MODEL: An Enhanced Term Classification Language for Describing Knowledge Level Models

Mihai Barbuceanu

Expert Systems Laboratory
Research Institute for Informatics
8-10 Averescu Avenue,
71316 Bucharest
ROMANIA

Abstract: Knowledge-based problem-solving comprises two processes, modelling and programming. Current AI languages and environments support only the programming process. Next generation languages and environments will have to support the intellectually challenging modelling component in the first place. On the road toward knowledge level modeling environments able to assist the full range of modelling activities involved in problem-solving, the first goal to be achieved is the design of a language for representing problem solving models at the knowledge level. This paper describes MODEL, an enhanced term classification language in the KLONE family which extends classification technology to allow the description of the domain, inference, task and strategic levels knowledge based problem-solving models are composed of. The paper gives a detailed account of the language and illustrates it with an encompassing analysis of the problem-solving and knowledge acquisition components of SALT, a well-known generic problem-solving model for constructive tasks.

Keywords: knowledge acquisition, knowledge modelling, term classification languages, generic problem-solving models, knowledge processing mechanisms, ontological analysis, KADS.

1. Introduction

Most people would probably agree that computer based problem-solving has two major components. The first is a modelling stage in which the goals, the required knowledge and the mechanisms which bring it to bear are identified and articulated in a model of the problem-solving process. The second is the programming process which encodes these elements as programs and data structures to be fed to a computer. In the realm of expert systems these stages have been termed knowledge level analysis and symbol level encoding [Newell 81].

According to this view, modelling is the "intellectual", creative side while programming is a tiresome and routine activity. In consequence, most people would also agree that automating programming as much as possible while supporting modelling through effective methods and tools is a worthwhile research goal for both software and knowledge engineering.

The current picture is different from such an ideal. Existing methods and tools have been developed almost exclusively for the programming side. We can count here many representation and inference methods supported by programming languages, shells or environments. Methods for knowledge acquisition and expert system construction at the knowledge level however, have just begun to be studied. Relevant examples include KADS [Breuker and Wielinga 87], role-limiting methods [McDermott 88], generic task [Chandrasekaran 87], inference structures [Clancey 85]. Very few tools supporting them exist but more are under research and development [Klinker et al 90].

Given this state of affairs, we have embarked on a research project aimed at constructing a new brand of knowledge engineering environment called "knowledge level modelling environment" (KLME). Unlike existing programming environments like KEE [Fickes and Kehler 85] or KnowledgeCraft [Wright and Fox 84] which provide exclusively symbol level languages and tools, KLME-s are devoted to supporting the whole range of knowledge level modelling activities. The programming activity is only one of

the problems addressed in a KLME. The most important component of a KLME is a knowledge modelling language able to describe knowledge level problem solving models. We have designed and implemented such a language as a major extension of terminological languages in the KLONE family. Its presentation is the focus of this paper.

Specifically, Section 2 of the paper describes the context of our research by reviewing a number of important issues in KL modelling, describing the structure of a KL model and presenting the functional architecture of the KLME we are building. Section 3 presents the MODEL language in detail. It discusses the terminological, assertional and refinement services of the language as well as the extensions we brought to make the language a substrate, or carrier, for the tools of a KLME. Section 4 attempts to prove that the language can be used for formalizing KL problem-solving models by providing a detailed analysis of the SALT [Marcus and McDermott 89] problem-solving and knowledge acquisition mechanisms. The analysis is carried out according to the KADS layered approach, showing how inference, task and strategic knowledge can be described in the proposed formalism.

2. Issues in KL Modelling

To place the MODEL language in a proper perspective we first discuss three important issues for knowledge level modelling: what are KL problem-solving models and how are they structured, why is it important to formalize these models and what services should a KLME provide.

2.1 What's in a Model?

A problem-solving model embodies a specific problem-solving method suitable to a specific problem type. This method defines the different knowledge types playing relevant roles in problem-solving. The method identifies generic and instantiated concepts and behaviours. The method should be enough abstract to warrant its applicability to problems of a given type and enough specific to rely on well-delimited knowledge types and roles.

Recently, several modelling styles have been proposed making specific distinctions along the above lines. KADS proposes a layered framework distinguishing among domain, inference, task and strategy layers. Role limiting methods [McDermott 88] build models using a few carefully selected knowledge roles identified in a problem class. Generic tasks [Chandrasekaran 87] build models from functional units packaging specific representation and control mechanisms. Program components [Klinker et al 90] implement the role-limiting approach by constructing and assembling "usable and reusable" knowledge processing mechanisms according to a perceived decomposition of the application. Finally, components of expertise [Steels 90] suggest a modular framework for system construction stressing pragmatic constraints on the task. All these approaches provide a notion of separately described components which are assembled in a problem solving-system.

2.2 Formalized Models

The formalization of problem-solving models has drawn only limited attention up to date. Nevertheless, it is a very important issue, some of reasons being the following.

1. Formalization makes models clear and understandable due to the declarative semantics it imposes. We believe that procedural semantics of the "look in the interpreter's code" type is unacceptable for describing KL models. On the other hand, efficient heuristic implementations of models must be accommodated.

2. Formalization makes it possible to consistently create new models by modifying, extending and combining existing ones.

3. Formalization makes it possible to develop life-cycle support tools applicable to all models expressible in the formal modelling language. Such tools provide essential support for model-driven knowledge acquisition, model maintenance, model integration and combination, model validation and verification, model explanation, etc.

4. Formalization helps automate the shell generation process, that is the production of an efficient programming shell supporting knowledge acquisition and problem solving according to a given model.

5. Formalizations which are also operational (simulable) make knowledge level debugging possible and in general help ensuring correctness in the initial phases of modelling.

If formalization is desirable, what formalisms are appropriate for KL modelling? Some of the solutions proposed up to now are:

(a) Procedural encoding of components and some network formalism for describing assemblies, as in the program components approach of [Klinker et al 90], object-oriented encodings as made for the KADS method by languages like MODEL-K [Karbach et al 91], OMOS [Linster 92] or more recently MOMO [Walther et al 92].

(b) Logic oriented languages such as algebraic specifications in ontological analysis [Alexander et al 86], order sorted logic and dynamic logic as in $(ML)^2$ [Akkermans et al 90, vanHarmelen et al 92], combinations of logic and algebraic specifications as in VITAL-CML [Jonker and Spee 92], first order logic with procedural mechanisms [Wetter 90] or Horn logic as in KARL [Fensel, Angele and Landes 90].

(c) Well-defined representation languages like terminological languages used by [Gaines 90], [Abrett and Burstein 88] and [Barbuceanu 91] or more hybrid frame languages like in [Skuce, Shenkang and Beauville 89].

Our choice - an enhanced terminological language-will be explained and motivated in Section 3.

2.3 Toward a Solution: Knowledge Level Modelling Environments

Given the broad research programme related to knowledge level modelling, we note that to date progress has been made especially in defining and structuring the content of KL models. Our belief is that in order to get to a practical KL modelling environment we must also solve the formalization problem as well.

We call knowledge level modelling environments knowledge engineering environments able to formally describe problem-solving models and to support the major activities related to the acquisition, use and evolution of these models. We consider that KLME-s should not be biased towards any of the existing modelling styles. On the contrary, by providing a general modelling language KLME-s should accommodate any existing modelling style and also encourage their experimentation, evaluation, integration and evolution.

