Reusable Knowledge-based Components for Building Software Applications: A Knowledge Modelling Approach

Martin Molina, Jose L. Sierra, Jose Cuena

Department of Artificial Intelligence, Technical University of Madrid

Campus de Montegancedo s/n 28660 Boadilla del Monte, Madrid, SPAIN

Tel: 34-91-3367390, Fax: 34-91-3524819

{mmolina,jlsierra,jcuena}@dia.fi.upm.es

http://www.dia.fi.upm.es

This version corresponds to a preprint
of the actual paper published in:
International Journal of Software Engineering and Knowledge Engineering,
Vol. 9 No. 3 (1999) 297-317.

Abstract.

In computer science, different types of reusable components for building software applications were proposed as a direct consequence of the emergence of new software programming paradigms. The success of these components for building applications depends on factors such as the flexibility in their combination or the facility for their selection in centralised or distributed environments such as internet. In this article, we propose a general type of reusable component, called *primitive of representation*, inspired by a knowledge-based approach that can promote reusability. The proposal can be understood as a generalisation of existing partial solutions that is applicable to both software and knowledge engineering for the development of hybrid applications that integrate conventional and knowledge based techniques. The article presents the structure and use of the component and describes our recent experience in the development of real-world applications based on this approach.

Keywords:

reusable software components, knowledge modelling, software engineering and knowledge engineering integration

1 Introduction

Software reuse is one of the most important areas of interest in computer science as it may improve the productivity and quality of software development. Since the consideration of this activity as an engineering process, different proposals about reuse have been made following different models. As new programming paradigms for application development were proposed (e.g., structured programming, functional programming, object-oriented programming, etc.) new approaches for reuse were also considered, giving rise to interesting contributions that have reduced considerably the effort required for building new applications. In particular, the scenario provided by internet has promoted the interest in this approach because it provides a virtual platform with searching facilities that significantly increases the number of potential users of available reusable components.

In this scenario, it is desirable to have powerful building blocks that are adaptable enough to a wide range of potential situations and, at the same time, a natural and intuitive combination scheme to be easily understood by developers. In this article, we present a proposal in this direction inspired by recent advances in the field of knowledge engineering. We describe a new type of component called *primitive of representation* (that generalises and integrates software and knowledge engineering) together with a combination scheme based on a knowledge modelling approach. The proposal is based on several years of experience in the use of a software environment developed by our group that assists developers in building knowledge-based or conventional applications. This environment, called KSM (Knowledge Structure Manager), allows a developer to construct an application following a knowledge modelling philosophy, using basic reusable pre-programmed components.

The article is organised as follows. Section 2 provides an overview of the different approaches about reusable components in the fields of software and knowledge engineering. Section 3 describes our framework for component reuse, based on a knowledge modelling approach. Section 4 presents our proposal of type of reusable component, the primitive of representation. This section outlines: the structure of a primitive, how primitives are combined for building complex architectures, how they are

organised in library and, finally, their computational support. Section 5 describes examples of our practical experience in using this type of components in the development of real world projects.

2 General View of Software/Knowledge Reuse

In *software engineering*, two main approaches to reusability can be identified [2, 16]: one based on a generative approach, where the goal is to reuse the processes involved in the generation of applications, and another one based on building blocks, where the principle of reuse is the composition of atomic reusable software components. From the point of view of building blocks, there are different degrees of software reuse, ranging from the simple reuse of libraries of subroutines to complex cases such as libraries of classes using an object oriented approach [3, 7], templates [26] or partial software architectures considered as generic applications. Recently, even more complex solutions have been proposed, such as the concept of object oriented framework [17] that provides a general environment of interrelated classes following standard design patterns [12] for the development of applications in a particular domain.

In *knowledge engineering*, reuse has also been considered to improve the development of knowledge-based systems. In the early times, one of the most successful results for reuse was the proposal of the *shell* concept, where the whole knowledge-based architecture is reused, except the content of the knowledge base, which has to be written for each application using a symbolic declarative representation. In the last decade, the development of knowledge-based systems has been mainly considered from a model-based point of view [4, 27], which has provided new opportunities for reuse. Using this paradigm, a developer constructs an abstract conceptual model in order to emulate a certain class of problem-solving behaviour. The model can be used as a guide of the knowledge acquisition activity and finally the resulting product is operationalised by using the corresponding programming languages and software tools. Basically, two different approaches for reuse may be identified in this field [20]. On the one hand, the *domain-oriented* approach tries to reuse a declarative domain description among different kind of problems. This approach uses the notion of ontology to describe explicit specifications of the domain

elements [14]. On the other hand, the *task-oriented* approach has the goal of reusing problem-solving knowledge in different domains. This approach uses the concept *of problem-solving method* (PSM) which is present in different knowledge engineering frameworks [6, 19, 24]. From a task-oriented approach, the management of libraries of reusable generic PSMs has been proposed in order to suggest to the developer possible knowledge structures to be considered during the knowledge acquisition process [1, 5].

To facilitate the implementation of knowledge models and libraries of PSMs, instead of using just programming languages, several research groups have proposed certain pre-programmed reusable software constructs. These proposals include *mechanisms* (knowledge-based components implementing a basic problem-solving method that carries out a generic task) used in the PROTÉGÉ-II software tool [23], the *application kit* with solution methods (LISP code fragments implementing basic functions) that is used in the KREST software tool [18], and also the type of components that we propose in this article (called *primitives of representation*), used within the KSM software tool [8, 9]. This type of components presents two main differences with the type of components used in software engineering: (1) they include an internal knowledge-based architecture, with a local knowledge base with a rich declarative representation which increases the flexibility for their adaptation, and (2) they present an important analogy with the descriptive entities used during the formalisation of knowledge models.

In summary, there exists a wide range of solutions for reuse from both the software and the knowledge engineering fields. One significant common trend is that component reuse provides an advanced view of software development where, instead of considering the software programming activity as the traditional process of writing sets of imperative sentences of code using a particular programming language, the new approach is based on a *modelling* view where the developer selects, adapts and combines basic building blocks to construct a model of an observed world. From this point of view, a complete reusable component paradigm needs to answer at least the following four questions:

- *How to choose an appropriate component?*. Components need to be stored in a server which can be either centralised in a computer or distributed in a network of servers such as internet. Here, search

facilities are needed to help users in selecting the appropriate components. These facilities can range from simple traditional approaches (e.g., catalogues of components) to more advanced ideas recently proposed in the field of knowledge engineering: management of expressive abstract semantic descriptions of components, fuzzy pattern-matching, case-based reasoning, etc.

