A Language to Formalize and to Operationalize Problem Solving Strategies of Structured Knowledge Models

Martin Molina, Jose L. Sierra, Juan M. Serrano

Department of Artificial Intelligence
Technical University of Madrid SPAIN
mmolina@dia.fi.upm.es
http://www.dia.fi.upm.es

Abstract. This paper describes a proposal of a language called *Link* which has been designed to formalize and operationalize problem solving strategies. This language is used within a software environment called *KSM* (Knowledge Structure Manager) which helps developers in formulating and operationalizing structured knowledge models. The paper presents both its syntax and dynamics, and gives examples of well-known problem-solving strategies of reasoning formulated using this language.

1. Introduction

In recent proposals of methodologies and tools within the knowledge engineering field, the model-based knowledge acquisition has been emerged as a solution to guide the development of knowledge based systems. Within this context, the concept of problem-solving method has been established by different authors [McDermott, 88; Chandrasekaran et al., 92; Wielinga et al., 92] with the goal of improving the process of development and the quality of complex architectures of knowledge-based systems. A problem-solving method (PSM) defines a strategy of reasoning on a predefined set of types of knowledge, formulating how the goal of a task can be achieved by executing a collection of sub-tasks.

One of the recent areas of interest in this field is the identification of methods and tools to *formalize* and *operationalize* PSMs. The first goal, formalizing, is to find formal representation languages expressive and intuitive enough to describe the different components of PSMs without ambiguity and well established semantics. The second goal is to provide computational solutions to make operational the PSM (executable on the computer) taking into account issues such as efficiency, portability, etc. In the community of knowledge engineering, different languages have been already proposed to satisfy some of these needs (OMOS, MODEL-K, MOMO, FORKADS, $(ML)^2$, QUIL, KbsSF, etc.). Especially, the issue of formalization has been object of high interest. For instance, the language $(ML)^2$ has been proposed to formalize models of expertise following the KADS methodology [Harmelen, Balder, 92]. From the point of view of operationalization, one of the most significant proposals has been the KARL language [Fensel, et al., 91] (with the recent version NewKARL that is more adapted to PSMs). This language produces an executable version of the PSM but, as the authors of KARL claim, the language is useful to validate and refine partial prototypes of knowledge models but it is not a valid solution to implement final real-world systems. Thus, although operationalization has been provided by some of these languages, it has been considered more from the point of view of a tool for validating a model specification instead of a tool for building the final system.

Therefore, there is still a need of tools for building operational versions for PSMs to be applied during the development and maintenance of complex real-world systems. One of the main difficulties in the design of representation languages for knowledge modeling is to provide at the same time a good level of understanding near to personal intuitions and good computational qualities (e.g., efficiency). In order to satisfy these two needs, our approach to develop operational models is based on the assembly of heterogeneous components. Thus, instead of looking for a universal modelling and operational language capable of providing the best solution to formulate the complete model, we believe that it is more appropriate to use a set of different primitive representation languages, each one specialized into different basic problem-solving tasks with inference procedures that provide the required level of efficiency. Together with these primitives, it is also required a flexible assembly mechanism to build complex models. This mechanism must to be capable of formulating control knowledge to establish strategies of reasoning for the use of simpler components.

What we present in this paper is a proposal in this direction. We have designed a language called *Link* to formulate control knowledge about how to articulate simpler inference steps. This language is one of the modeling tools that the KSM environment provides [Cuena, Molina, 97]. From the point of view of representation, the language includes production rules that allow to dynamically determine the control regime by developing local search spaces for each method. In addition to that, methods can be defined at domain-independent level, making references to generic classes of knowledge, and instances of such classes within domain models can share generic strategies of reasoning by inheritance. On the other hand, from the operational point of view, this language allow developers to build and maintain the problem-solving strategies of knowedge models. We have developed an interpreter for this language within the KSM environment with which we have built different real-world knowledge systems in different domains, providing good levels of efficiency and flexibilty for maintenance.

The paper presents first a brief introduction of the KSM enviroment and describes the use of representation primitives as basic components to build knowledge models. Then, the paper describes the Link language presenting the syntax and the dynamics. Finally, an example that illustrates the use of the language is presented.

2. The KSM Tool

This section briefly presents the KSM tool, in which the Link language is integrated as a language specialized in formulating problem solving strategies. KSM is described here in order to introduce certain modeling concepts (such as the knowledge-area, the task and the method) that are used within the Link language. KSM (Knowledge Structure Manager) (`http://www.isys.dia.fi.upm.es/ksm`) [Molina, 93; Cuena, Molina, 94; Cuena, Molina, 97] is a software environment that supports a methodology for building and maintaining knowledge models (figure 1). The methodology is a useful tool for developers who need to build large and complex knowledge models in real world projects. The environment helps developers in applying the whole methodology in order to build the operational version of the final system and it also assists end-users during the operation and maintenance of existing knowledge models.

Basically, to formulate a knowledge model in KSM, three main perspectives are defined: (1) the knowledge-area perspective, which plays the role of central structure of the model as a structured collection of knowledge bodies, (2) the task perspective, that describes the problem-solving behaviour of the model using tasks-subtasks hierarchies 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 *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. The whole knowledge model is a hierarchical structure of knowledge areas in such a way that there is a top-level area representing the entire model. This area is divided (using the part-of relation) into other more detailed subareas that, in their turn, are 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 using basic software building blocks.