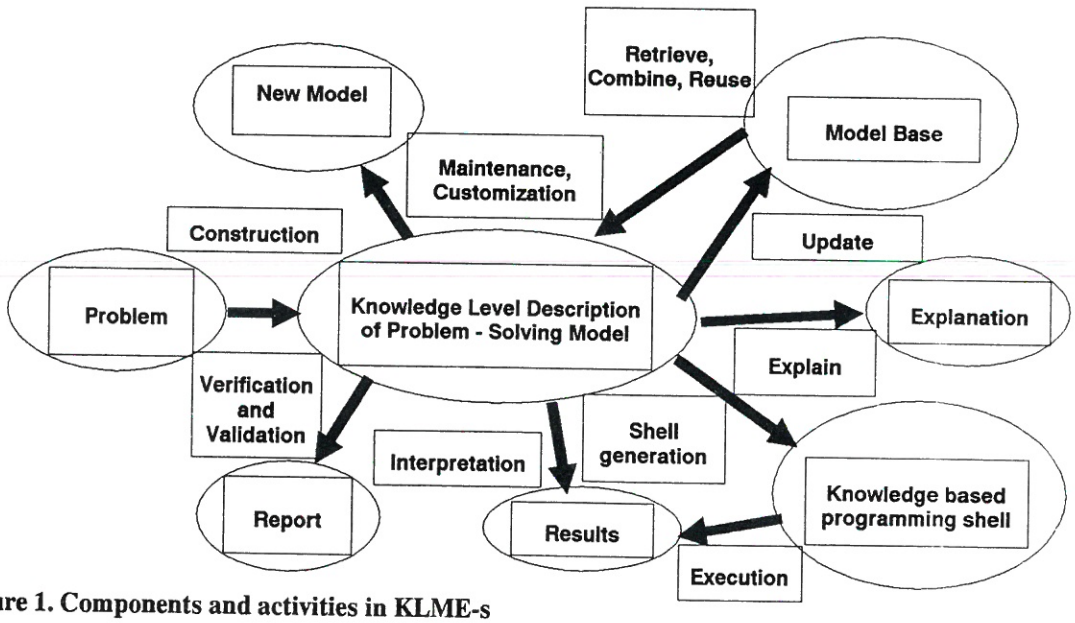


Figure 1. Components and activities in KLME-s

Figure 1 shows our current view of the components and activities of a KLME. Given a specific problem or problem type, model construction tools will help produce a model appropriate to the specific problem/context/domain requirements. This can be done in several ways such as retrieving a model from the model base, modifying an existing model, combining existing models or building up a brand new model. Each of these possibilities can be supported by specific tools. Case based reasoning could be used for retrieving models based on features of the problem, direct knowledge editing or tools based on psychological theories (e.g. repertory grids or concept maps) could be used to build new models. A very important service is shell generation. This transforms a conceptual model into an efficient programming shell having components for knowledge acquisition, inference, explanation, etc. The shell should be compiled into existing languages (e.g. KEE, CLOS, OPS5, C++) to ensure efficiency and portability. Another facility of the KLME is model base management. Issues of indexing and feature-based retrieval must be solved here. For better retrieval, models could be related to more general ontologies describing domains and problem types. Other activities supported include model interpretation (or simulation) useful for designing, debugging and validating models, model verification and validation, model maintenance and model explanation.

3. The Model Language

A programming language used in a programming environment is essentially a means for a human to communicate commands to a computer. A modelling language used in a KLME must be a means of communicating knowledge among humans. This places specific demands on the modelling language to be described next.

3.1 Requirements for the Modelling Language

Given the above distinction and the nature of problem-solving models we can state the following requirements for the modelling language.

1. Well-defined, declarative semantics. This stems from the need of clarity and uniformity in describing models.

2. Ability to describe both static - e.g. concepts and relations - and dynamic - actions or behaviours - components. Actions and behaviours are required not only for describing domain knowledge but also for expressing control in problem-solving models.

3. Capabilities for representing both generic and non-generic components.

4. Strong organizational capabilities for representing domain and problem-solving ontologies. This is part of the very idea of a general modelling environment.

5. Operational character allowing models to be mechanically interpreted. This allows models to be run without being compiled, allowing knowledge level design, debugging, enhancement and validation.

Terminological languages in the KLONE [Brachman and Schmolze 85] family, such as LOOM [McGregor and Bates 87] BACK [Luck et al 87] KL-TWO [Vilain 85] or recently CLASSIC [Brachman et al], are a tempting choice for the modelling language. Their semantics is formally defined and implementing algorithms has been analysed for complexity([Brachman and Levesque 84][Nebel 88],etc). They provide general representational services (completion, classification, recognition) and presently accommodate rules, dependency management and query services. With extensions to be discussed later on, classification technology can represent and simulate control and behaviour in various ways, including KADS- like layered models which may at first sight seem awkward to model.

3.2 . Overview of the Language

The MODEL language integrates standard features of terminological languages - completion, classification, recognition - with recent additions - rules and patterns - and with its own enhancements - methods, constraints and refinement. MODEL is the only terminological language which supports a second interpretation of descriptions, besides the usual meaning as intensional specifications

representing classes of objects. This second meaning is that of plans which specify how one can construct objects with a given structure. In MODEL this interpretation is supported by the refinement service useful, among others, for knowledge elicitation.

3.2.1 Concepts, Roles and Instances

MODEL is object centred in that it concentrates on describing, organizing and manipulating classes of objects in the domain of discourse. There are three kinds of formal entities in MODEL:

Concepts, which are complex aggregates composed of a limited set of description-forming operators. Concepts correspond to one- place predicates being applied to one individual object at a time.

Roles, which describe properties of or relations among objects. Roles correspond to two-place predicates and thus relate two individual objects at a time, one of them belonging to the domain of the role and the other to its range.

Instances, which directly represent objects in the domain of interest. Instances satisfy concepts (e.g. John is a Person) and have roles filled with instances of other concepts (e.g. John is married-to Mary).

< concept definition > →(concept < concept name > [:primitive]

(:and < concept > $^{+}$)

[< annotations >]

[< methods >]

< concept > → < concept name >

(:and < concept > $^{+}$)

(:all < role name > < concept >)

(:the < role name > < concept >)

(:some < role name > < concept >)

(:atleast < positive integer > < role name >)

(:atmost < non negative integer > < role name >)

(:same (< role name > $^{+}$)(< role name > $^{+}$))

(:oneof < constant > $^{+}$)

< annotations > → < annotation > $^{+}$

< annotation > → (:has < role name > (< annotation name > < constant > $^{+}$) $^{+}$)

(:self (< annotation name > < constant > $^{+}$) $^{+}$)

< annotation name > → < symbol >

< methods > → < method > $^{+}$

< method > → (:method < role name > (< message > < method expr > $^{+}$) $^{+}$)

(:method self (< message > < method expr >) $^{+}$) $^{+}$)

< message > → < symbol >

< method expr > → < symbol > | < Common LISP expr >

< constant > → < Common LISP constant >

Note: (:the R C) = $_{def}$ (:and (:all R C)(:atleast 1 R)(:atmost 1 R))

(:some R C) = $_{def}$ (:and(:all R C)(:atleast 1 R))

a - syntax of concepts

< role definition > →(role < role name >

(:and < role name > $^{+}$)

(:domain < concept >)

(:range < concept >)

(:inverse < role name >))

b - syntax of roles

< instance > →(instance < instance name > < concept >)

(fills < role name > < instance name > < constant > $^{+}$)

< instance name > → < symbol >

c - syntax of asserting instances

Legend:

$^{+}$ = at least one occurence

[...] = optional element

Figure 2. Syntax of MODEL Entities

The syntax of concepts, roles and instances is shown in Figure 2. Concept definition (Figure 2-a)

has four main sections. The first is the structured definition of the concept, that is a conjunction of terms formed with a vocabulary of descriptors shared by several terminological languages. Among these descriptors we mention the following. The :all descriptor restricts the concept type of role fillers, :atleast and :atmost restrict the cardinality of the filler set, :same imposes equality of fillers found by the following two role chains, and :oneof specifies a concept as a choice from an enumerated set of individuals.

The last two sections contain enhancements specific to MODEL. The methods section extends MODEL with method attachment and message passing mechanism in a manner common to object-oriented languages. Methods can be attached to each role in part or to the concept (self) as a whole. Assume that I is an instance of concept C, C has an R role and both R and C have a **simulate** method. MODEL allows **simulate** messages to be sent to I or to its R role according to the following patterns:

(i) (simulate I $argument_1$...$argument_n$:control *Control-fn*)

- Message sent to I

(ii) (simulate I R $argument_1$...$argument_n$:control *Control-fn*)

- Message sent to the R role of I

Here, the :control argument holds a function which will combine the methods found under the **simulate** message type. Because a concept is a conjunction of terms possibly inherited from other concepts, methods placed on roles and on the concept are also inherited according to a simple scheme: all methods from super concepts are accumulated in a set associated with the message type. The :control argument is used to combine these methods in user specified ways. This scheme also provides demon methods by means of special message names composed of a prefix - before, after and when - giving the moment of activation and a suffix - e.g assertion, retraction - giving the situation of activation.

The annotations section holds meta-information about the concept. Meta-information may be specified for each role in part (the :has operator) or for the concept as a whole (the :self operator). The meta-information is structured in an annotation-values format as shown in the syntax. Values of each annotation are also inherited. As in CLASSIC, it is possible to specify a predicate which will check whether a host language object belongs to a concept. This is done using a special test method.

Finally, MODEL concepts can be primitive or defined. Defined concepts contain necessary and sufficient conditions for an individual to be recognized as belonging to the concept. Primitive concepts contain only necessary conditions and thus are not fully specified. Several primitive concepts can be explicitly declared as being disjoint. This ensures that there will be no individuals "shared" by such concepts.