- *How to adapt the selected component?*. A second important point is the possibility of adaptation of components to a particular problem to ensure certain degree of generality. Some solutions to the adaptation problem use parameters that need to be specified for the particular problem or rely on a transformational approach that considers the adaptation process as a sequence of successive transformations from the original component to the final version. From the side of knowledge engineering, it is possible to use richer declarative knowledge representations that offer more flexible adaptation.
- *How to combine several components?*. Another key issue is the assembly of a set of components to configure a complete architecture. For this purpose it is necessary: to give a solution to the problem of *interoperability*, i.e., a standard and homogeneous communication between components together with a solution for software/knowledge sharing in order to avoid redundancy and inconsistency, and a *combination scheme* based on the use of descriptive entities and relations between them. Here, the knowledge engineering field can offer intuitive combination schemes inspired by natural descriptions of reasoning processes according to recent knowledge modelling proposals.
- *How are components programmed?*. Finally, a last important characteristic is to define how each component is programmed, i.e. which is its computational support taking into account issues such as efficiency, portability, etc. This point is sometimes missed by certain theoretical proposals of reusable components, particularly in the knowledge engineering community, but it is very important for the success of reusability.

In summary, the goal is to provide integral and uniform solutions for component reuse, applicable to the development of both conventional and knowledge-based applications, to produce a significant

decrease of the required effort for building and maintaining software applications compared to the traditional development methods. In our view, currently, there is an interesting set of partial solutions and individual theoretical or practical efforts, but they still need to be adapted to propose unified and complete instrumental solutions applicable for real problems.

In particular, in this article, we present a proposal in this direction which is based on our recent work and experience using a knowledge modelling approach for building software applications. We have developed a methodology to design, implement and maintain knowledge models based on reusable knowledge-based components. The methodology is supported by a software environment called KSM and has been validated in the context of the development of different real-world projects. The KSM approach provides some advanced answers to the previous four questions within a uniform framework by integrating solutions from the software and knowledge engineering fields. In particular, the methodology proposes (1) a general type of component, applicable to conventional as well as to knowledge-based systems, that can be more adaptable due to the use of domain representation formalisms and (2) proposes a way to combine components based on a knowledge modelling conception where new descriptive entities are used. In addition to that, the KSM approach provides an efficient computational support for components. What still needs to be improved within this framework is to provide better automatic tools for component selection and validation of configurations (for this purpose, we are currently studying different solutions based on explicit semantic formal characterisation of components, case-base reasoning and partial pattern-matching using different uncertainty models).

The following sections introduce first of all our view of knowledge modelling that serves as a framework to formulate hybrid (knowledge-based and conventional) software architectures at a high level of abstraction. Then, our proposal of reusable component is presented, describing its structure and how it is used for building software architectures.

3 A Knowledge Modelling Paradigm to Formulate Software Architectures

The model-based approach follows an intuition in which the developer observes a world of reference and constructs a model whose behaviour, simulated by computer, reproduces the observed behaviour. In particular, the knowledge modelling approach is centred in building models of expertise based on observed problem-solving actions in human specialists. For this process, it is necessary to use appropriate descriptive entities, intuitive enough, that naturally allow the developer to describe reasoning processes, together with a computational translation to produce the operational version. For this purpose, the knowledge engineering community has recently proposed high level concepts (e.g., the problem-solving method, hierarchy of tasks, ontology for domain description, etc.) that are useful to formulate complex knowledge models at different levels of abstraction. This is, actually, a better way to describe a software system because it is closer to the human reasoning and, from the architectural point of view, offers a more natural framework to combine reusable software components for building the whole system.

According to this view, we developed a software environment that helps developers to build models following a knowledge modelling view [8, 9] (`http://www.isys.dia.fi.upm.es/ksm`). This environment, called KSM (Knowledge Structure Manager), provides the developers with a set of high level descriptive entities to formulate a complex knowledge model (figure 1). Basically, each model is conceived as a hierarchically structured collection of knowledge areas which are refined internally by means of complementary views using conventional knowledge engineering entities (tasks, methods, etc.). In addition, KSM provides a library of configurable knowledge-based components to select the appropriate representation and to operationalise each basic module. The environment helps developers in applying a knowledge modelling methodology in order to build the operational version of the final system and it also assists end-users during the operation and maintenance of knowledge models.

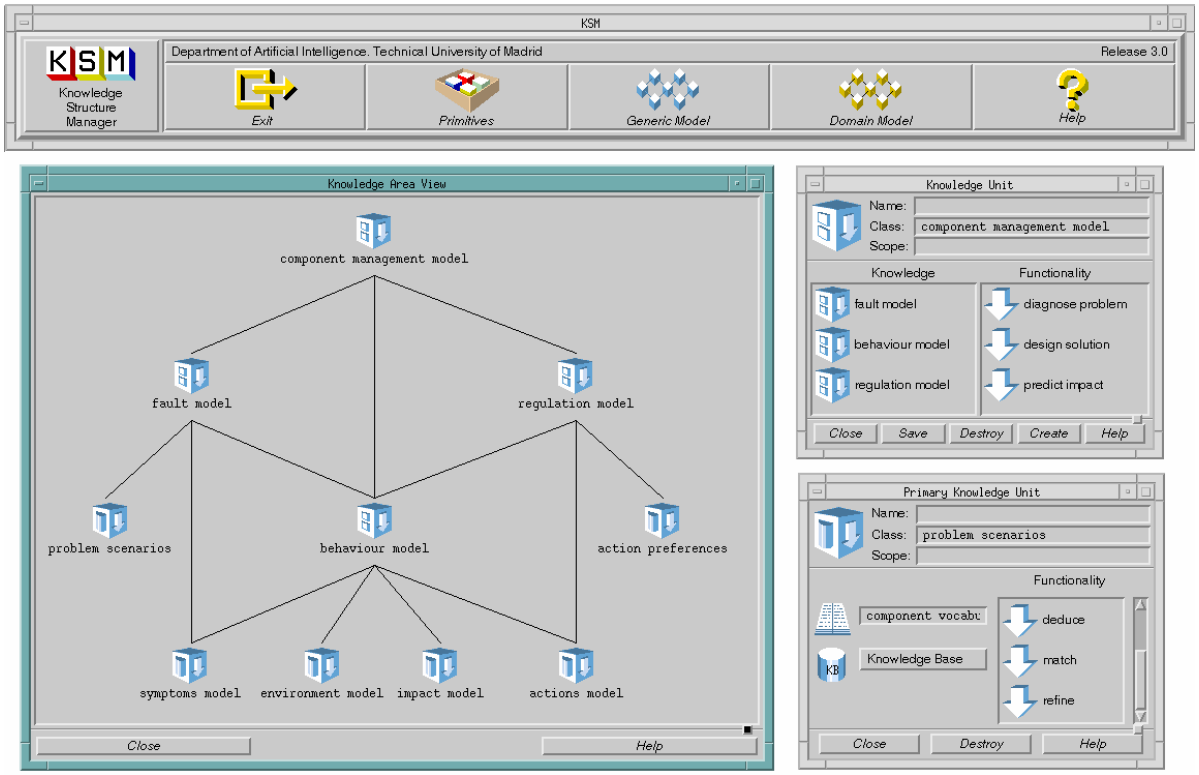

Figure 1: Example screen of the KSM environment.