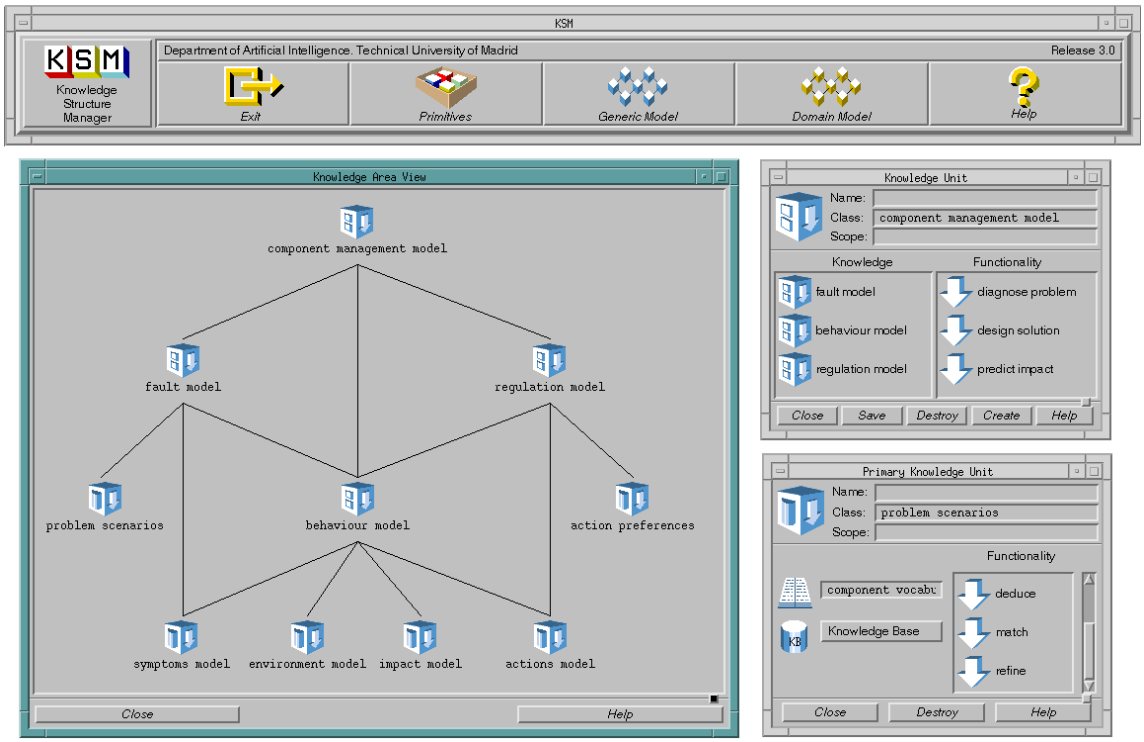

Figure 1: Example screen of the KSM environment.

The task perspective presents a functional description for 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 a set of input data and generates a set of output data as a result of its reasoning. Examples of tasks are: medical diagnosis of a patient, assignment of a set of offices to a group of people, 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. Examples of methods are: establish-and-refine, propose-and-revise, generate-and-test and heuristic classification. In KSM, methods are formulated using the *Link* language which is described in detail in the rest of this paper.

Finally, the vocabulary perspective is formulated by means of a set of components 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 it defines a partial view including 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.

The structure of knowledge-areas, tasks and vocabularies is called *generic* model, given that it is general and reusable. To develop a model for a particular domain the developer creates a quasi-isomorphic structure of knowledge areas specialised in the domain as an instantiation of the general description. For each generic knowledge area there will be one or more domain knowledge areas, following the same relations established by the generic model. The developer particularises the domain structure writing particular knowledge bases (using the set of knowledge acquisition facilities provided by primitives), creates domain conceptual vocabularies and he/she may also redefine at domain level the generic control knowledge defined in methods using the Link language.

The previous model is operationalised by using computational constructs that produce the executable version on the computer. In order to do so, it is necessary either (1) to translate the model into an executable version by applying conventional software engineering methods, or (2) to use high level reusable software components that implement basic problem-solving techniques. In KSM, the second solution is applied by using a particular type of basic component (called *primitive of representation*) together with the interpreter of the Link language. Primitives of representation implement basic primary inference steps, and methods of tasks (that invoke inference steps and subtasks) are implemented using the Link language.

In more detail, a primitive of representation is defined as a generic technique (supported by a reusable software tool) for solving certain classes of problems. Each primitive defines a particular domain representation using a local declarative language *L* together with several inference procedures that provide a problem-solving competence. The language *L* can be different for each type of primitive and it is usually 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 representations, etc. Also other parameterized or conventional representations can be considered (such as a graph-based language that defines nodes and arcs, where arcs include tags indicating their cost). The use of knowledge based representation provides flexibility to adapt this type of components to particular domains. A more detailed description of the structure and use of primitives of representation can be found in [Molina et al., 97].

3. Syntax of the *Link* Language

In general, the strategy of reasoning followed by a method to solve a particular type of tasks can be formulated by a procedural representation which is used to automatically deduce the sequence of calls to subtasks. In general, procedural knowledge may be represented following different approaches that essentially could be divided into: (1) *algorithmic approach*, which follows the traditional definition of algorithms used in conventional languages such as Pascal, Fortran, C, etc., based on control mechanisms like sequences, loops, conditions, etc., (2) *search approach*, that are used when non-deterministic solutions are produced and a systematic search is carried out by using certain strategic knowledge, and (3) *distributed approach*, where there is not a centralised mechanism for controlling the problem-solving strategy, but it emerges as a result of the cooperation of different components. Recently, in operational languages for knowledge modelling, the first approach has been mainly followed (for instance within the KADS approach, the KARL language). Here we follow the second approach with the Link language (using production rules), given that it easily includes the first one and, typically, knowledge models require search methods to carry out tasks. The third approach is also an interesting paradigm to be used in distributed knowledge models for multi-agents organizations.

This section describes in detail the Link language which is specialized in formulating the control knowledge of a method. The method establishes how to use a set of subtasks to carry out a global task. Basically, using the Link language, the method formulation includes on the one hand, the data connection among subtasks and, on the other hand, the execution order of subtasks (this separation of data flow and control regime to formulate procedures has been already used within different methodologies of software engineering). The following subsection describes in more detail this division and then, the rest of the subsections describe the syntax and semantics of the language.

3.1 General Structure of a Method

The view of each particular subtask to be used by a method is divided into two levels: the data level and the control level. The *data level* shows input data and output data. For instance, the task of classification receives as input measures and generates as output a category. Likewise, the task of medical diagnosis receives as inputs symptoms and the case history of a patient and generates as output a disease and a therapy. On the other hand, the *control level* offers a higher level view of the tasks showing an external view about how the task works. This level includes two elements of information: control parameters and control states. A control parameter selects how the task must work when it accepts different execution modes. For instance, a classification task classifies into categories measures received as input data according to a similarity degree. The similarity degree may be considered as a control parameter. In the context of a real time system, other examples are the maximum reasoning time or the maximum number of answers, when more than one could be expected.

```
METHOD Method Name

ARGUMENTS
    [INPUT   List of Inputs]
    [OUTPUT  List of Outputs]

DATA FLOW
    Data connection among subtasks

[CONTROL TASKS
    Data connection among control tasks]

CONTROL FLOW
    Rules describing the control flow

[PARAMETERS
    Default values of parameters]
```

Figure 2: General format of a method formulated using the Link language.