3.2.2 Patterns and Rules

Terminological languages generally do not allow arbitrary computations to be carried out. Patterns and rules are assertion-time extensions which permit this.

Patterns are conjunctions formed by instance asserting expressions (Figure 2-c) where instance names and role fillers can also be variables. Figure 3-a shows the syntax of patterns, Figure 3-b the syntax of rules and Figure 3-c a rule example. Patterns appearing in the left hand side of rules are instantiated by existing instances retrieved by the rule system. Patterns in the right hand side of a rule represent instances to be asserted once the variables are bound. Thus, the rule in Figure 3-c asserts an instance of the Working-family concept for each male ?h and female ?w who are employed and married to each other. Rules conditions may involve besides a conjunctive pattern an arbitrary predicate. Rules belong to rule sets which can be manipulated (loaded, executed, etc) individually. Rules and patterns are implemented through a RETE network. Like in CLASP [Yen, Juang and McGregor 91] they allow arbitrary forward driven computations to be performed.

< pattern > →(:and < element > $^+$)

< element > → (instance < pattern object > < concept name >)

(fills <role name> <pattern object> <pattern object>)

<pattern object> → <variable> | <constant>

a - pattern syntax

<rule definition> → (rule <rule name> :in <rule set name>

<pattern>

(:if <predicate>)

(:do <action>))

b - rule syntax

(rule Assert-working-family :in Rule-set-007

(:and (instance ?h (:and Employee Male Married))

(fills salary ?h ?h-s)

(fills married-to ?h ?w)

(instance ?w (:and Employee Female Married))

(fills salary ?w ?w-s)

(fills married-to ?h))

(:do (instance wf-0077 Working-family)

(fills husband wf-0077 ?h)

(fills wife wf-0077 ?w)

(fills family-income wf-0077 (+ ?h-s ?w-s))))

c-rule example

Figure 3. Syntax of Patterns and Rules

3.2.3 Constraints

Constraints represent a programming paradigm useful for solving search problems [Steele and Sussman 80], [Kanal and Kumar 88]. MODEL is currently the only language integrating term classification with constraint propagation. As the constraint mechanism employed is rather sophisticated [Trausan, Barbuceanu and Ghiculete 92], only its major lines will be reviewed here.

In principle, a constraint is a predicate among given variables. The domains of variables can be discrete or continuous sets. Solving a constraint satisfaction problem means finding an assignment of values to the variables such that all constraints are satisfied. Many algorithms for solving such problems have been reported [Davis 87]. Constraints have important applications in planning [Stefik 81], scheduling [Fox 83], [Mott et al 88], model based design [Murtagh and Shimura 90], etc.

Due to their declarative character and well-defined semantics, constraints deserve a place in a knowledge level modelling environment and can bring new problem-solving capabilities to a terminological language. As a simple example of what can be done with constraints and cannot be done otherwise, consider the concepts in Figure 4-a. It would be nice if we were able to state that in any instance of working-family the family-income is equal to the sum of the salaries of the husband and wife and if we could make sure that whenever two of these values are known the third gets automatically computed and asserted as well. There is however no way to declaratively state and enforce this in current terminological languages. With an explicit constraint language of the kind provided in MODEL this becomes easy. One simply loads the library constraint shown in

(concept working-family

(:and (:the husband employee)

(:the wife employee)

(:the family-income real-number)))

(concept employee

(:and (:the employer company)

(:the salary real-number)))

a - concepts

(constraint sum

(:parameters (N))

(:variables ((input 1 n) sum))

(:cases ((sum-unknown (and (unknown sum)

(all-known (input 1 N))))

(one-input-unknown (and (known sum)

(one-unknown (input 1 N))))

(all-known-ok (and (known sum)

(all-known (input 1 N))

(= sum (compute-sum (input 1 N)))))

(all-known-err (and (known sum)

```
(all-known (input 1 n))
(not ( = sum (compute-sum (input 1 N))))))))
(:actions ((all-known-ok do-nothing)
(all-known-err (error "inconsistent sum"))
(sum-unknown (assign sum (compute-sum (input 1 N))))
(one-input-unknown (assign (the-unknown (input 1 N))
(- sum (compute-sum (the-known (input 1 N))))))))))
```

b - constraint definition

```
(constraints S-2
(:constraint (sum (N 2)))
(:and (instance ?f working-family)
(fills husband ?f ?h)
(fills wife ?f ?w)
(fills family-income ?f ?income)
(fills salary ?h ?h-salary)
(fills salary ?w ?w-salary))
(:mapping (?income sum)
(?h-salary (input 1))
(?w-salary (input 2)))
(:actions ((all-known-err (error "inconsistent income for family ~S" ?f)))))
```

c - constraint instantiation and installation

Figure 4. Constraint Definition and Usage

Figure 4-b, instantiates it with two inputs (Sum works with any number of inputs) and installs the instance to watch over instances of the working-family concept. Figure 4-b shows the MODEL representation of the general Sum constraint and Figure 4-c the MODEL construct which simultaneously instantiates Sum for two inputs and connects it through a conjunctive pattern to the individuals it controls.

The MODEL scheme of specifying constraints has some specific features illustrated in Figure 4-b. Constraints can be parameterized allowing a variable number of variables and components. The constraint itself is represented as a bundle of rules or "cases", each case having a name, a predicate and an action. Cases allow clear specification of local propagations and, as shown in a next section, of computations outside constraints as well. Not shown in the example is another feature allowing constraints to be defined by connecting lower level constraints. These features make constraint specification more understandable and facilitate maintenance.

The instantiation and installation of constraints consist in creating an instance (S-2 in Figure 4-c) associating a pattern which detects the individuals to whom the constraint applies and specifying the correspondence between constraint variables and these individuals. As shown in Figure 4-c, it is also possible to add actions overriding those provided in the constraint definition.

3.3 Terminological and Assertional Services

MODEL terminological services are related to the construction of conceptual taxonomies. Assertional services handle the creation, modification and retraction of individuals - objects described by concepts but which cannot be further instantiated.

As MODEL also supports the meaning of concepts as plans, MODEL provides an additional refinement service which builds an individual by instantiating the "plan" contained in its concept. This service is usable at assertion time, hence can be considered an assertional service. We will treat it separately however, due to its novelty in the context of terminological language mechanisms.

MODEL terminological services include:

(i) *Completion* - a process which puts a concept in its "complete" form. This form contains all inherited features, unifies descriptions of multiply inherited roles, checks consistency of unified descriptions and of the concept as a whole.

(ii) *Subsumption checking*. A concept **a** subsumes a concept **b** iff all individuals described by **b** are also described by **a**. The set of individuals described by a concept is called the extension of the concept. Terminological languages admit axiomatic specifications of their semantics, which helps prove properties of the language such as the relation between expressive power and complexity of the subsumption algorithm [Brachman and Levesque 84]. Considering results such as [Nebel

88], subsumption in MODEL is CO-NP hard due to the existence of subroles and cardinality restrictions. Our implementation is however complete, covering the expensive cases as well.

(iii) *Classification*. This is the process of determining the most specific subsumers and the most general subsumees of a concept. Classification is implemented essentially as a network traversal process doing subsumption checking among relevant pairs of concepts. MODEL concept bases are built in two phases. In the first, concepts are acquired and/or edited with minimal checks from the system. In the second, concepts are put in the complete form and classified. We also note that terminological services do not take into consideration rules, implications and constraints.

Concerning assertional services, MODEL provides an assertion language for creating instances and filling their roles, a query language for retrieval and a retraction language for removing fillers and instances. Boolean constraint propagation [McAllester 90] is used for keeping dependencies and handling retractions.

Implications, rules and constraints are considered by the assertional services. Whenever an individual is recognized as belonging to a concept, the implication rule is applied and its consequent concept is asserted of the individual.

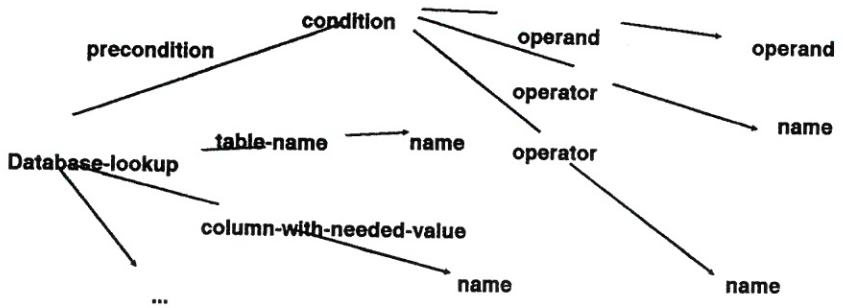
Patterns are held in a RETE network which keeps track of the tuples matching each pattern. When such tuples come into existence rules and installed constraints (Figure 4-c) whose patterns are satisfied by the tuples are executed.

