Towards An Uniform Language for Knowledge Representation Including Procedural Knowledge Expressed Declaratively

Liviu Badea

Expert Systems Laboratory
Research Institute for Informatics
8-10 Averescu Avenue,
71316 Bucharest
ROMANIA

Abstract: Recent studies in Knowledge Representation have emphasized the importance of a formal theory in this domain. Such a theory should be able to describe not only specific domain objects, but also actions that modify them, meta-rules that govern the application of actions, etc. Among the numerous candidates to the universal language of Knowledge Representation, the term subsumption languages (TSL) seem to be the most appropriate ones because of their elegant uniformly formalized semantics. They seem however to be unable to describe actions, procedures or rules (even though some of them claim to have integrated actions or rules, the integration is only superficial).
This paper presents a term subsumption language able to describe and reason about actions and procedures by treating them exactly like any other language concept. In other words, the procedures are built using the same formal constructors as for any other concept and there is no formal distinction between a procedure and a concept. The feature of uniformity in representation is entirely new. Although some languages (as LISP for instance, but at a lower level) claim that they treat data and procedures uniformly, this is not actually so, because the integration is only superficial (the interpreter of the language treats data and procedures as totally different entities).
In our new Knowledge Representation language, data and procedures are treated by the Assertion Language Interpreter essentially the same way and the difference between a procedure and some other concept is only in the programmer's mind.
Another important feature of this language is its very natural way of building higher order structures (higher order procedures/meta-rules that reason about procedures/actions). This is especially important in view of an observation made by most of AI researchers that a system only behaves intelligently when it is capable of reasoning about its own actions and behaviour (i.e. using higher order logic, which our language can easily provide).
Our new Knowledge Representation language can also be viewed as a mathematical theory of KR and/ or computation, since it provides a well-defined semantics. This semantics has two components: the usual model-based semantics, which we call assertional semantics, and the terminological semantics. As far as we know, the terminological semantics is an original approach to the semantics of a language. Unlike the assertional semantics that describes the various constructors of the language by means of concept/role extensions in models, the terminological semantics is some sort of a meta-semantics as it describes the behaviour of the language constructors without any reference to extensions or models (by only reasoning in the terminological component of the language).
The following TSL constructors are shown to be a minimal set (in order to be able to describe procedures):
and, or (we adopt some kind of "positive logic"), all, $\subset$ (role-value-map), inv (inverse roles), comp (role composition used to build role chains).
We introduce the mathematical concept of role conjugation that turns out to be the correspondent of a recursive procedure call in ordinary languages.
We also show on a toy example how the Assertion Language interprets a procedure exactly the same way as it interprets an instance assertion.
We argue that such a uniform representation language can be used as an "universal" Knowledge Representation language (a standard in KR - namely a Knowledge Interchange Format has been proposed, but we need more than a common format, we need a common theory !).

Keywords: Knowledge Representation, Term Subsumption Languages, procedural knowledge, models of computation, terminological and assertional semantics.

Liviu Badea graduated from the Faculty of Automatic Control and Computer Science, the Polytechnical Institute of Bucharest in 1990.
From 1990 up to present he has been working in the Expert Systems Laboratory of the Research Institute for Informatics. He has also had teaching activities at the Faculty of Automatic Control and Computer Science, where he is also completing his Ph.D. thesis (in the AI field). His research interests include the theoretical foundations of artificial intelligence, logic, knowledge representation and problem-solving in AI and the relationship between mathematics and computer science (especially AI).

1. The Term Subsumption Language

This section is an overview of the term subsumption language (TSL) constructors needed in order to be able to represent actions and

procedures. Such a language consists of a terminological language (the TBox) and of an assertional language (the ABox). The assertional language provides the means of describing the instances of the TBox concepts (which can be viewed as classes of objects). The relations between TBox concepts are expressed by roles. The assertional semantics of the concept and role constructors are defined by their behaviour in extensions (models):