Control states, in their turn, indicate the degree of success or failure of the task after the reasoning. For instance, the medical diagnosis task may have as possible control states: insuficient data (when there are not enough data to give a result), healthy patient (when the patient does not have any disease), no therapy proposed (when the patient has a disease but the system does not find out a therapy) or therapy proposed (when the patient has a disease and the system finds out a therapy). Note that control states do not provide the actual results of the task, but they give an abstract information about how the tasks worked. In summary, at the control level of a task, control parameters selecting modes are received as input and control states informing about the reasoning are generated as output.

According to this division, the formulation of a method using Link language includes several sections (Fig. 2). After the name of the method, the first section, that is called

arguments, indicates the global inputs and outputs of the method. Then, there are two main sections: the data flow and the control flow. The *data flow section* describes the data connection of subtasks at the data level, indicating how some outputs of a task are inputs of other tasks. The *control flow section* describes the execution order of subtasks using control rules that include control states and parameters. In addition, there are also other two optional sections: the control tasks and the parameters. The *control tasks section* allows the developer to include tasks that decide the execution of other tasks, and the *parameters section* is used to write default values for control parameters. Next paragraphs describe in more detail these sections.

3.2. The Data Flow Section

The data flow section describes the data connection of subtasks showing how some outputs of a subtask are inputs of other subtasks. The developer here writes input/output specifications of subtasks using what is called flow. A *flow* identifies a dynamic collection of data, for instance the symptoms of a patient in medical diagnosis or the resulting design of an elevator. For a given method, there are several names of variables identifying the different flows that will be used to connect subtasks. These variables represent plain flows, i.e. flows whose internal organization is not known at this level. In addition, there are complex flows, called flow expresions, that are composition of others. A flow expression can be one of the following:

- empty flow: `nil`
- constant value: `constant S`
- plain flow: `P`
- list of flows: `[F1, F2 , ... , Fn]`
- conjunctive flow: `F1 & F2 & ... & Fn`
- disjunctive flow: `F1 | F2 | ... | Fn`
- labelled flow: `label(F, label)`
- selective flow: `select(F, selection-criteria)`

Where the meanings of letters are: S is a symbol or a number, P is a plain flow, F and F_i are flow expressions. The conjunctive flow represents the concatenation of flows. The disjunctive flow means the first flow (from left to right) with a value different from empty flow. The labelled flow is a flow with a *label* (a label is a number or a symbol). The label can be explicitly given or it can be the content of a flow expression.The selective flow selects a part of a flow F taking into account a *selection-criteria.* This criteria is written as a list of selectors where each selector can be either a number of order or a label, and the value can be either explicitly written or the content of a flow expression.

```
(Knowledge Area Name) Task Identifier

  [INPUT
         { [Mode] Flow Expresion } + ]

  [OUTPUT
         [Mode]  { Flow Identifier } + ]
```

Figure 3: Format of an input/output specification. The meaning of the symbols used here are the following: [X] means X is optional and {X }+ means that X can be one or more.

Typically, flows are divided into input flows, output flows and intermediate flows. Input and output flows identify the global inputs and outputs of the method and connect them to intputs and outputs of subtasks. An intermediate flow identifies a flow connecting the output of a subtask to the input to another subtask. Note that flows do not identify knowledge bases given that knowledge is considered static and it is viewed permanently

associated to tasks. Thus, the whole data flow section is viewed as a set of flows and subtasks transforming input flows into output flows. To formulate this inference structure, the developer writes a collection input/output specifications (i/o specifications). Each i/o specification includes, first, the subtask name as a pair made of the knowledge area name and the subtask identifier. Second, it is defined the input of the subtask. Basically, the input is defined with names identifying flows (plain or complex flows). Each input flow accepts a *mode* that may be the default mode or the one-of mode. The default mode works as follows. When the subtask is going to start the reasoning, considering that the input flow is a list of elements, the subtask gets as input all the elements of the flow. On the other hand, the *one-of* mode considers just one single element of the list each time. The particular element is selected depending on the number of times that the same subtask has been executed (the first time, the first element is selected, the second time the second element, and so on). This second mode is useful to formulate non-deterministic search methods using Link.

Finally, it is defined the output of the task. The output is defined with a list of single identifiers giving names to the output flows. In Link language, in general, subtasks are considered non-deterministic processes. This means that as a result of a reasoning, a task may generate not just one result, but several ones. For instance, in the context of medical diagnosis, the task may deduce as a result several diseases and several therapies for the same symptoms. Thus, when tasks are going to be connected in the data flow section this possibility must be taken into account. This is managed with two output modes. Modes select whether the whole set of outputs must be generated one by one element considering that there is a non-deterministic result (this is the default mode) or, on the contrary, they select whether it must generate all the outputs at once as a list of single elements for each output flow, which is called the *all* mode.

In summary, the data flow section describes the static connection among subtasks explaining how they are comunicated. However the dynamic behaviour of the method is not defined here. A given data flow with several subtasks may be later executed in different ways according to different execution order of subtasks. Therefore, it is necessary to complement this view with another one that allows the system to dynamically deduce the execution order of subasks. This information is given in the control flow section.

3.3 The Control Flow Section

The purpose of the control flow section is to provide a formal description of a control regime that determines the execution order of subtasks. In general, this sequence has to be deduced dynamically by using control knowledge formulated using production rules. The use of rule-based languages to model control has been already proposed in first generation knowledge based systems. The representation used here has been partially inspired in the proposal of Chandrasekaran [Chandrasekaran et al., 92] to dynamically determine the control regime using a Problem Space Computation Model (PSCM) [Johnson, 91]. One of the advantages of this representation is that it makes possible to define local search spaces considering the non-deterministic behaviour of subtasks. At the same time, the representation is simple enough to be used easily due to this language tries to be not a complex programming language but, on the contrary, it was designed to serve as an easy description to formulate procedural knowledge (a method usually has a small number of rules, less than 10). Production rules provide a intuitive representation and flexibility for maintenance.