In more detail, a KSM knowledge model comprises three main perspectives: (1) the knowledge-area perspective, which is the central structure of the model (conceived as a structured collection of knowledge bodies), (2) the task perspective, that describes the problem-solving behaviour of the model and (3) the vocabulary perspective, which includes the basic terms shared by several knowledge modules. The knowledge-area perspective is used for presenting a general image of the model, where each module represents what is called a *knowledge area*. In general, a knowledge area identifies a body of expertise that explains a certain problem-solving behaviour of an intelligent agent. Typically, a knowledge area is associated to a professional skill, a qualification or speciality of an expert. For instance, in a medical domain, there could be a knowledge area about clinical frames in the field of infectious diseases relating symptoms and diseases, and another area about therapies with includes treatments to be applied to certain diseases. The whole knowledge model is a hierarchical structure of knowledge areas with a top-level area representing the entire model. This area is divided (using the part-of relation) into other more detailed

subareas which, again, are further divided into other simpler areas and so on, developing the whole hierarchy (where some areas may belong to more than one higher level area). A bottom level area is called *primary knowledge area* and corresponds to an elementary module that may be directly operationalised by means of basic software building blocks.

Knowledge areas are active modules that, in general, provide different services represented by a set of tasks. The task perspective presents a functional description of each task using a tree of task-method-subtasks. A *task* is a goal that can be achieved by knowing about a certain knowledge area. The task receives input data and generates output data as a result of its reasoning. Examples of tasks are: medical diagnosis of a patient, design of the machinery of an elevator and mineral classification. The *method* describes how to carry out the task by using a particular problem-solving strategy. When a developer decides to use a certain method to accomplish a task, this task is divided into a set of subtasks. These subtasks may also be refined by methods generating new subtasks. In KSM, methods are formulated using a particular language called *Link* that is supported by a run time interpreter [22].

Finally, the vocabulary perspective is formulated by means of the so-called conceptual vocabularies. A *conceptual vocabulary* defines a basic terminology used by several knowledge areas. A vocabulary is not a description of the whole domain knowledge, but rather defines a partial view of the basic terms that are common to different knowledge bases. In KSM, vocabularies are formulated using a particular language called *Concel* that uses a concept-attribute-facet representation together with an organisation in classes-subclasses-instances [9].

The developer formulates the cognitive architecture of the knowledge model as a structure of knowledge-areas, tasks and vocabularies. The resulting model is an abstract description of the types of knowledge and strategies of reasoning that supports a certain problem-solving behaviour. However, in order to produce the operational version of such a model (executable on the computer) the developer needs to make design decisions, selecting appropriate software constructs to support knowledge representation and inference methods for each primary area. For this purpose, KSM provides the so-called

primitives of representation. The rest of the article describes in detail the characteristics of these primitives.

4 A Reusable Knowledge-based Component: The Primitive of Representation

This section describes our proposal of reusable component, called *primitive of representation*, that was defined to implement KSM knowledge models. A primitive of representation is a reusable pre-programmed software component that implements a generic technique for solving certain classes of problems. The primitive defines a particular domain representation using a declarative language together with several inference procedures that provide problem-solving competence. In a simplified way, the structure of the primitive is defined by a pair $<L, I>$, where L is a formal language for knowledge representation and $I = \{i_j\}$ is the set of inferences, i.e., a collection of inference procedures that use the declarative representation written in L.

The module defined by a primitive is a design decision that is mainly influenced by the representation language L. This language is usually homogeneous, declarative and close to personal intuitions or professional fields. In a particular primitive, this language can adopt one of the representations used in knowledge engineering such as: rules, constraints, frames, logic, uncertainty (fuzzy logic, belief networks, etc.), temporal or spatial representations, etc. Also other parameterised or conventional representations can be considered, such as the parameters of a simulator or a graph-based language. According to the expressiveness of language L, primitives can be classified into three categories: (1) *knowledge-based primitives*, that rely on a complex language such as representations used in knowledge engineering (2) *parameterised primitives*, that use a simple language based on the use of several parameters (for instance, a simulator of a river where a set of physical parameters describe the characteristics of the river) and (3) *black box primitives* that do not have any explicit language, for instance, a software module for statistic operations or a module specialised in time series management. Each primitive of representation, viewed as a basic software tool, provides local knowledge acquisition

facilities to help developers in writing a domain model using the language *L*. For instance, primitives can provide knowledge base editors where the specific appearance of the declarative language can be diverse (e.g., text-based, graphical-based, etc.).

Each element of the set of inferences *I* expresses an inference procedure that uses the knowledge formulated in the language *L*. For instance, the rule-based primitive may have an inference, called *forward chaining*, that implements an inference procedure following a forward chaining strategy to determine whether a goal can be deduced from a set of facts given the rules of the knowledge base. In addition, there may be also another inference that follows the backward chaining strategy for the same goal. Each inference i_j defines a pair $<P, C>$ where *P* is a set of inputs (premises) and *C* is a set of outputs (conclusions).

This theoretical and simplified description of the structure of primitives needs to be considered in a more complete way from the point of view of a developer who wants to use primitives for building a particular application. Thus, the user of primitives needs an explicit description to decide upon its applicability to a particular problem. Figure 2 shows a complete description of a primitive. This description is divided in two main parts: the domain representation of the primitive, and the functional description, i.e., the set of inferences. Concerning the domain representation, different issues are considered such as characteristics of the representation language, domain assumptions, knowledge editing and acquisition tools provided by the primitive and the technical performance. From the functional point of view, each inference is described by different slots: the goal, inputs and outputs, inference assumptions, the strategy of reasoning, the explanation facilities and the performance.

Characterization of a Primitive of Representation		
Name:		*Name of the primitive of representation*
	Domain Representation:	
Representation Language:		*A set of syntax and semantic rules to formulate a domain model using the representation provided by the primitive*
General Domain Assumptions:		*A collection of general necessary characteristics of the domain that must be satisfied in order to be able of applying the primitive of representation.*
Knowledge Ed./Acquisition Tools:		*Available tools supplied by the primitive to help in the acquisition and edition. For instance, language translators to import fragments of knowledge bases, automatic learning tools or graphical editors for the knowledge base.*
Performance:		*A set of properties about the performance of the domain representation such as knowledge-base memory consumption or loading/compilation time.*
	Functional Description	
Inference-1:		*Name of an inference procedure.*
	Goal:	*A summary of the objective of the inference, formulated as a set of sentences about input and output roles.*
	Inputs:	*A list of names for input roles, together with a description of expected syntax and, optionally, their relation with the domain description. In addition to this, a list of names for parameters and type of values to select operation modes.*
	Outputs:	*A list of names for output roles, together with a description of the expected syntax and, optionally, their relation with the domain description. In addition to this, a list of the potential control states that express degrees of success of the task execution.*
	Inference Assumptions:	*A set of declarative sentences about necessary properties of the input roles and the domain that must be satisfied in order to be able of applying the inference method.*
	Strategy of reasoning:	*An abstract description of the prblem solving strategy in terms of knowledge base manipulation oriented to facilitate the understanding of explanations.*
	Explanation Facilities:	*A set of available types of questions accepted by the to produce explanations (e.g., why, why not, how, what if,*
	Performance:	*A set of properties about the inference execution performance, for instance, the average answer time (in terms of components of the knowledge base), computational complexity, etc.*
Inference-2:		***...***
...		***...***
Inference-N:		***...***

Figure 2: Information associated to a primitive of representation.