$E[\,(\text{and } C1\ C2)\,] = E[\,C1\,] \cap E[\,C2\,]$

$E[\,(\text{or } C1\ C2)\,] = E[\,C1\,] \cup E[\,C2\,]$

$E[\,(\text{all } r\ C)\,] = \{\, x \mid E[\,r\,](x) \subset E[\,C\,] \,\}$

where $E[\,r\,](x) = \{\, y \mid (x,y) \in E[\,r\,] \,\}$

$E[\,(\text{atleast } n\ r)\,] = \{\, x \mid \text{card}(\, E[\,r\,](x)\,) \geq n \,\}$

$E[\,(\text{atmost } n\ r)\,] = \{\, x \mid \text{card}(\, E[\,r\,](x)\,) \leq n \,\}$

$E[\,(\subset r1\ r2)\,] = \{\, x \mid E[\,r1\,](x) \subset E[\,r2\,](x) \,\}$

$E[\,(=\ r1\ r2)\,] = \{\, x \mid E[\,r1\,](x) = E[\,r2\,](x) \,\}$

$E[\,(\text{and } r1\ r2)\,] = E[\,r1\,] \cap E[\,r2\,]$

$E[\,(\text{or } r1\ r2)\,] = E[\,r1\,] \cup E[\,r2\,]$

$E[\,(\text{inv } r)\,] = \{\, (y,x) \mid (x,y) \in E[\,r\,] \,\}$

$E[\,(\text{comp } r1 \ldots rn)\,] = \{\, (x_0,x_n) \mid \exists\, x_1, x_2, \ldots, x_{n-1}$ such that $(x_i,x_{i+1}) \in E[\,r_{i+1}\,]$, $i = \{0,\ldots, n-1\}$

$E[\,(\text{domain } C)\,] = \{\, (x,y) \mid x \in E[\,C\,] \,\}$

$E[\,(\text{range } C)\,] = \{\, (x,y) \mid y \in E[\,C\,] \,\}$

where E denotes the extension, C, C1, C2 are concepts, r, r1, r2 are roles and x, y are instances.

The concept forming operators are: and, or, all, atleast, atmost, $\subset$ (role value map), $=$, while and, or, inv, comp, domain, range are role forming operators.

For instance, a binary tree can be represented as follows:

```
(concept Tree
  (or Leaf Node))
(concept Leaf :primitive
  Tree)

(concept Node :primitive
  Tree)

(role left-son :primitive
  (and (domain Node)
       (range Tree)))

(role right-son :primitive
  (and (domain Node)
       (range Tree)))

(disjoint Leaf Node)
```

2. Representing Recursive Procedures in the TSL

In this section we show how recursive procedures can be represented in a TSL without introducing new constructors. We note that in all systems and languages created and implemented up to now, the procedural knowledge (functions, actions, etc.) was only partially integrated with the data-description language since it provided some special constructors with some additional procedural semantics (which could not be fully expressed in the semantics of the data-description language). Our representation of procedural knowledge relies only on the data descriptors of the terminological language and their semantics.

A toy example will be given. Suppose we want to write a procedure that determines the leftmost leaf of a binary tree. In PROLOG, such a problem is solved by the following program:

```
left_leaf( Leaf, Leaf ) :-
  Leaf = leaf( Leaf_name ).
left_leaf( node( Left_son, Right_son ), Left_leaf ) :-
  left_leaf( Left_son, Left_leaf ).
```

In our TSL, we can define, for each procedure (or PROLOG predicate), a concept whose instances will represent the activations of the respective procedure. For instance, we can define:

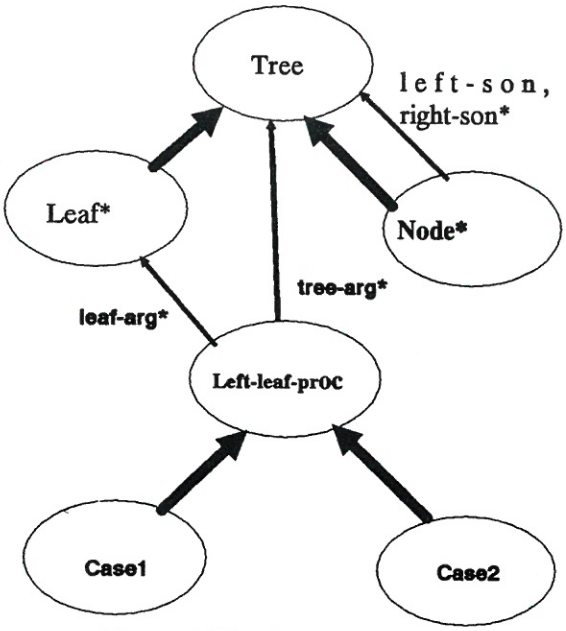

Figure 1. The Concept Lattice

```
(concept Left-leaf-proc
  (or Case1 Case2))
```

