PERGAMON

Expert Systems With Applications 15 (1998) 33–46

Expert Systems with Applications

KROL: a knowledge representation object language on top of Prolog

Khaled Shaalan[a], Mahmoud Rafea[b], Ahmed Rafea[a,*]

[a]*Computer and Information Sciences Dept., Institute of Statistical Studies and Research (ISSR), Cairo University, 5 Tharwat St., Orman, Giza, Egypt*
[b]*Central Laboratory for Agricultural Expert Systems (CLAES), PO Box 100, Dokki, Giza, Egypt*

Abstract

This paper presents a knowledge representation object language (KROL) on top of Prolog. KROL is aimed at providing the ability to develop second-generation expert systems. The main aspects of KROL include multi-paradigm knowledge representation (first-order predicate logic, objects, rules), inference mechanisms at different levels of granularity, explanation facility, object-oriented database management module, and user-friendly interface. KROL has sufficient expressive power to be used in applying demanding knowledge-based modeling methodologies, such as KADS and Generic Task, which are the major landmarks of the second-generation expert systems technology. Four successful agricultural expert systems have been developed in the last 6 years using KROL. To demonstrate the language capabilities, we present an example of disorder diagnosis.

1. Introduction

Object-oriented technology has become a powerful means of handling the complexity inherent in many systems. Object-oriented technology has influenced and benefitted from research in the arena of artificial intelligence and knowledge-based systems (KBSs) (Schroeder et al., 1994; Harmon, 1995). As far as the representation of the real world is concerned, the goal of knowledge representation in KBSs and object modeling is the same. An object-oriented knowledge structure not only models the problem domain closely, but also facilitates the implementation of a generic inference component (Lyndon et al., 1995). Object-oriented methodology is used in designing a KBS to assist expert systems designers from the conceptual design phase to the validation phase (Akoka et al., 1996). Conversely, a knowledge-based approach is used to make preparing object-oriented code for reuse significantly easier and more quantifiable (Etzkorn et al., 1995). There is a long history of representing knowledge in the form of rules. The advantages of rule-based expert systems have been documented in the literature (Hayes-Roth et al., 1983).

The combination of object and rule processing provides a firm foundation for addressing more complex problems (Kowalski et al., 1990). Both paradigms increase programmer productivity during application development and reduce maintenance costs. The synthesis of object and rule processing provides all the advantages of both technologies, as well as some advantages available only as a result of their combination. Augmenting rules with objects makes it possible to refine knowledge about the application even than with rules alone. The rule-based application can exploit data abstraction and encapsulation principles to structure knowledge. Enhancing objects with rules provides a more powerful paradigm for reasoning about objects. In addition, the inference engine in a rule-based system allows reasoning knowledge to be expressed more clearly and concisely. All these have motivated us to develop a new language, a knowledge representation object language (KROL), on top of Prolog.

Prolog has been a primary language for artificial intelligence, and we have chosen Prolog as KROL's implementation language. Prolog has three positive features (Vranes et al., 1994).

First, Prolog syntax and semantics are much closer to formal logic, the most common way of representing facts and reasoning methods used in artificial intelligence. Second, Prolog provides automatic backtracking, a feature making for considerably easier search, the most central of all artificial intelligence techniques. Third, Prolog supports multidirectional reasoning, in which arguments to a procedure can freely be designated inputs and outputs in different ways in different procedure calls, so that the same procedure definition can be used for many different kinds of reasoning. Besides this, new implementation techniques have dramatically improved the efficiency of the current versions of Prolog. KROL has been implemented on top of SICStus

* Corresponding author. Tel.: (+20) 2 361177; fax: (+20) 2 36044727; e-mail: shaalan, mahmoud, rafea@esic.claes.sci.eg.

0957-4174/98/$19.00
PII: S0957-4174(98)00009-8

Prolog (SICS, 1995) for many good reasons, including: portability to many computer systems; interface between C and Prolog; the range of libraries that have developed. It is aimed at further extending the expressiveness of representing knowledge and information in Prolog.

Several attempts have been made to extend logic programming for better knowledge representation. Much of that work supports one paradigm of knowledge representation, e.g. frame (Iline et al., 1987), semantic network (Yoshiyuki and Koseki, 1987), and blackboard (Brogi et al., 1991; Vranes et al., 1994). Though some researchers have proposed hybrid knowledge representation (Xu et al., 1994), they have no support for second-generation expert systems. Distinguishing from the above work, our objective is to develop a knowledge representation language that has expressive power to be used in applying demanding knowledge-based modeling methodologies, such as KADS (Schreiber et al., 1993) and Generic Task (GT) (Chandrasekaran, 1986), which are suitable for large-scale knowledge bases (David et al., 1993).

KROL provides facilities that are important in knowledge representation and processing, such as:

- the expressive power to represent complex knowledge. The multi-paradigm knowledge representation of KROL avoids restricting users to a single way of expressing either knowledge or data;
- the facility to modularize effectively a knowledge base and to construct a hierarchy of concepts;
- the facility to control inheritance of properties through a concept hierarchy;
- the facility to deal with inference mechanisms at different levels of granularity;
- the knowledge base development tools that facilitate application development;
- the primitives that allow for higher level knowledge base modeling approaches to scale to large problems, e.g. KADS;
- the synergy of different inference mechanisms in one system;
- a strict typing facility.

In agriculture, the potential offered by expert systems opens up a whole new dimension in the transfer of knowledge to extensions service and farmers. This has led us to develop KROL. KROL is in active use; it has been working on the development of four reliable and robust crop management expert systems during the last 6 years. We took the advantage of proven knowledge engineering methodologies, such as KADS and GT, for constructing a model or representation of the underlying domain, and then for designing reasoning mechanisms that can be used, together with the model, to develop these expert systems.

This paper is organized as follows. Section 2 describes KROL as a knowledge representation language. We present the knowledge representation paradigms with support tools that are needed for application development. Section 3 presents the implementation aspects of our language. We show how we can implement KROL as an efficient programming language. Section 4 demonstrates how KROL can be used to develop a KADS-based expert system; in this section, we show an example of a disorder diagnosis system. Section 5 gives a brief overview of our experiences to date with developing expert systems using KROL. Section 6 concludes the article.

2. KROL: the language

A new knowledge representation language that combines logic, object-oriented and rule-based programming paradigms is designed. This provides a good medium for second-generation expert systems development. From a practical point of view, we have found it more attractive to design it as an extension of the existing Prolog system to support the combination of knowledge representation paradigms. Knowledge base development tools that facilitate application development are also designed. The approach considered in the current work is based on the expectation that the developer of KBSs is already proficient in Prolog, and this is the target user.

