# Towards A Knowledge Representation Language Based On Open Architecture Model (OAM)

Mahmoud Rafea
Central Lab. For Agricultural Expert Systems (CLAES), P.O. Box: 100 Dokki, Giza, Egypt

Khaled Shaalan                    Ahmed Rafea
Computer and Information Science Dept., Institute of Statistical Studies and Research (ISSR), Cairo Univ., 5 Tharwat St., Orman, Giza, Egypt

E-mail:{mahmoud,shaalan,rafea}@esic.claes.sci.eg

## Abstract

This work was initiated for the development of a Knowledge Representation Object Language (KROL), implemented in SICStus Prolog, that supports the integration and cooperation of multiple-problem solving methods, inference mechanisms, and software programs. This paper addresses an environment based on OAM which is the foundation of KROL. OAM consists of components which are called agents. The cooperation of these components support the process of developing expert systems. The key component of this model is a cooperator agent that uses a reasoning strategy and blackboard structures. A number of problem solving-methods have been implemented and integrated successfully with the proposed architecture. Consequently, domain knowledge can be represented using heterogeneous representation which is the major landmark of the second generation expert systems.

**Keywords:** knowledge representation, knowledge acquisition, models integration, problem-solver, reasoning strategy, blackboard, software agents.

## 1 Introduction

Knowledge acquisition is dominated by two basic ideas. The first is that knowledge acquisition is an active modeling process, rather than a transfer process moving knowledge from one place to another. The second is that the modeling process must take place at a higher conceptual level that abstracts from implementation detail [1]. The introduction of a conceptual level improves the efficiency in the process of building knowledge systems. The conceptual level allows a high level of defining reusable specifications of knowledge modules to be customized and assembled into systems [2]. This leads to an expert system design that reflects and makes explicit the structure and function of the involved knowledge thus improving its quality.

This paper describes the Knowledge Representation Object Language (KROL) which is at the core of a complete environment for the development and use of large knowledge bases, characterized by the cooperation and integration of components which are called agents. Furthermore, the approach followed is based on acquiring knowledge in relation to what has been called: *the open architecture model (OAM).* In this model, domain knowledge can be presented using heterogeneous representation which is the major landmark of the second generation expert systems [12]. This will facilitate the knowledge engineering tasks as knowledge engineers and/or programmers will need not to twist things in order to adapt the knowledge of a problem to a single model.

The motivation behind this work is presented in Section 2. In Section 3 we give an overview of the OAM. In Section 4 we give a brief and informal summary of the proposed language. Section 5 demonstrates the applicability of KROL as an implementation language for agents of the OAM. Moreover, the main components that are implemented and integrated using the OAM are addressed. Section 6 concludes the article.

## 2 Motivation

The increasing desire to put expert systems into particular use has motivated some researches to look into software engineering aspects of developing expert systems. The current and future trends [11] in developing expert systems shows that the direction is towards having more structural and formal development approaches, leaving behind the traditional exploratory development approach either as totally being our of favor or as just a small part of a more comprehensive approach. This is because the traditionally popular exploratory approach has been criticized as being weak due to its ad hoc nature and lack of underlying engineering principles.

The concepts and implementations, presented in this work, have been produced as a result of suffering while developing a number of expert systems, ranging from small to sufficiently large ones. The work presented here is successfully used in developing deployed expert systems [3]. The proposed model is employed in implementing a language that supports knowledge systems development. This language is dynamically growing to cope with the needs of expert system development. This is in fact because of its nature, which facilitates the integration of new software and problem-solving methods.

The first thing to think about when considering an efficient knowledge representation language, is the problem-solving methods that this language should contain. One cannot ignore rules when considering real expert systems because of their simplicity in representing associations and domain expert rules of thumb. Also, one cannot ignore the evolving problem-solving methods which have established an excellent reputation like case-based-reasoning (CBR) [4], [5] and [6], generic tasks [7] and [13], such as hierarchical classification (HC-GT) [8], and also, one cannot ignore future invented problem solvers.

Secondly, it was found that the problem of integrated knowledge systems, is how the OAM will be implemented and how its component will be built so that they become loosely and/or tightly coupled. Fortunately, object-oriented (OO) techniques, including OO design (OOD), OO programming (OOP), and OO databases (OODB), are enabling software technology that attempt to fulfill the functional and computational needs essential to build such systems.

Prolog has been a primary language for artificial intelligence. We have chosen SICStus Prolog as implementation language for our proposed language for many good reasons, including: portability to many computer systems, interface between C and Prolog, the range of libraries that have developed, and our experience in implementing SICStus

Prolog Objects System (the short name is Prolog Objects) in cooperation with the Swedish Institute of Computer Science (SICS).

