Beyond Objects

Luc Steels

Artificial Intelligence Laboratory,
Vrije Universiteit Brussel,
Pleinlaan 2, B-1050 Brussels, Belgium,
E-mail: steels@arti.vub.ac.be

Abstract. An agent-oriented approach to the design, implementation and maintenance of reusable software components is discussed. Main features of the approach are: (1) the use of a formal description to enable meta-level reasoning and reflection, (2) the use of automated coding and instantiation of computational fragments, (3) the use of a distributed network and interfaces allowing the browsing, indexing and retrieval of fragments from remote sites.

1 Introduction

The world of information processing is currently undergoing a phase transition due to the widespread availability of networked computers. In the United States alone, the Internet had early 1994 15 million users and has been growing at a rate of 10 percent per month. This development is causing a sudden exponential growth in the usage of computers and telecommunications. Email and other networking activities were for a long time the almost exclusive province of academic researchers, but at the moment commercial and personal transactions are increasingly taking place over computer networks, thus for ever transforming the way developed societies operate. In the not too distant future, almost everybody will have their personal digital assistant that is directly connected via telecommunications to more powerful computational networks.

One of the most interesting functionalities that is becoming practical in this highly networked, computer-rich environment, is a software agent ([15]). Software agents are computer applications which autonomously execute a particular task for their owners, for example find the best airline connections and schedule other practical details of a trip. Agents are in principle continuously in operation, for example to monitor e-mail traffic, or to handle requests from other agents. They can delegate subtasks to other agents, and take the initiative to seek out needed information. Software agents can migrate from one machine to another, seeking out opportunities and resources, and 'following' their owners wherever they go. More ambitiously, we expect that they learn and adapt like human agents.

Object-oriented programming [16] and the message passing framework [6] appear to be suited very well for designing and implementing software agents that operate in distributed environments. But they only provide the computational framework. Much further work needs to be done on many issues: defining common languages for communication between agents, study mechanisms for

achieving autonomy, have global scripts so that agents may install themselves on many different machines, etc. A lot of research on this topic has been conducted in the context of AI research on distributed intelligence [2], [4]. Also research on intelligent robotic agents within the context of Alife may prove of high value [13].

This paper reports on research to build software agents that have a formal description of their internal structure and functioning so that (1) other agents can find out what they are capable of doing, and (2) they can reconfigure themselves when needed to cope with changed environments or changes in tasks. The formal descriptions act as the meta-level for an object-oriented software layer. Agents are assumed to be capable of jumping from the programming level to the meta-level (for example to reconfigure themselves) or map the meta-level onto the object-level (for example when they need to re-install themselves on another machine). Agents that have these properties will be called *reflective agents.* Some researchers have already suggested that a reflective capability is needed (see in particular [3]) and others have suggested the use of formal languages to define the capabilities of agents and use it as a basis for interaction among agents [9] . Our work contributes by using a particular kind of description for the meta-level inspired by the notion of the knowledge level, first introduced by Newell [8], [14], and by using techniques from program synthesis and formal specification theory to relate the meta-level with the object-level. This work builds further upon results obtained in the context of enhancing the knowledge engineering process [13]. Most of the ideas and techniques contained in this paper have all been implemented in the form of a workbench known as KREST. This workbench is already distributed over 30 sites throughout Europe and is available through ftp. A large variety of applications, mostly in the domain of knowledge-based applications has already been constructed. The encapsulation in terms of agents as discussed in this paper is in progress.

The paper has the following parts. First the general principles of our reflective agents are discussed. Then three scenarios of usage are presented. The first scenario concerns the development of an application on a single machine. The second scenario focuses on the sharing of agents over multiple machines and through multiple projects. The final scenario illustrates the use of reflection. Some general conclusions and future research topics end the paper.

2 The architecture of reflective agents.

A software agent will need on the one hand some basic functionalities to negotiate entry into a computer system, migrate over a network, install and invoke itself, report back to other agents that request its services, etc. These functionalities are essential but are not discussed further in this paper. On the other hand, an agent needs functionalities that are specific to the tasks that it executes for its owner. These task-specific functionalities will here be our primary concern.

