

# hyperRemora

**Maria Ruey-Yuan Lee\***  
**Bob Jansen\*\***  
**Carine Souveyet†**

Technical Report TR-FD-91-03  
August, 1991

\* CSIRO Division of Information Technology, P.O.Box 1599, North Ryde,  
AUSTRALIA, fax: +61-2-888-7787, e-mail: lee@syd.dit.csiro.au

\*\* CSIRO Division of Information Technology, P.O.Box 1599, North Ryde,  
AUSTRALIA, fax: +61-2-888-7787, e-mail: jansen@syd.dit.csiro.au

† Université de Paris 1 - Sorbonne, UFR 06, 17 rue de la Sorbonne, 75231 Paris Cedex  
05, France, e-mail: souveyet@masi.ibp.fr

## Abstract

This paper describes *hyperRemora*, a prototype software tool for knowledge systems design. The design and maintenance methodology supported by the tool is summarised, and then the tool itself is described in detail. The key feature of this tool is that it removes the problem of representational mismatch between knowledge structure and knowledge execution. An automatic transformation from a knowledge structure (which is suitable for domain experts) to one which is suitable for efficient execution is performed and creates the input/output behaviour of the system. Of particular interest is the demonstration of designing inference trees using hyperRemora.

**Keywords:** Knowledge Based Systems, REMORA, Database, Software Engineering

Paper presented at IJCAI-91 Software Engineering for Knowledge Based Systems Workshop, Sydney, 24 August, 1991.

## 1. Introduction

The art of constructing knowledge systems is both part of and an extension of the programming art. It is the art of building complex computer systems that represent and reason with knowledge (Hayes-Roth *et al* 1983). We argue that the application of computer software design criteria to knowledge systems could make the construction work easier. There are four main design criteria for computer systems that should be met:

- Effectiveness.
- Flexibility.
- Efficiency.
- Maintainability.

To achieve first two criteria, acquired knowledge should be based on a representation that is easy to understand. Further, this representation should be modifiable by a domain expert. To achieve the last two criteria, the execution of the knowledge base should be performed using a representation that is a perfect match between the structure of the knowledge and the execution of knowledge, and which lends itself to conceptually simple implementation. Manually producing the executable representation is prone to errors, while automatic translation between forms can be error free, and makes representation mismatch irrelevant.

HyperRemora is a prototype tool which facilitates the design and development of information and knowledge based systems. The use of a conceptual schema in the system is an extension of the REMORA methodology (Rolland & Richard 1982), developed by Université de PARIS 1 - Sorbonne<sup>1</sup>. The tool reduces the significance of representational mismatch between the initial knowledge structure and knowledge translated for execution. The representation of knowledge structure aids the effectiveness and flexibility of a system; the transformation of knowledge for execution leads to efficient systems. An automatic transformation from a knowledge structure (which is suitable for domain experts) to one which is suitable for efficient execution can be performed which preserves the input/output behaviour of the system.

The purpose of this paper is to present a coherent framework for the development of a method for the design and construction of knowledge based systems. The second section of this paper describes some of the problems which occur in knowledge based systems. Section 3 describes a model to overcome these problems and discusses the design life cycle of this model. Section 4 introduces the implementation of the model. Section 5 shows the application of hyperRemora in designing inference trees. It gives graphical representations of dynamic specifications and static specifications of inference tree structures. It also describes a functional prototype of the specifications, code generation for automatically transferring the specifications into computer code, and a user friendly interface.

## 2. Problems

Knowledge engineering has proven to be difficult (Waterman 1986). Major difficulties are involved in acquiring knowledge from domain experts (Hayes-Roty *et al* 1983). One of the main difficulties in knowledge acquisition is representation mismatch: the difference between the way a domain expert normally states knowledge and the way it should be represented in a program. To decrease the representation mismatch between the domain expert and the program under development, knowledge acquisition should be based on a representation as close as possible to the way domain experts think about a problem.

---

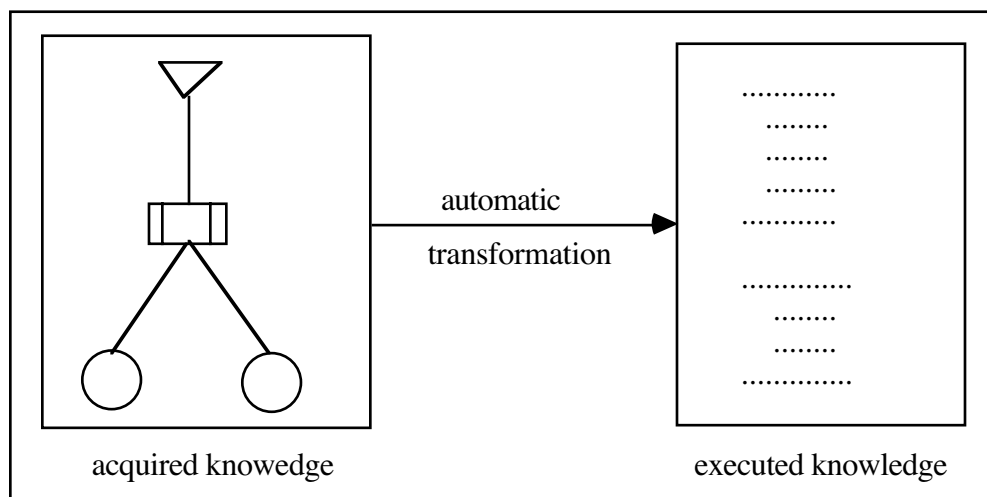
<sup>1</sup> HyperRemora has been developed as a collaboration between the CSIRO Division of Information Technology and the Université de PARIS 1 - Sorbonne.

