Hybrid AI Development Tools

Juha Kuusela
Esko Nuutila
Helsinki University of Technology
Laboratory of Information Processing Science

Abstract - This article describes hybrid AI tools. The basic idea of hybrid tools is explained and some examples of existing commercial hybrid AI tools are given. An experimental hybrid tool [1] developed by the authors is described in more detail. Finally the whole idea of hybrid tools is evaluated.

1 Hybrid systems

Traditionally AI programs are based on a single methodology, such as frames, rules or logic programming. Furthermore every methodology is claimed to be natural, efficient and uniform. There is, however, strong evidence that no single methodology is appropriate for all applications, and it is believed that naturalness, efficiency and flexibility are all increased by trading uniformity for the power that is provided by combining a small set of different methodologies in the same AI tool /7/. Such 'hybrid' AI tools like ART /1/, KEE /6/ and Loops /10/ have become increasingly popular in the expert system development.

Usually a hybrid tool combines some frame based knowledge representation, rule based reasoning and a procedural language. The commercial products also have highly sophisticated graphic based user interfaces which include interactive facilities for building and examining the knowledge base of the application and monitoring the program execution.

There are several possibilities for combining the different methodologies. In ART the emphasis is on the fast execution of a powerful production rule language. The execution

[1] This work was done at Nokia Electronics and it was partially supported by TEKES

is based on the RETE matching algorithm /4/, so ART is basically a forward chaining system, but backward chaining is also possible. ART has a frame-like knowledge representation language sugared on top of the working memory elements of the production system. Knowledge can also be directly represented with the working memory elements. LISP functions can be called from the condition and action parts of the rules. The top level of the program is always the production system.

In KEE frame based knowledge representation, rule based reasoning, LISP, interactive graphics and active values (procedures or sets of rules attached to slots of frames that are invoked when the slots values are accessed or stored) are integrated into a single representation facility. The frame system acts as the unifying concept. The components of KEE itself are represented as frames that the user may access. That makes the system extendable and modifiable.

Loops builds on the function-oriented programming of LISP and adds paradigms of rule based, object-oriented and access-oriented programming. The access-oriented component contains active values and so called property annotations.

The combination of different AI techniques in single tool is a hard problem that is not entirely solved in none of the abovementioned tools. We will discuss the pros and cons of these solutions later, but first we will describe yet another hybrid tool called ET.

2 Description of the ET system

ET is hybrid AI tool that combines a semantic network -based knowledge representation, a powerful production system language and procedural programming in LISP. Because ET is an experimental tool it contains no sophisticated graphical user interface and there is no support for development of graphical user interfaces for the application programs.

The production system component and the semantic network component were implemented as two separate programs with a well defined interface. This approach was taken in order to make the components as independent as possible so that they could easily be documented as our master theses /8,9/.

The main design goals were power of expression and efficiency. The basic structure of ET resembles that of ART. We wanted ET to be faster than ART and yet have the same expression power. This goal was achieved.

Following sections describe informally with examples the main components of ET and the interface between the components.

2.1 SNET

The knowledge representation language of ET is called SNET. Objects and relation instances are the basic entities of SNET language. Objects are unique named entities that can be members of relation instances. Relation instances are n-tupples of objects or relation instances. Size n and other features of the members of the tupple and side effects of instantation can be defined in relation definition. Relation instances form a set, because system prevents instantation if a similar instance already exists.

Example of a relation instance:

(color :of Clyde :is grey)

A frame like structure is sugared on top of this basic mechanism. On this level objects are divided in concepts and instances which provide convinient means to declare concept hierarchies and put all properties of an object to a single place. Same functionality can be achieved using directly predefined relations *'is-a'* and *'instance-of'*.

Example of concepts and an instance:

```
(defconcept mammal
  relations
    (is-a :who self :what concept))

(defconcept elephant
  relations
    (is-a :who self :what mammal)
  instance-relations
    (color :of instance :is grey)
    (location :of instance :is Africa))

(definstance Clyde
  instance-of
    elephant
  relations
    (location :of self :is India))
```

Concept hierarchies like the one shown in example become useful if they allow the definition of default properties. SNET allows objects to directly or indirectly inherit relation instances. How to inherit is defined in relation definition, it is not only a special property of *is-a* and *instance-of* relations.

In the example the concept elephant inherits relation instance *(is-a :who elephant :what concept)* from mammal via relation instance *(is-a :who elephant :what mammal)*. This type of inheritance is called direct because the inherited relation instance already exists.