3.4 The Refinement Service

Descriptions can also be seen as plans for constructing structured objects. As this view is useful in the modelling framework for supporting knowledge acquisition and compilation services, MODEL offers a special refinement machinery to be described next.

3.4.1 Descriptions as Plans

Let us consider a piece of domain knowledge taken from the SALT [Marcus and McDermott 89] system. In certain case SALT uses database-lookup procedures to determine values for design variables. A database-lookup procedure specifies a table to look into, a column where the needed value can be found, a number of tests performed on entries belonging to other columns to determine viable candidate values and an ordering scheme consisting of an ordering column and an ordering criterion for ranking candidates. The general scheme of such a procedure can be described as a MODEL concept as shown in Figure 5-a.



(concept Database-lookup

(:and Procedure

(:the precondition Condition)

(:the table-name Name)

(:the column-with-needed-value Name)

(:all parameter-test Condition)

(:the ordering-column Name)

(:the optional (:oneof minimum maximum))))

a - The concept

b - Fragment of the equivalent goal tree

Figure 5. Database-lookup Concept and its Goal Tree

A plan interpretation of this description should be read like this: "to construct a Database-lookup procedure one has to define a precondition, specify the name of a data table and the column where the value can be found, define a set of tests and specify the ordering column together with the ordering criterion". In other words, the description supplies the equivalent of a goal tree where the concept is the top goal and the roles represent subgoals, as in Figure 5-b.

The component descriptions supply additional information about the corresponding subgoal. Cardinality restrictions specify limits to the number of times a goal must be achieved - e.g. only one precondition but possibly more than one parameter tests. Role chains specify that one goal's instantiation should also be used as other goal's instantiation.

In order to carry out the plans represented by descriptions a collection of planning operators must also be specified. These operators will carry out actions to instantiate goals. Possible operators in the above example include prompting the user for values, recursive goal expansion into subgoals, knowledge based selection from alternatives or search for instances or subsumee concepts.

To summarize, descriptions formed with concept-forming constructs can be viewed as goal trees which a planning process can use to build instances of concepts. The process of building instances of a concept is called refinement and the MODEL service for this is the refinement service or R-box.

3.4.2 The MODEL R-box

The R-box is a general-purpose agenda problem solver. Each concept to be refined is placed as a goal on the agenda and each of its refinable roles is placed as subgoal.

Figure 6-a describes the basic loop of the refinement system mentioning some of the options it has at each step. Users can program the R-box in two ways. The first is by defining refinement rules - with the MODEL rule language - as illustrated in Figure 6-b. The second is to use annotations on the concepts. Annotations contain meta-information used by the refinement process, as shown in Figure 6-c.

Basic Loop of the Refinement Process

1. **Determine number of instances of concept to be refined.** Can be done by
   - prompting the user
   - applying a rule
   - applying a knowledge based method attached to concept.

2. **Determine basic refinement mode**: interactive prompting, search or recursive expansion. Can be done by
   - prompting the user
   - applying a rule
   - reading an annotation

3. **Carry out concept refinement** according to the determined refinement mode (step 2) as many times as determined at step 1.

3.1 If refinement mode is **interactive prompting**, then
   - determine a screen configuration
   - carry out prompting for each role

3.2 If refinement mode is **search, then use**
   - **either** a general search procedure to retrieve an instance or subsumed concept
   - **or** a specially tuned search method attached to concept or role

3.3 If refinement mode is **recursive expansion** then
   - create subgoals for refinable roles
   - use a special annotation to specify the order of handling subgoals

a - basic control loop of the refinement process

```
(rule Refinement-Rule-007 :in Procedure
AcquisitionRuleSet
(:and  (instance ?g RBox-goal)
(instance ?g Database-lookup)
(fills Table-name ?g Machine))
(:if (unknown $Refinement-mode))
(:do
 (assign ?g $Refinement-mode 'interactive-prompting)
(assign ?g $Screen-format
(get-annotation ?g 'screen-format))))
```

b - Refinement rule example

```
(concept Database-lookup
```

(:and ...)

(:self (refinement-mode 'interactive-prompting)

(screen-format 'role-prompting-box)))

c - Use of refinement annotation

Figure 6. The Refinement Process and its Programming

The rule in Figure 6-b states that any Database-lookup procedure which searches the Machine table must be acquired through interactive prompting. An annotation placed on the concept tells the R-box what screen configuration to use for this prompting, and information is transferred to the R-box by the rule. In Figure 6-c the same refinement mechanism is used, but without any rule. Relevant information is placed on the concept as annotations directly used by the R-box.

The refinement process works in a forward driven manner. A goal has attributes for the control information needed to carry it out - e.g. the refinement mode or subgoal ordering. This information is supplied by rules, methods or annotations. A goal is ready for processing when all requested information becomes available. At each cycle, the refinement processor executes the first ready goal from the agenda.

3.4.3. Using the Refinement Service

We presently see three major directions in which refinement is useful for knowledge level modelling.

The first is as a mechanism for prototyping knowledge acquisition scenarios. In the MODEL framework domain knowledge bases consist of instances of MODEL concepts. For example, in a SALT system represented in MODEL, Database-lookup procedures will appear as instances of the concept shown in Figure 5. With the refinement service one can program various model driven knowledge elicitation strategies of the kind illustrated by PROTEGE [Musen 89]. The example in Figure 6 is illustrative for programming a knowledge elicitation session as well. Constraints can also be employed at knowledge acquisition time to validate entered knowledge and to infer consequences of it. Acquisition strategies can be encoded in rule sets and used or reused for rapid prototyping of models. More on these issues will be presented in the next Section.

The second use of refinement is in the process of compiling target shells from KL models. Refinement is useful here for compiling procedures described in the terminological language into another language. This is based on the fact that refinement traverses recursively a tree of concepts and can assemble code during this traversal.

Finally, refinement is also useful as a general problem-solving method. We have implemented and reported several versions of the incremental refinement idea embodied in the refinement service. One version was in the context of object-oriented representations [Barbuceanu, Trausan and Molnar 90] and another was defined for a more general description language [Barbuceanu, Trausan and Molnar 87,89]. We have shown that these refinement mechanisms are usable as generic problem-solving models especially for synthesis tasks like design [Barbuceanu 85]. In [Barbuceanu 86] we show the use of refinement in describing and compiling program components.

3.5 Modelling Control and Behaviour

While it is clear that terminological languages are suitable for representing "static domain knowledge", one may wonder whether control and behaviour can be represented equally well. In this Section we show that this is entirely possible. Representing such knowledge is a prerequisite to using a terminological language for KL modelling.

First, a lot of domain knowledge is about "how to do" something, as is the case with the piece of knowledge in Figure 5 which is about how to look for a value in a relational table. Second, all modelling styles use abstract control schemes in their problem-solving methods. The fact that some approaches view control schemes as strongly related to the domain representation - e.g. generic task - while others think of them as independent - like - KADS - only places demands on the flexibility of the needed mechanisms, if one wants to accommodate all these modelling styles.

Another requirement comes from the layered control architecture of KADS which separates domain, inference, task and strategy levels.

Control schemes and behaviours - generally called actions - must be represented in the terminological language in a way that meets four basic requirements:

1. Actions must be represented as concepts with their relevant features represented as roles.

2. There must exist a way to interpret actions.

3. There must exist a way to compile actions into usual "procedural" languages.

4. The interpretation and compilation mechanisms must directly use the represented features of actions. Modification of action roles should directly reflect in their compiled and interpreted behaviour.

These conditions ensure that actions can be classified as any other concept, that models built with them can be made operational and that terminological maintenance and modification of their feature will reflect in their actual behaviour.

3.5.1 The Basic Action Representation

Actions are represented as concepts with roles describing their relevant features. A good example is the Database-lookup procedure in Figure 5. We have called elsewhere [Barbuceanu, Trausan and Molnar 87] such representations semantic as they exhibit the meaning of the action through semantic relations. Another example is shown in Figure 7-a which presents another SALT inspired piece of procedural knowledge for determining constraints on a design variable (the term constraint in this example is different from the MODEL constraints previously discussed). Figure 7-b shows part of a possible terminology involving SALT constraints.

```
(Concept FormulaCalculationConstraint
(:and Constraint
(:the constrained-value Variable)
(:the constraint-type (:oneof Minimum
Maximum))
(:the precondition Condition)
(:the procedure Calculation)
(:the formula Formula)))
```

a - Procedure concept for determining a constraint

```
Procedure
Constraint
FormulaCalculationConstraint
Maximum FCC ; (:the constraint-type maximum)
Minimum FCC ; (:the constraint-type minimum)
SideDoorFCC ; (:the precondition (=
door-opening side))
CentralDoorFCC ; (the precondition
(=door-opening center))
DataBaseLookupConstraint
```

b - Partial terminology for SALT like constraints

Figure 7. Actions

To simulate such procedures during problem solving, MODEL provides the method attachment mechanism. Specialized interpreters can be attached as methods to the concepts describing actions and used to execute them.