## 3 The Open Architecture Model (OAM)

OAM can be defined as a system that can incrementally integrate different problem-solvers, inference methods, and software programs. We are going to refer to the inference methods as primitive agents and to software programs as software agents. By software agents we mean DBMS, simulation programs, network programs, etc. Notice that the term agent will be used to reference all or one of the OAM components, depending on the context.

Accordingly, the purpose of this model is to exploit all the available *agents***,** in handling a single domain problem, through cooperation. The handling of the domain problem includes using the proposed model for knowledge acquisition and reasoning through these agents. If an agent can be used to acquire a particular problem, easily and naturally, in some domain and another agent can acquire another problem in the same domain, why not use both agents in handling problem-solving in this domain. In fact, this is how humans act. These criteria have been noticed by many people working in the field between different problem-solvers. For instance, CBR systems are well suited for experience-rich but knowledge-poor domains [9].

This cooperation necessitated the definition of the relationship among the agent or what we call a reasoning strategy (RS). For instance, the RS for diagnosis may be as follows:

- try rules
- if rules cannot reach a diagnosis, try HC-GT
- if HC-GT cannot reach a diagnosis, try CBR
- if CBR cannot reach to diagnosis, try asking the user
- if the user answers and a diagnosis is available, acquire this diagnosis.

Cooperation can be performed at two levels: a macro-integration level and a micro-integration level. The previous example demonstrates the macro integration level. Micro-integration is required when an agent asks another to solve part of its problem. An example of micro-integration is the activation of a database query agent, by another agent, to get a value for a particular attribute. If the query fails, another agent may be called, e.g., call the user-interface agent to get the value from the user. It should be remarked that this technique can be enhanced to cope with the multi-agent distributed knowledge system.

Another example of cooperation, is when the RS is used to integrate two or more expert systems, in the same domain, built by different domain experts and knowledge engineers. These expert systems cooperate to reach a conclusion. If they generate the same conclusion, the conclusion is strongly recommended. But, if they disagree, a strategy must be defined to resolve the conflict.

In order to achieve this behavior, it was found that it is necessary to have a top level agent which acts like a cooperator. The role of this agent is to cater the system agents by providing them with the needed inputs. To achieve this goal, the top level agent first tries to check in the blackboard if a solution is already available (i.e., inferred before) or if it has been proved that it cannot be available. Consequently, it fetches the RS and controls the calling of the proper agents according to the strategy order. Figure 1, depicts the relationship between the entities of the OAM.

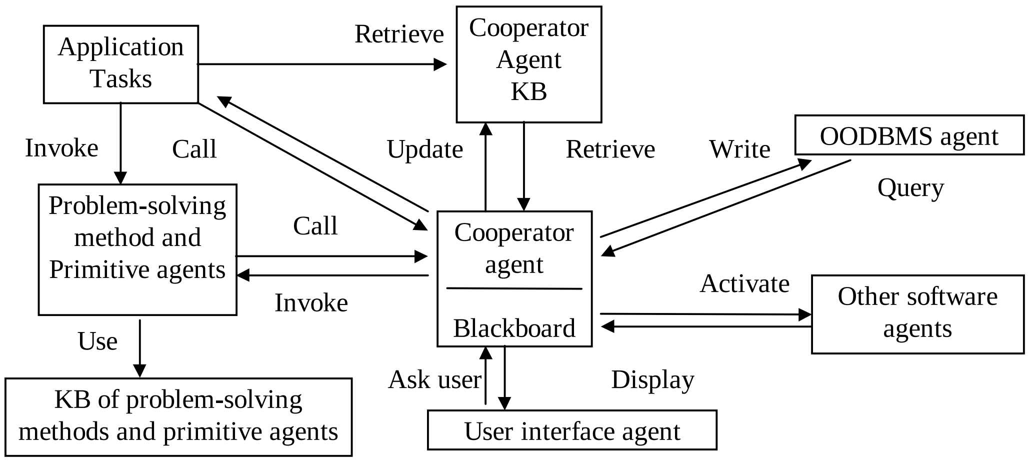

**Figure 1:** The open architecture model

In general, the component of a knowledge system applications based on the proposed model can be classified into:

- The cooperator agent and its knowledge representation
- The primitive agents and their knowledge representations
- The problem-solvers needed by the application and their knowledge representations.
- The needed software (e.g., DBMS) and their interfaces.
- Application tasks and sub-tasks

## 4 KROL: The language

A new language has been designed and implemented using object-oriented programming, logic programming, and rule-based programming. KROL is a multi-agent programming language. The basic model is that of agents communicating over a blackboard allowing for different readings, depending on the context, where agents compute rules or relations, serve a user-defined task, or apply problem solving methods. The agents of OAM correspond to statements being executed at runtime.

