to denote both specialization and attachment.

Any OBKRS is provided with pre-defined inference mechanisms. They are activated either by the knowledge base user or by another process, in order to deduce new knowledge according to already represented knowledge. Most of these mechanisms stem from frame language inference mechanisms, such as filters or default values. Yet the main inference mechanism is the so-called *classification*, also defined by DLSs [10]. Type checking and type inferences are actually essential during the execution of any of those mechanisms.

3 Characterization of Types in KRSs

Types as dealt in KRSs are attribute descriptions, and further class descriptions as records of attribute descriptions.. They are subsets of values built over the set captured by the initial data type associated with each attribute. Those powersets are incrementally — by inheritance — constructed through descriptors applied on a same attribute.

Whatever the initial type of an attribute is (`int`, `list`, etc.), the powersets are always used on the same way during type checking and inferences, according to a set-based semantics. Given the definitions of necessary conditions for both attachment and specialization and the way attribute types are used, we define an algebra which any data type of the KRS collection should implement.

Each data type $T = \langle V; O \rangle$, where V is a set of values and O is a set of operations on these values, is associated with an algebraic structure over the powerset of V. This algebraic structure specifies the way the KRS operates in a common way on all subsets of any data type. We write this algebra $\mathcal{A}_T = \langle 2^V ; E \rangle_T$ where E contains set operations.

On one hand, data types are implemented in the host language. On the other hand, in order to perform type checking, the KRS relies on the implementation of $\mathcal{A}_T$. Yet a new data type is generally introduced in the KRS through a membership predicate while its actual implementation is a black box. As a consequence, the KRS is no longer able to generate $\mathcal{A}_T$. Then no type checking nor type inference is processed over those powersets.

For example, assuming that `e-int` (even integer) is a data type implemented in the host language, let us then consider the four attribute domains below:

a_1 `in integer; interval` $[0;+\infty]$; a_2 `in e-int; interval` $[4;+\infty]$;
a_3 `in e-int; interval` $[0;+\infty]$; `except` 4 6 8;
a_4 `in e-int; interval` $[0;2]$ $[10;+\infty]$;

The KRS can infer neither $||a_3|| = ||a_4||$ nor $||a_2|| \subset ||a_1||$, unless `e-int` has been defined as a sub-type of `integer`. Yet in this last case, there is no way for the KRS to check the consistency of such a subsumption relationship as long as the system cannot interpret host language predicates.

The KRS cannot guarantee the safety and the completeness of specialization over powersets built over new data types. Thus, subsumption tests which involve classes containing these attribute descriptions are incomplete.

Avoiding the above incompleteness requires the KRS to be able to access the properties and relevant features of any new data type. The way the new data type is related to existing data types must also be known by the KRS. In other word, the solution is to connect any data type $T = \langle V ; O \rangle$ to the KRS while extracting from O the operations and properties that are essential for the managing of 2^V. For example, when a data type is characterized as a total order (property defined in O), any of its powersets may be uniquely expressed by an ordered list of intervals. There is the idea underlying the design of the type system sketched in the next section; the type system gives several implementations of $\mathcal{A}_T$ according to several characterized and important properties. Then, adding a new data type is partly carried out through its attachement to predefined properties. Such an attachment leads to the inheritance of the implementation of $\mathcal{A}_T$ according to the specified generic properties.

4 The METEO Type System

As pointed out in the previous section, a solution to type collection extensibility in an OBKRS is the design of a type system where both host data types and knowledge term types are to be handled. The type system METEO (Module of Extensible Types for Expressive Objects) resulting from our study is presented in this section. It is made up of two type levels:

- *C-types*: encapsulated and formated implementations of host data types
- *δ-types*: canonical expressions for powersets.

4.1 C-types as Implementations of Data Types

METEO contains a whole set of C-types. A C-type is defined, in a minimal fashion, by two operations. Let $\mathcal{U}$ be the set of universal values.