A useful organization of concepts describing actions is to consider them parameterized by their roles. If an instance is created with all roles closed, then the corresponding procedure will have no formal parameters. If unfilled roles exist, then these can be considered formal parameters and actual values will have to be supplied at activation time. Figure 8 shows examples of this where simulate is the method used to execute an action.

```
(concept MaximumFCC
(:and FormulaCalculationConstraint
(:the Constraint-type maximum)))
(instance MaxFCC-1 MaximumFCC)
(simulate MaxFCC-1 :constrained-value
CarJambReturn
:precondition T
:formula '(* panel-width stringer-quantity))

(concept SideDoorFCC
(:and FormulaCalculationConstraint
(:the precondition '(= door-opening side))))
```

```
(instance SideDoorFCC-1 SideDoorFCC)
(simulate SideDoorFCC-1 :constrained-value
'CarJambReturn
:constraint-type 'maximum
:formula '(*panel-width stringer-quantity))
(instance MaxCarJambReturn (:and
MaximumFCC SideDoorFCC))
(fills constrained-value MaxCarJambReturn
CarJambReturn)
(fills formula MaxCarJambReturn (* panel-width
stringer-quantity))
(simulate MaxCarJambReturn)
```

Figure 8. Procedures and Their Arguments

Actions are compiled by supplying specialized compilation methods attached to them. To support the above variability in the number of arguments, these methods must be able to make partial evaluation.

Finally, we note that the use of a terminological language to represent procedures improves their reusability and maintainability. Classificatory services permit that these program components be organized in taxonomies and be retrieved according to the powerful subsumption-based and content-directed mechanisms available. This also makes procedure descriptions be rightfully considered knowledge-level.

3.5.2 Composition and Layering

Besides proper parameters or features, roles can also contain other actions which are components of the given action. This creates a hierarchical representation in which actions are explicitly assembled to result in higher order actions.

It is useful to distinguish between two kinds of component actions: actions representing "normal" components or sub-actions and actions representing control components which apply the non- control actions and assemble their results. In the simplest case there should be one control action and as many non-control actions as needed.

If this discipline is obeyed, a single interpretation method is sufficient for all actions. This method contains a few lines of code which apply the control action to the sub-actions and to some global data base. For the same reasons a single compilation method will suffice. The scheme can specify any kind of control structure starting from a general one (e.g. if-then-else) to customized, domain specific ones. The scheme naturally extends to the situation where the control and/or non-control actions are composed actions themselves, thus being able to create meta-level architectures important e.g. for the KADS approach. In this situation, if the control action is composed then it will return a simple action which will be applied to the non-control actions as arguments. If a non-control action is composed, its application will determine procedure on an elementary action which will then be applied to the database.

As an example of a model specific action, we consider the selection of fixes in the SALT system. SALT uses pieces of knowledge named fixes to suggest possible repairs to violated constraints. In general, there are several possible fixes per a given constraint violation. Each fix has an associated desirability rating. Selection of the preferred fix is made according to this rating. This is an example of knowledge which would be considered task level according to KADS. Figure 9 shows the action for this task level selection, while Figure 10 shows how roles for control and non-control components must become sub-roles of the general control-component and non-control-component roles in order to be treated properly by the interpretation/compilation methods.

```
(concept Fix
(:and Procedure
(:the violated-constraint Constraint)
(:the value-to-change Node)
(:the change-type (:oneof increase decrease))
(:the step-type StepType)
(:the step-size Number)
(:the preference-rating (:oneof 1 2 3 4 5 6 7 8))))
(concept PreferenceBasedFixSelection
(:and ControlledAction
(:all possible-fixes Fix)
(:theprefer-lowest-rated PreferLowestRatedFn)))
```

Figure 9. A Task Level Action

(concept ControlledAction

(:and Action

(:the control-component Action)

(:all non-control-component Action))

(:method :self (simulate Simulate Controlled ActionFn)))

(role possible-fixes (:and non-control-component)

(:range Fix))

(role prefer-lowest-rated (:and control-component))

Figure 10. The General ControlledAction and Action Subroles

3.5.3 Constraint Based Control Schemes

We remember that MODEL constraints are specified as a logical order of cases. The cases in which an installed constraint can find itself at any time constitute the state the constraint finds itself in. This creates the possibility for constructing some useful control structures in which control flow is related to the state of constraints. For example, consider the Sum constraint with cases as shown in Figure 4 and an installed instance of it. Then, one can specify actions taking place when/while/until/etc. the constraint instance is in one or several states. For illustration, Figure 11 shows an action which repeats the Find-a- value action until the Sum-007 constraint instance enters the all-known-ok state.

We have shown elsewhere [Barbuceanu, Trausan and Molnar 89] that constraint driven control eases maintenance because constraint modifications, done through creating, removing, merging or splitting cases, are easier to propagate over procedures whose control explicitly uses constraints and cases.

(instance MakeSumHold ConstraintDrivenUntil)

(fills constraint-instance MakeSumHold Sum-007)

(fills until-condition MakeSumHold all-known-ok)

(fills action MakeSumHold Find-a-value)

Figure 11. Constraint Driven Action

4. Ontologies for Knowledge Level Modelling

In this Section we show how MODEL can be used to build a conceptual model for the well-known problem-solving system SALT [Marcus and McDermott 89]. Our analysis will be made according to the KADS approach thus showing how KADS modelling can be accommodated in our framework. Because the SALT architecture integrates problem-solving with knowledge acquisition, our discussion will tackle both aspects in part. For each aspect we provide the KADS description and the MODEL formalization.

4.1 The SALT System

4.1.1 The SALT Problem Solving Process

SALT is a generic problem-solving system based on a constraint satisfaction process, being suitable to a class of (synthesis) problems where knowledge exists for proposing values, specifying constraints values must satisfy and specifying remedies when constraints are violated.

Being a shell, SALT does not come with any specific domain knowledge for the **static domain knowledge level** of KADS.

At the **inference level**, SALT assumes the existence of variables (a KADS metaclass) which are assigned values through several inferential processes (KADS knowledge sources). The most important are procedures which propose values, constraints which enforce restrictions on values and fixes which provide ways to determine new values in case constraints are violated.

At the **task level**, SALT recognizes three important goals, extending a design by proposing a value for a variable, checking that constraints are satisfied and fixing violated constraints. These goals are accomplished by associated tasks which propose a design extension, identify a constraint and propose a fix.

At the **strategy level**, the SALT problem solver simply cycles through the three above tasks. When a value is proposed, SALT checks constraint

violations and if any is discovered it applies fixes according to existing preference ratings.

4.1.2 The SALT Knowledge Acquisition Process

One major lesson from the role limiting method is that knowledge roles identified during conceptualizing problem-solving also structure the knowledge acquisition process. Taking knowledge acquisition as a problem-solving activity one can come with a layered KADS analysis of it. Given the above observation, the structure of the knowledge acquisition process would mirror the structure of the problem-solving process as they both rely on the same knowledge roles. This is apparent from the following analysis of the SALT knowledge acquisition process.

At the **inference level**, the acquisition process manipulates variables, procedures, constraints and fixes as data, or metaclasses. The knowledge sources which act on these data are e.g. the individual mechanisms for eliciting procedures, constraints and fixes. Note that for the SALT problem solver procedures, constraints and fixes are canonical inference steps.

At the **task level**, the main goals are those of acquiring a procedure, constraint or fix. For each goal there is an associated acquisition task which uses the individual elicitation mechanisms.

At the **strategy level**, the SALT acquisition process similarly cycles through the three major tasks selecting one according to what is probably SALT's most important source of power, a thorough analysis of the completeness, uniqueness, correctness and convergence of the collection of procedures, constraints and fixes.

4.2 Modelling the SALT Problem Solving Process

As static domain knowledge - such as cables, machines or cars in an elevator design domain - is the least problematic to represent in a language like MODEL, we will concentrate on the other layers.

4.2.1 The Inference Level: SALT Knowledge Bases

SALT metaclasses and knowledge sources define a SALT specific representation language in terms of which SALT knowledge bases are acquired.