KROL is written in Prolog, so prolog syntax is used for KROL knowledge structure. An object object-identifier is declared by writing it in the following form:

```
object-identifier :: {
        statement-1 &
        statement-2 &
        :
        statement-n
}.
```

where object-identifier is a Prolog term that is either an atom or a compound term of the form functor(V1,…,Vn), where V1,…,Vn are distinct variables. The body consists of a number of statements, possibly none, surrounded by braces. The statements in object body have the same control structure as any Prolog clause.

The behavior of an object is represented by methods and its characteristics are represented by attributes. Attributes may have facets (value type, value source, possible values, and if the attribute takes a single or multiple value). Objects are based

and provides a mechanism in addition to inheritance known as method delegation. Multiple inheritance is allowed as well as light weight objects called instances. Heuristic knowledge is represented as rule clusters. A particular rule cluster is maniftested as a set of declarative rule instances defined in an object. A rule instance is declared by writing it in the following form:

*ruleid(conclusion)* ***if*** *premise*

In the following section we demonstrate the applicability of KROL to implement the components of the OAM.

## 5 Demonstration of KROL capabilities to implement agents of OAM

### 5-1 COOPERATOR AGENT

As previously described, agents used in an application should be accessed through the cooperator agent. In this way, the integration of a heterogeneous representation can be implemented without burden on the application developer(s). This problem-solver exploits its own knowledge representation to achieve its objectives. The knowledge representation is based on the declarative nature inherited from logic language, meant Prolog.

Here, it should be stressed that micro-integration means using more than one inference mechanism and/or knowledge representation and/or software system within another agent to solve sub-problems on its behalf. While macro-integration is based on a reasoning strategy (RS) to solve a particular problem or a particular sub-problem which means that if the sub-problem cannot be solved by one agent, the cooperator agent will use the next one in the declared strategy.

In figure 2, an example of the cooperator knowledge base is presented where two object slots (attributes) are declared in the object *plant*. This object is a part of hierarchical organization of the domain concepts. Notice the meta-attribute declaration: type/2, legal/2, ll/2 (lower limit), and ul/2 (upper limit). Not all these meta-attributes are used by the cooperator agent; some of them are used by the user-interface agent and for application verification. There are also some other meta-attributes which can be used but not needed in this example, Also, notice that the age/1 attribute has no declaration for the source_of_value/2. This means that the problem-solver will call the user-interface agent to fetch its value. If the user-interface agent needs to be integrated with another agent, it must be explicitly declared in the source_of_value/2.

The source_of_value/2 declaration of the growth_stage/1 attribute, inform the problem solver that the value of this attribute can be derived using some rule(s), encoded in the object *age_determine_growth_stage*. Accordingly, the cooperator agent will call the rule agent and store the fetched value in the blackboard.

```
plant :: {
        type(age/1, integer) &
        type(growth_stage/1, nominal) &
        legal(growth_stage/1, [seedling, tillering, extension, booting, heading, ripening, harvest] ) &
        ll(age/1, 1) &
        ul(age/1, 160) &
        source_of_value(growth_stage/1, [derived(age_determine_growth_stage)]) &
        super(domain_class)
}.
```

**Figure 2:** An example showing part of the knowledge base of the cooperator agent**.**

## *5-2 PRIMITIVE AGENTS*

Primitive agents are used as part of a problem solving method, application tasks, or generic tasks. Some of the primitive agent can be considered as problem-solvers, e.g., the KADS problem-solver [10]. In this article, two primitive agents are briefly described: the rule agent and the intelligent database agent.

### 5-2-1 Rule Agent

The first developed expert systems have been built using the production rules model. The main draw back of such systems is their maintenance and the difficulties faced when the number of rules grows enormously. The proposed model exploits rules as logical declarations which describe some criteria of the domain knowledge with no intended reasoning chaining. In this way, the robust nature of rules is exploited while their accompanied problems disappear. Of course one can chain rules whether forward or backward, if that is needed. However, in this case, the conventional problems should be considered.

The role of the rule agent is to focus on the rule which can evaluate the value of an input attribute and execute the rule body to fulfill this target. The evaluation process succeeds when either the given attribute and its value are valid or a value for this attribute can be fetched.