2.1 Types of agents.

We make first of all a distinction between four types of agents:

- **Task agents**: They are responsible for a particular task, something that needs to be accomplished. For example, look up an element in a database, compute a graph, display a picture, acquire through a dialog the structure of a causal network, filter out and re-route e-mail messages, etc.
- **Resource agents**: They are responsible for a particular resource (an interface to a user, a data object, a model). Resources can be sources as well as sinks of information.
- **Method agents**: They are responsible for a particular method, i.e. a procedure that can be used to resolve a task.
- **Project agents**: They are responsible for maintaining a group of agents which share a common context. This includes a set of task agents relevant for achieving a particular project, as well as resource and method agents.

Each of these agents can be generic or specific. A generic agent holds descriptions and structures that only partially define the agent. For example, a generic task agent may have information about a task decomposition and needed resources but may lack information about which specific resources are available for the task. Or, a generic causal network agent may have information on how to represent causal networks but may not contain information about any specific network in particular.

Task, resource, and method agents are related to each other. For example, a method agent may be associated with a particular task agent, a task may have access to several resources to accomplish the task, a task may be related to several subtasks. The agents involved in a particular relation are said to form an *agent network*. The relation between tasks, resources and methods is represented in *dependency diagrams* such as the one in figure 1. and in *task structures* which represent task/subtask relations. The agents which form part of a network must first be located (or locate themselves) inside a project agent. In addition, we have introduced a special agent called a *kit*, which acts as a support agent for the construction of other agents. The term kit emphasises that this agent consists largely of reusable agents or agent network fragments which can be used in the assembly of other agents and agent networks. A kit maintains the vocabulary for building up formal descriptions as explained in more detail later. Kits are organised in inheritance hierarchies where lower level kits are refinements of higher level kits. At the top we find a so called *base kit* which contains the most fundamental vocabulary, e.g. terms like task, method, resource, or terms for defining abstract datatypes, computational constraints on methods, etc. The base kit contains mappings from formal descriptions to code as explained later. More specialised kits contain vocabularies for a particular domain, for example a scheduling kit contains terms for scheduling tasks. They contain fragments which are agent networks (possibly consisting of generic agents) relevant for the domain of scheduling, for example a particular task decomposition for solving scheduling

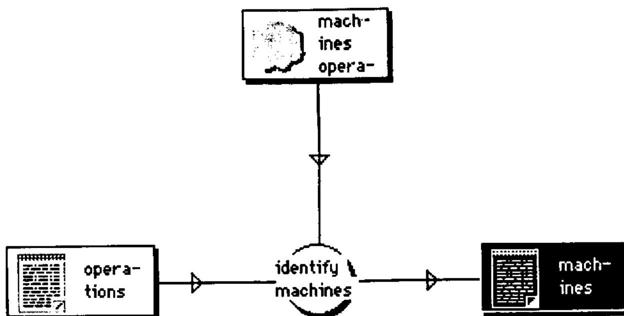

Fig. 1. A dependency diagram for a task that acquires a causal model in the form of a network of symbols by interacting with a domain expert.

problems. The base kit (and its refinements) 'know' about each other in the sense that they can identify the most relevant kit for working on a particular project.

2.2 Agent components

Each agent has three major components: (1) a formal description of the capabilities and internal structure of the agent, (2) code fragments implementing these capabilities, and (3) execution objects, i.e. instantiated datastructures, ready-to-run methods, etc.

(1) The formal description.

The formal description spans a continuum from 'knowledge level' descriptions that are close to the conceptual domain of the user to 'symbol level' descriptions that define the abstract datatypes and constraints on the methods used to achieve a task. The formal description describes the task that the agent can perform as well as the decomposition into subtasks, the resources needed to handle a task, and the methods that will be used. The formal description could, in the case of generic agents, be partial. For example, it could consist of the description of a task together with a definition of the number and kind of resources, but without a method to achieve the task. Or it could consist of the definition of the contents of a domain model but without a specification of how these contents will be represented (as a network, hierarchy, set of symbols, etc.). The repertoire to build up formal descriptions should at any time be extendable.

We use at present feature structures to formulate formal descriptions (see figure 2). A feature structure is a record-like structure that consists of a set of attributes and associated values, together with equality constraints among values of attributes. A value may itself be a feature structure. Feature structures have been widely used in natural language processing but are a general representational formalism equivalent to predicate calculus in expressive power [7]. There is an inferencing mechanism operating over feature structures which is akin to

unification. Feature structures have therefore not only a formal semantics, but also an inferential semantics which is effective, i.e. deductions can be made in bounded time.

```
[content-form   [duplicates       no
                 elements         [content-form   [136]  symbol]
                 ordered          no
                 structure-type   collection
                 structured       yes
                 uniform          yes           ]
 content-type   components
 fills-roles    {[owner       [228]  identify machines
                  role-name   [237]  Covering set     ]}          ]
```

Fig. 2. Formal descriptions of agents take the form of feature structures. The example shows a feature structure for a method agent.