- $\in_T : \mathcal{U} \mapsto$ Bool, is the membership predicate, from any value, to the type T
- $=_T : T \times T \mapsto$ Bool, is the equality predicate between two values of T

A C-type is a data type, that is a double $\langle V; O\rangle$ where V is a set of values and O is a set of operations applying to values. O must include the two above operations, especially since the membership predicate actually characterizes V. In addition to the minimal operations above, many others which are useful in a knowledge base may be implemented in the definition of a C-type. For example, the C-type `list` of METEO contains all the usual operations applying to lists, such as `map`, `catenate`, etc.

C-types are ordered according to C-sub-typing. It is a set-based partial order. $T_1 = \langle V_1; O_1\rangle$ may be a C-sub-type of $T_2 = \langle V_2; O_2\rangle$, written $T_1 \preceq T_2$, if $V_1 \subseteq V_2$. A rich tree of C-types, named $\mathcal{T}$, is provided by METEO, as shown on figure 1.

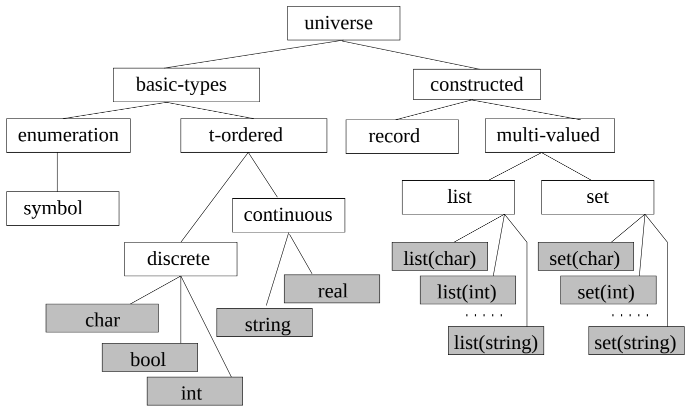

Fig. 1. METEO C-type tree $\mathcal{T}$. Only white C-types actually do exist in the minimal setting of METEO. Although grey tint C-types do not exist in the minimal setting, they can be easily integrated by the designer as addressed further on.

In $\mathcal{T}$, middle C-types are mainly used to factor out generic properties about lower C-types. $\mathcal{T}$ is actually built with object-oriented principles, namely inheritance of properties and methods. Hence, operations that are based on generic properties are defined within middle types as second order functions and are inherited along the C-sub-typing order. For example, the C-type `t-ordered` factors out all kinds of totally ordered types. It implements general algorithms, such as sorting. Then, whatever the total order relation may be, all its C-sub-types (`discrete`, `int`, etc.) inherit those general algorithms.

In addition to its usual definition, each C-type $T = \langle V; O \rangle$ is associated with an algebraic structure $\mathcal{A}_T = \langle 2^V; E \rangle$ where E contains the usual set operations. METEO defines, for each C-type, the implementation of this structure, according to the explicited generic properties of the C-type. Hence, the management of powersets is implemented in C-types, through several operations of E, mainly membership ($\in_{\delta,T}$), inclusion ($\leq_{\delta,T}$), GLB and LUB (resp. $\sqcap_{\delta,T}$ and $\sqcup_{\delta,T}$), and set difference ($\setminus_{\delta,T}$). In METEO, the powersets are named δ-types.

4.2 Internal Language for Powersets

The second level of types in METEO deals with the management of normal forms that express subsets of C-type values, called δ-types.

δ-types are intended to express powersets of C-types. It is a double $\delta t = \langle T; e(T) \rangle$ where T is the C-type the δ-type is attached to and $e(T)$ is a syntactic expression representing the set denoted by δt. The expression $e(T)$ is a normal form, *i.e.* that two different δ-types from the same C-type never denote the same set of values.

The syntax of δ-types is distributed among C-types. It takes into account the specific properties of C-types that are useful in order to make the syntax optimal. For this purpose, C-types are grouped into algebraic classes of C-types, according to the useful properties they share. For instance, powersets of `discrete` are expressed by lists of closed-bound intervals, which specializes the way its C-super-type `t-ordered` expresses δ-types, by lists of open-bound intervals.