2.1. Knowledge representation paradigms

The choice of representation can be crucial to the tractability of the problem. Newell (1981) states that:

$$\text{Representation} = \text{Knowledge} + \text{Access}$$

For a representation to be adequate, it must have sufficient expressiveness to capture the knowledge and sufficient pragmatic utility to allow for manipulation of the knowledge (Hartley, 1985). The most widely used and best-understood representation systems are rule-based systems, frame and, more recently, object systems. If the relations and inferences in the domain are mostly of a heuristic nature, then a rule-based system is usually suitable. However, relations and inferences that are mostly of the hierarchical inference type (generalization, refinement, and inheritance of properties from class to subclass) are usually more adequately represented with frame or object systems (Batarekh et al., 1991). The combination of object and rule processing provides a firm foundation for addressing more complex problems.

In KROL, knowledge is represented by a single formalism: the object. Objects correspond to real-world concepts or rules. Rules are uniformly handled in an object-oriented manner. The behavior of a concept is represented by methods, and properties (attributes) represent its characteristics. The relationship between concepts is governed by the applied inference. For example, inheritance, a built-in inference in KROL, is a specialization where all subobjects inherit the behavior of their super-objects. 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 `f`unctor (V1,…,Vn), where `V1, …, Vn` are distinct variables. The body consists of a number of statements, possibly none, surrounded by braces. The statements in the object body are described below.

2.1.1. Rules

A rule allows information about an object to be inferred, rather than retrieved using a traditional message passing against stored data. Thus, rules provide an information derivation mechanism that results in greater informational content than is present in the stored data alone. The major disadvantage of rule systems is knowledge maintenance. It has been recognized that a decomposable KBS leads to computationally efficient inference design and increases maintainability. So, KROL provides a convenient mechanism for rule clustering. A particular *rule cluster* is manifested 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`

where `ruleid` is any label that uniquely identifies the rule and *conclusion* part is a list structure. An element of this list is either a derived value of the form

`attribute of object = value`

or a message. The *premise* part has the same control structure as any Prolog clause body. However, rules in KROL differ from prolog rules in that they are order insensitive and their execution is handled by a hypothesis-interpreter, which is a facility to customize the inference control strategy. Atomic statements in premise are either a derivation of an attribute value of the form

`attribute :: object rel_op value`

Prolog goal, or a message, where `rel_op` is any Prolog relational operator. The following is an example of a rule instance:

```
r4([growth_stage of plant = develop-
ment, current_date of plantation =
[D,M,Y]]) if

system :: time(_, _, _, D, M, Y), % message
ending_dev_date :: plant = Date, %
derivation
:compare_date( > = , Date, [D,M,Y]) %
prolog call
```

2.1.2. Methods

Methods are used to perform data manipulation and implement applications. Additionally, methods may be written directly in Prolog, thus giving the programmer more freedom in terms of programming constructs and access to Prolog predicates. A method has a clausal syntax similar to that of Prolog, but instead of the usual predicate calls in the body of a clause there are *method-calls*. Ordinary Prolog goals are also allowed in a prefixed form, using ':' as a prefix. Atomic goals, i.e. messages, in the body of a method may be in a number of forms, besides being Prolog goals:

- `goal` to send the message goal to the object **self**
- `object::goal` to send the message to object `object`
- `object < :goal` to delegate the message goal to object `object`
- `::goal` to send the message goal to a method that may be defined locally or inherited by the `object`
- `< :goal` to delegate the message goal to a method that may be defined locally or inherited by the object.The following is an example of a method.

```
plant(Species, Lwb, Upb):-
subclass(Subclass), % send to self
text(subclass, Text),
menu(subclass, Menu),
self(Self), % inlined method
  < :ask(subclass(Self), Text, Menu, -
Subclass), % send to object
Subclass::plant(Species, Lwb, Upb)   %
send to object
```

Note that objects are based on the notion of *prototypes* (Lieberman, 1986), which basically allows 'classes' to be first-class objects, and provide a mechanism in addition to inheritance known as method delegation. In general, a set of objects declared may form an inheritance hierarchy. Since objects with multiple supertypes are allowed, the hierarchy is generalized into a lattice. Hence, KROL supports multiple inheritance. Immediate superobjects are declared by defining the method `super` within the object in the following form

`super(object − identifier)`

In the programming language literature the restriction on the use of inheritance varies considerably. In KROL, it takes the form of *differential inheritance*, where an optional list of excluded methods (*don't-inherit-list*) may be specified in super definition. For example, consider the declaration of the object `set` with superobject `bag`. To exclude the

inherited method `numberOfOccurrences` from `set`, we write

```
super(bag, [numberOfOccurrences/1])
```

Furthermore, each superobject possesses automatically the method `sub` which returns its subobject, a labor-saving feature. Thus, an object hierarchy is created with a double link that represents a super-sub relationship. This feature allows for easier application of different search algorithms.

2.1.3. Attributes

An object may have attributes that are modifiable. Attributes declaration takes the form

```
attributes(Attributes)
```

where `Attributes` is a list of compound terms specifying the attributes with their initial values. For example, the object *point* that defines a movable point in a two-dimensional space may contain the definition

```
attributes([x(0), y(0)])
```

with `x` and `y` initialized to the value 0.

2.1.4. Meta-attributes

A set of facets can be attached to an attribute. In KROL, facets are predicates, which are used as rules associated to events such as value range and value set, and rules for inferring attribute values. The following facets are provided by KROL:

1. *type*. KROL has five primitive data types, namely: nominal, integer, real, string, and date. Each attribute must have a type declaration, e.g.

```
type(disorder_name/1, nominal)
```

Some primitive-type data types require additional facet declarations that are needed for dynamic consistency checking:

- *legal*. The value set of a nominal type should be defined. These possible values can be explicit or implicit, e.g.

```
legal(colors/1, [green, red, white, -
black, brown]) & % explicit
legal(disorder_name/1, Disorders):- %
implicit
disorder::leaves(disorders)
```

- *range*. The value range of an integer or real type should be defined, e.g.

```
range(current_year, 1996 – 2000)

% lower_bound – upper_bound
```

2. *single* or *multiple*. An attribute of a primitive type may take either a single value or multiple values. The default is single-valued.

3. *source*_of_value. Attributes' values can come from user input, database table, and a conclusion of a rule. In KROL, they are known as: user, database, and derived respectively. The value of an attribute may be tried in order from a combination of sources, e.g.

```
source_of_value(disorder_name/1,

[derived, user])
```

Notice that in case of `derived`, a predetermined rule cluster may be given, e.g.

```
source_of_value(growth_stage/1,

[derived([age_growth_stage])])
```

It should be stressed that the integration of different inference mechanisms in one system can be achieved by the virtue of this meta-attribute. Suppose that an attribute value can be derived using a generic task or routine design problem solvers, one can declare that in the `source_of_value` meta-attribute.If `source_of_value` is user, additional facets may be declared:

- *prompt*. This specifies the text or window that is to be displayed when the system asks the user about the attribute value, e.g.

```
prompt(average_temperature/1,''What
is the leaves color in ~w?'', [Current_
Month]):-

plantation::month(Current_Month)
```

- *necessary*. This enforces the user to give a value during the session, e.g.

```
necessary(average_temperature/1)
```

Otherwise the response with 'unknown' as an attribute value is permitted.