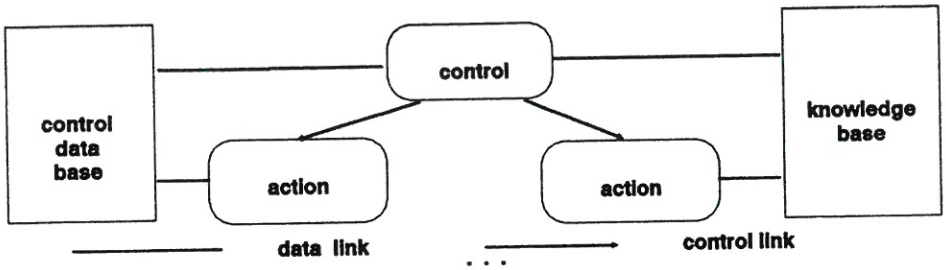
Figure 12 shows a possible organization of this layer. Figure 12- a illustrates the MODEL concept for SALT knowledge bases and the concept of control database holding control data manipulated by inference and other processes together with some of its components. Figure 12-b shows how a particular SALT knowledge base is created as an instance. Figure 12-c shows a possible taxonomic organization of some SALT components. Examples of Procedure-s, Constraint-s and Fix-es have been provided in the previous section. In the shown organization we have separated control concepts (nodes and their components) from the knowledge base as they play a special role as private data for the problem-solving mechanisms at various levels.

```
(concept KnowledgeBase :primitive)
(concept SALTKnowledgeBase
(:and KnowledgeBase
(:some procedures Procedure)
(:some constraints Constraint)
(:some fixes Fix)))
(concept ControlDataBase :primitive)
(concept SALTControlDB
(:and ControlDataBase
(:some nodes Node)))
(concept Node
(:and MainSALTKBComponent
(:the name Symbol)
(:all contributes-to Contribution) (:the contributed Node)))
(:all constraints Constrainer)
(:all suggests-revision-of SuggestedRevision)
(:the type (:oneof input parameter constraint))))
(concept Contribution
(:and AuxSALTKBComponent
(:the procedure Procedure)
(:the contributed Node)))
```

a- Scheme of SALT knowledge bases

```
(instance VT-SALT SaltKnowledgeBase)
```

b - creating a SALT knowledge base



KnowledgeBaseComponent

SALTKBComponent

MainSALTKBComponent

Procedure

Calculation

DataBaseLookup

Constraint

Fix

AuxSALTKBComponent

Test

Table

Formula

ControlDataBaseComponent

SALTControlDBComponent

Node

Contribution

Constrainer

SuggestedRevision

c - Possible organization of KB components.

Figure 12. MODEL-ing the Inference Layer of SALT

4.2.2. Task and Strategic Levels: Knowledge Processing Mechanisms

To represent the task and strategic layers, we first introduce the notion of knowledge processing mechanism (KPM). A KPM is a MODEL hierarchically recursive action which can act on a knowledge base and on a control database. Figure 13-a shows the KPM concept, Figure 13-b a graphic depiction of it and Figure 13-c a fragment of a KPM ontology.

(concept KnowledgeProcessingMechanism

(:and ControlledAction

(:the knowledge-base KnowledgeBase)

(:the control-data-base ControlDataBase)

(:all actions KnowledgeProcessingMechanism)

(:the control KnowledgeProcessingMechanism)))

a - The KnowledgeProcessingMechanism concept

b - Control and data links in a KPM

KnowledgeProcessingMechanism

Simple KPM

SimpleAction

SequencedActions
SortedSetSelection
KnowledgeDrivewKBSearch
SelectedActions
KnowledgeDrivewSelectedActions
CycleActions
OrderedCycle
KnowledgeDrivewSelectionCycle
KnowledgeTool
KnowledgeAcquisitionTool
InferenceTool
ExplanationTool

c - Fragment from a possible KPM ontology

Figure 13. Knowledge Processing Mechanisms

With these preparations we can describe some task and strategy level mechanisms.
Figure 14 illustrates two task level mechanisms, Propose-a- design-extension and Propose-a-fix. Figure 14-a shows the KPM-s by means of which these tasks are modelled. The Knowledge DrivewKBSearch mechanism searches for a knowledge base component which satisfies a search criterion. Once found, the when-found-action is applied and its result is left in a specified component (role) of the control database.

The Propose-a-design-extension task is an instance of this. It looks for nodes which have all contributors known and applies an applicable procedure for determining the value of the node's variable.

(concept KnowledgeDrivenKBSearch
(:and SimpleKPM
(:the kb-component KnowledgeBaseComponent)
(:the result-cdb-component ControlDataBase Component)
(:the search-criterion Condition)
(:the when-found-action SimpleKPM)))
(Concept SortedSetSelection
(:and SimpleKPM
(:the sorted-kb-component KnowledgeBase Component)
(:the select-components SimpleKPM)
(:the sorting-criterion Symbol)
(:the selection-criterion (:oneof minimum maximum))
(:the process-selected SimpleKPM)))

a - Used KPM-s

(instance Propose-a-design-extension Knowledge DrivewKBSearch)
(fills kb-component Propose-a-design-extension Node)
(fills result-cdb-component Propose-a-design-extension proposed-extension)
(fills search-criterion Propose-a-design-extension HasAllContributorsKnown)
(fills when-found-action Propose-a-design-extension ApplyAProcedure)
(instance Propose-a-fix SortedSetSelection)
(fills sorted-kb-component Propose-a-fix Fix)
(fills select-sorted-components Propose-a-fix SelectFixesForViolatedConstraint)
(fills sorting-criterion Propose-a-fix preference- rate)
(fills selection-criterion Propose-a-fix minimum)
(fills process-selected Propose-a-fix ApplyAFix)

b - SALT tasks

Figure 14. Task Level Mechanisms for the SALT Problem-Solver

The SortedSetSelection Mechanism is a bit more complex. First, it selects a set of knowledge base components of the sorted-kb- component type by applying the select-sorted-components action. Second, it sorts these components according to the sorting-criterion. Third, it selects from among the sorted set a subset using the selection-criterion and finally it applies the process- selected action upon each component from this subset. The SALT task for proposing a fix when a constraint violation occurs is implemented by instantiating this mechanism as shown in Figure 14- b. The task searches for the Fix components which apply to the violated constraint and sorts them according to their preference rating. The Fix-es with the smallest value for the preference rate are then selected and applied.

Finally, Figure 15 shows the strategic problem-solving mechanism of SALT. Figure 15-a shows the KnowledgeDrivenSelectionCycle mechanism which cycles through a set of actions selecting one at each step on the basis of specific knowledge. The SALT problem-solving strategy is an instance of this. It selects one of the three basic SALT tasks according to a fairly simple scheme: first a design extension is proposed, then constraint violations are checked and last fixes are applied if necessary.

(concept KnowledgeDrivewSelectionCycle
(:and SimpleKPM
(:some cycled-actions SimpleKPM)
(:the selection-action SimpleKPM)))

a - The KnowledgeDrivewSelectionCycle mechanism

(instance SALT InferenceTool (:and InferenceTool KnowledgeDrivewSelection Cycle))
(fills cycled-actions SALT InferenceTool Propose-a-design-extension Identify-a-constraint Propose-a-fix)
(fills selection-action SALT InferenceTool SALTProblemSolvingStrategy)

b - The SALT inference tool

Figure 15. Strategic Mechanism for the SALT Problem-Solver

4.3 Modelling the SALT Knowledge Acquisition Process

At the inference level, the SALT knowledge acquisition system is concerned with eliciting the major pieces of knowledge, namely procedures, constraints and fixes. One way to carry this out in MODEL is to use the refinement service. Figure 16 shows how the refinement process can be declaratively programmed to elicit a piece of Fix knowledge. The refinement process will build an instance of the Fix concept. The instance will be built by recursive expansion, that is every role will be considered as a subgoal and refined in a specific manner given by its annotation. The order of refining subgoals is given by a special annotation.

```
(concept Fix
(:and SALTKBComponent
(:the violated-constraint Constraint)
(:the value-to-change Node)
(:the change-type (:oneof increase decrease))
(:the step-type StepType)
(:the step-size Number)
(:the preference-rating (:oneof 1 2 3 4 5 6 7 8))))
(:self (refinement-mode recursive-expansion)
(refinement-order
(violated-constraint value-to-change
change-type step-type step-size
preference-rating)))
(:has violated-constraint
(refinement-mode interactive-prompting)
(dialogue-box Selection-box)
(box-info-fn Find-unfixed-constraints))
(:has value-to-change
(refinement-mode interactive-prompting)
(dialogue-box Indented-selection-box)
(box-info-fn Find-contributors-to-value))
(:has change-type
(refinement-mode interactive-prompting)
(dialogue-box Selection-box)
(box-info-fn Read-role-value)) ...))
```

Figure 16. Refinement Annotations Directing the Fix Elicitation Process

As shown in Figure 16, some roles are instantiated by interactive prompting. A specification of the dialogue box in which this takes place is given together with a function supplying the information presented to the user when prompted. For example, to determine the violated-constraint the user is presented a list of constraints for which fixes have not been provided yet. For determining the value-to-change in response to a constraint violation, the user is presented an indented list showing the contributors to the constraint violation determined following contributes-to links in the control database (see [Marcus and McDermott 89] for details).