A regular occurrence is for a knowledge engineer to build a representation that matches domain expert requirements, and then completely reengineer the system for 'runtime' execution<sup>2</sup>. This is done because the structure of knowledge as supplied by the expert is not necessarily computationally efficient. However, a computationally efficient system may not be understandable to the expert. The manual transformation from a knowledge structure which is suitable for the domain expert to one which is suitable for efficient execution seems to be a weak link in the chain since it allows for the introduction of errors. To avoid introducing errors, the transformation should be automatic and preserve the input/output behaviour of the system.

Recently, the designers of knowledge system have used a host of new tools or shells for building knowledge based systems. The tools are typically large and complex, requiring a major amount of time and effort to learn and use. Even experienced developers may feel that some of these tools are not easily understood and lack flexibility (Rothenberg 1989). The designed models are often inadequate in representing the structural and functional features/aspects of a domain, so the interface provided by the tool is not very good for communicating these features to users. To improve the communication ability, an automatic transformation from a more "natural" structure of knowledge should be preferred.

### 3. The Proposed Model

We propose a model that automates the translation from the acquired knowledge structure to the execution structure and thereby avoids errors arising from representational mismatch. Both designers and users can easily understand and have flexible access to the system, but from different perspectives. An automatic transformation between the two representations is made generating runtime code and the (graphical) human interface for the system. The code for the system execution is automatically installed into system components. The result obtained from this is that the system should correctly perform the actions expected by the user, because the representation is accurate and its transformation is efficient. Figure 1 shows the automatic transformation from the acquired knowledge to the executed knowledge in the model.



**Figure 1.** The proposed model with an automatic transformation from acquired knowledge to executed knowledge.

<sup>2</sup> Debenham 1989 recommends three stages of re-engineering, one to "normalize" the model, one to reduce computational complexity, and one for memory space/execution speed optimization.

### **3.1 Software life cycle of the proposed model**

As a knowledge system is a software product, it is natural to propose a software life cycle for it. Our definition of the four phases of the life cycle of software are:

*Phase 1* - Identifying the functional and structural specifications.

*Phase 2* - Functional prototype construction.

*Phase 3* - Code generation and installation.

*Phase 4* - Operation, maintenance and execution.

The first two phases produce the representation of acquired knowledge. The last two phases result in a representation of knowledge which can be executed.

#### **3.1.1 Identification of functional and structural specifications**

The architecture behind the identification of functional and structural specifications comes from consideration of the division of a system into useful modules and integration of those modules to yield a whole. The fundamental concept is that a knowledge system can be considered a *finite-state machine* (Lewis *et al* 1976). Since the simulation of a finite-state machine requires only few operations to process an individual input, it operates rapidly. Generally a module should be small enough that a knowledge engineer or domain expert can understand it fully.

Components of each module are :

1. a finite set of inputs.
2. a finite set of states.
3. a dynamic transition function which assigns a new state to every combination of state and input.

We hypothesise that a rule is an association of output states with input states and is therefore a component of three. Firing a rule implements a dynamic transition. A finite number of rules will imply a finite number of possible states.

The conceptual schema of modules represents the domain structure and function. Graphical interfaces make the structure and function of the module available to the user for inspection, extension and modification. The definitions of the logical relationships between modules (precedences, conditions, constraints, etc) are explicit.

#### **3.1.2 Functional prototype**

The main goal of a functional prototype is to validate and refine the conceptual schema. The principle task of the functional prototype is to produce a simple mapping from components in order to control and manage the execution sequence of components. The prototype mechanism gives a flexible representation, and highlights the graphical component which is executed to provide 'execution animation' for the user during the validating process.

#### **3.1.3 Code generation**

The code generation facility is used to build the prototype system. The principle of code generation is to transfer the dynamic transition schemas into an operational environment. To ensure that the execution functions won't corrupt the conceptual schema, the executed knowledge and the conceptual schema are separated. The definition of the representation can be systematically transferred into computer code for system execution, thus aiding the maintenance of the system and providing a ready

means to re-use software. Code generation provides an interactive function for automatically generating input interfaces for acquiring knowledge from the experts.