2.2. Inference mechanism

A built-in inference is provided for most common uses. The inference follows the open-world assumption, where either positive or negative values of attributes are recorded. Owing to the increased complexity of KBSs, appropriate inference mechanisms at different levels of granularity are designed and implemented. A major feature of this inference is that it is a reusable component that can fit into different domains. Its methods are encapsulated in an object, known as *inference_class*, which can be classified as follows.

4. *Methods that directly reason about attribute values*. A core operation of the inference is the `get_value` operation that provides the mechanism for heuristically determining the value of an inferred (derived) attribute or proving that a given attribute can have a specific value. It differentiates between two cases during the course of reasoning: the single-valued attribute and multi-valued

attribute. The general nature of the `get_value` operation is similar to the goal satisfaction process provided in an inference engine. When an access to an attribute is requested during the course of the inference process, the `get_value` is automatically invoked in order to determine or prove an inferred attribute value for the object of interest. The `source_of_value` facet guides this operation. Thus, a generic inference strategy can be devised to search the domain knowledge for evidence that will establish a value for an attribute.

5. *Methods that directly invoke the inference in order to reason about attribute values*. These methods provide the capability to express fine-grained inference mechanisms in a flexible and efficient way, meaning that the relevant parts of the knowledge are involved in the derivation process. There are two defined methods that can be used to deal with rules: `focus` and `invoke`. The former tries to prove all rule instances that drive a given attribute. The latter tries to prove a particular rule.
6. *Methods that directly invoke the inference and indirectly reason about attribute values*. These are middle-grained or coarse-grained inference mechanisms that act upon rule clusters. There are two defined methods that can be used to deal with rules: `conclude_relation` and `conclude_all`. The former tries to prove all rule instances in a given object. The latter is more general and tries to prove all rule instances in a given object and recursively its descendant objects.

2.3. Knowledge base development support tools

Explanation facility, user interface, and database tools are provided to the developer with KROL for convenient application development. These tools are briefly described below.

2.3.1. Explanation support tool

The ability to explain reasoning processes used for problem solving distinguishes the expert system from other decision support systems (Swinney, 1995). The provision of an explanation facility may actually lead to a higher probability of acceptance of the system output (Ye, 1990) and allow the user to establish deeper understanding of the system. KROL supports the most widely used types of explanation facility to explain the behavior of rules to the user. They are 'WHY' the system asks this question and 'HOW' the system has reached this conclusion. Consequently, the attribute values and their corresponding sources are recorded during the course of the inference process. Also, the order in which rule clusters were employed are recorded and traced for an explanation of the system's conclusion. Since a trace of the system execution history is unlikely to be very illuminative to the user, the system design is augmented to respond with a customized explanation text template. This template contains explanations (clarification, textbook references, case citation, multimedia, etc.) that can be used as long as needed, without adding overhead to the knowledge. KROL provides the object `explanation` that defines the methods `how` and `why`.

2.3.2. User interface support tool

The user interface handles the interaction between the user and the system that is 'comfortable, easy to use, and friendly' (Roesner, 1988). The end-user can interact either in Arabic or English, i.e. it is bilingual. In KROL, it takes the form of: issuing a prompt that asks the user for some information, accepting an attribute's legal values whose source of values is defined as user, reporting an error message, reacting to an explanation request, and generating a report. Consequently, different styles of interface programming, such as dialogue and menu, are supported. The multimedia facility is provided; this is helpful in explanation, especially for new personnel (Rafea, 1995; Rafea et al., 1995a). Note that the user's reply will be recorded in order not to ask the same question twice. However, the user is permitted to respond with 'unknown'. Users are never asked about the attributes that take an unknown value. This is very useful, because sometimes we can work with incomplete knowledge or apply heuristics whenever necessary. KROL provides the object `user_interface`, written in C and interfaced to Prolog, that defines the user interface methods.

2.3.3. Object-oriented data base management system (OODBMS) support tool

As mentioned in Yahaya (1994), the practical expert systems in the future will have to increase in terms of size and complexity. In addition, as their overall size grows, the non-knowledge-based components, such as the data processing components, are also expected to increase in size. To allow the knowledge engineers to use a single way to represent data and knowledge in an application, we have developed an OODBMS support tool. Although important, it is beyond the scope of this article; refer to Kim (1990) for an excellent overview and motivation for the subject. The `dbms` object provides the methods that enable us to define a schema, create a data object, create a view, process a query, and maintain the integrity of databases in an object-oriented manner.

3. Implementation aspects

Object-oriented languages have an undeserved reputation for inefficiency because some early languages were interpreted rather than compiled (Rumbaugh et al., 1990). KROL is a compiled language that expands into SICStus Prolog code. The expansion of KROL definitions to Prolog definitions is based on source-to-source transformations. The transformation rules given in ESICM (1992) and SICS (1995) provide the definitions of general rewrite rules for

expanding definitions. Hence, the operational and declarative semantics of KROL programs are given in terms of their translations to Prolog.

First of all, every defined object will translate to several Prolog clauses belonging to a unique `object module` with the same identity as the `object-identifier`. Object modules are significantly cheaper to create than ordinary modules, as they do not import the built-in Prolog predicates. The module will contain the predicates implementing an object declaration, the method code, the rule code, the attribute code and the imported methods. It should be noted that SICStus Prolog uses a flat, not hierarchical, module system with access control mechanisms for exporting methods. These mechanisms provide for efficient encapsulation of object primitives and enhance the code execution through direct accessing of the object code. The following sections discuss the implementation aspects of KROL.

3.1. The inheritance mechanism

One aspect of object-oriented languages that seems inefficient is the use of method resolution at runtime (also known as *dynamic binding*) to invoke methods. Method resolution is the process of matching an operation on an object to a specific method. This would seem to require a search up the inheritance graph at runtime to find the object that implements the operation. KROL optimizes the look-up mechanism to make it more efficient; a method dispatches in a constant time once its target object becomes determinate regardless of the depth of the inheritance graph or the number of methods in the object. Moreover, the method dispatcher is cleanly captured and will only contain the relevant information where all the excluded entries are removed. The inheritance mechanism is based on the module access control mechanisms. All the methods visible in the immediate supers are collected after subtracting those that are specified in the *don't-inherit-list*, the resulting set is made visible in the module of the inheriting object by means of importation.