In principle, this description is present for each available primitive, using a textual format with natural language understandable by human developers. However, it is desirable here to use a general, formal and computational representation for this description in order to be able to provide automatic tools that help developers in the selection, adaptation and combination of primitives. This is still an open issue

that needs to be practically solved by researches and developers. For the moment, there are certain interesting partial proposals considering formal characterisations for domain assumptions, non-functional requirements, etc. One interesting possibility for this need is the use of ontological descriptions (using standard languages such as KIF-Ontolingua). This type of languages provides rich, expressive and formal declarative representation that could be useful for this purpose, even across distributed environments such as internet. However, it is still necessary to propose and agree on standard valid descriptions for knowledge/software components in order to make extensive use of this approach.

4.1 Adaptation of Primitives of Representation

This section describes how the primitive of representation is refined to be used in a particular domain. The basic idea is that the developer adapts the primitive (1) by defining the role of the primitive within the global architecture, and (2) by writing its domain knowledge base following its particular representation using the local acquisition facilities.

As presented in previous sections, during the formulation of a particular knowledge model, the developer defines an abstract structure that constitutes a description of a cognitive architecture. Basically, the central structure of this model is defined as a hierarchy of knowledge areas, where each area is divided into subareas until elementary areas are reached (called *primary* knowledge areas). Once this pattern has been formulated, a particular knowledge representation must be selected for each primary area. The purpose of a primitive of representation is to provide a valid local representation for primary areas and to serve as a computational construct to produce their operational version. Thus, for every primary knowledge area of the model, the developer selects the most appropriate primitive of representation.

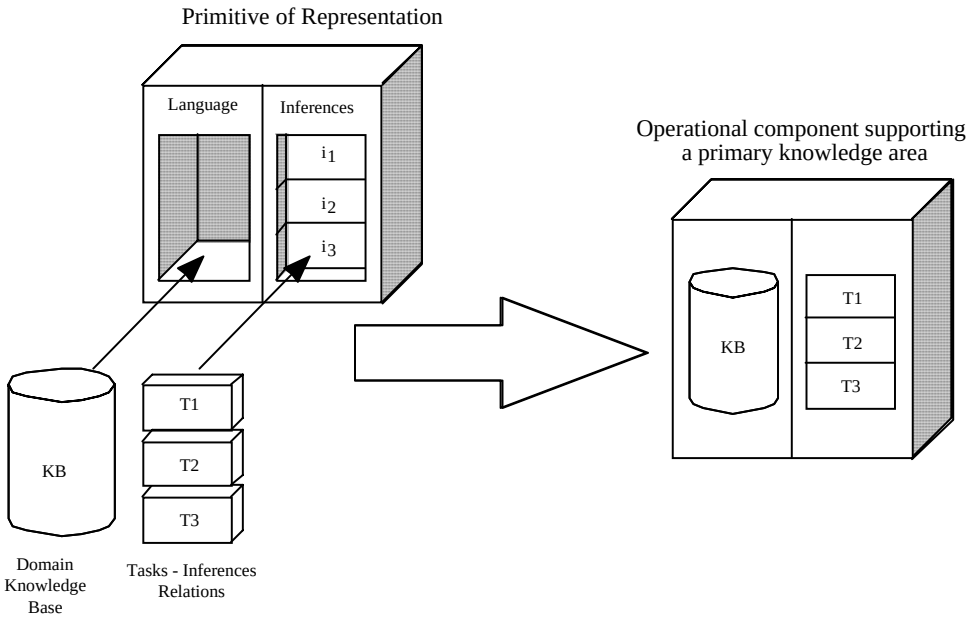

Figure 3: The primitive serves as a template that has to be filled with a domain knowledge base and a set of inference mappings for building the operational component that supports a primary knowledge area.

The process of implementing a primary area by using a primitive is viewed as a specialising process of the general technique supported by the primitive. During this specialisation, the primitive is considered as a template that is filled with two classes of information (figure 3): (1) *task-inference relations*, where the developer relates each task of the primary area with one inference of the primitive (obviously, it is not necessary to use all the inferences of a primitive to implement a particular primary area), and (2) *domain knowledge base*, that includes the specific knowledge of the particular application using the representation language provided by the primitive. For example, consider a frame-based primitive of representation where one of its inferences is called *match* that identifies the frame (or frames) closest to a partial description. This inference receives as input a *partial description* and generates as output a *category*. This primitive can be used to construct a primary area about problem scenarios in an urban network of streets, where it is necessary to identify traffic problems using a partial description provided by sensors on the roads. This primary area, called *problem scenarios*, includes a task called *identify problem* that receives as input *observables* provided by sensors and produces as output the

detected *traffic problem*. In order to use the primitive in this domain it is necessary to map the tasks of the primary area (as well as their inputs and outputs) into inferences provided by the primitive. Thus, in this case, the task *identify problem* of the primary area is associated with the inference *match* of the primitive and the corresponding equivalencies between inputs and outputs are established (*observables* are the *partial description* and *traffic problem* is the *category*). In doing so, the developer specialises the primitive in a particular domain expressing the role that the primitive plays in the knowledge model. For this purpose, the KSM environment provides a specific formal language called *Link-S* [25].

In a second step, the developer acquires the specific domain knowledge. For instance, following the previous example, the generic area *problem scenarios* (supported by the frame-based primitive) is used for building a domain knowledge area in the particular case of the city of Madrid. Thus, the developer acquires the particular information for creating the content of the knowledge base and writes the specific traffic problem scenarios of the city of Madrid using the frame representation language provided by the primitive. Then, the resulting module is ready to be executed in the computer for detecting problems. The same generic area could be also used to construct the problem scenarios of a different city, for instance the city of Barcelona by writing different frames.

In summary, implementing knowledge models is viewed as a process of specialisation of building blocks where the developer first specifies the role that the knowledge block plays in the whole model and, then acquires the specific knowledge of the domain to create the final component. One of the interesting advantages provided by the primitive is that there is a clear analogy between primitives and knowledge areas, so this offers an easy transition from the implementation-independent model (as a result of analysis phase) to the operational version. This continuity between both phases reduces the required effort for operationalising knowledge models and, since the implementation preserves the structure defined by the abstract model, it also improves the understandability and flexibility of the final system.