The vocabulary out of which the features in a formal description can be defined constitute a particular *ontology*. At the moment we define the ontology by listing the terms and constraints on the terms (e.g. which possible values a feature may have). Such an ontology is maintained by a kit. In principle, the ontology could be further constrained using a formalism like KIF [5], so that wider interoperability would be guaranteed.

A formal description in the form of a feature structure can be turned into an ASCII representation and as such be transmitted without any restriction from one machine to another anywhere on the network. Receiving sites must of course have the kits necessary to work with the formal representation, e.g. to extend formal descriptions or to map them onto code.

(2) Code fragments

Some agents have representational or computational abilities. This then requires datastructures and/or runnable procedures. A code fragment contains the code to instantiate these datastructures or procedures in a textual format so that it can be stored and transmitted. Figure 3. contains an example of a code fragment based on an object-oriented implementation (CLOS). The code defines a set of symbols which is partially filled by various instances of symbols. This code could be the definition for a datastructure representing "a list of possible symptoms". Clearly the code fragment is a way to get persistence and to transmit agents from one computer to another one, if both have interpreters for the same language.

(3) Execution objects.

Execution objects are the datastructures and procedures as installed on a particular machine. They are the most compact and efficient implementation

```
'identify machines (...g Original:methods:)
(make-instance 'cl-symbol-level-method
  :name '"identify machines"
  :CLOS-method 'cover
  :roles '(("Set to be covered" ("operations" SL-MODEL))
           ("Mapping" ("machines operations mapping" SL-MODEL))
           ("Covering set" ("machines" SL-MODEL) ))
```

Fig. 3. This figure shows a code fragment using CLOS. The fragment defines the form and contents of a datastructure.

but cannot be transferred. Figure 4. shows an example of an execution object.

```
#<CL-SYMBOL-LEVEL-METHOD #x9F3689>
Commands   Resample   Edit   Inspect
#<CL-SYMBOL-LEVEL-METHOD #x9F3689>
Class: #<STANDARD-CLASS CL-SYMBOL-LEVEL-METHOD>
Wrapper: #<CCL::CLASS-WRAPPER CL-SYMBOL-LEVEL-METHOD #x9F31D1>
Instance slots
NAME: "identify machines"
CLOS-METHOD: COVER
ROLES: (("Set to be covered" ("operations" SL-MODEL)) ("Mapping" ("machines oper
```

Fig. 4. Example of the execution object for the code fragment shown earlier. The figure shows the object as seen through the inspector.

3 Scenario 1: Agent development

We now describe a first typical scenario on how a single application (or set of related applications) may be built to run on a single machine. Development starts by creating a project agent which will act as the context for further development. A kit is chosen with which the agents in a project will be constructed. By default, this will be the base kit. A project agent has an interface to access the different agents associated with it (task, method, and resource agents) (figure 5.).