3.2. Object attributes

Attributes are based on efficient term storage associated to modules. The attributes for an object are collected from its ancestors and itself at compile time and used for initialization at load time. The methods for accessing and deriving attributes are in-lined to primitive calls whenever possible.

3.3. Methods

The method body is translated to a Prolog-clause body. The code is traversed, and the method-calls are transformed according to the following transformation pattern:

$$\text{Method} - \text{call} \begin{cases} \text{Module} \times \text{Message} \times \text{Self} \times \text{Myself} \\ \text{Module} \times \text{Message} \times \text{Self} \end{cases}$$

where *Module* is the target `object module`, the argument *Message* is the received message, the argument *Self* is bound to the current contextual object that is needed for dynamic binding of attributes and methods to objects at runtime, and the argument *Myself* is the parameter needed to cater for passing object parameters, if any, to the method-call.

Methods are used to perform data manipulation, and implement applications. The good programming styles that are employed at the Prolog level can also be employed at the KROL level. This language efficiency is realized by:

7. representing objects as special lightweight Prolog modules;
8. exploiting the first argument indexing of the Prolog compiler, leading to direct access to the method clauses;
9. preserving the last call optimization in recursive methods, i.e. the tail primitive is expanded into tail recursive Prolog code. Thus, the expanded code will invoke the expanded code directly instead of calling the dispatcher.

3.4. Rules

Rules are transformed into Prolog code in a way similar to methods transformation with additional information. This information provides for flexible pattern matching that improves a great deal with inference process. This is realized by:

- storing for each attribute the rules that derive a certain value. This is very useful when considering the method `focus` of the inference;
- storing for each rule cluster all rule heads, each in the form of a catchall goal. This is very useful when considering the methods `conclude_relation` and `conclude_all` of the inference.In general, KROL provides the capability to express fine-grained inference mechanisms explicitly in a way such that the granularity, i.e. target rules, used in a program can be adjusted to invoke the inference directly.

4. Demonstration of KROL capabilities to develop a KADS-based expert system

An expert system is one of the most successful applications of artificial intelligence. With the increase on demand of using expert systems, they become bigger in size and more complex in structure. A large and complex expert system must be engineered carefully if it is to function properly and to be modified easily. Thus, creating an expert system requires a development methodology that emphasizes a good structure for the knowledge within the system.

A common knowledge engineering methodology is to partition the task into smaller components, rather than have one big system. Each of these modules should be a

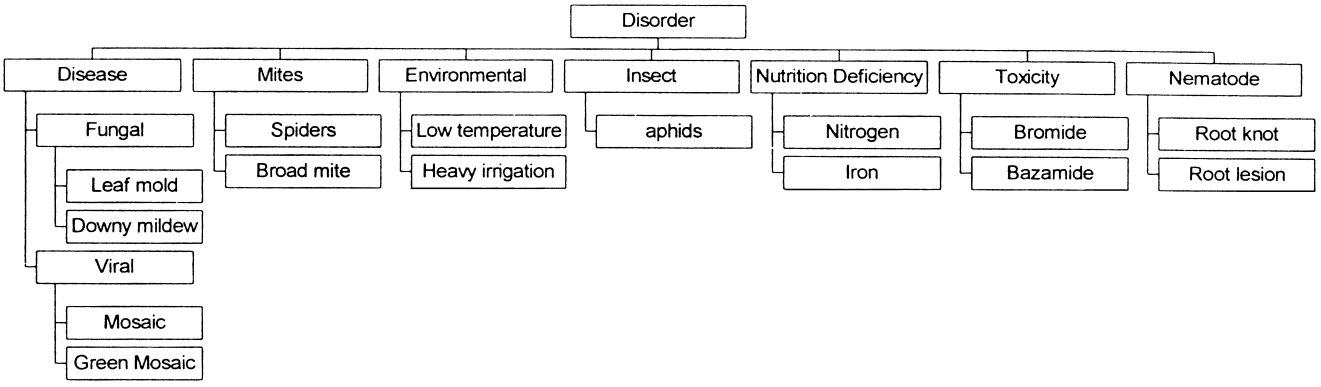

Fig. 1. Disorder concept hierarchy.

well-defined portion of the system, with carefully defined inputs, outputs and functions. The modularization of a system can be done in different ways. The most appealing one is the knowledge-level approach, which consists of breaking the system into a number of layers or levels. One of these approaches is KADS, which is a major landmark of the second-generation expert systems engineering methodologies. Thereby, a structured systematic development of KBSs is achieved. In the following subsections we give a brief overview of KADS and show how KROL can support their implementation in terms of an example.

4.1. KADS: a brief overview

KADS is a methodology that has been developed in the framework of the Esprit program. The KADS-I project has been succeeded by the KADS-II (KommonKADS) project. The theories concerning the modeling of knowledge according to KADS are based on the work of Professor Wielinga and Professor Breuker of the University of Amsterdam. The model-based approach according to KADS is rapidly becoming the de facto standard in Europe. In KADS, the development of a KBS is viewed as a modeling activity. The KADS methodology is based upon a number of principles derived from cognitive psychology, artificial intelligence, and software development. In this paper, we assume that the reader is familiar with KADS methodology; for more details see Schreiber et al. (1993).

The KADS expertise model distinguishes three types of knowledge, and prescribes specific relations between these knowledge-types.

- The first category of knowledge is called *domain knowledge* and concerns domain-specific knowledge. Such knowledge describes the objects of discourse in a particular domain, facts that hold about such objects, relationships among them. Rules, facts, hierarchies, objects, properties, relations, etc. often represent this type of knowledge. A crucial property of this first category of knowledge is that it is represented, as much as possible, as being independent from how it will be used. Thus, we state which properties and relations hold in a particular domain, but we do not state how these properties and relations will be used in the reasoning process. That is the concern of the second category of knowledge.
- The second category of knowledge is called *inference knowledge*. Here, we specify: (a) what the legal inference steps are that we can use in the reasoning process, (b) which role the domain knowledge plays in these inference steps, and (c) what the dependencies are between these inference steps. Again, a crucial property of this type of knowledge is what it does not contain: although we specify what the legal inference steps are, we do not specify the sequence in which these steps should be applied.
- This sequence of steps is exactly the concern of the third type of knowledge, the *task knowledge*. This specifies in which order the inferences from the second category should be executed. This type of knowledge is concerned with actions, sequences, iterations, state-transitions, etc.

4.2. Disorder diagnosis: an example

Diagnosis is the problem of trying to find the causes of abnormal observations. We chose an example that investigates the application of KROL to a domain theory for diagnosing disorders in a cucumber production management system. This system contains ten concepts, 23 attributes, one relation between concepts, 11 relations between concepts instances, six relations between expressions, and 109 relations between expression instances.