where Case1 and Case2 correspond to the two PROLOG clauses. For each argument of the procedure (or PROLOG predicate), the corresponding TSL concept must have a role:

```
(role tree-arg :primitive
  (and (domain Left-leaf-proc)
       (range Tree)))

(role leaf-arg :primitive
  (and (domain Left-leaf-proc)
       (range Leaf)))
```

The first case, Case1, corresponds to the PROLOG fixed-point clause:

```
(concept Case1
  (and (all tree-arg Leaf)
       (= leaf-arg tree-arg)))
```

Note that (= leaf-arg tree-arg) expresses the equality of the two (input and output) arguments of the procedure.

The second case, Case2, contains a recursive call to Left-leaf- proc:

```
(concept Case2
  (and (all tree-arg Node)
       (all (and (comp tree-arg left-son (inv tree-arg))
                 (comp leaf-arg (inv leaf-arg)))
            Left-leaf-proc)))
```

The construction above contains only standard concept forming operators, but it behaves, at assertion time, like a recursive procedure. Note that the differences between data (ordinary concepts) and procedures only exist in the programmer's mind.

Let x be an instance of Case2. Then $x \in$ Case2 iff $x \in$ (all tree-arg Node) and $x \in$ (all (and (comp tree-arg left-son (inv tree-arg)) (comp leaf-arg (inv leaf-arg))) Left-leaf-proc) i.e. tree-arg (x) $\subset$ Node (the tree argument is a Node) and (tree-arg^{-1} o left-son o tree-arg $\cap$ leaf-arg^{-1} o leaf-arg) (x) $\subset$ Left-leaf-proc.

Note that tree-arg (x) is the tree argument of the procedure, left-son o tree-arg (x) is its left son, and tree-arg^{-1} o left-son o tree-arg (x) is an instance y of Left-leaf-proc that has the respective left-son as a tree argument (because tree-arg (y) = left-son o tree-arg (x)).

The recursive call to Left-leaf-proc leads to a construction containing the conjugation of the role left-son through the role tree-arg: (all tree-arg^{-1} o left-son o tree-arg Left-leaf-proc). Such a role conjugation appears whenever the corresponding function is primitive recursive. Thus, the important link between primitive recursion and role conjugation has been revealed by a term subsumption language.

Let us now turn to the functioning of the mechanism at assertion time. Suppose the assertion of an instance x of Left-leaf-proc (in the ABox) having, for example, tree-arg equal to a

binary tree rooted at node12 and with two leaves (leaf1 and leaf2), while leaf-arg remains unspecified (we are practically calling the respective procedure with the given argument):

```
(assert (Leaf leaf1)
        (Leaf leaf2)
        (Node node12)
        (left-son node12 leaf1)
        (right-son node12 leaf2)
        (Left-leaf-proc x)  ;procedure call
        (tree-arg x node12)) ; input argument
```

The ABox instance-recognizer will assert for x the constraints mentioned in the definition of the concept Left-leaf-proc. Because Left-leaf-proc is a disjunction: (or Case1 Case2), the recognizer will first try x against Case1. But it will fail when trying x ∈ (all tree-arg Leaf), because tree-arg (x) is a Node and not a Leaf (since nodes and leaves have been declared disjoint). The recognizer thus backtracks to Case2, (all tree-arg Node) is true for x, so the test should proceed

```
x ∈ (all (and (comp tree-arg left-son (inv tree-arg))
              (comp leaf-arg (inv leaf-arg)))
         Left-leaf-proc).
```

Because tree-arg (x) = node12, left-son o tree-arg (x) = leaf1 and leaf-arg (x) is unbound (unknown), the recognizer will have to check that every instance y with tree-arg (y) = left-son o tree-arg (x) and with leaf-arg (y) = leaf-arg (x) is an instance of Left-leaf-proc. This is the equivalent of a recursive procedure call on the left-son. The instances x and y of Left- leaf-proc can be regarded as activation records of the corresponding procedure.