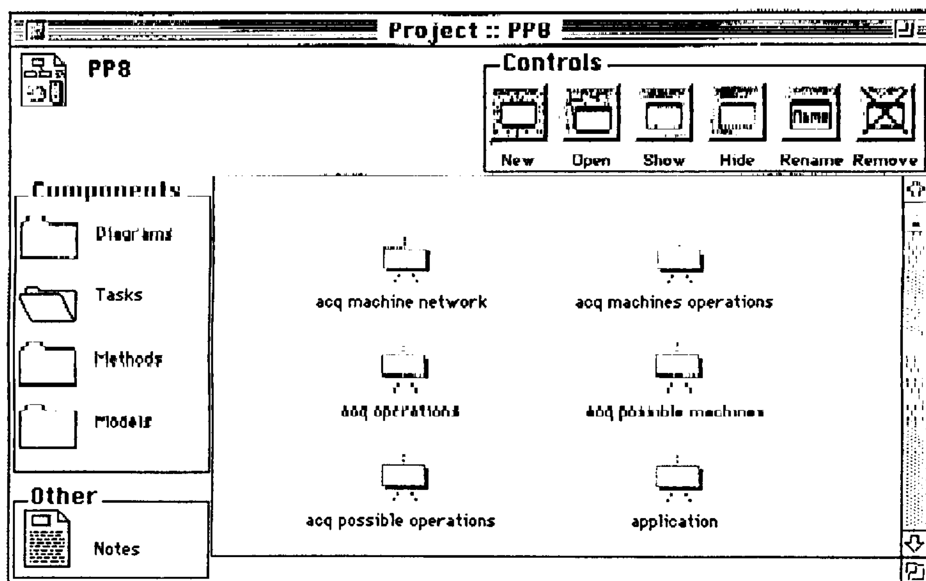

Fig. 5. Figure of interface to a project agent. The task agents are highlighted and are accessible by clicking on them. New agents can be constructed at any time.

The development of specific agents within a project starts with the definition of parts of the formal description. Each agent has an interface which shows the formal description as well as other graphical representations to show dependencies to other agents (figure 6.). The formal description is editable under the guidance of the kit associated with the agent. The kit ensures that only defined terms can be used to formulate feature structures. A kit enforces constraints that are associated with certain features. The formal description is extended through manipulations of graphical representations. For example, a resource can be associated with a task by graphically making a link between the two. The formal descriptions of both agents involved are then updated.

The formal description will on the one hand contain domain-specific features relevant for indexing or for recording design decisions. But it can be expanded to contain information relevant for coding. The basekit [1] that is currently operational contains the vocabulary for describing a wide arsenal of fundamental datatypes (sets, symbols, numbers, trees, networks, etc.) as well as fundamental methods operating over these dataypes. We have developed the necessary components so that the basekit can construct code and execution objects based on formal descriptions. Using these tools, a developer can instantiate execution objects and code fragments without having to write code herself. There is a way to inspect the code associated with the agent through an editing window, and there is a way to inspect the execution objects through an inspection window. Many datastructures have an associated browser to see the contents of the datastructure independently of how they have been represented internally.

When a network of agents has been completely worked out, a running application is automatically synthesized. This requires that there is at least one

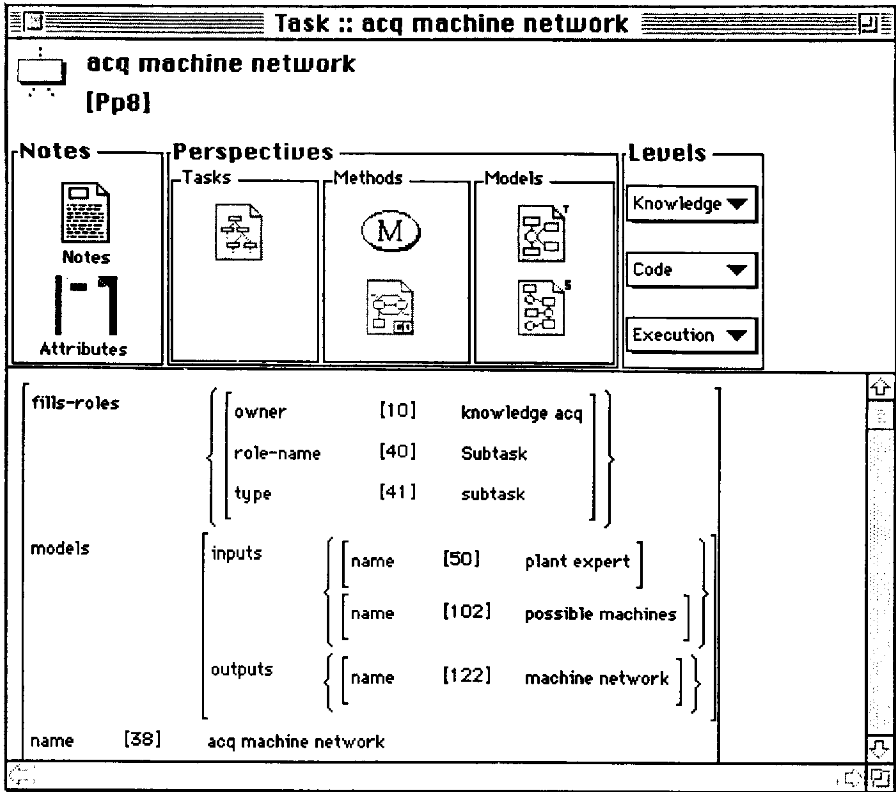

Fig. 6. Figure of interface to a task agent. There are ways to navigate to the task structure and the dependency diagram associated with the task. There is a window (at the bottom) that contains the formal description.

task agent, that a method agent has been associated with the task, and that the resources required by this method are available as fully instantiated agents. The application can then be activated by clicking on the task agent, similar to the way an application is activated on a desktop.