4.2.1. Domain knowledge

The domain *concepts* has two types of concept: the first type is simple concepts, such as `soil`, `water`, `climate`, `plant`, and `plantation`. The second type is hierarchical concepts, such as `observation` and `disorder`. The concepts representing the domain were implemented as objects. For example, Fig. 1 illustrates a concept hierarchy for identifying disorders that infect the plants in a farm.

```
disorder :: {
            super(domain_class) &
            attributes([value([]), verify([]), confirmed([]),
            infection([]),  spread_range([])])

            source_of_value(value/1, [derived])&
            source_of_value(verify/1, [derived])&
            source_of_value(confirmed/1, [derived])&
            source_of_value(infection/1, [derived, user])&
            source_of_value(spread_range/1, [derived, user])&

            prompt(value/1, disorder_value, [])&
            prompt(infection/1, disorder_infection, [])&
            prompt(spread_range/1, disorder_spread_range, [])&

            type(value/1, nominal)&
            type(verify/1, nominal)&
            type(confirmed/1, nominal)&
            type(infection/1, nominal)&
            type(spread_range/1, nominal)&

            multiple(value/1)&
            single(verify/1)&
            single(confirmed/1)&
            single(infection/1)&
            single(spread_range/1)&

            legal(value/1, Ds):-
               disorder :: leaves(Ds)&
            legal(confirmed/1, confirmed)&
            legal(verify/1, yesno)&
            legal(infection/1, disorder_infection)&
            legal(spread_range/1, disorder_spread_range)

            }.

            root_knot:: {
            super(nematode)&
            attributes([root_knot_infection([])])&
            source_of_value(root_knot_infection/1, [user])&
            single(root_knot_infection/1)&
            type(root_knot_infection/1, nominal)&
            prompt(root_knot_infection/1, root_knot_nematode_infection,
            [])&
            legal(root_knot_infection/1, root_knot_nematode_infection)
            }.
```

Fig. 2. An implementation of two concepts from the diagnosis hierarchy using KROL.

Fig. 2 shows the implementation of two concepts from such a hierarchy. At the topmost is the `disorder` object, which is inherited by all of the other objects in the hierarchy. In general, the domain concepts inherit the behavior of the generic object `domain_class`. At any point in the hierarchy, however, an object has the option to override the defaults with ones that are specialized for the problem area, e.g. the object `root_knot`. The object hierarchy is used to represent the implicit relationships, that is disorder `is_a` disorder, between different classes of disorder concepts, starting from objects at a general level down to objects at specific levels.

Properties with their initial values are implemented as attributes. Some facets are associated with these attributes. For example, the attribute *value* of disorder object has the facets *legal* that specifies leaves, the most specific objects in the hierarchy, as its legal values.

A second type of relation is the *relation between expressions* about property values. It is worth noting that the relations between expressions are grouped according to the semantic of the relation and the concepts to which the relation operands, which are properties, belong. The right-hand side of a relation is the properties of one object, whereas the left-hand side of a relation may be the properties of more

Left Hand Side			Right Hand Side		
Concept	**property**	**value**	**concept**	**property**	**value**
Leaves_ Observation	L_O_Color	Yellow	Disorder	Value	salt_injury, nitrogen_def, zinc_def, heavy_irigation, high_light_intensity
Leavs_ Observation	L_O_Color	yellow edges; brown edges	Disorder	Value	jassid, potassium_def
Leaves_ Observation	L_O_Color	light green; lemon_yellow	Disorder	Value	iron_def
Leaves_ Observation	L_O_Color	yellow but main veins still green	Disorder	Value	magnesium_def, manganese_def
Leaves_ Observation	L_O_Color	Purple	Disorder	Value	phosphorus_def, low_temperature
Leaves_ Observation	L_O_Color	brown	Disorder	Value	low_temperature
…	…	…	…	…	…

Fig. 3. A subset of `caused_by` relation of disorder diagnosis.

than one object. This grouping of relations between expressions came about as a result of our experience in order to establish a clear and clean mapping between the knowledge sources and the relations between expressions. Each relation is implemented as a rule cluster, a set of declarative rule instances defined in an object. It refers to the sub- or system to which it belongs. For example, the relation `caused_by` specifies a *caused by* relationship between disorder and observations. A sample of this relation is shown in Fig. 3 and its implementation is shown in Fig. 4.

4.2.2. Inference knowledge

The inference structure of the disorder diagnosis receives the user's complaint, usually symptoms, and obtains the case description from the farm database to predict the assumed disorders. It selects observations and factors

```
observation_caused_by_disorder::{

super(diagnosis_system)&

r1([value of disorder = nitrogen_def,
    value of disorder = zinc_def,
    value of disorder = salt_injury,
    value of disorder = high_light_intensity,
    value of disorder = heavy_irrigation]) if
     l_o_color :: leaves_observation = yellow &
r2([value of disorder = jassid,
    value of disorder = potassium_def]) if
    (l_o_color :: leaves_observation = 'yellow edges' -> true
    ;l_o_color :: leaves_observation = 'brown edges'
    )&
r3([value of disorder = iron_def]) if
    (l_o_color :: leaves_observation = 'light green' -> true
    ;l_o_color :: leaves_observation = 'lemon yellow'
    ) &
r4([value of disorder = magnesium_def,
    value of disorder = manganese_def]) if
    l_o_color :: leaves_observation = 'yellow but main veins
still green'&
r5([value of disorder = phosphorus_def,
    value of disorder = low_temperature]) if
    l_o_color :: leaves_observation = purple &
r6([value of disorder = low_temperature]) if
    l_o_color :: leaves_observation = brown

}.
```

Fig. 4. An implementation of subset of `caused_by` relation using KROL.