MODEL constraints (no relation to SALT constraints) can be used to make certain inferences at elicitation time. In this example, the constraint-type, change-type and value-to-change are constrained by certain relations such as that if a maximum constraint is violated it can be fixed only by increasing the maximum or by decreasing the value. Figure 17-a shows a constraint enforcing that the change-type is correctly asserted whenever the constraint-type and the value-to-change are known. If the constraint is installed as shown in Figure 17-b, the change-type will not be prompted for anymore. Similar constraints or MODEL roles could be used to check consistency of the elicited knowledge.

At the task and strategy levels, knowledge mechanisms can be defined in a similar manner to what has been presented for the problem-solving system. The task of acquiring a Fix for example, can be described as a

```
(constraint Fix-change-type
(:variables constraint-type change-type
value-to-change)
(:cases
(decrease-1 (and (unknown change-type)
(eq constraint-type 'maximum)
(is-value value-to-change)))
(increase-1 (and (unknown change-type)
(eq constraint-type 'maximum)
(is-constraint value-to-change))) ...)
(:actions
(decrease-1 (assign change-type 'decrease))
(increase-1 (assign change-type 'increase))...)
```

a - Constraining the change-type

```
(constrains Fix-change-type-i
(:constraint Fix-change-type)
(:and (instance ?f Fix)
(fills violated-constraint ?f ?v)
(fills constraint-type ?v ?co-type)
(fills change-type ?f ?ch-type)
```

(fills value-to-change ?f ?val-ch))

(:mapping (?co-type constraint-type)

(?ch-type changr-type)

(?val-ch value-t-change)))

b - Installing the constraint

Figure 17. Constraint for Automated Inferencing of the Change Type

sequence of three steps. The first prepares the information used in the elicitation process (such as the list of contributors to a value). The second is the elicitation process described in Figure 16 and the third constructs and updates pieces of the semantic network (composed of Node-s) in the control database. The acquisition strategy is similar to the problem-solving strategy - a KnowledgeDrivewSelectionCycle (Figure 15) - but the control action is different as it tries to delay the acquisition of fixes as much as possible.

4.4 Supporting Compilation and Other Services

Problem-solving and knowledge acquisition are the most important activities supported in a KLME. As shown in Figure 1, KLME-s should support other activities as well, and the modelling language must lay the foundation on which to build tools supporting all activities.

An almost equally important activity is model compilation, that is the generation of shells from models. Our view is that models should be compiled into shells written in state-of-the-art AI languages like CLOS [Keene 89] or KEE [Fickes and Kehler 85]. This ensures efficiency and portability of the generated expert system. Assuming that the target language has standard features like objects, methods, demons and rules, the compilation into shells can be done in two steps. The first step is the specification of what has to be compiled and how. The second step is actual compilation.

Concepts from the knowledge base - e.g. procedures or fixes - can be compiled into objects. Those in the control database - e.g. nodes - can be compiled into more specific data structures. KPM-s can be compiled into methods, rules or demons. As all these are MODEL concepts, the complete form of concepts will be used as the compilation source. This form is constructed by completion service of the modelling language and is characterized by the fact that all inheritable features are inherited, all descriptions are unified and all detected inconsistencies are eliminated. In a first version of compilation, the specification step can be taken by means of a special editor/browser which would help annotate concepts with specifications of how they should be compiled. The Fix concept, for instance, can be annotated with specifications determining its compilation into e.g. a CLOS object inheriting from a more general Action object. KPM-s can be compiled as generic methods attached to the ControlDataBase and KnowledgeBase objects or to some of their components.

Given such specifications, the compilation stage can be implemented as a program whose input is a collection of annotated concepts and whose output is a collection of target language constructs. The MODEL language provides features useful in supporting other activities in knowledge level modelling. Model validation can be supported by building rule and/or constraint systems able to analyse the model. Watching over the problem-solving process by means of rules and/or constraints is also useful for generating explanations and for knowledge level instrumentation (debugging and others) of any on-going process.

5. Conclusions

Computer based problem-solving comprises two processes, modelling and programming. For knowledge based problem solvers these are known as knowledge level modelling and symbol level encoding. The existing AI programming environments currently support only the programming side of the process. The next generation of tools will have to address the modelling side, as this is the place where the intellectually challenging problems arise and the greatest potential for improving the knowledge and software engineering practice stems from.

As a solution to this, we have presented the notion of a knowledge level modelling environment which would support the full range of knowledge level

modelling activities involved in problem-solving. Programming is just one of these activities.

On the road toward building a knowledge level modelling environment the first problem we encounter is designing a knowledge modelling language able to describe at the knowledge level the problem-solving methods people invent and improve. Unlike programming languages which are means of communicating commands to a machine, modelling languages should be means of communicating knowledge among people. This raises specific requirements for these languages, among which most important are clearly defined semantics and powerful general-purpose representational services.

Our choice was to build such a language on the basis of term classification technology as this already provided much of the needed functionality. Our language extends this technology with features from object-oriented systems - methods and annotations - constraints, rules, a new refinement service and a way of hierarchically constructing knowledge processing mechanisms.

These extensions help the modelling language scale up from describing static domain knowledge to describing the inference, task and strategy level mechanisms problem-solving models are composed of.

To substantiate these claims, we have presented a detailed analysis of the problem-solving and acquisition components of SALT [Marcus and McDermott 89], a well-known generic problem-solving model for constructive problems.

As the implementation of the language is practically finished, our work will continue with implementing the other components of the knowledge level modelling environment.

REFERENCES

ABRETT, G. and BURSTEIN, M.H., **The Kreme Knowledge Editing Environment**, in J.H.Boose and B.R.Gaines (Eds.) Knowledge-Based Systems, Vol. 2, ACADEMIC PRESS, 1988.

AKKERMANS, J.M., VAN HARMELEN, F., SCHREIBER, A. and WIELINGA, B.J., **A Formalization of Knowledge-Level Models for Knowledge Acquisition**, INTL. JOURNAL OF INTELLIGENT SYSTEMS, 1990.

ALEXANDER, J.H., FREILING, M.J., SHUMAN, S.J., REHFUSS, S. and MESSIK, S.L., **Ontological Analysis: An Ongoing Experiment**, Proceedings AAAI Workshop on Knowledge Acquisition for Knowledge Based Systems, Banff, Canada, 1986.

BARBUCEANU, M., **An Object Centered Framework for Expert Systems in CAD**, in I.S. Gero (Ed.) Knowledge Engineering in Computer Aided Design, NORTH HOLLAND, 1985, pp. 223- 253.

BARBUCEANU, M., **Knowledge Based Development of Reusable and Evolutionary Software**, Proceedings of the Sixth International Workshop on Expert Systems and Their Applications, Avignon, France, 1986.

BARBUCEANU, M., TRAUSAN-MATU, S., and MOLNAR, B., **Concurrent Refinement of Structured Objects: A Language for Declarative Knowledge Programming Based on Specifications and Annotations**, Proceedings of Expert Systems '89, Brighton, UK, 1989.

BARBUCEANU, M., MOLNAR, B. and TRAUSAN-MATU, S., **Concurrent Refinement: A Model and Shell for Hierarchical Problem Solving,** Proceedings of Tenth International Workshop on Expert Systems and Their Applications, Avignon, France, 1990.

BARBUCEANU, M., **The Models Environment for Knowledge Acquisition: Automating The Task-specific Architecture Approach**, The AAAI Knowledge Acquisition For Knowledge Based Systems Workshop, Banff, Canada, November 1990.

BARBUCEANU, M., Models: **Towards Knowledge Level Modeling Environments**, Australian Workshop on Knowledge Acquisition for Knowledge Based Systems, Sydney, Australia, August 1991.

BARBUCEANU,M. and TRAUSAN-MATU, S., MODELS: **Towards a Language Construction Approach to Expert System Development**, in STUDIES AND RESEARCHES IN COMPUTERS AND INFORMATICS, Bucharest, Romania, 1990.