Normalization of δ-types in METEO has the same purpose as normalization described in [1]. It is an operation that computes the normal form δt^N associated with any newly created δ-type δt. A normal form is itself a δ-type. METEO deals only with normal forms. The normalization operation is used to guarantee the following property: $\forall \delta t_1^N = \langle T; e_1(T) \rangle, \delta t_2^N = \langle T; e_2(T) \rangle : e_1(T) \neq e_2(T) \Longleftrightarrow \|\delta t_1^N\| \neq \|\delta t_2^N\|$.

Since the δ-type syntax is distributed among C-types, the normalization operation is defined for each class of C-types, according to each syntax. Normalization is achieved through a syntactic rewritting system which is fully presented, as well as proven sound, complete, and confluent in [4].

δ-sub-typing and Other Set-Based Operations. δ-types from the same C-type $T = \langle V; O \rangle$ are partially ordered according to δ-sub-typing. δ-sub-typing is directly issued from set inclusion available in $\mathcal{A}_T$ (section 4.1). Like normalization, δ-sub-typing is implemented according to the distributed syntax, *i.e.* for each class of C-types. For example, δ-sub-typing between two `discrete` δ-types leads to the computation of inclusion between two ordered sets of closed-bound intervals. Following the same sketch, the others set-based operations of E are distributed among classes of C-types.

γ-Sub-Typing. In order to be able to check set inclusion between two δ-types from different C-types which are related by C-sub-typing (*e.g.* `integer` and `e-int`), δ-sub-typing is extended as γ-sub-typing.

γ-sub-typing is a relation that links two δ-types from different C-types. It is computed by the combination of both C-sub-typing and δ-sub-typing. Let $t_1 = \langle T_1; e_1(T_1)\rangle$ and $t_2 = \langle T_2; e_2(T_2)\rangle$ be two δ-types, and $T_2 \preceq T_1$. γ-sub-typing between t_2 and t_1 is written $t_2 \preceq_\gamma t_1$ and is defined as follows: $t_2 \preceq_\gamma t_1 \iff (T_1 = T_2 = T \text{ and } t_2 \leq_{\delta,T} t_1)$ or $(T_2 \prec T_1 \text{ and } t_2 \leq_{\delta,T_2} h(t_1))$, where h is a non-strict inclusion-preserving mapping between δ-types from two different but related C-types [4]. Basically, h maps a δ-type of the upper C-type to a δ-type of the lower C-type, such that inclusion is preserved through h.

4.3 Connecting Meteo to the OBKRS Through δ-types

The design of Meteo is independent of the KRS, yet δ-types are the results of the typing of knowledge terms; *i.e.* Meteo can be instantiated by the typing of a knowledge base. The typing of a knowledge base is thus the core of the interfacing between the KRS and Meteo.

The typing of a knowledge term is made up of two main steps. First, attribute descriptions are translated towards δ-types according to the initial data type of each attribute. Then resulting δ-types are normalized. Only the first step takes part in the interface between the KRS and Meteo, while the other one, unlike it is carried out in [1], is an internal process of Meteo.

Due to the typing of the whole knowledge base, each knowledge term is associated with a δ-type. The set-based operations applying to δ-types are equivalent to the type checking operations of the KRS. Consequently, each operation that was used to manipulate knowledge term descriptions is now associated with a semantically equivalent operation in Meteo that manipulates δ-types. For example, δ-sub-typing between record δ-types is equivalent to specialization between class descriptions.

4.4 Extensibility of Meteo

The extensibility of Meteo is independent of the KRS it is intended to be dedicated to. This section addresses the way Meteo fully deals with type collection extensibility while preserving knowledge base type consistency.