When checking y ∈ Left-leaf-proc, the recognizer will first try y ∈ Case1, i.e. y ∈ (all tree- arg Leaf) (which succeeds because tree-arg (y) = left-son o tree- arg (x) = leaf1 is a Leaf) and y ∈ (= leaf-arg tree-arg) (which also succeeds because leaf-arg (y) = leaf-arg (x) is unknown). As y is asserted to be an instance of (= leaf-arg tree-arg), leaf-arg (y) = leaf-arg (x) will be bound to tree-arg (y) = leaf1 i.e. leaf-arg (x) = leaf1. Thus the procedure call x terminates with the response leaf-arg (x) = leaf1.

In order to support concept disjunction and inverse roles, the assertional language must provide backtracking and instance retraction (it should be nonmonotonic).

The above example suggests that any recursive procedure can be implemented in the TSL. For instance, an indirect recursion like

```
proc1( arg1( Part1 ) ) :- proc2( Part1 ).
proc2( arg2( Part2 ) ) :- proc1( Part2 ).
```

can be represented by

```
(concept Proc1          (concept Proc2
  (or Case1 ...))         (or Case2 ...))

(role arg1 :primitive     (role arg2 :primitive
(domain Proc1))             (domain Proc2))

(concept Case1                          (concept Case2
(all(comparg1part1(invarg2)) (all(comparg2part2(invarg1))
  Proc2))                                 Proc1))
```

The next example shows the treatment of multiple occurrences of a variable V:

```
p( a(V) ) :- q( b(V) ), r( c(V) ).

(concept P
  (or P1 P2 ...))

(role a :primitive      (role part-a :primitive
  (and (domain P)       (domain Ra))
       (range Ra)))
(concept P1
  (and (all (comp a part-a (inv part-b) (inv b)) Q)
```

```
(all (comp a part-a (inv part-c) (inv c)) R)))

(role b :primitive     (role part-b :primitive
  (and (domain Q)        (domain Rb))
       (range Rb)))

(role c :primitive     (role part-c :primitive
  (and (domain R)        (domain Rc))
       (range Rc)))
```

We may also use "local variables" in a procedure (they are used to store intermediate computations). In the previous example, the role chain (comp a part-a) is used in two different places and could be replaced by a local variable as follows:

```
(role local-var :primitive
  (domain P))

(concept P1
  (and (= local-var (comp a part-a))
       (all (comp local-var (inv part-b) (inv b)) Q)
       (all (comp local-var (inv part-c) (inv c)) R)))
```

3. Terminological and Assertional Semantics

The tendency of uniformization of all structures of a language (whether descriptive or procedural) has its roots in the essay of conceiving a reflective knowledge representation language (i.e. a language that is able to reason about its own constructions). The property of reflectivity can be viewed as a form of informational feedback (that is, the results of the reasoning process may be used to guide further problem-solving) and is an inherent feature required for an intelligent behaviour.

Almost all computer languages let the data structures be viewed more as static and empty frames with a weak semantics (as opposed to the term subsumption languages that provide necessary and sufficient concept definitions and have thus strong semantics). In classic languages, procedures are built using predefined control structures which have predefined semantics usually known by the programmer and not by the system itself, therefore making true reflection impossible. Our approach integrates descriptive and procedural knowledge while allowing procedures to be built only by using the standard concept forming operators with no any other special control operators (which would be primitive to the system itself, i.e. their semantics is not intrinsic to the system and the system would not be able to reason about them).

In order to clarify this distinction, we shall split the semantics of a TSL into two components: the terminological semantics and the assertional semantics. The assertional semantics of the language is the usual model-based semantics that describes the concept and role forming operators in terms of instances (belonging to the ABox). Section 1 presented the assertional semantics of our TSL. It describes how the language interpreter treats instances in the ABox. However, one major weakness is perceivable: the system cannot reason about itself using the assertional semantics (unless it has a set-theory based theorem prover).