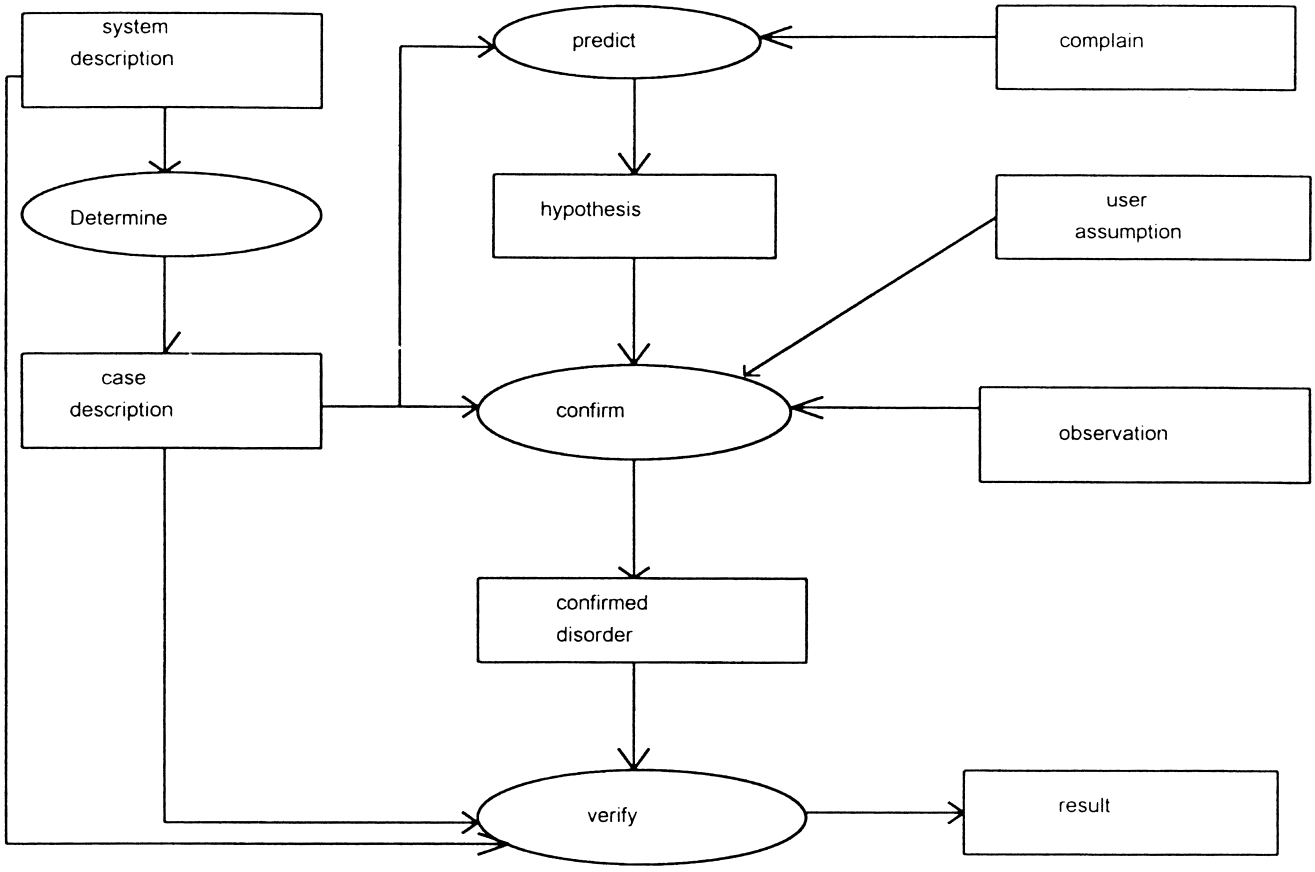

Fig. 5. An inference structure for disorder diagnosis in a crop management system. Rectangles represent roles; ovals represent inference steps. Arrows are used to indicate input–output dependencies.

related to the assumed disorders, prompts the user, analyzes the user response and confirms the possibility of disorders existence. Then, the confirmed disorder is assigned a certainty factor, which is either 'likely' or 'most likely'. Fig. 5 shows the inference structure of the disorder diagnosis. Fig. 6 shows a sample of the implemented inference step. Associated with each inference step is an explanation module. Fig. 7 shows the implementation of the `predict` explanation module.

4.2.3. Task knowledge

The goal of the disorder diagnosis task is to provide the user with a diagnostic disorder which causes problems on plantation or verifies a user's assumption. The task structure of the diagnosis is expressed in the pseudo code segment shown in Fig. 8. The implementation of this task is shown in Fig. 9. It should be noted that the task at the very beginning differentiates between disorders that infect the plants at different growth stages, i.e.

```
diagnosis_inference ::{

super(inference_class)&

determine
predict:-
   (inference_class::conclude_all(observation_caused_by_disorder)->true;true),
   (inference_class::conclude_all(observation_confirm_disorder)->true;true),
   (inference_class::conclude_all(observation_plant_caused_by_disorder)->true;true),
   (inference_class::conclude_all(plant_observation_confirm_disorder)->true;true)&

confirm
verify
}.
```

Fig. 6. An implementation of subset of disorder diagnosis inference using KROL.

```
predict :: {

super(diagnosis_sys) &
dynamic rel/1 &
type(rel/1, nominal) &
long_prompt(rel/1,2,54,Mdiag) :-
        appl_presentation :: fetch(text(exdiag3, Mdiag), _) &

legal(rel/1, L):-
findall(X, (:: map(X,Y),
        (history :: premise_data(Y, _RuleId, disorder, value, _V)  % fired rules
        ;history :: premise_data(Y, _RuleId, _O, confirmed, _V)    % fired rules
        )), L1),
        sort(L1, L) &

% map(text describing the relation, relation name)
map('possible disorders which cause primary observations',
observation_caused_by_disorder) &
map('possible disorders which cause primary observations on fruits',
observation_plant_caused_by_disorder) &
map('disorders which confirmed by primary observations',
observation_confirm_disorder) &
map('disorders which confirmed by primary observations on fruits',
plant_observation_confirm_disorder)

}.
```

Fig. 7. An implementation of an explanation module.

development, middle, and late, upon which the disorder diagnosis is derived.

5. Experience

The current work is aimed at further extending the expressiveness of representing knowledge and information in Prolog. Plain Prolog represents knowledge in the form of Horn-clauses. The inference mode is a backward-chaining ordered sensitive system which differs from conventional rule-based systems. This is because, in rule-based systems, the order in which rules are applied depends both upon data values changing within the system and on the rule control methodology adopted. In fact, the evolution of KROL came about because of the need to bridge the gap between the

```
Task:  Disorder Diagnosis
Goal:    finding  causes of user complaint
            or verifying of user assumption

DETERMINE(System Description → Case Description)
IF  the goal is to  find causes of user complaint
THEN
     OBTAIN (Complaint)
      PREDICT(Complaint + Case Description  → Hypothesis )
      CONFIRM(Hypothesis+Case_description+Observation→Confirmed Disorder)
      VERIFY(Confirmed Disorder+ System_Description +Case Description → Result)
       PRESENT(Result)
ELSE
       OBTAIN(user assumption)
        CONFIRM(user assumption+Case_description+Observation→Confirmed Disorder)
        VERIFY(Confirmed Disorder+System_Description+Case Discription→ Result)
              IF Result contains an unlikely assumption
              THEN
                   PRESENT the unlikely assumption
```