The clustering of a collection of rules in one object ensures modularity and hence improves maintainability, verification, re-usability and safe performance. Notice that the body of a rule may contain calls to the cooperator agent. In figure 3, the statement

not have to think of how the plant age will be retrieved. Furthermore, during development, this value may be retrieved as an input from the system tester and in the final version it may be retrieved from a database without making any changes in the rule. Figure 3, illustrates the rule used to determine the growth_stage/1.

### 5-2-2 Intelligent Database Agent

The intelligent database is represented as object clauses, i.e., encapsulated inside an object. Those clauses represent an association between knowledge attributes. To explain this idea consider the example illustrated in figure 4.

```
age_determine_growth_stage :: {
      r([growth_stage of plant = Stage]) if
           age :: plant = Age,                          % top value-finding call
           (    Age =< 21 -> S = seedling ; Age =< 45 -> S = tillering ; Age =< 79 -> S = extension
           ;    Age =< 89 -> S = booting ; Age =< 140 -> S = heading ; Age =< 150 -> S = ripening
           ;    S = harvest
           ) &
      super(plant)
}.
```

**Figure 3:** Example rule showing some knowledge style

Notice that the major difference between this representation and the rules, is that the association does not have the ability to implement chaining. Consequently, these declarations represent an association between one attribute (argument number one), in one hand, and a group of attributes, in the other. The intelligent database agent will focus on the clause from which it can fetch the value of the input attribute which correspond to argument number one. If the attributes (argument number 2) values are not available in the blackboard, then the call to the intelligent database agent fails.

```
general_associations :: {
       assoc(leaf_abnormal of leaf = X, [leaf_rust of leaf = X]) &
       assoc(area of plantation = L * W, [length of plantation = L, width of plantation = W]) &
       super(wheat_irrigation)
}.
```

**Figure 4: Knowledge representation of Intelligent database**

## *5-3 PROBLEM-SOLVING METHODS*

A number of problem-solver have been implemented and integrated successfully using the cooperator agent. It is important that OOP has facilitated the development of such a heterogeneous system. Each type of a problem solver is encapsulated in an object and linked to the application when needed. Also the application domain knowledge base consists of object declarations assembled according to the problem-solver requirements. An important representation is the domain ontology. The domain ontology is found to be sharable between a number of problem-solver. In the following sections some of the implemented problem solvers are briefly described.

### 5-3-1 CBR

CBR, as a problem-slover, has a large granularity that can be divided into simpler granules. Each of those granules is a problem-solver having its own behavior. The use of the cooperator agent can integrate the newly developed CBR problem-solvers to give the conventional behavior. The advantage gained is that those CBR-problem-solvers become more reusable and more efficient when considering the cooperation of problem-solvers in handling a knowledge problem.

### 5-3-2 Hierarchical Classification Generic Task (HC-GT)

Generic tasks are problem solving types corresponding to substructures into which domain knowledge is decomposed. To characterize HC-GT, the problem solving is based on the establish and refine control strategy which is applied to a node structure assembled in a hierarchy. Consequently, the nodes are mapped to objects with the same representation as those of the cooperator agent. In each node a rule is declared which determines whether the node is established. The HC-GT method starts from the top node and calls the rule agent to evaluate the rule attached to a named node. Accordingly, it either considers the node siblings (node established) or traverses the neighboring node. If at least one of the leaf nodes is established, then the task has succeeded. The established leaf node(s) represent the result.

## *5-4 SOFTWARE AGENTS*

It was found that expert systems which we develop most often, need to be integrated with intelligent and elegant user-interfaces, and databases. In order to re-use the developed software, we consider modularity, generality, and flexibility. Currently, our model contains an integrated OODBMS and a multilingual user-interface capability.

### 5-4-1 Object-Oriented Database Management System

The implemented OODB is an extension to the relational model. The relation schema is declared and encapsulated in an object. Consequently, the schema and its instances can be inherited by the sibling objects or can be re-defined and overridden by the ancestor declaration. This polymorphic behavior was very helpful. One faced example is that a reference data about the climate, water and soil need to be stored in a static structure that can be only accessed, if the user does not have his own data. Consequently, the different scheme of those relations were defined twice: once for the reference data and the other for the user data, while the query is called once so that the OODBMS will retrieve the reference data inherited from the reference object when the user does not store his data.