In addition to this, theoretically, it would also be possible to consider a greater degree of adaptability of the primitive if we consider the possibility of adapting the representation language for a particular

domain. The idea is that the primitive provides the language *L*, but in a particular domain a different language *L´* is used for representing the domain knowledge. This provides as advantage that the resulting knowledge base uses the terminology of the final domain, instead of the abstract terminology used by the primitive (such as rules, frames, etc.). Thus, the knowledge acquisition process can be easier since concepts are expressed using domain references that are more meaningful to professional experts. A way of implementing this idea is to include a parameterised module that translates the language *L'* to the language *L* (where parameters are used to define the characteristics of the language *L*). Another solution is to define mappings between *L* and *L´*, following a similar idea implemented in the Protégé-II environment [13]. The main practical problem of these approaches is the production of explanations. Explanations have to be expressed in the same domain language that is used for building the knowledge base, so it would also be necessary to translate the explanation to the domain language after reasoning. This could be done by establishing a general format for explanations, but it imposes an internal structure for all the primitives. This shows a trade-off between the degree of adaptability of existing primitives and the flexibility that a programmer has for building new primitives. Therefore, although there are some proposals for adapting the language *L* to a particular domain, this problem is not solved completely yet, so further research in this direction is necessary.

4.2 Combination of Primitives of Representation

The combination scheme established for primitives of representation follows the model defined as a structure of knowledge areas (figure 4). Each primitive is associated with one or more primary areas and then, each primary area is part of higher level knowledge areas. Inferences provided by primitives are associated with primary tasks that, in their turn, are combined to define higher level strategies of reasoning of problem solving methods. Methods are formulated by using the Link language [22] defining a line of reasoning with two main parts: the data flow section, that defines how sub-tasks are connected, and the control flow section, that uses rules to establish the execution order of sub-tasks.

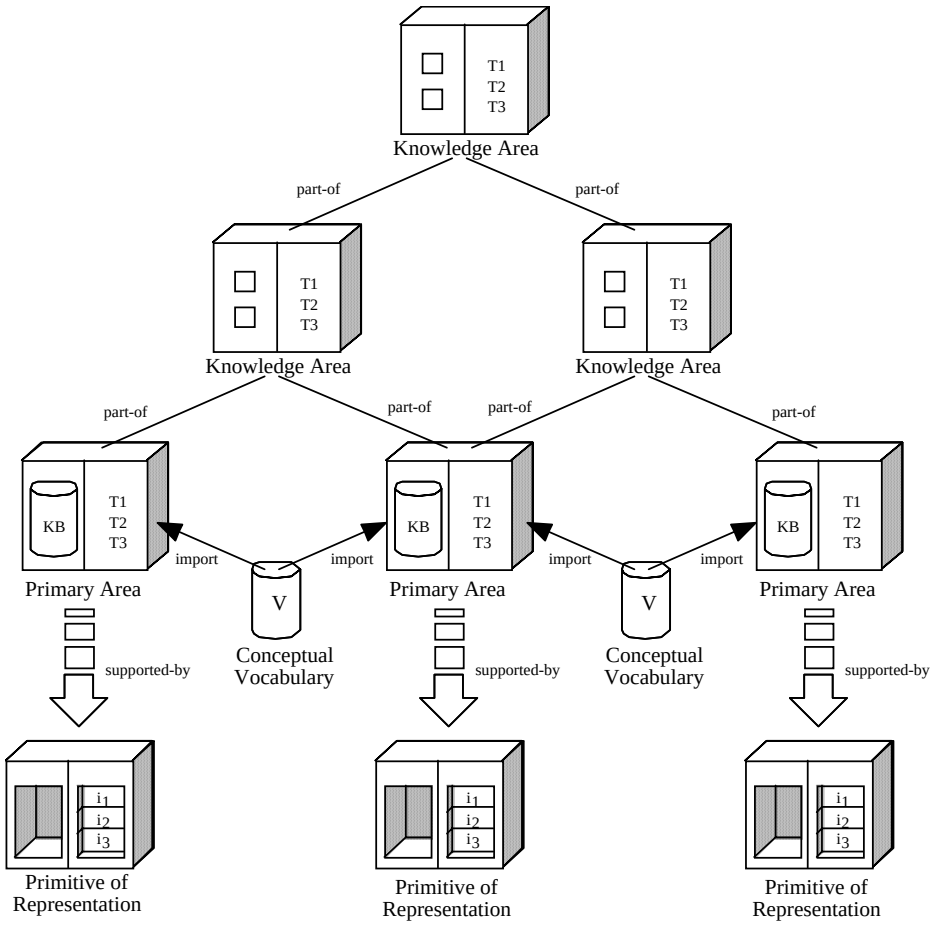

Figure 4: Combination of primitives of representation. Each primary area is implemented by one primitive and a primary area can be part of other knowledge areas. Primary areas share conceptual vocabularies by importation to use a common terminology in local knowledge bases.

On the other hand, the representation language of the primitive is used to formulate a declarative model of the domain knowledge. However, other different primitives can share part of the domain-specific information, so it is also necessary here to give a solution for knowledge sharing in order to avoid redundancy and inconsistency. This idea about common concepts is considered in the KSM environment with the use of conceptual vocabularies. Vocabularies define global sets of concepts to be shared by different knowledge areas and, therefore, they have to use a general representation, the Concel language. From the point of view of primitives of representation, they must be capable of sharing vocabularies. The solution to this is that the primitive provides mechanisms to import Concel definitions that are translated to the local representation language of the primitive. Thus, when the user of the primitive needs to write a

particular local knowledge base during the knowledge acquisition phase, the vocabularies shared by the primitive are previously imported to be part of the base, in such a way that vocabularies are directly available in the language of the primitive. To implement this idea, every primitive may have its own specific procedures to import vocabularies that create the corresponding data structures according to the internal representation followed by the primitive. An alternative approach to this is described in [25], which is based on a more general view.

4.3 **Selection of Primitives of Representation**

To facilitate the construction of knowledge models, a library of pre-programmed reusable primitives of representation may be taken into account. This library provides a developer with certain facilities to select and to use primitives according to specific requirements. The library may include as primitives some of the most extended reusable techniques for knowledge representation: production rules, logic clauses, frames, belief networks, qualitative constraints, artificial neural networks, etc. In addition, the library may include other conventional reusable software tools such as simulators, statistics tools, etc. However, the library must not be a closed environment. It has to be open in order to be able to add new primitives according to the necessity of the development of particular knowledge models. For instance, to construct a knowledge model for urban traffic management, it may be necessary, among others, a knowledge area to classify traffic problems, which may be implemented by a rule-based primitive, and a knowledge area to model the traffic behaviour, which must be implemented by a traffic simulator. The rule-based primitive may be already be present in the library, so it will be directly reused to create the knowledge area for traffic problems. But the traffic simulator could not be in the library. In this case, a new object will have to be programmed, implementing the traffic simulator, and it will be included in the library to be considered a new primitive. This new primitive will be useful for developing the corresponding knowledge area, as well as to be available to be reused in the development of other knowledge models.