### 3.1.4 Execution

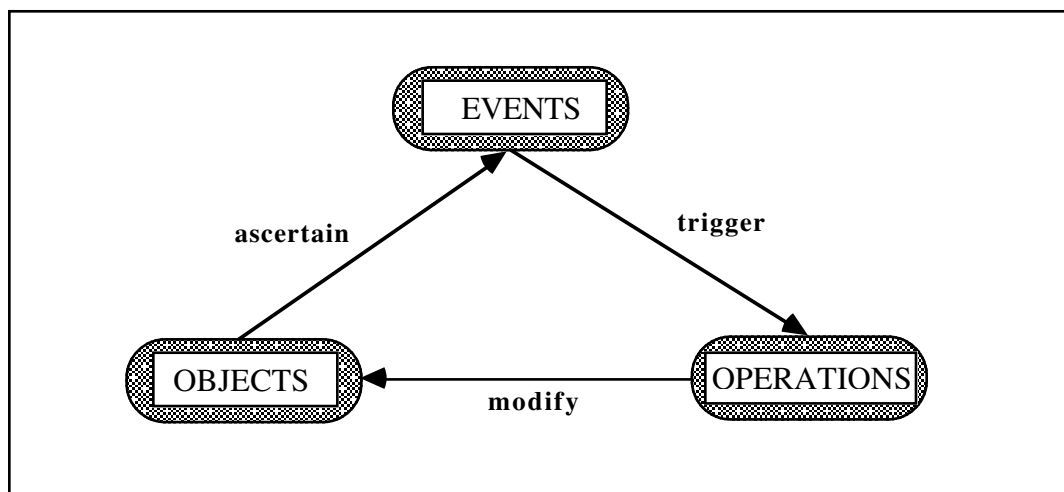
The final phase drives the system execution. It supports system use and operation, monitors system performance, and stores data.

## 4. hyperRemora - The Implementation of the proposed model

We have carried out initial experiments to test and validate this approach in Hypercard (Apple 1987). HyperRemora is a prototype tool for information and knowledge based systems design and development. The conceptual schema of hyperRemora is an extension of the REMORA methodology (Rolland & Richard 1982).

### 4.1 REMORA Methodology

The REMORA methodology provides static and dynamic aspects to describe a data base application. Static aspects represent the structure of an application which are modelled by **objects**; dynamic aspects are modelled by representing the system as a set of dynamic transition relationships - **events, operations** and **objects**; Those components are acted upon by the behaviour a phenomena of **triggering** (event to operation), **modifying** (operation to object) and **ascertaining** (object to event). Figure 2 shows a causal chain between system components and their actions.



**Figure 2.** A causal chain between system components and their actions (adapted from Rolland & Richard 1982).

A triggering action indicates that an event causes the execution of one or more operations. A modifying action is an operation which modifies one or more objects. An ascertain action specifies a particular state change in an object, and this change could drive an event.

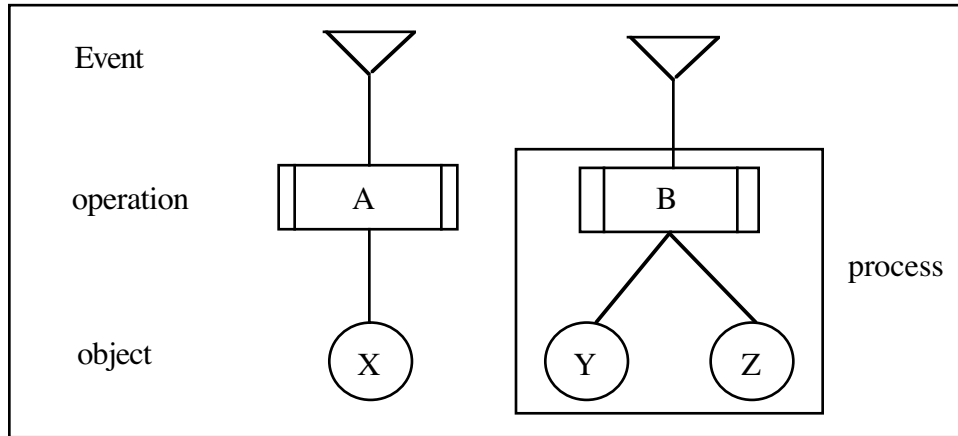
#### Events

An event represents a stimulus to a system. When an event arrives, a **predicate** checks the event, and determines whether or not it should affect a dynamic transition. If the event situation is satisfied, the the event could trigger operations. A distinction is made between an **external event** (which represents an external message which arrives from outside), an **internal event** (which represents elementary state changes of objects), and

a **temporal event** (a time event). An external event **structure** describes the external values associated with the external message.

### Operations

An **operation** is an action that can be executed at a given time and modify objects. An operation can only alter (write) its 'own' objects but has the ability to get information (read) from unrelated objects. Figure 3 shows the relationship between operations and objects. In the two dynamic transitions shown, operation\_B has both read-access and write-access to object\_Y and object\_Z, but only read-access to object\_X.