4 Scenario 2: Transfer of agents between different projects

We now describe a second scenario how agents are exported and incorporated in other projects, possibly across the network. An agent or a coherent collection of agents can be cut out of a project and installed as a fragment in a kit. The agents can be at different levels of grainsize (i.e. their formal descriptions need only to be partial) and the code fragment can or cannot be included. Execution objects are never included. For a particular fragment, one agent is chosen as the major focus of attention. There is support in the workbench for performing these operations (figure 7.).

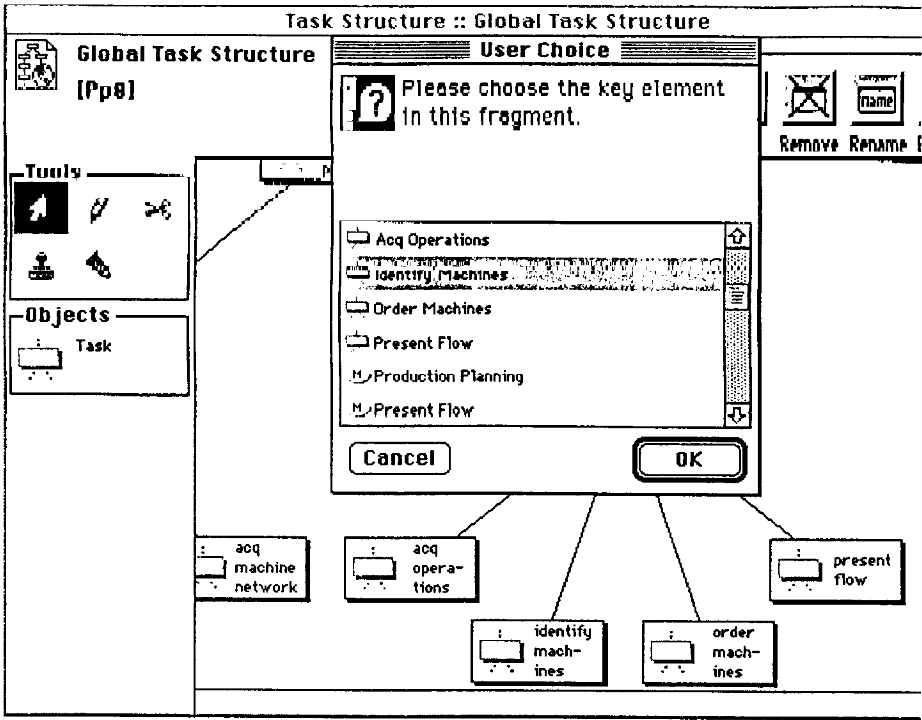

Fig. 7. Formulation of a fragment which is put in an application kit. The user needs to select an agent which acts as the focal point of view of the fragment.

A fragment can be retrieved out of a kit and imported in a project which is being built. Once a fragment is imported it can be properly inserted using cut and paste operations and a glueing operation which allows the unification of two agents. Formal descriptions play an important role because they are a basis for testing whether fragments can be fused or not. Because code and execution objects are constructed automatically by the base kit, they can be reconstructed in a new project and this way a new application can be made operational, possibly on another site.

To support indexing and the distributed use of fragments, we use the World-Wide Web as a general interface to a kit. Fragments are 'published' in a kit catalog located on a KREST server and they can then be imported in new applications.

5 Scenario 3: Reflection

In one mode of operation, a developer or user is in complete control: she

constructs interactively the formal descriptions, extends or refines the kits, causes execution objects to be created, browses through the kit catalog and retrieves fragments. We can expect however that increasingly the agents will need to start having a 'life of their own' because the complexity of managing large collections of kits distributed over networks of computers. This is where reflection comes in.

Reflection is the capability of a computer system to jump out of an execution process (for example when an error occurs), perform reasoning or computation at a meta-level, translate the impact of the meta-level decisions back to the execution level and proceed the computation [10]. This capability is possible in the KREST framework because of the availability of formal descriptions for every unit of code/execution, the availability of a formal inferencing schema at the meta-level, and the availability of a facility that automatically translates formal descriptions to the code/execution level. Some concrete examples of usage of reflection facilities in the KREST framework are reported in [11].