In more detail the representation of rules is the following: (1) the left hand side includes a set of conditions about intermediate *states of task execution*, and (2) the right hand side includes a sequence of *specifications of task execution* . Each of one of the first elements (states of task executions) is a triplet $<K,T,S>$ where K is a knowledge area, T is a task

identifier and *S* is a control state. This means that the result of the execution of the task *T* of the area *K* has generated the control state *S*. The value of *S* is control information such as successful execution or failure of different types, which may be used as premises to trigger other production rules. The representation of the elements in the right hand side (specification of task execution) is another triplet *<K,T,M>*, where *K* is a knowledge area, *T* a task and *M* an execution mode. This representation means that the task *T* of the knowledge unit *K* must be executed with the execution mode *M*. The execution mode expresses the conditions limiting the search such as: maximum number of answers allowed, threshold for matching degree in a primary unit using frame representation, time-out, etc. For instance, the following rule is an example of this representation:

```
<K: validity, T: establish, S: established>,
<K: taxonomy, T: refine, S: intermediate>
->
<K: taxonomy, T: refine, M: maximum 3 answers>
<K: validity, T: establish, M: null>.
```

However, in Link language, this representation has been modified to include some syntactic improvements (for instance, the name of the knowledge area is written in brackets before the name of the task). For example, the above rule is expressed:

```
(validity) establish IS established
(taxonomy) refine IS intermediate
->
(taxonomy) refine MODE maximum answers = 3
(validity) establish.
```

This representation must also include references to the beginning and the end of the execution to indicate the first set of actions to be done and when it is considered that the process has reached a solution of the problem. The beginning of the execution is referred as a state of the execution (to be included in the left hand side of the rules) and it is written with the reserved word `START`. The end of the execution is considered as an action (to be included in the right hand side of the rules). It is written with the reserved word `END` and, optionally, can be followed by a symbol that expresses the control state that has been reached. A complete example of a method formulation for hierarchical classification using the establish-and-refine strategy is the following:

```
METHOD establish and refine
ARGUMENTS
       INPUT description
       OUTPUT category
DATA FLOW
      (validity) establish
                INPUT description, hypothesis
                OUTPUT category
      (taxonomy) refine
                INPUT category
                OUTPUT hypothesis
CONTROL FLOW
      START
      -> (taxonomy) refine, MODE maximum answers=3,
         (validity) establish.

      (validity) establish IS established,
      (taxonomy) refine IS intermediate hypothesis
      -> (taxonomy) refine MODE maximum answers=3,
         (validity) establish.

      (validity) establish IS established,
      (taxonomy) refine IS final hypothesis
      -> END.
```

In this example two subtasks are considered, establish (that belongs to the knowledge area called validity) and refine (that belongs to the knowledge area taxonomy). In the control flow section, the first rule indicates that the first step is to call the task refine (which at this moment generates the top-level hypothesis given that its input is unknown) and, then, to call the task establish in order to check the validity of the hypothesis. The second rule is applied several times generating each time more specific hypotheses. The last rule defines when the current hypothesis is considered as a solution.

3.4 The Reflective Level

In addition to the previous representation, the Link language includes also the possibility of formulating a more complex control mechanism by using what is called *control tasks*. These tasks are included in the *control task* section in the same way that is formulated in the *data flow* section. The main difference is that control tasks produce as output, instead of only flows (at data level), tasks to be executed, formulated as task specifications. These task specifications can be included in the right hand side of control rules to determine when they must be executed. This is done by using a special action called EXECUTE, using the following syntax:

```
EXECUTE task-specification
```

Here, *task-specification* is the name of a variable that is output of a control task and dynamically contains as value the specification of a task execution, with the previous presented syntax (i.e., a triplet<*K, T, M*>). In addition, a control task can get as input the execution state of another tasks. This is formulated in the form:

```
STATE OF (knowledge-area) task
```

In this way, it is possible to build models that include specific knowledge bases containing criteria to select the next tasks according to the execution of previous tasks. These solutions provide the required freedom to use the most appropriate knowledge representation and inference for different control strategies. For instance, in order to illustrate this idea, consider the following example:

```
METHOD external scheduling
ARGUMENTS
     INPUT  problem description
     OUTPUT solution
DATA FLOW
     (area 1) task 1 INPUT problem description
                     OUTPUT solution, problem description
     (area 2) task 2 INPUT problem description
                     OUTPUT solution, problem description
     (area 3) task 3 INPUT problem description
                     OUTPUT solution, problem description
CONTROL TASKS
     (scheduling knowledge) select next task
            INPUT  problem description, STATE OF (area 1) task 1,
                   STATE OF (area 2) task 2, STATE OF (area 3) task 3
            OUTPUT task
CONTROL RULES
     START
     -> (scheduling knowledge) select next task.

     (scheduling knowledge) select next task IS no more tasks
     -> END.

     (scheduling knowledge) select next task IS task selected
     -> EXECUTE task,
        (scheduling knowledge) select next task.
```

This example shows how to formulate a strategy of control in which one of three tasks is executed according to an external scheduling knowledge. The knowledge area called scheduling knowledge includes a set of criteria formulated, for instance, using a declarative symbolic representation (rules, frames, etc.) to select the appropriate next task to be executed according to the current state of the problem description.

4. Dynamics of Link Methods

The execution of a method formulated using the previous language basically follows the control established by the set of control rules. Each rule includes in the right hand side a sequence of subtasks execution. In the simplest case, when this sequence is previously known and it is permanent, there is just one rule with the explicit order at the right hand side. Figure 4 shows a graphical view of the execution of this sequence considering a simple rule such as: `START -> (area 1) task 1, (area 2) task 2.`

Figure 4: This figure shows the graphical representation of the simplest case of the execution of a method. It is a linear execution of a sequence of two tasks, task 1 and task 2 (in order to simplify the figure, references about areas are not written).