**Figure 3.** A dynamic transition and the relationship between operations and objects.

If an operation can modify the state change of more than one object, then the operation and objects form a **'process'**, as in the second dynamic transition in figure 3. The concept of a process is an extension of REMORA methodology. The change has been motivated by the need to express the execution order between operations, which are not available in the original REMORA methodology.

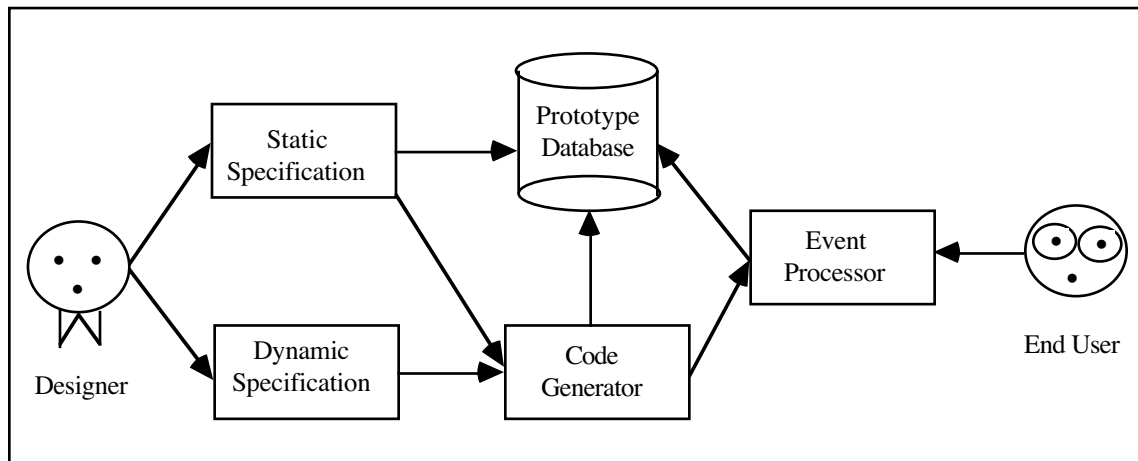
An operation **condition** describes the triggering function of an operation. If the condition is true, then the operation executes. A **factor** represents the iterative process of an operation. A factor determines the number of times that the operation must be executed on object instances.

### Objects

The static aspects are modelled using objects. An **object** represents entities and attributes in an relational model (Codd 1970).

## **4.2 The structure of hyperRemora**

HyperRemora assists a knowledge engineer in designing and building a knowledge based system by using abstracted representations to define the rule structure. It supports designers in specifying static (structural) and dynamic (behavioural) aspects of a system. The structure of hyperRemora is shown in Figure 4.



**Figure 4.** The structure of hyperRemora.

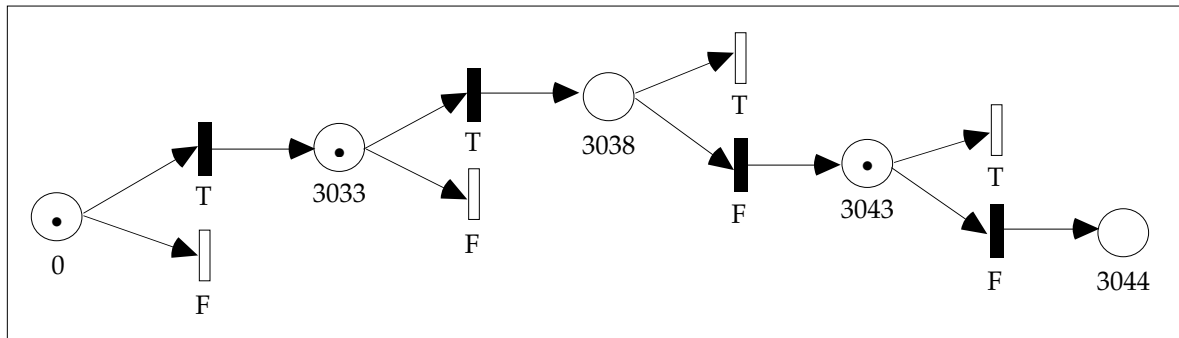
Static specification uses a semantic network representation (Quillian 1968) for defining the structure of objects. A prototype database is then automatically created from the definition of objects. Dynamic specifications represent dynamic transitions, which consists of events with all the operations triggered by those events and all the associated objects which are modified by those operations. Static and dynamic components can be input to a code generator which automatically generates fully functional code for system execution and creates the input/output interface. The execution is controlled by an event processor.

## 5. An Application of hyperRemora - Inference Tree Design

The prototype tool has been applied to the process of designing an inference tree, based on *ripple down rules*, a minimal context based knowledge acquisition system (Compton & Jansen 1990, Compton & Preston 1990). With ripple down rules, rules can be entered into an expert system exactly as provided by the expert. Those rules will only be tested after all existing knowledge in the knowledge base has been considered (Compton & Jansen 1990). One main advantage of using ripple down rules is that since the rules are inserted without modification of the existing knowledge, rules are added to a knowledge base far faster than in a conventional rule based system.