After evaluation of the definitions in our example relation instances in which Clyde is a member are:

1. *(color :of Clyde :is grey)*
2. *(location :of Clyde :is India)*

First of them is an example indirect inheritance. It was inherited via relation instance *(instance-of :who Clyde :what elephant)*, but relation instance *(color :of elephant :is grey)* doesn't exists. Instead there exists relation instances:

a) *(instance-relation :of elephant*
:is (color :of instance :is grey))

b) *(instance-relation :of elephant*
:is (location :of instance :is Africa))

So indirect inheritance means instantation of a relation list that is member in the indirectly inherited relation instance. Why was relation instance *(location :of Clyde :is Africa)* not inherited? Actually it was, but it was overridden by relation instance *(location :of Clyde :is India)*, which was defined within the description of *Clyde*.

There is two ways to refer to relations. The existence of an relation instance can be checked out by function *relation-exists* and a relation instance member can be defined to be function of other members. This allows not only LISP code but also relation instances themselves to refer to existing relation instances. This refence is implemented so that it allows both direct and indirect reference to function value. For example let us consider statement: "Location of Bill is the same as location of Clyde". This can be interpreted either directly as in statement "Location of Bill is India" or indirectly as in statement "Location of Bill is same as location of Clyde even if location of Clyde has changed".

Relation instances are the basic elements of SNET language and consequently relation definition most complicated language feature. For example let us consider definition of relation *color*:

```
(defrelation color
  actors
    (of :single-p t)
    (is :resriction (relation-exists (is-a :who is :what color)))
  function-p
    is)
```

to be used like *(color :of Clyde :is gray)*.

This definition restricts the set of legal first member for instance of color relation by stating that every object can be member in only one color relation (*single-p*). It also restricts the set of legal second member to the set of colors. After first restriction it is safe to define color to be function of an object (*function-p is*).

Connection between different parts in a hybrid AI tool is crucial. In ET connection between SNET and EPSL is clear: relation instances in SNET are data elements in EPSL. Every access to data is made via SNET.

2.2 EPSL

The production system language component of ET is called EPSL. EPSL is basically a forward chaining system, but backward chaining may also be used as in ART.

As usually, a production system written in EPSL consists of a working memory and a rule memory.

The working memory contains a set of data elements that are arbitrarily nested LISP lists of atoms, integers and floating-point numbers. The semantics of the data elements are defined purely by the rules that use them. Thus the user may structure data elements as he (currently there are only male EPSL users) likes. When EPSL is used with SNET the data elements have the same structure as SNET relation instances (except that in SNET the relation instances are implemented as arrays).

Examples of data elements:

```
(goal (has monkey bananas))
(= (* 2 3) 6)
(color :of clyde :is gray)
```

The rule memory contains an arbitrary set of forward chaining and backward chaining rules.

A forward chaining rule consist of the condition part and the action part. The condition part is a logical expression of patterns and LISP tests. The logical operators are *not*, *and*, *or*,*exists* and *forall*. LISP tests are arbitrary LISP expressions that have no side-effects.

Patterns are descriptions of data elements contained in working memory. Structurally they are like data elements, but a number of special constructions may be used to form a pattern that describes many data elements. The special constructions include variable bindings, calls to LISP functions and various constructions for representing alternatives and repetitions.

The action part of a rule is a sequence of LISP expressions. The variables that were bound in the condition part may be refered in these expressions.

Here is an example of a forward chaining rule:

```
(defrule monkey-reaches-bananas
  (height bananas ?h)
  (holds monkey ?x)
  (OR (AND (ladder ?x)
           (height ?x ?lh)
           (TEST ( >= ?lh ?h))
      (AND (stick ?x)
           (length ?x ?sl)
           (TEST ( >= ?sl ?h)))
   =>
  (assert '(monkey reaches bananas)))
```

A backward chaining rule consists of the goal part and the condition part. The goal part is a pattern that describes a data element which can be derived using the rule. The condition part is similar to that of a forward chaining rule.

Here is an example of a backward chaining rule:

```
(defrule current
```

```
(current ?+x ?-i)
<=
(voltage ?x ?v)
(resistance ?x ?r)
(BIND ?v (* ?i ?r)))
```

The execution of an EPSL production system happens in the same kind of match - conflict resolution - action -loop as in OPS5.

In the match phase the working memory is searched for rule instances i.e. sets of data elements that satisfy the condition part of some rule. If there are no rule instances the execution stops. If there are more than one rule instance the conflict resolution strategy is used to find out the most appropriate rule instance. In the action phase the chosen rule instance is executed.