The current version of the KSM environment provides a simple method, based on the idea of catalogue, to store and select primitives. Primitives are classified into three categories: knowledge-based, parameterised and black box. Within each category, they are linearly organised, so the developer need to search in a list of names the most appropriate primitive. In addition to that, KSM is able of reusing abstract configurations of primitives. This is provided by the management of generic models as a structured collection of abstract knowledge areas. Of course, the catalogue approach is a valid solution when the number of primitives is low. However, in a different scenario where there are a larger number of primitives produced by several development centres it is necessary to provide a richer indexing method to facilitate the selection of the most appropriate primitive.

In summary, primitives are stored in an open library that includes and organises the collection of existing primitives, but accepts the inclusion of new ones according to the necessity of particular applications. The use of this kind of library avoids the commitment to a particular knowledge representation, allowing the developer to use the most convenient for each case and to develop architectures that include multiple. This is particularly important in real world applications where efficiency must be taken into account and it is a solution for building hybrid architectures that include knowledge-based components and algorithmic components, so it is appropriate as an integrated approach of knowledge-based and conventional techniques.

4.4 **Computational Support of Primitives: an Object Oriented Solution**

At the implementation level, the primitive is a software module designed and implemented as a *class* (from the object-oriented development point of view), i.e. programmed with a hidden data structure and with a collection of operations which are activated when the class receives messages. A class implementing a primitive (figure 5) includes, on the one hand, an internal data structure divided into three

basic parts: (1) a data structure to support the *local vocabulary* used by the knowledge base (for instance, in the case of a representation of rules, this part contains the set of concepts, attributes and allowed values that will be valid in the knowledge base), (2) a data structure that implements the internal structure that supports the *knowledge base* as a result of the compilation of the language provided by the primitive, and (3) a *working memory* that stores intermediate and final conclusions during the reasoning processes together with traces that can serve to justify conclusions through explanations. The data structures (1) and (2) are created during the knowledge acquisition phase and the data structure (3) is created and modified during the problem-solving phase when inference procedures develop their strategies of reasoning.

On the other hand, the class implementing a primitive includes a set of external operations that can be classified into three types: (1) *knowledge acquisition operations*, whose purpose is to help the user in creating and maintaining the knowledge base, (2) *problem-solving operations* that execute the inferences provided by the primitive; they receive a set of inputs and generate responses using the internal structure representing the knowledge base, and (3) *explanation operations*, that justify the conclusions using the traces stored in working memory. If the primitive is not knowledge based, the corresponding object includes neither knowledge acquisition nor explanation operations.

Each particular primitive is implemented as a different class. When a primary knowledge area is built using a primitive of representation, internally an instance of the corresponding class is created automatically. The instance inherits characteristics of the class such as inferences and knowledge representation. Each instance of the class representing the primitive includes two types of specific information: (1) task-inference relations, the primitive is associated with the primary area by using associations that relate tasks of the primary area and inferences of the primitive, and (2) a particular knowledge base that includes the domain knowledge for the particular primary knowledge area.

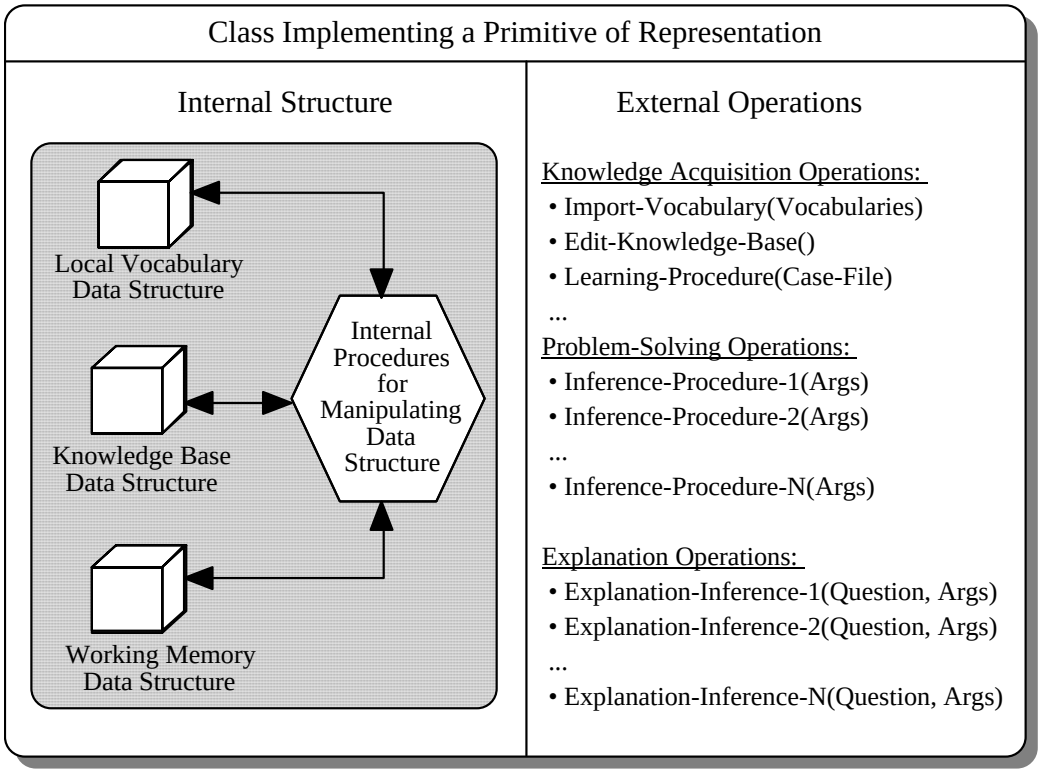

Figure 5: Structure of the class that implements a primitive of representation

During the creation of a knowledge model, the developer constructs each primary knowledge area using a copy of the corresponding primitive. This creation is internally supported by the creation of an instance of the class that implements the primitive. The primitive provides operations for editing the knowledge base. These operations present an external user-friendly view of the knowledge base to the operator, with facilities to create and modify the knowledge base. Note that, with this scheme, there is not a unique global acquisition method for a given knowledge model that uses several primitives of representation, but each primary knowledge area uses its own method supplied by its particular primitive. Therefore, the knowledge acquisition facilities provided by a particular knowledge model are the union of the acquisition facilities provided by all the used primitives. A basic operation to create the knowledge base is to import a conceptual vocabulary. This means that the primitive is capable of importing a global definition of concepts (written in Concel language) in order to be able to share common definitions with other modules.