The structure of ripple down rules is a binary tree. Each rule is a node of the tree with two branches depending on whether the rule is satisfied or not by the data being considered. The insertion of any new rule creates a new leaf node attached to the existing leaf node at which the expert system previously ended, i.e. knowledge is added to the last level in the order of rule firing. Figure 5 shows a petri net (Peterson 1981) view<sup>3</sup> of a rule tree. In the diagram, a circle represents a numbered rule; a bar shows two branches of the binary tree; a black bar indicates the inference path, and a token (•) marks a fired rule.

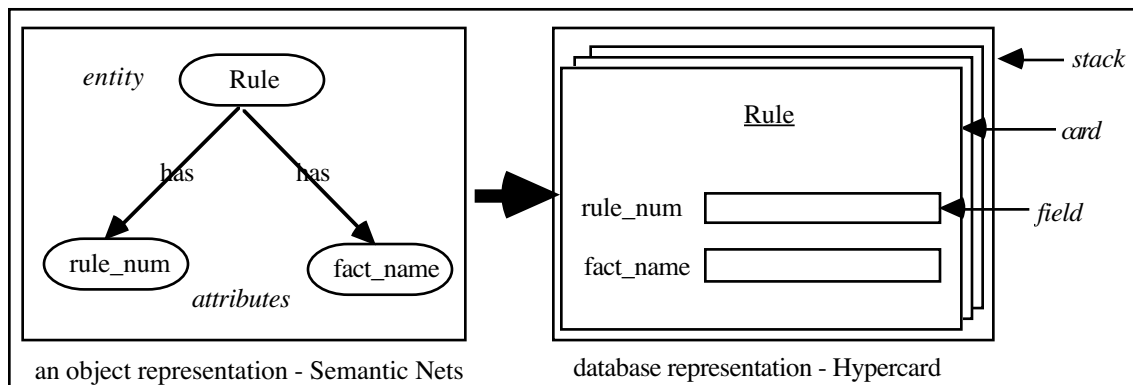
<sup>3</sup> The authors have carried a visualisation interface to expert systems based on the Petri net modelling. The interface is distinctive in that it enables the inference tree to be viewed from two separate objectives - rule-based and data-based.



**Figure 5.** A petri net graph of a rule tree.

### 5.1 Static Specification of Inference Tree Design in hyperRemora

The structure of an object is defined by a semantic net graph, and is a static aspect of hyperRemora. An object represents a rule, which in our hypothesis, can be considered as defining a dynamic transition in a finite-state machine. Hence, using a finite number of rules results in a finite state machine. A semantic net uses nodes to refer to objects and arrows to express directed relationships. The structure of an object then provides an interactive framework for creating a relational database table interface. The prototype database is implemented in Hypercard. An entity object is represented by a *stack* in Hypercard, with a two dimensional storage for data. An attribute is represented by a *field* in Hypercard. A *card* stores a rule and its associated facts. Figure 6 is an example of transforming an object which represents a semantic net graph into a Hypercard data base.