BOOSE, J. and BRADSHAW, J., **Expertise Transfer and Complex Problems,** INTL. JOURNAL ON MAN MACHINE STUDIES, 26, 1987, pp. 3-28

BORNING, A., DUISBERG, R., FREEMAN-BENSON, B., KRAMER, A. and WOOLF, M., **Constraint Hierarchies,** OOPSLA '87 Proceedings, pp. 48-60.

BRACHMAN,R.J., MCGUINESS,D.L., PATEL- SCHNEIDER,P.F.,RESNIK, L.A. and BORGIDA, A., **Living with Classic,** in J.F. Sowa(Ed.) Principles of Semantic Networks, Explorations in the Representation of Knowledge, MORGAN KAUFMANN, San Matteo, CA., 1991.

BRACHMAN, R.J., GILBERT, V.P.and LEVESQUE , H.J., **An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of KRYPTON,** Proceedings of Ninth International Joint Conference on Artificial Intelligence, Los Angeles, CA.,1985.

BRACHMAN, R.J. and LEVESQUE, H.J., **The Tractability of Subsumption in Frame-Based Description Languages,** Proceedings of the American Association for Artificial Intelligence, Austin, Texas, 1984.

BRACHMAN, R.J. and SCHMOLZE, J.G., **An Overview of the KLONE Knowledge Representation System,** COGNITIVE SCIENCE 9(2), 1985.

BREUKER, J. AND WIELINGA, B., **Use of Models in the Interpretation of Verbal Data,** in A.L. Kidd (Ed.)Knowledge Acquisition for Expert Systems: A Practical Handbook, PLENUM, New York, 1987.

CHANDRASEKARAN, B., **Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks,** Proceedings of Tenth International Joint Conference on Artificial Intelligence, Milan, Italy, 1987.

CLANCEY, W., **The Knowledge Level Reinterpreted: Modeling How Systems Interact,** MACHINE LEARNING, 4, 3/4 December 1989, pp. 285-293.

DAVIS, E., **Constraint Propagation with Interval Labels,** ARTIFICIAL INTELLIGENCE, 32, 1987, pp. 281-331.

ESHELMAN, L. and MCDERMOTT, J., **MOLE: A Knowledge Acquisition Tool That Uses Its Head,** Proceedings of the American Association for Artificial Intelligence, Philadelphia, PA., 1986.

FENSEL, D., ANGELE, J. and LANDES, D., **KARL: A Knowledge Acquisition and Representation Language,** in J.C. Rault (Ed.) Proceedings of the 11-th Intl. Conference on Expert Systems and Their Applications, Vol.1, Avignon, France, 1991, EC2, pp. 821-833,.

FICKES, R. and KEHLER, T., **The Role of Frame Based Representations in Reasoning,** COMMUNICATIONS OF THE ACM, 28, 1985.

FOX, M., **Constraint Directed Search: A Case Study of Job-Shop Scheduling,** Research Report CMU-RI-TR-83-22, Carnegie-Mellon University, 1983.

GAINES, B., **An Architecture for Integrated Knowledge Acquisition Systems,** Proceedings of 5-th AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Canada, November 1990.

GRUBER, T. and COHEN, T., **Principles of Design for Knowledge Acquisition,** Proceedings of IEEE Conference on Artificial Intelligence Applications, 1987.

JONKER, W. and SPEE, J.W., **Yet Another Formalization of KADS Conceptual Models,** in T. Wetter, K.D. Althoff, J. Boose, B. Gaines, M. Linster, F. Schmalhofer (Eds.) Current Developments in Knowledge Acquisition: EKAW 92, SPRINGER-VERLAG, 1992.

KARBACH, W., LINSTER, M. and VOSS, A., **Models of Problem- Solving in Knowledge-Based Systems: Their Focus on Knowledge Acquisition,** Proceedings of 5-th AAAI Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff, Canada, November 1990.

L. Kanal and V. Kumar (Eds.) Search in Artificial Intelligence, SPRINGER-VERLAG 1988.

KEENE, S., **Object-oriented Programming in COMMON LISP: A Programmer's Guide to CLOS,** ADDISON-WESLEY PUBLISHING COMPANY, Reading, MA., 1989.

KLINKER, G., BHOLA, C., DALLEMAGNE, G., MARQUES, D. and MCDERMOTT, J., **Usable and Reusable Programming Constructs**, Proceedings of 5-th AAAI Workshop on Knowledge Acquisition for Knowledge Based Systems, Banff, Canada, November 1990.

LUCK, K., NEBEL, B., PELTASON, C. and SCHMEIDEL, A., **The Anatomy of the Back System**, KIT Report 41, Fachbereich Informatik, Technische Universität Berlin, Berlin (West), 1987.

MARCUS, S. and MCDERMOTT, J., **SALT: A Knowledge Acquisition Language for Propose and Revise Systems**, ARTIFICIAL INTELLIGENCE, 39, 1989, pp. 1-37

MCALLESTER,D.,**Truth Maintenance**, Proceedings AAAI-90, pp. 1109-1116.

MCDERMOTT, J., **Preliminary Steps Towards a Taxonomy of Problem Solving Methods**, in S. Marcus (Ed.) Automating Knowledge Acquisition for Expert Systems, KLUWER ACADEMIC PUBLISHERS, 1988.

MCGREGOR, R. and BATES, R., **The Loom Knowledge Representation Language**, ISI IRS-87-188, USC/ISI, Marina del Rey, CA., 1987.
MCGREGOR, R. and BURSTEIN, M., **Using a Description Classifier to Enhance Knowledge Representation**, IEEE EXPERT, June 1991, pp. 41- 46.

MOTT, D. H., CUNNINGHAM, J., KELLEHER, G. and GADSDEN. J. A., **Constraint-based Reasoning for Generating Naval Flying Programs**, EXPERT SYSTEMS, Vol. 5, No. 3, August 1988, pp. 226- 246.

MUSEN, M.A., **Knowledge Acquisition at the Meta Level: Creation of Custom-Tailored Knowledge Acquisition Tools**, SIGART Newsletter 108, 1989.

NEBEL, B., **Computational Complexity of Terminological Reasoning in BACK**, ARTIFICIAL INTELLIGENCE, 34, 1988.

NEWELL, A., **The Knowledge Level**, ARTIFICIAL INTELLIGENCE, 18, 1982, pp. 87-127.

PATEL-SCHNEIDER, P.F., **Small Can be Beautiful in Knowledge Representation**, Proceedings IEEE Workshop on Principles of Knowledge Based Systems, Denver, Colorado, 1984.

SCHMOLZE, J.G., **The Language and Semantics of NIKL**, BBN Inc., Cambridge, MA., 1985.

SKUCE, D., SHENKANG, W. and BEAUVILLE, Y., **A Generic Knowledge Acquisition Environment for Conceptual and Ontological Analysis**, Proceedings of 4-th AAAI Workshop on Knowledge Acquisition for Knowledge Based Systems, Banff, Canada, November 1989.

STEFIK, M., **Planning with Constraints (Molgen: Part1)**, ARTIFICIAL INTELLIGENCE, 16, 1981, pp. 111-140.

SUSSMAN, G.J. and STEELE, G.L., **Constraints: A Language for Expressing Almost Hierarchical Descriptions**, ARTIFICIAL INTELLIGENCE, 14, 1, 1980, pp. 1-39.

TRAUSAN-MATU, S., BARBUCEANU, M. and GHICULETE, GH., **Designing a Flexible Constraint Propagation System,** submitted for publication.

VAN HARMELEN, F., BALDER, J. R.,ABEN, M.W.M.M. and AKKERMANS, J.M., **(ML)2: a Formal Language for KADS Models of Expertise**, Report KADS II/T1.2/TR, October 1991.

WALTHER, J., VOSS, A., LINSTER, M., HEMMANN, TH., VOSS, H. and KARBACH, W., **MoMo, Report GMD AI Division**, March 1992.

WETTER, T., **First Order Logic Foundation of the KADS Conceptual Model**, in B. Wielinga, J. Boose, B. Gaines, G. Schreiber and M. Someren (Eds.), Current Trends in Knowledge Acquisition, Amsterdam 1990, IOS, pp. 356-375.

WIELINGA, B.J., SCHREIBER, A.TH. and BREUKER, J.A., **KADS: A Modeling Approach to Knowledge Engineering**, Report KADS II/T1.1, University of Amsterdam, 1992.

WRIGHT, J.M. and FOX, M.S., **SRL 15 User Manual**, Tech. Rept. Carnegie Mellon University, Robotics Institute, 1984.

YEN, J., JUANG, H.L. and MCGREGOR, R., **Using Polymorphism to Improve Expert Systems Maintainability**, IEEE EXPERT, April 1991, pp. 48-55.