During the execution of tasks of the knowledge model, the problem-solving operations of the corresponding objects of the primitives are invoked with input data. Those local operations manipulate the

internal data structure of the knowledge base to generate outputs. During the problem solving reasoning, the operations produce intermediate and final conclusions that are stored in the working memory. This information is used later, when the user of primitives wants to get explanations that justify the conclusions of the reasoning.

In summary, the consideration of the primitive as a class understood in the context of object-oriented development provides modularity and a solution at implementation level for structuring and sharing the software elements that support the primitive. The modularization philosophy of the object-oriented viewpoint, where each module is associated with an intuitive entity that belongs to a model of a certain world is very appropriate to be used in primitives, where each module identifies a representation technique for operationalising knowledge models. The encapsulation provided by the class allows implementing the most efficient solution to support each primitive hiding the data structures (with internal processes) corresponding to the management of the knowledge base and the working memory.

5 Practical Experience and Examples

This section summarises the experience of our research group in the last five years (since the proposal of the KSM environment) using this scheme of component reuse. We have developed different real applications as well as academic projects in different domains where it has been necessary to programme and reuse primitives. For instance, a group of primitives was used to develop applications in domains such as traffic control [10, 11, 21], intelligent user interfaces [15], emergency management (e.g., in the hydrology field) and design of the machinery of elevators. Figure 6 shows some of the primitives of representation that were developed or reused in these projects.

Category	Representation	No. Lines	Language	Comments
Generic	Hierarchies of concepts	12100	C++	A primitive with a language of concept-attribute-value with classes and instances with inheritance and selection procedures
	Rules with Uncertainty	11000	C	A rule-based primitive with uncertainty using the Mycin and the Dempster and Shafer models
	Production Rules	29300	C++	A rule-based primitive with backward chaining inference, for the development of production systems

	Qualitative Constraints	9500	C	A constraint-based primitive for qualitative variables and an inference procedure based on the Waltz algorithm
	Parametric Constraints	2700	Prolog	A constraint-based primitive for numerical and qualitative parameters and procedures for verification
	Patterns	28200	C++	A pattern-based primitive with partial pattern-matching inference procedures
	Logic Clauses	450	Prolog	A clause-based primitive with inference procedures based on automatic deduction of logic programming
	Belief Networks	26100	C++	A primitive for belief network representation with inference procedures for probabilistic reasoning
Domain-Specific	Road Network	7800	Prolog	A primitive for representing road networks with procedures for simulating the traffic behaviour
	Traffic Scenarios	4600	Prolog	A primitive for representing scenarios of traffic problems with matching procedures using uncertainty models
	Dates and Calendars	600	Prolog	A primitive for interpreting dates using a calendar and types of days (weekends, summer holidays, etc.)
	Demand Structures	900	Prolog	A primitive for representing patterns of traffic demand on a traffic network according to a temporal classification
	Signal Plans	1600	Prolog	A primitive for representing traffic signal plans together with their estimated effect and consistent traffic conditions
	Hydrologic Models	12500	C	A model-based representation of dependencies between variables and procedures for prediction and consistency verification

Figure 6: Some examples of primitives of representation developed in our research group. They are divided into two categories: generic (for general purpose) and domain specific (for specific domains such as traffic control or hydrology).

The examples of primitives shown in this section are divided into two categories: generic, that are general purpose primitives, and domain-specific, that are primitives that are specific for a particular domain (such as hydrology or traffic control). Generic primitives follow a general knowledge representation technique (such as rules, frames, constraints, logic clauses, etc.) with their corresponding inference procedures (e.g., forward and backward chaining, uncertainty management, matching procedures, constraint satisfaction techniques, etc). Once the primitives were developed, they were extremely useful to increase the productivity in the development of both prototypes and final applications.

Some domain-specific primitives were also programmed as a result of the development of particular projects in the domains of traffic control and hydrology. As a fundamental difference with the

generic primitives, they use domain specific languages for domain knowledge representation that are closer to the professionals in particular fields. Here, it was especially important that the explanations were formulated in the domain language, together with an efficient operation. Within this category, there are primitives that use certain efficient adaptations of knowledge-based representations (e.g., a primitive for representing traffic-scenarios that uses fuzzy logic for matching partial descriptions) and primitives that use conventional algorithmic techniques (e.g., a primitive for simulating the behaviour of traffic on a road network). The management of such primitives in these domains has shown a high level of reuse of the resulting applications. For instance, different particular realisations were developed for traffic control for several cities. The highest degree of reuse was achieved in the case of transporting primitives for different road networks in Barcelona and Madrid (the same set of primitives was used in those applications).

In all these cases, the code of primitives was totally reused. The required adaptations for particular applications were carried out by means of two possibilities: (1) by writing the particular domain knowledge base using the declarative language provided by the primitive, and (2) by reconfiguring the global knowledge model where the primitives are integrated. The use of the high level representation language showed that the required effort for reuse was significantly lower than writing new software modules. Domain-specific primitives showed a greater degree of usability given that they present a language closer to the specific fields where they were applied. The management of primitives also showed a good support for maintenance. From the original design of some applications, it was necessary to substitute certain primitives according to new requirements. This is an interesting solution to develop applications that evolve from original prototypes, where simple primitives are initially used, until final applications where some of the initial primitives are substituted by more efficient or specific techniques to achieve the required degree of performance. In final applications, it was also necessary to write *ad hoc* primitives without representation language whose purpose was to perform simple information management (such as data transformation, combination of alternatives, etc.). This type of primitives was

about 30% of the global set of primitives and presented a low degree of reuse because they had a strong dependence with data structures.

The primitives were developed using the C, C++ and Prolog programming languages on Unix workstations. The C and C++ languages were used to achieve a high level of efficiency in certain primitives, although the recent versions of Prolog (that use efficient interpreters based on the Warren Abstract Machine) are quite efficient and give the possibility of good integration in general purpose environments. Presently, we are planning to increase the number of existing primitives with other generic and domain-specific representations. In this future work we will use also other languages (e.g., Java) and, even, other tools (commercial or distribution-free) will be integrated in the library of KSM as primitives.

6 Conclusions

The traditional interest for component reuse in software and knowledge engineering has recently been promoted within the context of reuse across internet, which provides a virtual platform that significantly increases the accessibility and the number of potential users of the available reusable software components. Within this context, it is desirable to have powerful building blocks, adaptable enough to a wide range of potential situations together with a combination scheme natural and intuitive enough to be easily understood by developers. This article presents a proposal of a reusable software component, called primitive of representation, that can be used for building software applications with hybrid architectures including knowledge-based and conventional techniques.