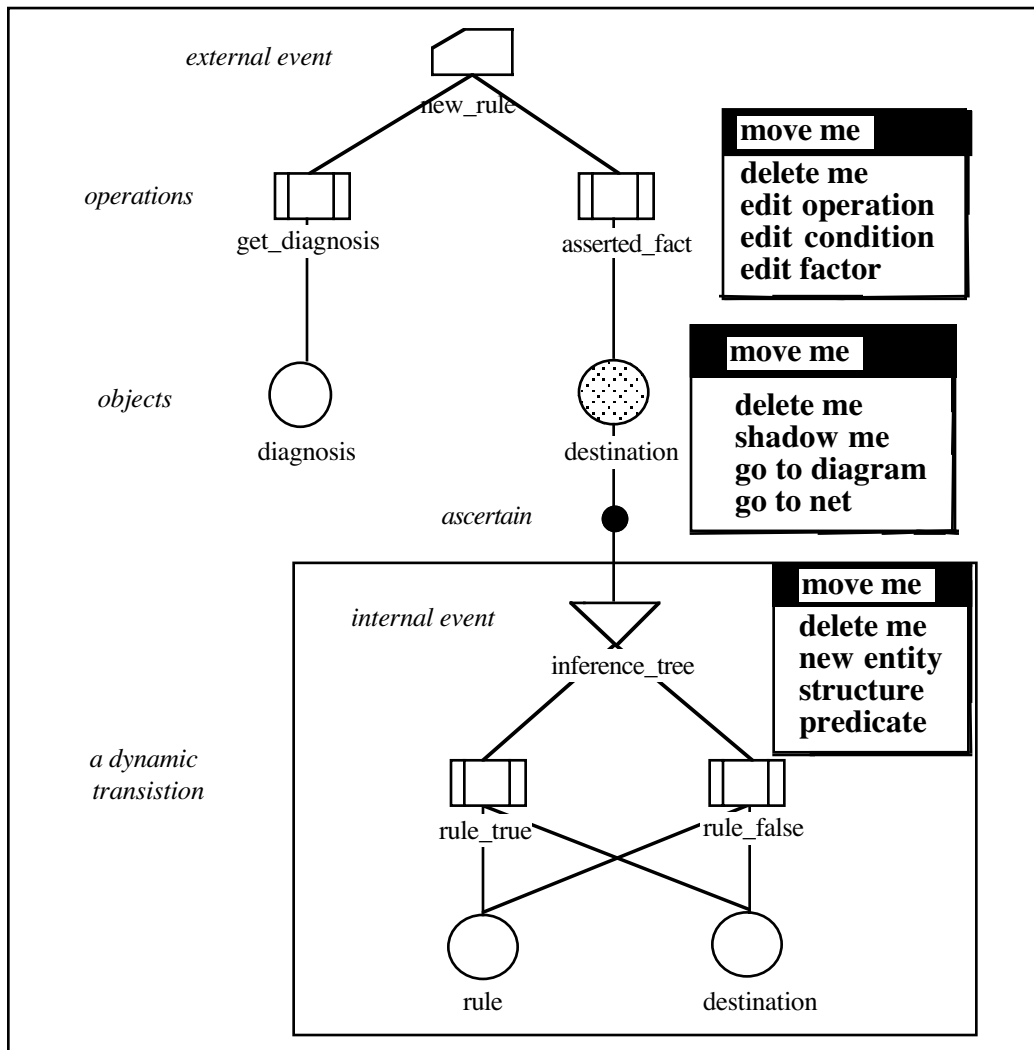


**Figure 6.** Transform a semantic net graph into Hypercard data base.

The database table not only stores data but also stores the executable code. A number of SQL-like functions manage i/o access to the relational database. Message inheritance is used for passing messages between different database tables and executing operations.

### 5.2 Dynamic specification of inference tree design in hyperRemora

The dynamic specification describes the data processing of a system. The graphical representation of an inference tree design in hyperRemora is shown in Figure 7. This representation introduces dynamic transitions and shows their predecessor and/or successor. In the diagram, a circled shadow pointer is used to link two dynamic transitions; a punched card represents an external event; a circle represents an object; a triangle represents an internal event; a rectangle represents operations; a dot represents an ascertain; and the pop-up menus show the association between different components.



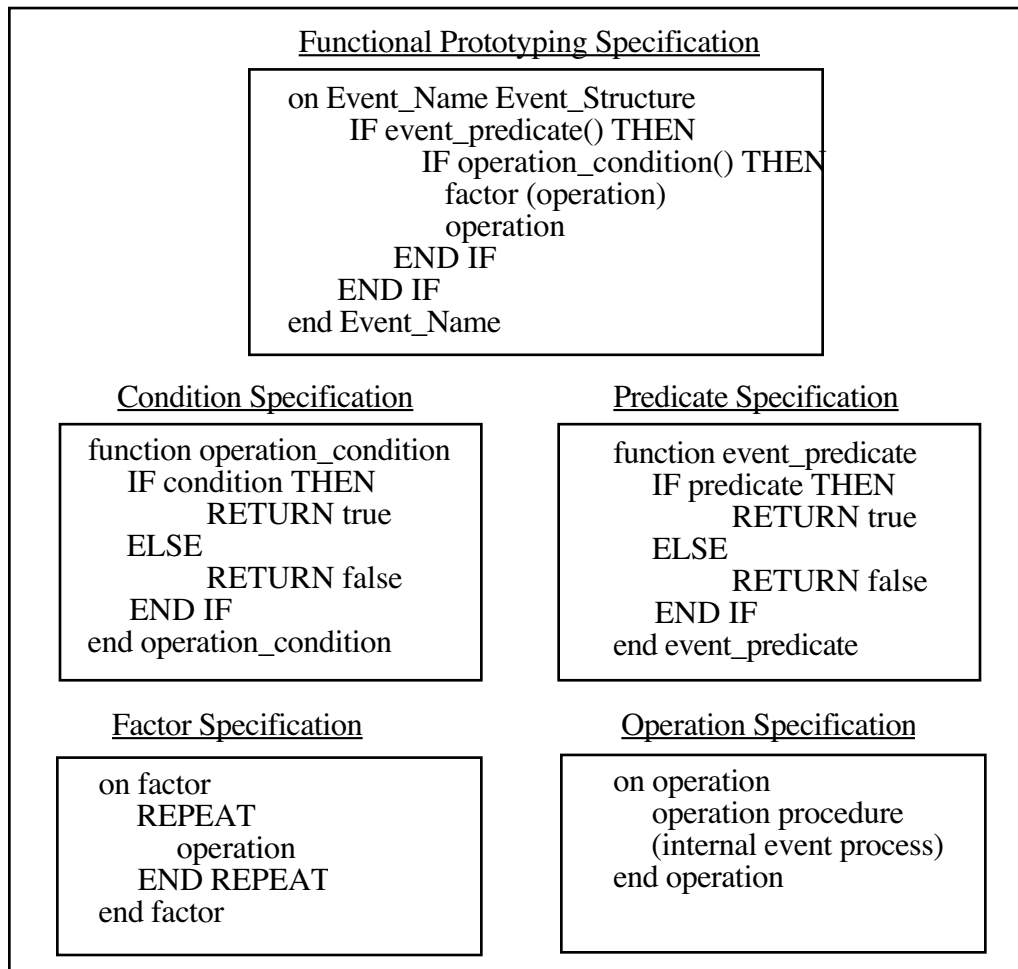
**Figure 7.** Graphical representation of inference tree design in hyperRemora.