In the figure, the square under the word *start* is the first task that is executed (in this case task-1) and it is executed considering as input data what is defined in the data flow section (they are not explicit in this scheme). After the execution of this task the output is data-1. The following action is to execute task-2 using as input data what is defined in the data flow section and considering that task-1 generated data-1. Finally, task-2 produces as output data-2. This example illustrates the simplest case where there is a linear execution of two tasks showing explicitly their outputs. However, the use of control rules allows to define more complex situations. First, it is possible to represent control structures such as if-then, loops, repeat, etc. In order to do so, control states are used. For instance, consider the set of rules:

```
START -> (area 1) task 1.
(area 1) task 1 IS a -> (area 1) task 1.
(area 1) task 1 IS b -> (area 2) task 2, END.
```

This set of rules defines a loop executing task-1 until the execution of task-1 produces the state called *b*, when task-2 is executed. For instance, a particular execution with this scheme could be what is presented in figure 5. Notice that control states are written on the left of arcs and rules are on the right. In the picture, at the beginning, task-1 is executed, following the first rule. After that, the state called *a* is reached. Then task-1 is executed again (following the second rule) and this time the state called *b* is reached, so finally task-2 is executed (following the third rule) and the process finishes.

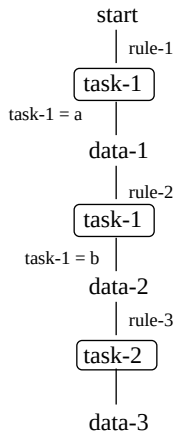

Figure 5: Graphical representation of the execution of a loop where task-1 is executed twice.

In addition, in Link language is useful to define a more powerful execution with a non-linear sequence. This is possible due to two reasons: on the one hand, for a given state more than one rule may be triggered and, on the other hand, in general a given subtask may generate more than one result (e.g, a diagnosis subtask can generate two different possible causes for the same set of symptoms). For instance, figure 6 illustrates this situation. In this case, task-1 is executed after the initial situation using the rule-1. The execution of task-1 generates two different options data-1 or data-2, so there is an arc for each option. Following the data-1 option, rule-2 is applied with which task-2 is executed producing data-3. However, after this step it is not possible to continue applying more rules (an impasse is reached) so it is necessary to backtrack following another option. Now, the data-2 option is followed. In this case there are two possible rules to be applied, rule-3 and rule-4. Aplying rule-3 means to execute task-3 and to finish. Likewise, applying rule-4 means to execute task-4 and to finish. In summary, there are two possible sequences to reach a solution: one is {task-1, task-3} and the other is {task-1,task-4}.

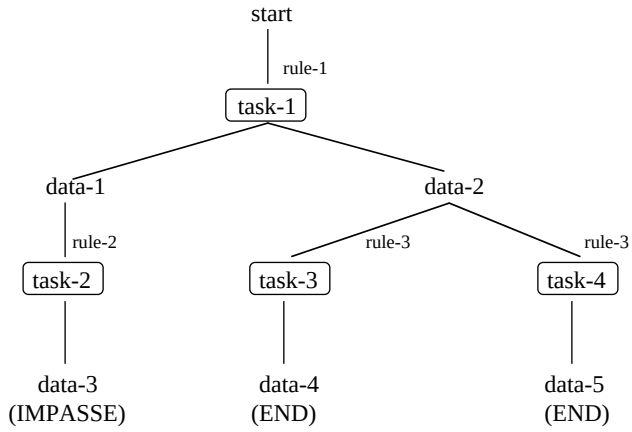

Figure 6: Graphical representation of the execution of a method where there is a non linear reasoning.

This possibility of non-linear executions is a poweful technique that allows the developer to define more easily problem-solving strategies where there are search procedures. The developer can also modify the search control strategy using some tools provided by Link: input modes (one-of or set), output mode (all, one-each-time), and search parameters (such as maximun number of outputs and time-out). According to this, Link develops a search space for the execution of a particular problem-solving method. The particular way in which the Link interpreter develops the tree depends on the number of outputs that have to be

generated. If the method has to produce just one output, the tree is developed following a depth-first search strategy until the first *end-leaf* is reached (a leaf of the tree where the END action has been reached). If the method has to produce two outputs, the tree is developed until the second end-leaf, and so on. If there is not limited number of outputs, the interpreter develops the complete tree. A complete example of an execution corresponding to the establish-and-refine problem solving method which was presented in Link language previously is presented in figure 7.