Unlike the assertional semantics, the terminological semantics is a sort of meta-semantics as it describes the composition rules of the various language constructors without any reference to instances of the ABox. The terminological semantics is an equivalence relation in the space of all syntactic constructs (i.e. it identifies semantically equivalent syntactic constructs) and has to be consistent with the assertional semantics. An example of a non-trivial terminological semantics rule is the following:

$(\subset r1\ (\text{and}\ r2\ (\text{range}\ C))) = (\text{and}\ (\subset r1\ r2)\ (\text{all}\ r1\ C))$

where r1, r2 are roles and C is a concept.

Proof: $x \in (\subset r1\ (\text{and}\ r2\ (\text{range}\ C)))$ iff

$r1(x) \subset r2(x) \cap C$ iff

$r1(x) \subset r2(x)$ and $r1(x) \subset C$ iff

$x \in (\subset r1\ r2)$ and $x \in (\text{all}\ r1\ C)$ iff

$x \in (\text{and}\ (\subset r1\ r2)\ (\text{all}\ r1\ C))$ q.e.d.

In an ideal case we would like a complete terminological semantics, but it seems to be a very difficult problem in a reasonably expressive

language like ours(Romanian). If the terminological language incompletely treats a given constructor, whenever such a constructor is encountered in a concept definition, that concept should be declared primitive (i.e. incompletely defined in the TBox). Whenever the terminological semantics is weaker than the assertional one, we could expect that some inconsistencies only manifest at assertion time.

4. Concluding Remarks

Our approach to expressing procedural knowledge by only using common concept forming operators can be viewed as a mathematical theory unifying the theory of computation and of knowledge representation. It also proves, and this is of no little importance, that a term subsumption language (with the following constructors: and, or, all, $\subset$, inv, comp) has the computational power of a language of general recursive functions and is so computationally equivalent to a Turing Machine.

BIBLIOGRAPHY

BAADER, F., **Terminological Cycles in KL-ONE Based Knowledge Representation Languages.**

BRACHMAN, R.J., GILBERT, V.P.and LEVESQUE, H.J., **An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of KRYPTON,** Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, CA.,1985.

BRACHMAN, R.J. and SCHMOLZE, J.G. **An Overview of the KL-ONE Knowledge Representation System,** COGNITIVE SCIENCE, 9(2), 1985.

BRACHMAN, R.J. and LEVESQUE, H.J., **The Tractability of Subsumption in Frame-Based Description Languages,** Proceedings of the American Association for Artificial Intelligence, Austin, Texas, 1984.

BRACHMAN, R.J. et al, **Living with Classic,** in J.F. Sowa (Ed.) Principles of Semantic Networks, pp. 401-456.

MCGREGOR, R., BURNSTEIN, M.H., BERANEK, B. and NEWMAN, **Using a Description Classifier to Enhance Knowledge Representation,** IEEE EXPERT, June 1991.

MCGREGOR, R., **The Evolving Technology of Classification- Based Knowledge Representation Systems,** in J.F. Sowa (Ed.) Principles of Semantic Networks , pp. 385-400.

NEBEL, B., **Computational Complexity of Terminological Reasoning in BACK,** ARTIFICIAL INTELLIGENCE, 34, 1988.

PATEL-SCHNEIDER, P.F., **A Four-valued Semantics for Terminological Logics,** ARTIFICIAL INTELLIGENCE, 38, 1989, pp. 319- 351.

PATEL-SCHNEIDER, P.F. et al, **Term Subsumption in Knowledge Representation.** AI MAGAZINE, Summer 1990. p. 16.

PELTASON, C., SCHMIEDEL, A., KINDERMANN, C. and QUANTZ, J., **The BACK System Revisited,** KIT Report, Technische Universität Berlin, September 1989.

SCHILD, K., **Undecidability of Subsumption in U,** KIT Report, Technische Universität Berlin, October 1988.

WOODS, W.A., **Understanding Subsumption and Taxonomy,** in J.F. Sowa (Ed.) Principles of Semantic Networks, pp. 45-94.

YEN, J., JUANG, H. and MCGREGOR, R., **Using Polymorphism to Improve Expert Systems Maintainability,** IEEE EXPERT, Spring 1991.