Adding a New Data Type. Due to the C-type hierarchy, Meteo is both a polymorphic and an extensible type system. Adding a C-type may be achieved dynamically by the designer. First, the designer selects a C-super-type. Second, (s)he declares links towards required operations which are implemented in the host language. Let us recall that these operations are mainly the membership and equality predicates, plus possibly some specific operations that represent properties required by the C-super-type. Once the previous two steps are achieved, the new C-type is fully and immediately integrated into Meteo, as any other predefined C-type. Indeed, the formated implementation of the new C-type is generated by METEO by setting up dynamic links between METEO predefined modules and the actual implementation of the new data type. The function used to link a host language data type to METEO is named `create-adt`. For example, adding the C-type `date` $\preceq$ `discrete` is performed by executing the following code:

```
(create-adt 'date
     supertype: 'discrete
     equal: 'dateq
     member: 'datep
     order: 'ldate
     succ: '+day
     pred: '-day )
```

where the quoted symbols are names of functions implemented in the host language. The designer must define the `order` function on `date` values, as well as `pred` and `succ`, because they are required by the given C-super-types `t-ordered` and then `discrete`. These three functions are mandatory for they allow δ-types of `Date` to be represented by lists of closed-bound intervals. In addition, the designer may define any other function applying to `date` values that would be used in the knowledge base.

Extensibility and δ-type Management. The extensibility of METEO is actually protected along δ-type management. Indeed, whenever a designer adds a new data type, (s)he does not have to define the sub-language associated to the new type; the syntax of δ-types is inherited from the C-super-type, as well as all the set-based operations applying to them. This possibility is due to the fact that for any new implemented data type T, the powerset algebra $\mathcal{A}_T$ is automatically generated. Therefore, the designer does not have to deal with δ-type management and further attribute domain management.

As previously said, no KRS is able to deal with the example section 3 in a sound manner. With the connection of METEO to the KRS, not only the new data type `e-int` can be added, but this addition goes with the automatic generation of all set-based operations that manage subsets of even values, such as normalization or subsumption through inclusion. Hence, METEO computes the type of all four attributes:

$$\text{type }(a_1) = \langle\ \texttt{integer}\ ; [0;+\infty_{\texttt{int}}]\ \rangle \qquad \text{type }(a_2) = \langle\ \texttt{e-int}\ ; [4;+\infty_{\texttt{e-int}}]\ \rangle$$
$$\text{type }(a_3) = \text{type }(a_4) = \langle\ \texttt{e-int}\ ; [0;2] + [10;+\infty_{\texttt{e-int}}]\ \rangle$$

Then METEO is syntactically able to infer that $||a_3|| = ||a_4||$, thanks to the normalization process which ensures a syntactic equality over δ-types whenever (and only when) there is a semantic equality. Moreover, γ-sub-typing is used to check whether or not $||a_2|| \subseteq ||a_1||$, where h is the homormorphism that translates δ-types of `integer` towards δ-types of `e-int`: $h(\text{ type}(a_1)\ ) = \langle\ \texttt{e-int}\ ; [0;+\infty_{\texttt{e-int}}]\ \rangle$. Then, using δ-sub-typing among δ-types of `e-int`, it is straightforward for METEO to conclude that $h(\text{type}(a_2)) \leq_{\delta,\texttt{e-int}} \text{type }(a_1)$, *i.e.* according to the definition of γ-sub-typing, $\text{type}(a_2) \leq_\gamma \text{type }(a_1)$.

4.5 About the Implementation of METEO

METEO has been implemented for TROPES [12] with the programming language ILOG-TALK [9]. Since METEO carries out the descriptional operations of TROPES, it cooperates with all inference and checking processes of TROPES, such as classification [5] and categorization [13] among others, as well as with error handling or knowledge revision [6]. At last, METEO weakly cooperates with the constraint management system of TROPES [8] during the computation and storage of attribute domain reductions for arc-consistency.

5 Related Works

5.1 METEO vs. Other Type Systems

No type system has been designed so far to deal with the specificities of a KRS that stem from frames: powersets of any data type are to be dynamically handled in order to ensure the type consistency of knowledge bases. Although some programming languages permit constructions of powersets (*e.g.* `range` over numeric types in Ada), those constructions remain poor and no dynamic management over subsets is allowed. Actually, no type system do handle the dimension corresponding to powersets (δ-types) of usual data types (C-types), whereas it is one of the specificities of METEO.