The *external event* 'new rule' adds a new rule to a system. An *event predicate* determines the acceptance of this external message. If it is acceptable then the external event triggers operations. The *operations* are the actions of asserting facts and getting interpretations. Those actions then modify *objects*. The object 'diagnosis' and object 'destination' are associated with the external event. The object 'diagnosis' has a rule number and an interpretation. The object 'destination' keeps a record of its parent node and its two branches - true or false.

The *ascertain* action specifies a state change in object 'destination', and that this change could drive an *internal event* 'inference tree'. The *operation conditions* determine whether the rule falls into the true branch or false branch; then update object 'rule' and object 'destination'. The object 'rule' stores a rule number and its associated fact names.

### 5.3 Functional Prototype

The functional prototype is available for validating the conceptual schema. An automatic transformation from conceptual schema to handlers or functions (which are Hypertalk concepts for program execution and message passing) is shown in figure 8.



**Figure 8.** Functional prototype specifications.

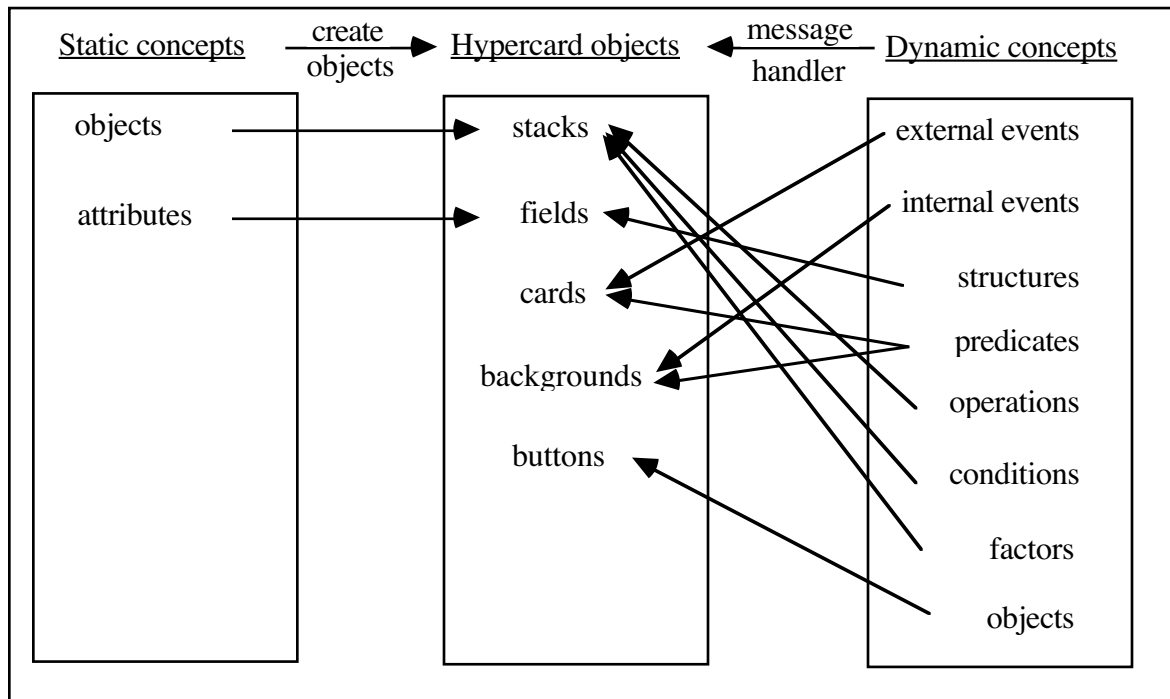
## 5.4 Code Generation

The principle task of code generation is to systematically transfer the dynamic transition specifications into corresponding Hypercard objects: buttons, fields, cards, backgrounds and stacks. Hypercard is activated by using Hypertalk to control the actions of Hypercard objects. Hypertalk can send messages to and from Hypercard objects. How a given objects responds to a particular message depends on its script. The three concepts of objects, messages and scripts are essential to a Hypercard system.