Compared to other approaches about reusable components for building applications, first, the primitive presents a large size, which in principle decreases the effort for building complex architectures. The required adaptability of the primitive to be applied to different domains is achieved by using the domain knowledge base and task specialisations. The use of declarative knowledge representations increases the level of generality (by abstracting the domain knowledge) and provides a flexible solution for adapting primitives to particular domains. The primitive provides a way to be combined following a

knowledge modelling philosophy that makes use of mechanisms for sharing a common terminology (conceptual vocabulary in KSM). The consideration of the primitive as a class, understood in the context of object-oriented development, provides modularity and an efficient solution at implementation level for structuring and sharing the software elements that support the primitive.

Primitives are stored in an open library that includes and organises the collection of existing primitives, but accepts the inclusion of new ones according to the necessity of particular applications. The use of this kind of library avoids the commitment to a particular knowledge representation, allowing the developer to use the most convenient one for each case and to develop architectures that include multiple representations. This is particularly important in real world applications where efficiency must be taken into account.

The primitive of representation was defined in the context of a software environment called KSM that supports the development of applications following a knowledge modelling philosophy. This environment has been used for the development of real world applications in different domains (hydrology, traffic control, intelligent interfaces, etc.) and includes a library of primitives of representation developed and reused in these projects. Our experience shows that the use of such primitives provides an important increase of the productivity in the development of complex applications that integrate both conventional and knowledge-based techniques.

Acknowledgements

We would like to thank Alberto Gómez for his work related to primitives and Sascha Ossowski for his comments in previous versions of this article. Special thanks to the members of the Intelligent Systems Research Group of the Technical University of Madrid for their participation in the development of primitives.

References

1. Benjamins R.: "Problem-solving Methods for Diagnosis". PhD Dissertation. Universiteit van Amsterdam. 1993.
2. Biggerstaff T.J., Perlis A.J. (eds): "Software Reusability". ACM Press. 1989.
3. Booch G.: "Object-oriented software development". IEEE Trans. Software Engineering pp.211-221, Feb. 1986.
4. Bradshaw, J.M., Ford K.M., Adams-Webber J.R., Boose J.H.: "New approaches to constructivist knowledge acquisition tool development" Int. J. Intell. Syst. 8 (2). 1993. Also in "Knowledge Acquisition as Modelling", Ford K.M. and Bradshaw K.M. (eds) Wiley, New York, 1993.
5. Breuker J., Vand de Velde W. (eds): "CommonKADS Library for Expertise Modelling: Reusable Problem Solving Components". IOS Press. 1994.
6. Chandrasekaran B., Johnson T.R, Smith J.W.: "Task Structure Analysis for Knowledge Modelling", Communications of the ACM, 35 (9), 124-137. 1992.
7. Cox B.: "Object-oriented Programming: An Evolutionary Approach". Reading, Mass.: Addison Wesley, 1986.
8. Cuena J., Molina M.: “KSM: An Environment for Knowledge Oriented Design of Applications Using Structured Knowledge Architectures” in “Applications and Impacts. Information Processing’94”, Volume 2. K. Brunnstein y E. Raubold (eds.) Elsevier Science B.V. (North-Holland), IFIP. 1994.
9. Cuena J., Molina M.: "KSM: An Environment for Design of Structured Knowledge Models”. Chapter of the book "Knowledge-based Systems: Advanced Concepts, Techniques and Applications". S. G. Tzafestas (Ed.). Publisher World Scientific Publishing Company. 1997.
10. Cuena J., Ambrosino G., Boero M.: "A General Knowledge-based Architecture for Traffic Control: The KITS Approach". Proc. International Conference on Artificial Intelligence Applications in Transportation Engineering. San Buenaventura, California. June, 1992.

11. Cuena J., Hernández J, Molina M.: "Knowledge Oriented Design of an Application for Real Time Traffic Management: The TRYS System". Proc. 12th European Conference on Artificial Intelligence ECAI 96. Budapest, Hungary, 1996.
12. Gamma E., Vlissides J., Johnson R., Helm R. :”Design Patterns: Elements of Reusable Object Oriented Software”. Addison-Wesley. 1994.
13. Gennari J.H., Tu S.W., Rothenfluh T.E., Musen M.: "Mapping Domains to Methods in Support of Reuse". International Journal of Human-Computer Studies. 41, 399-424. 1994.
14. Gruber T.R.: "A Translation Approach to Portable Ontology Specifications". Knowledge Acquisition, 5. 1993.
15. Hernández, J., Molina, M.: "Advanced Human-Computer Interaction for Decision Support Systems Using Knowledge Modeling Techniques". Proceedings 15th IFIP World Computer Congress, IFIP'98. IT&KNOWS Conference, Information Technology and Knowledge Systems, Viena-Budapest, 1998.
16. Krueger C.W.: “Software Reuse”. ACM Computing Surveys. Vol. 24, N 2, 131-183. Jun 1992.
17. Lewis T., Rosenstein L, Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J., Shumucker K.: “Object Oriented Application Frameworks”. Manning Publications Co. 1995.
18. Macintyre A.: “KREST User Manual 2.5”. Vrije Universiteit Brussel, AI-lab. Brussels. 1993.
19. Marcus S. (ed): "Automatic Knowledge Acquisition for Expert Systems". Kluwer, Boston. 1988.
20. Molina M., Shahar Y.: "Problem-solving Method Reuse and Assembly: From Clinical Monitoring to Traffic Control". Proc. 10th Knowledge Acquisition for Knowledge-based Systems Workshop KAW-96. Banff. Canada. 1996.
21. Molina, M., Hernández, J., Cuena, J.: "A Structure of Problem Solving Methods for Real Time Decision Support in Traffic Control". International Journal of Human Computer Studies.1998 (49)
22. Molina M., Sierra J.L., Serrano J.M.: "A Language to Formalize and to Operationalize Problem Solving Strategies of Structured Knowledge Models". 8th Workshop on Knowledge Engineering: Methods & Languages KEML 98. Karlsruhe, Germany, 1998.

23. Puerta A.R., Tu S.W., Musen M.A.: “Modelling Tasks with Mechanisms”. International Journal of Intelligent Systems, Vol. 8, 1993.

24. Schreiber A. Th., Wielinga B.J., Breuker J.A. (eds): "KADS: A Principled Approach to Knowledge Based System Development". Volume 11 of "Knowledge-Based Systems Book Series", Academic Press, London. 1993.

25. Sierra, JL.,Molina, M.: "Terminological Importation for Adapting Reusable Knowledge Representation Components in the KSM Environment". Workshop on Applications of Ontologies and Problem-Solving-Methods. 13th European Conference on Artificial Intelligence. ECAI'98, Brighton, England. 1998.

26. Volpano D.M., Kieburtz R.B.: "The Templates Approach to Software Reuse". In "Software Reusability" Biggerstaff T.J., Perlis A.J. (eds). ACM Press. 1989.

27. Wielinga B.J., Schreiber A.T., Breuker J.A.: "KADS: A modelling approach to knowledge engineering". Knowledge Acquisition. 4, 5-53. 1992.