Fig. 8. Task structure of disorder diagnosis.

```
diagnosis_task ::{

main:-
       system :: time(_HH,_MM,_SS, _DD, Month, _YY),
        plant :: update(current_month(Month)) &
        diagnosis_inference :: determine,
        inference_class :: get_value(plant, growth_stage(Gr)),
        :diag_menu(Gr),
        diagnosis_inference :: predict,
forall(disorder :: value(D), D:: assert(verify(no))),
% invalidate disorders
        diagnosis_inference :: confirm,
       diagnosis_inference :: verify,
        :: present_value &

present_value:-
       :: get_likely(Disorders1),
       :: get_most_likely(Disorders),
       :: present_diagnosis(Disorders, Disorders1) &

get_most_likely(Disorders):-
     findall(Disorder,
            (disorder :: leaf(Disorder),
             Disorder :: confirmed('most likely')), Disorders)&
get_most_likely([])&

get_likely(Disorders):-
findall(Disorder,
       (disorder :: leaf(Disorder),
        Disorder :: confirmed(likely)), Disorders)&
get_likely([])&

present_diagnosis([], []):-!,
       inference_class :: get_value(plant, growth_stage(Gr)),
       :: check_normal_observation(Gr) &
present_diagnosis(Disorders, Disorders1):-
       :diag_results(Disorders, Disorders1) &

check_normal_observation(Gr):-
       Gr==development,
inference_class :: get_value(leaves_observation, l_o_color(LC)),
inference_class :: get_value(leaves_observation, l_o_shape(LS)),
       inference_class :: get_value(stem_spot, s_s_exist(SE)),
inference_class :: get_value(stem_observation, s_o_shape(SS)),
       LC==normal, LS==normal, SE==no, (SS==normal; SS==cutting),!&
check_normal_observation(Gr):-
       (Gr==mid; Gr==late),
       inference_class :: get_value(leaves_observation, l_o_color(LC)),
       inference_class :: get_value(leaves_observation, l_o_shape(LS)),
       inference_class :: get_value(stem_spot, s_s_exist(SE)),
       inference_class :: get_value(stem_observation, s_o_shape(SS)),
       inference_class :: get_value(fruit_observation, f_o_shape(FS)),
       inference_class :: get_value(fruit_observation, f_o_color(FC)),
       LC==normal, LS==normal, SE==no, (SS==normal; SS==cutting),
       FS==normal, FC == normal&
}.
```

Fig. 9. An implementation of diagnosis task using KROL.

modeling and the implementation of expert systems methodologies. The first version was actually a side-effect of research that investigated the application of KBSs in the agriculture sector. The problem was sufficiently complex so that modularization became an imperative. Owing to its inherent modularity, Prolog Objects as a layer on top of SICStus Prolog appeared to have the greatest potential for KBSs modularization. As a matter of fact, the task was a collaborative research interest between the Swedish Institute of Computer Science (SICS) and the Central Laboratory For Agricultural Expert Systems (CLAES). However, modularization alone was not the answer. It made the system maintainable. To make it fit into the second-generation expert systems methodologies, such as KADS and GT, a fully fledged representation language was designed and implemented. This turned out to be KROL.

KROL has been of real practical use in developing expert systems for crop production management (Rafea et al., 1992). This is one of the problems that involves many parameters, and requires very complicated optimization and modeling steps. The overall production management problems involve, among other aspects, water requirements calculations, determining fertilizers and pesticides needs, water and soil salinity calculations, diagnosing the disorders or malnutrition that cause symptoms noticed by the growers, scheduling of agricultural operations and tasks, and advising about remedial and protective measures. The nature of the systems that deal with such a problem involve a large amount of non-numeric data manipulation and a lot of heuristic procedures to get near optimum solutions. Moreover, the size of the problem under consideration suffers from the lack of enough experts to support the agriculture growers, and the heavy dependence upon the experience of these experts, all of which makes the choice of the knowledge-based approach for the solution of this problem a most suitable one.

While developing these applications, the following benefits can be noted.

- Representing data in the form of objects is more modular and efficient than other forms of representation used with KBSs. The knowledge base is split into smaller, more manageable parts. Moreover, the knowledge about a problem is organized as the interaction of several well-defined, semantically related parts of knowledge.
- Organizing objects in a hierarchy reflects a top-down methodology, where a complex problem is decomposed into smaller parts that are visible to the entire system indicated by the top-level object. This characteristic lends itself to bridging the gap between the design and implementation when considering the knowledge-level modeling of KADS and task decomposition into subtasks of GT.
- Message passing allows the system to keep knowledge about data separated from knowledge about reasoning, which is critical for good data abstraction and the encapsulation of knowledge. No object-processing system is complete without full message-passing capabilities.
- Pattern-matching rules enable a clear and concise specification of the algorithm. These rules are easily implemented, understood, and maintained. They can keep track of a dynamically changing situation automatically, so their performance is superior to that of procedural rules.
- Capturing experts' reasoning knowledge and behavior are directly mapped owing to the declarative nature supported by the logic style of programming.
- Representation paradigms and inference schemes allow that a system can be developed through incrementally encoding domain-specific knowledge.
- Direct mapping of expert systems modeled through KADS and GT into KROL code. This has the effect of increasing the productivity during application developments as well as reducing the maintenance costs.

6. Conclusion

This paper describes the KROL. This language involves two aspects: the multi-paradigm knowledge representation based on logic, object-oriented, and rule-based programming paradigms; the knowledge base development support tools, such as object-oriented database management system, user interface, explanation facility, that are convenient for application development. Moreover, owing to the increased complexity of KBSs, appropriate inference mechanisms at different levels of granularity are provided.

The use of the optimized compiler of SICStus Prolog and the schemes for developing an efficient implementation improve the performance of KROL. This implementation is based on a compiler. KROL programs translate into Prolog programs, producing a program that can be directly executed. The translation is based on rewrite rules.

With KROL, the foundation has been laid to develop valuable expert systems for active use by the agriculture sector in Egypt. Four expert systems that contribute to the transfer of knowledge to extension service and farmers have been developed. The expert systems being used are mainly for crop management, which have been developed by CLAES at the Agriculture Research Center of Ministry of Agriculture and Land Reclamation in Egypt. They are: the Cucumber Expert System (CUPTEX), the Citrus Expert System (CITEX), the Tomato Expert system (TOMATEX), and Neper Wheat. CUPTEX (Rafea et al., 1991, 1995b) is an expert system for cucumber production management under a plastic tunnel. CITEX (Salah et al., 1993) is an expert system for citrus production in the open field. TOMATEX (El-Shishtawy et al., 1995) is an expert system for tomato production in different environments, e.g. under plastic tunnels, open fields, and low tunnels. Neper Wheat (Schroeder et al., 1995) is an expert system for irrigated wheat management in the open field. CUPTEX,

CITEX and TOMATEX are implemented using KADS methodology, whereas Neper Wheat is implemented using GT methodology. These expert systems are intended to be used by the agricultural extensions service within the Egyptian ministry of agriculture and by the private sector. They have demonstrated the applicability of KROL in implementing second-generation expert systems.