The message passing order in Hypercard is:

button or field→card→background→stack→home→Hypercard.

Part of the code generation process is to find the relationship and links between conceptual objects and Hypercard objects. Figure 9 shows the mapping between conceptual objects and the Hypercard environment.



**Figure 9.** The relationship links between conceptual objects and Hypercard objects.

The mapping is as:

- event structure creates fields for input interface.
- external events are unique messages coming from the real world. The scripts of external events are put in the card level, which is the basic unit of storing information.
- internal events may be shared by more than one external event. The scripts of internal events are put in the background level.
- operations, conditions and factors can be reused. The scripts are in stacks.

The static concepts create the layout of the generated application, while the dynamic concepts specify corresponding message handlers. To ensure that the execution functions won't corrupt the conceptual schema, the executed system and the conceptual schema are separated.

### 5.5 Event Processor

The event processor handles the execution of hyperRemora. It presents the input and output interfaces. The input layout for end-users is automatically created by the definition of an external event *structure*. The input interface of the external event "new rule" is shown in figure 10. The user sends external values from the input interface. The message handler controls the execution flows associated with the event and handles the internal events and their synchronization. Results are stored in the database and presented through the output interface. These interfaces overlap each other to a large extent.

*New\_Rule*

**name** .....

**age** .....


**sex** .....

**weight** .....

**height** .....

**interpretation** .....

*Please Enter Above Information*



**Figure 10** The input interface of an external event in hyperRemora.

In the future a number of extensions to hyperRemora will be implemented. These include:

- use hyperRemora to re-build the Garvan ES-1 expert system (Horn *et al* 1985), a medical expert system for thyroid function tests. The Garvan ES-1 expert system contains 661 rules, 216 facts and 144 interpretations (Lee 1990).
- provide a checking module for validating and modifying the conceptual schema (Souveyet 1990).
- apply artificial intelligence techniques to assist designers in the specification process, such as a reasoning incomplete specification mechanism.
- implement the tool in an object-oriented environment.
- support more fully the generation of Hypercard applications.

## 6. Conclusion

We have shown that hyperRemora provides representations that give function and structure to knowledge, is flexible and is accessible to a domain expert. The definition of the knowledge representations can be systematically translated into computer code and created input/output interfaces. The executed knowledge is maintainable and reusable. The automatic translation between forms can be error free, and makes representation mismatch irrelevant.

## Acknowledgements

We wish to thank the following people for their active reading and commenting of this paper.

Craig Lindley of the CSIRO Division of Information Technology.

Phil Preston and Tim Menzies of the University of New South Wales.

## References

Apple Technical Publications, *Hypercard Script Language Guide : The HyperTalk Language*, 1987.

Codd EF, *A Relational Model for Data for Large Shared Data Banks*,

Communications of the ACM, Vol. 13, No. 6, 1970

Compton, P. & Jansen, R. *A Philosophical Basis for Knowledge Acquisition*, Knowledge Acquisition, Volume 2, Number 3, 1990, pp.241-259.

Compton, P. & Preston, P. *A Minimal Context Based Knowledge Acquisition System*, Proceedings of the AAAI Knowledge Acquisition Workshop, 1990.

Debenham, J. *Knowledge Systems Design*, Prentice Hall (Pub), 1989.

Giarratano, J. & Riley, G. *Expert Systems Principles and Programming*, Pws-Kent Company (Pub), 1989.

Guida, G. & Tasso, C. *Building Expert Systems : From Life Cycle to Development Methodology*, in Topics in Expert System Design, North-Holland (Pub), 1989, pp.3-24.

Hayes-Roth, F., Waterman, D., & Lenat, D. (ed), *Building Expert Systems*, Addison-Wiley Publishing Company (Pub), 1983.

Horn, K.A., Compton, P., Lazarus, L. & Quinlan, R. *An Expert System for the Interpretation of Thyroid Assays in a Clinical Laboratory*, The Australian Computer Journal, Volume 17, No 1, February, 1985.

Lee, M. *The implementation of a Knowledge Dictionary in SQL*, Proceedings of the ORACLE Asia/Pacific Users Conference, 1990.

Lewis, P., Rosenkrantz, R., & Stearns, R. *Compiler Design Theory*, Addison-Wesley Publishing Company (Pub), 1976.

Peterson, J.L. *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, 1981.

Quillian, R. *Semantic Memory*, Semantic Information Processing, ed. by M. Minsky, The MIT Press, pp. 227-270, 1968.

Rolland, C. & Richard, C. *The Remora Methodology for Information Systems Design and Management*, IFIP, 1982, pp.369-426.

Rothenberg, J. *Expert System Tool Evaluation*, in Topics in Expert System Design, North-Holland (Pub), 1989, pp.205-229.

Souveyet, C. & Rolland, C. *A Workbench for An Information System Design*, 1990.

Waterman, D. *A Guide to Expert Systems*, Addison-Wesley Publishing Company (Pub), 1986.