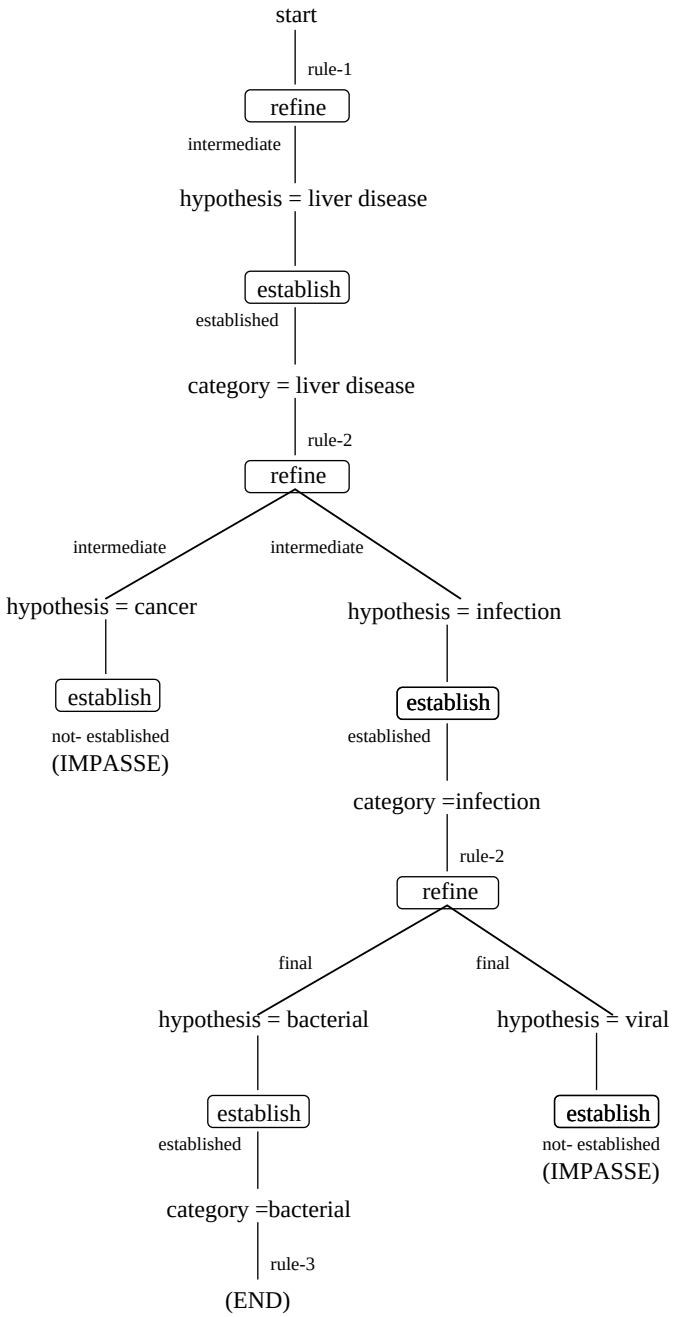

Figure 7: Example of the execution of a method where there is a non linear reasoning (taken from [Chandrasekaran et al., 92] and adapted to the Link execution).

At run time, the method of each task develops a local search space in such a way that when a subtask is executed a new local search space is developed for the method associated to the subtask. Thus, the execution of methods in general dynamically develops a tree of search

spaces, corresponding each space to a method. The leaves of such a tree corresponds to a method that executes subtasks associated to primary areas. In addition to this, usually, methods are described in domain-independent terms. Using the Link language, this means that the method is written using tasks that are associated to *generic* knowledge areas (classes of knowledge). Then, a domain model includes *domain* knowledge areas that are instances of such classes. Therefore, when a task of a domain knowledge area is executed, the Link interpreter applies inheritance from the upper level knowledge areas to inherite the corresponding generic method. This can be seen as a process of instantiation of the method that dynamically substitutes the names of classes of knowledge areas with names of instances of such classes before the method is going to be executed.

5. Example: The Propose-Critique-Modify Method

This section illustrates the use of the Link language to formalize a problem-solving method, using as example the *propose-critique-modify* method for design tasks [Chandrasekaran, 90]. The section starts with the task-method-subtask structure of the method as it was proposed by their authors, and it is described first how it is modelled using KSM components (knowledge-areas and tasks). Then, two Link methods are presented corresponding to the KSM model. The first one includes the general strategy or reasoning of the method. The second one shows how it is possible to write dynamic selection of methods using Link, by using control tasks and knowledge areas at reflective level.

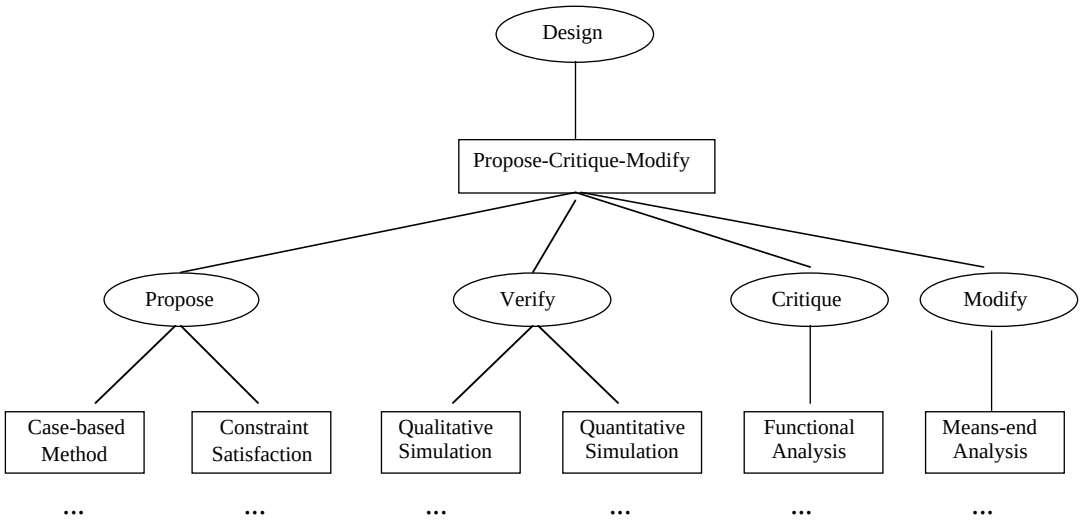

Figure 8: Task-method-subtask structure for the design task using the propose-critique-modify method.

Figure 8 shows a simplified version of the task structure proposed by Chandrasekaran for the design task [Chandrasekaran, 90]. The method, called propose-critique-modify, divides the top level task into four subtasks: propose, verify, critique and modify. The propose subtask generates an initial design proposal whose consistency is checked by the verify subtask. If the proposal does not satisfy the design constraints, it is analyzed (by the critique subtask) and corrected (by the modify subtask) starting again another cycle until no violations are found. In this example, we have considered that some of the subtasks have different alternative methods, that can be dynamically selected during the reasoning according to the particular characteristics of the case. For instance, the propose subtask can be carried out by two different methods: the case-based method that use a set of predefined design cases, and the constraint satisfaction method that proposes an initial design by applying constraint satisfaction procedure. Likewise, the verification of a design proposal, can be achieved through qualitative or quantitative simulation.