### 5-4-2 Multi-Lingual User-Interface Agent

The multi-lingual user-interface is based on the concepts of the implemented OODB. In our design the top object of the database is English. Each language inherits from the English object the defined intentions and the stored extensions. The override mechanism is applied on the record level. If the record(s) can be retrieved from an object, the object will not inherit the super object record(s). Any call in the user-interface agent is directed to what we call the presentation layer (object). The presentation layer translates calls from English to the currently active language and display those calls to the user. Then the user inputs are translated from the active language to English. Consequently, all values in the blackboard are in English. This is important because the code may contain conditional comparisons and necessitates the re-translation of the results needed to display the session conclusions.

### 5-5 APPLICATION TASKS

In fact, application tasks are domain specific, for example irrigation, fertilization, and pest-control, in the agriculture domain. These tasks are actually based on problem solving specific tasks such as: diagnosis, planning, assessment, etc. The granularity of such tasks are large. This means that they use a number of agents. The developer, of such tasks, can use any of the available agents.

Notice that the RS permits the developer to build more than one application task to solve a particular problem. Fore example, one can have two diagnosis tasks: one based on HC-GT and the other based on cover and differentiate method. The cooperator agent can use either or both tasks to diagnose a disorder. Another example of cooperation is when part of the problem can be modeled using a problem solver and the rest can be modeled using primitive agents. In the task code, the developer can insert a call to the top agent, if the task problem-solver could not reach to a conclusion.

## 6 CONCLUSION

It was found that the critical problem, in building knowledge base applications, is the gap between methodology and implementation. By methodology we mean models for knowledge elicitation, knowledge documentation, system design, and system implementation. In fact, this gap increases the difficulty of knowledge modeling, as knowledge engineers autonomously think of implementation. Consequently, this paper addresses the problem of integration and cooperation of multiple problem solving methods, inference mechanisms and software programs. It proposes a language based on an open architecture model proper to a general environment for the construction of large knowledge bases that assists the developer in building a knowledge base application, aiming at bridging this gap.

We believe that the model adaptability, modularity, and extendibility are potent in supporting knowledge system development. Object-orientation and the declarative nature of knowledge enhance maintainability.

## REFERENCES

1. Newell, A.; The Knowledge level. *Artificial Intelligence*, 18, pp. 87-127; 1981.
2. Barbuceanu, M.; Model: toward integrated knowledge modeling environments. *Knowledge Acquisition*, 5(3), pp. 245-304; 1993.
3. Rafea, A.; Agricultural Expert Systems Development in Egypt. *In Proceedings of the 2nd International Conference on Expert Systems for Development (ICES-94),* Bangkok, Thailand; IEEE computer society press; 1994.
4. Kolodner, J.L.; *Retrieval and organizational strategies in conceptual memory: A computer model*. Ph.D. thesis, Yale University; 1980.
5. Kolodner, J.L., Simpson, R.L., & Sycara, K.P.; A process model of case-based reasoning in problem solving. *In Proceedings of IJCAI-85*, pp. 284-290. Los Angeles, CA: Morgan Kaufmann; 1985.
6. Schank, R. C.; *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*, New York: Cambridge University Press; 1982.

7. Chandrasekaran, B.; *Towards a taxonomy of problem solving types*. AI Magazine, 4(1), pp. 9-17; 1983.
8. Chandrasekaran, B.; Towards a functional architecture for intelligence based on generic information processing tasks. *In Proceedings of the 10th International Joint Conference on Artificial Intelligence*, pp. 1183-1193, Milan, Italy; 1987.
9. Chi, R.T., Chen, M. & Kiang, M.Y.; A generalized case-based reasoning for personnel performance evaluation; *In Proceedings of the 24th Hawaii International Conference on System Sciences*, Kauai, HI, Jan 8-11, pp. 82-89; 1991.
10. Jonker, W. & Spee, J.W.; *Yet another formalization of KADS conceptual models*. In T. Wetter, K.D. Althoff, J. Boose, B. Gaines, M. Linster & F. Schmalhofer (Eds.); Current Developments in Knowledge Acquisition: EKAW 92. New York: Springer Verlag; 1992.
11. Yahaya, N.; On the Development of Environments For Developing Expert Systems; *In Proceedings of the 2nd International Conference on Expert Systems for Development (ICES-94)*, Bangkok, Thailand; IEEE computer society press; 1994.
12. David J. and Krivine J. (Eds.) (1993). *Second Generation Expert Systems*, Springer-Verlag.
13. Chandrasekaran, B. (1986). Generic Tasks in Knowledge-based reasoning: high-level building blocks for expert system design, *IEEE Expert*, 1, pp.23-30.