6 Conclusions

This paper reports on research laying the foundations for an agent-oriented architecture that supports the sharing and reuse of software components in a distributed environment. We believe that the essential ingredients to make this happen are (1) powerful formal descriptions and associated inference calculi, (2) a knowledge level framework for describing applications, (3) a 3-layered structure of any application fragment (formal description, code, and execution), (4) agentification of the components so that they can move independently and interact as first-class citizens with other components, and (5) publishing in a distributed context such as offered by the Worldwide Web. Our work so far should be seen as experimental in nature, despite the fact that the facilities discussed in this paper have all been implemented. Much more work needs to be done. For example, many details and conceptual gaps need to be cleared up in relation to the indexing of fragments in very large kits, the fusion of fragments into applications, the cutting operations to remove a fragment (while absorbing just enough context to make the fragment usable), and so on. On the other hand, it seems that we are approaching a new era of computer usage in which distributed computation and independently operating software agents are becoming a reality.

7 Acknowledgement

A large number of highly capable researchers have contributed to the ideas and programs discussed in this paper. Angus McIntyre constructed the first implementations of the KREST workbench in the in the context of the ESPRIT project CONSTRUCT. More recent technical contributions were made by Koen de Vroede (who implemented the first versions of the base kit), Luc Goossens (who has played a major role in making the feature structure formalism usable), Aurelien Slodzian (who implemented the first reflective capabilities) and

Kathleen Van den Abbeele (who implemented the distributed usage and servers through the World Wide Web). Sabine Geldolf, Viktor Tadjer, and Roumena Polianova have made important contributions to develop applications. Walter Van de Velde has made important conceptual contributions. This research is supported by the IUAP centre of excellence project of the Belgian Government.

References

1. de Vroede, K. , L. Goossens and A. Slodzian (1994) The structured base kit. VUB AI lab Memo.
2. Demazeau, Y., J.-P. Muller and E. Werner (1990) Decentralized AI 1,2, and 3. North-Holland, Amsterdam.
3. Ferber, J. and P. Carle (1990) Actors and agents as reflective concurrent objects: a Mering IV perspective. In: Proc. of the 10th Intern. Workshop on Distributed AI. Austin Texas.
4. Gasser, L. and M. Huhns (eds.) Distributed Artificial Intelligence. Vol. 2, Pittman, London.
5. Genesereth, M. and R. Fikes (1992) Knowledge Interchange Format, Version 3.0 Computer Science Department Stanford University. Tech Report Logic-92-1.
6. Hewitt, C. (1973) A Universal, Modular Actor Formalism for Artificial Intelligence. Proceedings of IJCAI 1973.
7. Johnson, M. (1991) Features and Formulae. Journal of the Association for Computational Linguistics. Vol 17, nr 2.
8. Newell, A. (1982) The Knowledge level. Artificial Intelligence, 18, 87-127.
9. Shoham, Y. (1993) Agent-oriented Programming. Journal of Artificial Intelligence. [to appear]
10. Smith, B. (1984) Reflection and semantics in LISP. In: Proc. 11th ACM Symposium on Principles of Programming Languages, 23-35, Utah.
11. Slodzian, A. (1994) Knowledge level reflection in practice. VUB AI laboratory. Master's thesis.
12. Steels, L. (1992) The componential framework and its role in reusability. In: David, J.M., and J.P. Krivine (1992) Second Generation Expert Systems. Berlin: Springer Verlag.
13. Steels, L. (ed.) (1994) The biology and technology of intelligent autonomous agents. Springer Verlag. Berlin.
14. Steels, L. and J. McDermott (eds.) (1994) The knowledge level in expert systems. Conversations and Commentary. Academic Press. New York.
15. Tokoro, M. (1993) The Society of Objects. Invited talk at OOPSLA'93. Addendum to the OOPSLA'93 Proceedings.
16. Yonezawa, A. and M. Tokoro (eds.) Object-oriented Concurrent Programming. MIT Press, Cambridge Ma.
17. Wielinga, B. and F. Van Harmelen (1993) Knowledge level reflection. In: Steels, L. and B. Lepape (eds.) (1992) Enhancing the knowledge engineering process. Contributions from ESPRIT. Amsterdam: Elsevier Publishing.