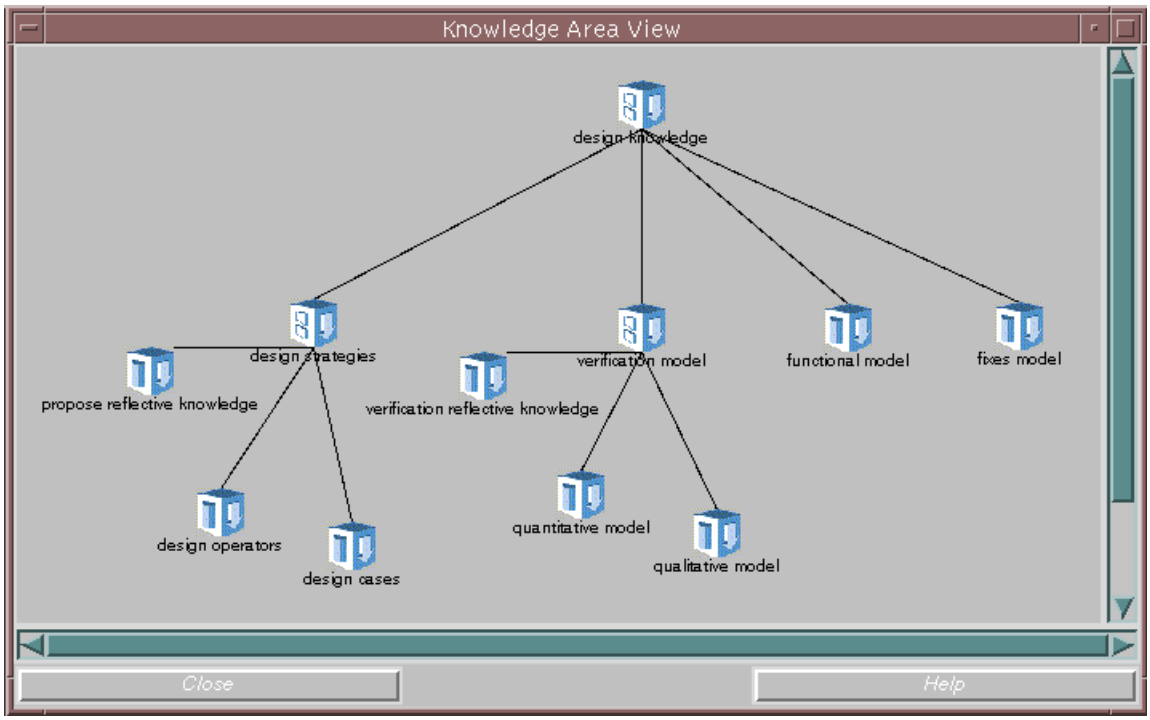

Figure 9: KSM window showing the knowledge-area structure for the design problem

This task structure analysis can be operationalized, following the KSM-based methodology, with the knowledge-area structure shown in figure 9. The top level knowledge area represents the whole knowledge used for design. This knowledge is decomposed in four main areas: knowledge about design strategies, the verification model, a functional model of the artifact, and knowledge about ways of fixing a design proposal that present inconsistency. Concerning the knowledge about design strategies, knowledge about design cases and design operators are the kinds of knowledge used for the proposal of designs. The verification model is decomposed into two areas, qualitative and quantitative models, to perform two different types of simulation.

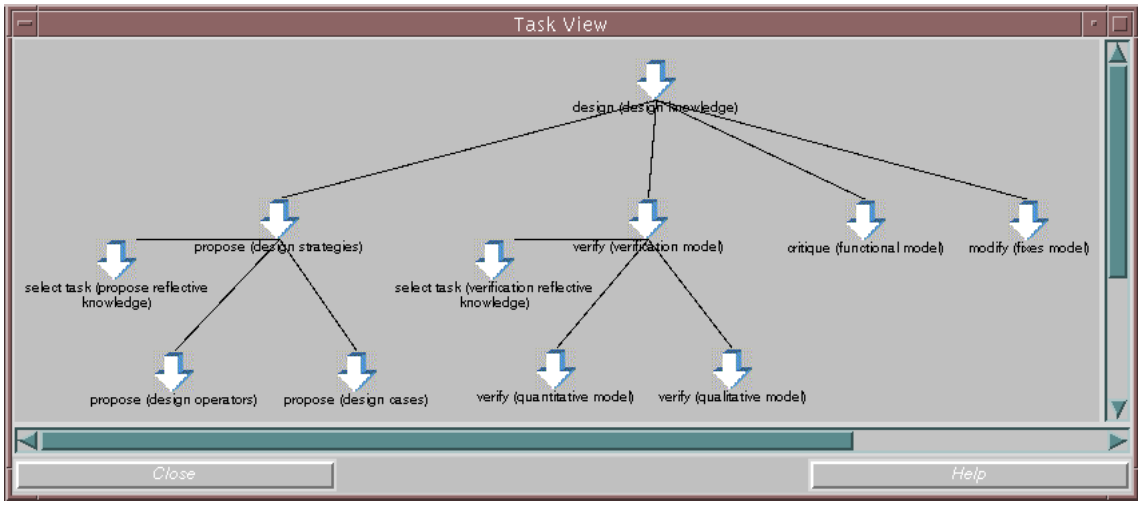

Figure 10: KSM window showing the task decomposition for the design task

The two knowledge areas about design strategies and verification model include also a reflective knowledge area that contains the criteria to dynamically select the appropriate subtask according to the characteristics of the problem. For instance, time or memory constraints in the problem solving process, degree of detail or optimality in the design solution, etc. In this way, the proposal of design solutions could make use of knowledge about design cases, following a case-based reasoning method, knowledge about design operators, by means of a constraint-satisfaction strategy, or even both kind of knowledge, providing various alternative design solutions (besides the alternatives that could be derived from the application of each method independently). Accordingly, a reflective knowledge area about verification lets determine dynamically the more suitable strategy to verify the design solution, using a qualitative or quantitative model of the designed artifacts.

```
METHOD propose critique modify

   ARGUMENTS
     INPUT   specification
     OUTPUT  design solution

   DATAFLOW

    (design strategies) propose
             INPUT  specification
             OUTPUT design solution

    (verification model) verify
             INPUT  design solution
             OUTPUT violations

    (functional model) critique
             INPUT  violations, design solution
             OUTPUT causes

    (fixes model) modify
             INPUT  causes, violations, design solution
             OUTPUT design solution

   CONTROL RULES

     START
      -> (design strategies) propose,
          (verification model) verify.

     (verification model) verify IS violation found
      -> (functional model) critique,
         (fixes model) modify,
         (verification model) verify.

     (verification model) verify IS correct design
       ->  END.
```

Figure 11: Link formulation for the method propose-critique-modify

The decomposition of the design task, associated to the design knowledge area is shown in figure 10 (the knowledge area associated to each task appears in parentheses). The Link methods of each task specifies the control knowledge that dynamically determines their execution order. In this hierarchy, for instance, the *propose* task is decomposed into two versions: the *propose* task of the *design operators* knowledge area and the *propose* task of the *design cases* knowledge area, which follow a constraint-satisfaction and a case-based reasoning methods respectively. The reflective task selects the most appropriate task (or tasks) given the characteristics of the current problem .