References

Akoka, J., & Comyn-Wattiau, I. (1996). UNIFESS: an object-oriented method for expert system design. In *Proceedings of the 3rd World Congress on Expert Systems* (pp. 614–624). Korea: Cognizant Communication Corporation.

Batarekh, A., Preece, A., Bennett, A., & Grogono, P. (1991). Specifying an expert system. *Expert Systems with Applications*, 2, 285–303.

Brogi, A., Turini, F., & Gaspari, M. (1991). Inheritance hierarchies in blackboard architectures. In E. Lenzerini (Ed.), *Inheritance hierarchies in knowledge representation and programming languages*. New York: Wiley.

Chandrasekaran, B. (1986). Generic tasks in knowledge-based reasoning: high-level building blocks for expert system design. *IEEE Expert*, 1, 23–30.

David, J., & Krivine, J. (Eds.) (1993). *Second generation expert systems*. Berlin: Springer-Verlag.

El-Shishtawy, T., Wahab, A., El-Dessouki, A., & El Azhary, E. (1995). From dependency networks to KADS: implementation issues. In *Proceedings of the 2nd FAC/IFIP/EnrAgEng Workshop on Artificial Intelligence in Agriculture*, The Netherlands.

ESICM (1992). Design of the compiler for a knowledge representation object language (KROL) on top of Prolog, Technical Report No. TR-88-024-27 Expert Systems for Improved Crop Management (ESICM), UNDP/FAO, EGY/88/024.

Etzkorn, L., & Davis, C. (1995). Knowledge-based object-oriented reusable component identification. In *Proceedings of the 8th Florida Artificial Intelligence Research Symposium (FLAIRS)*, Florida AI Research Society (pp. 97–101).

Harmon, P. (1995). Object-oriented AI: a commercial perspective. *Communications of the ACM*, 38 (11), 80–86.

Hayes-Roth, F., Waterman, D., & Lenat, D. (Eds.). (1983). *Building expert systems*. Addison-Wesley.

Hartley, R. (1985). Representation of procedural knowledge for expert systems. In *Proceedings of the 2nd Conference on Artificial Intelligence Applications: The Engineering of Knowledge-Based Systems* (pp. 256–531). Silver Spring, MD: IEEE Computer Society.

Iline, H., & Kanoui, H. (1987). Extending logic programming to object programming: the system LAP. In *Proceedings of the IJCAL* (pp. 34-39).

Kim, W. (1990). Object-oriented databases: definition and research directions. *IEEE Transactions on Knowledge and Data Engineering*, 2 (3), 327–341.

Kowalski, B., & Stipp, L. (1990). Object processing for knowledge-based systems. *AI Expert*, 34–41.

Lieberman, H. (1986). Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA* (pp. 214–223).

Lyndon, M., & Tan, C. (1995). An object-oriented knowledge base for multi-domain expert systems. *Expert Systems with Applications*, 8 (1), 177–185.

Newell, A. (1981). The knowledge level. *AI Magazine*, 2 (2), 1–20.

SICS (1995). *SICStus prolog user's manual*. Sweden, S-164 28, KISTA: Swedish Institute of Computer Science (SICS).

Rafea, A., Warkentin, M., & Ruth, S. (1991). An expert system for cucumber production in plastic tunnels. In *Proceedings of the World Congress on Expert Systems*, Orlando, FL, USA (pp. 909–916).

Rafea, A., Warkentin, M., & Ruth, S. (1992). Knowledge engineering: creating expert systems for crop production management in Egypt. In C. Mann, & S. Ruth (Eds.), *Expert systems in developing countries: Practice and promise* (pp. 89–103). Westview Press.

Rafea, A. (1995). On integrating agricultural expert systems with data bases and multimedia. In *Proceedings of the First International Conference on Multiple Objective Decision Support Systems for Land, Water, and Environmental Management: Concepts, Approaches, and Applications*, Honolulu, HI, USA.

Rafea, A., El-Azhari, S., & Hassan, E. (1995a). Integrating multimedia with expert systems for crop production management. In *Proceedings of the 2nd IFAC/IFIP/EnrAgEng workshop on Artificial Intelligence in Agriculture*, The Netherlands.

Rafea, A., El-Azhari, S., Ibrahim, I., Soliman, E., & Mahmoud, M. (1995b). Experience with the development and deployment of expert systems in agriculture. In *Proceedings of IAAI-95*, Montreal, Canada.

Roesner, H. (1988). Expert systems for commercial use. In S. Savoy (Ed.), *Artificial intelligence and expert systems chinester* (pp. 34–59). Ellis Horwood.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W. (1990). *Object-oriented modeling and design*. Prentice Hall.

Salah, A., Hassan, H., Tawfik, K., Ibrahim, I., & Farahat, H. (1993). CITEX: an expert system for citrus crop management. In *Proceedings of the 2nd National Expert Systems and Development Workshop (ESADW-93)*, Ministry of Agriculture and Land Reclamation, Cairo, Egypt.

Schreiber, G., Wielinga, B., & Breuker, J. (Eds.). (1993). *KADS: A principled approach to knowledge-based system development, knowledge-based systems*. San Diego, CA: Academic Press.

Schroeder, K., Kamel, A., Sticklen, J., Ward, R., Ritchie, J., Schulthess, U., Rafea, A., & Salah, A. (1994). Guiding object-oriented design via the knowledge level architecture: the irrigated wheat testbed. *Mathl. Comput. Modeling*, 20 (8), 1–16.

Schroeder, K., Kamel, A., Sticklen, J., El-Skeikh, E., Ward, R., Ritchie, J., Schulthess, U., Rafea, A., & Salah, A. (1995). Neper Wheat: an integrated architecture for irrigated wheat crop management. In *Proceedings of the 2nd IFAC/IFIP/EnrAgEng workshop on Artificial Intelligence in Agriculture*, The Netherlands.

Swinney, L. (1995). The explanation facility and the explanation effect. *Expert Systems with Applications*, 9 (4), 557–567.

Vranes, S., Stanojevic, M., Lucin, M., Stevanovic, V., & Subasic, P. (1994). A blackboard framework on top of Prolog. *Expert Systems with Applications*, 7, 109–130.

Xu, D., & Zheng, G. (1994). A hybrid knowledge representation based on logical objects. In *Proceedings of the 2nd International Conference on Expert System for Development* (pp. 153–159). Thailand: IEEE Computer Society Press.

Yahaya, N. (1994). On the development of environments for developing expert systems. In *Proceedings of the 2nd International Conference on Expert Systems for Development*, (pp. 24–29). Thailand: IEEE Computer Society Press.

Ye, L. (1990). User requirements for explanation in expert systems. Ph.D. dissertation, University of Minnesota, Minneapolis, MN.

Yoshiyuki, & Koseki (1987). Amalgamating multiple programming paradigms in prolog. In *Proceedings of IJCAI* (pp. 76–82).