5.2 METEO vs. KRS Similar Abilities to Manage Types

Type collection extensibility is still a problem in most KRSs in the area of description logic systems, for example. Despite some incomplete solutions, such as TEST-H in CLASSIC, only two systems provide the designer with type collection facilities in such a way that type checking remains sound. The TEST-H predicate of CLASSIC is currently studied in order to deal with concept subsumption [2]. The system K-REP [11] provides the designer with an API in order to facilitate the integration of a new data type T, although (s)he still has to program the set-based functions of $\mathcal{A}_T$. The design of METEO has been achieved according to formal observations close to those B. Gaines reported in [7] regarding data type extensibility and KRSs. As a consequence, the underlying mainframe of METEO is close to the open system of Gaines, especially on the way both deal with class types as constrained record types [4].

6 Conclusion

The usual data types available in a knowledge representation system (KRS), namely the type collection of the KRS, are not always sufficient for the development of knowledge bases that require complex data structures (*e.g.* molecular biology). This problem relies partly on type checking and completeness of inference processes within the knowledge bases.

This paper presented the type system METEO which has been designed to maintain the type consistency of knowledge bases in presence of external data types. METEO is a polymorphic type system due to the hierarchical and generic organization of data types. This means that new data types may be added through dynamic predefined links that METEO sets up between the implementation of the types in the host language and the knowledge representation system. When a new data type is added, the handling of its powersets (which are the types of knowledge term descriptions) is fully and automatically generated by METEO. There is the main originality of METEO, because no existing KRS is able to warrant the completeness of set-based reasoning processes as soon as a new external data type is used in the knowledge base. The main organization of METEO can be safely of use to any description logic system.

References

1. A. Borgida and R.J. Brachman. Protodl : a customizable knowledge base management system. In *1st CIKM*, pages 482–490, Baltimore (MA, US), 1992.
2. A. Borgida, C.L. Isbell, and D.L. McGuinness. Reasoning with black boxes: handling test concepts in CLASSIC. In *International Description Logics Workshop*, Cambridge (MA, US), November 1996.
3. R.J. Brachman and J.G. Schmoltze. An overview of the KL-ONE knowledge representation language. *Cognitive Science*, 9:171–216, 1985.
4. C. Capponi. Design and implementation of a type system for a knowledge representation system. Rapport de Recherche 3096, INRIA Rhône-Alpes, France, January 1997.
5. C. Capponi, J. Euzenat, and J. Gensel. Objects, types and constraints as classification schemes. In *Knowledge Retrieval, Use and Storage for Efficiency*, Santa Cruz (CA, US), August 1995.
6. Y. Crampé. A characterisation of revision in object-based knowledge representation. ISMIS'97, poster Session, Oak Ridge National Laboratory, October 1997.
7. B.G. Gaines. A class library implementation of a principled open architecture knowledge representation server with plug-in data types. In Ruzena Bajcsy, editor, *13th International Joint Conference on Artificial Intelligence*, volume 1, pages 504–509, Chambéry (France), September 1993. Morgan Kaufmann.
8. J. Gensel. Contraintes et représentation de connaissances par objets. application au modèle TROPES. thèse de 3ème cycle, Université Joseph Fourier, Grenoble, France, October 1995.
9. ILOG, Gentilly (France). *ILOG TALK, version 3.1 (Beta 1)*, 1994.
10. R. MacGregor. Inside the LOOM classifier. *SIGART Bulletin, Special issue on implemented knowledge representation and reasoning systems*, 2(3):88–92, June 1991.
11. F.J. Oles, E.K. Mays, and R.A. Weida. The algebraic essence of K-REP. In *International Description Logics Workshop*, Cambridge (MA, US), November 1996.
12. Projet SHERPA, INRIA Rhône-Alpes, Grenoble (France). *TROPES, version 1.0 reference manual*, June 1995.
13. P. Valtchev and J. Euzenat. Classification of concepts through products of concepts and abstract data types. In *1st International Conference on Data Analysis and Ordered Structures*, pages 131–134, Paris (France), June 1995.
14. W.A. Woods. Understanding subsumption and taxonomy: a framework for progress. In J.F. Sowa, editor, *Principles of Semantic Networks*, chapter 1, pages 45–94. Morgan Kaufmann, 1991.

This article was processed using the LaTeX macro package with LLNCS style