The Link methods of the *design* task (the top level task), and the *propose* task of the *design strategies* knowledge area are described in the following paragraphs. Figure 11 shows the Link formulation of the *propose-critique-modify* method. The argument part shows the specification of the design problem as its input, and the desing solution as its output. The dataflow section includes the tasks in which the method is decomposed, and the input-ouput relations between them. In this case, the *propose* task receives the same input as the design task. The output is the design solution that will be input of the *verify* task. This task produces as output the violations of the specification that could be derived from the current solution. The *critique* task gets as inputs these violations, as well as the current design solution, and provides the causes that justify the violations. The *modify* task provides a new design solution, based on the previous design solution, the violations and the associated causes. The control part of the Link method includes rules to deduce the execution order of the subtasks. In this case, the first rule (that is initially triggered given that it includes in the left hand side the START word) says that initially the procedure starts with the propose task, followed by the corresponding verification. In case that a violation is found (second rule), the design solution will be analyzed (critique task), modified accordingly (modify task) and it will be verified again (verify task). The process finishes when the output control state of the verify task indicates that a correct design has been found (third rule).

```
METHOD propose by dynamic selection

  ARGUMENTS
     INPUT  specification
    OUTPUT  design solution

  DATAFLOW

    (design cases) propose
            INPUT  specification
            OUTPUT design solution

    (design operators) propose
            INPUT  specification
            OUTPUT design solution

  CONTROL TASKS

    (propose reflective knowledge) select task
            INPUT  specification
            OUTPUT selected task

  CONTROL RULES

     START
      -> (propose reflective knowledge) select task,
          EXECUTE selected task,
          END STATE OF selected task.
```

Figure 12: Link formulation of a method that dynamically selects the most appropriate method to perform the *propose* task.

On the other hand, the *propose* method of the *design strategies* knowledge area (figure 12), shows the utility of the reflective capabilities of the link language to implement the dynamic selection of methods. The dataflow section includes the alternative tasks (that internally use different methods) for the propose task. The select task defined in the *control task* section of the method gets as input the specification of the design problem and produces as output the most appropriate task. As it is specified in the control section, first, it is executed the selection task, followed by the task selected as output. The method finishes with the same output state of the executed subtask.

6. Conclusions

The Link language helps to formalize and operationalize problem solving strategies for structured knowledge models. The language has been defined within a knowledge modelling philosophy that is supported by the KSM environment, in which knowledge models are viewed as a structured collection of knowledge areas (at generic and domain levels) that include hierarchies of task-subtasks. The role of the Link language within this environment is to allow to formulate the problem-solving knowledge associated to each task, that allow to dynamically deduce how to execute the set of subtasks.

In summary, the Link language follows a representation divided into two main parts: the data flow and the control flow. The data flow expresses how certain outputs of subtasks are input of other subtasks. The control flow includes knowledge (represented with production rules) that allow to deduce the execution order of subtasks. This representation allows to easily represent search procedures considering a non-deterministic execution of subtasks. An interpreter of the Link language was implemented (within the KSM environment) that executes methods developing a tree of local search spaces and applies inheritance to use generic methods within domain models.

This language has been used to develop real-world applications in different domains, implementing a wide range of different types of control strategies: simple sequences of tasks, typical strategies of problem-solving methods (propose and revise, establish and refine, etc.), and also reflective architectures that require external scheduling knowledge to establish the control regime. In all these cases, the Link interpreter has provided a good level of efficiency together with a simple representation that facilitates the understanding and maintenance of the problem strategies. Presently, we are working on two extensions of the Link language: an explanation generator based on the declarative representation used by the language, and a graphical version of the language (the Visual Link).

6. References

[Chandrasekaran, 90] Chandrasekaran B.: "Design Problem Solving: A Task Analysis". AI Magazine 11, 4, 59-71. 1990.

[Chandrasekaran et al., 92] Chandrasekaran B.. Johnson T., Smith J.: "Task-structure Analysis for Knowledge Modeling". Communications of the ACM, 35:124-137. 1992.

[Cuena, Molina, 94] 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 and E. Raubold (eds.). Elsevier Science B.V. (North-Holland). 1994.

[Cuena, Molina, 97] 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.), World Scientific Publishing Company. 1997.

[Fensel, et al., 91] Fensel D., Angele J., Landes D.: "KARL: A Knowledge Acquisition and Representation Language". Proc. Exper Systems and their Applications. Avignon, 1991.

[Harmelen, Balder, 92] Harmelen F.V., Balder J.: "$(ML)^2$: A Formal Language for KADS Models of Expertise". Knowledge Acquisition, 4. 1992.

[Johnson, 91] Johnson T.: "Generic Tasks in the Problem-Space Paradigm: Building Flexible Knowledge Systems while Using Task-Level Constraints". PhD thesis. The Ohio State University, Ohio. 1991.

[McDermott, 88] McDermott J.: "Preliminar Steps Towards a Taxonomy of Problem-solving Methods". In Marcus S. (ed) "Automated Knowledge Acquisition for Expert Systems". Boston, Kluwer. 1988.

[Molina, 93] Molina M.: "Desarrollo de Aplicaciones a Nivel Cognitivo Mediante Entornos de Conocimiento Estructurado". PhD Thesis. Technical University of Madrid. 1993.

[Molina et al., 97] Molina M., Gómez Alberto, Sierra J.L.: "Reusable Components for Building Conventional and Knowledge-Based Systems: The KSM Approach". Proc. 9th International Conference on Software Engineering and Knowledge Engineering, SEKE´97. Madrid. June, 1997.

[Wielinga et al., 92] Wielinga B.J., Schreiber A.T., Breuker J.A.: "KADS: A Modelling Approach to Knowledge Engineering". Knowledge Acquisition, 4(1):5-53. Special issue "The KADS Approach to Knowledge Engineering. 1992.