EPSL has three predefined conflict resolution strategies: the LEX and MEA strategies of OPS5 and a priority based strategy. A new strategy can easily be implemented by writing a LISP function that compares rule instances.

Backward chaining rules are executed on demand when there is no data element that satisfies a pattern in the condition part of a forward or backward chaining rule and the missing data element can be derived with a backward chaining rule. Thus backward chaining rules are principally used for fetching data that is needed in forward chaining.

Efficiency was one of the main goals of ET project. The RETE matching algorithm that was originally developed for the fast execution of OPS -family of production systems is considered currently the fastest way to implement the match phase of production system execution. Because all the logical operators and the LISP tests that are included in EPSL can't be handled in the original algorithm we had to extend it. An efficient execution of EPSL production systems is thus achieved by using the extended RETE algorithm and by compiling the RETE discrimination network into LISP functions.

3 Discussion

In this chapter we describe some problems in hybrid AI tools and their components. We also discuss the validity of the motivations behind hybrid AI tools.

The frame based component of a hybrid tool is designated to the representation of structural objects in an application. The problem is that all kinds of data structures cannot be efficiently represented with frames. Frames were initially developed for the representation of natural language statements, usually in some 'micro world'. Thus the representation of geometrical data structures, for example, is very inefficient with frames. Other kinds of structural data representation facilities must be used too, but this is not easy in any of the hybrid AI tools we know.

It is claimed that two of the main strengths of rule based systems are modularity and the possibility of incremental development /2,5/. We think that this is not the case.

In conventional programming modularity means decomposition of programs into relatively independent pieces that interact in a explicitely specified manner. This approach makes it possible to develop and maintain large systems. Key issue is that the modules can be implemented and tested independently. Rule based systems are however not modular in this sense. Usually they consist of one linear set of rules that communicate via global database. It is hard to believe that in any real application the rules could be totally independent. Furthermore, the interaction between different 'modules' of the program i.e. rules is not defined. Thus the rules cannot be implemented or tested independently. Every new rule can affect the behavior of the whole system, so the system cannot be developed incrementally.

The initial motivation for the development of hybrid AI tools was the observation that no single AI methodology is suitable for all applications. We agree in that because methods that are designed to be general are weak by their nature. As Feigenbaum states /3/, the search for general knowledge representations although apparently intuitively desirable, has no empirical justification. The key point is that this statement is not restricted to single methodologies. It is true for any combination of separate tools. Therefore a hybrid AI tool like ART, KEE or ET, that aims in generality will be inefficient no matter how well it is implemented.

Our conclusion is that another kind of approach to the development of knowledge based AI systems would be more fruitful. Instead of trying to develop general purpose facilities, tools that help the creation of special purpose knowledge representation facilities for specific applications should be designed. Special purpose knowledge representation facilities would be more natural and more efficient in their specific applications and with good tools their development would be a minor task compared to the acquisition and representation of the domain knowledge.

References

/1/ ART Reference Manual, Inference Corporation, Los Angeles, CA, 1985.

/2/ Barr, A., Feigenbaum, E.A. editors, The Handbook of Artificial Intelligence, vol.I, William Kaufmann, Inc., Los Altos, CA, 1981, pp. 193-194.

/3/ Feigenbaum, E.A. The art of artificial intelligence: Themes and case studies of knowledge engineering, IJCAI 5, 1977, pp. 1014-1029.

/4/ Forgy, C.L. Rete: A fast algorithm for the many pattern/many object pattern match problem, Artificial Intelligence, 19, Sep. 1982.

/5/ Hayes-Roth, F. Rule-based Systems, Comm. ACM, vol.28, no.9, Sep. 1985, pp. 921-931.

/6/ KEE Software Development System User's Manual, Intellicorp, Jan. 1985.

/7/ Kunz, J.C., Kehler, T.P., and Williams, M.D. Applications development using a hybrid AI development system, The AI Magazine, vol.5., no.3., Fall 1984, pp. 41-54.

/8/ Kuusela, J. Tietämyksen esittämiskieli ja sen liittyminen asiantuntijajärjestelmien rakentamistyökaluun, diplomityö, Otaniemi, 1986.

/9/ Nuutila, E. Asiantuntijajärjestelmien rakentamistyökalun sääntöpohjainen päättelymekanismi, diplomityö, Otaniemi, 1985.

/10/ Stefik, M.J., Bobrow, D.G. and Kahn, K.M. Integrating access-oriented programming into a multiparadigm environment, IEEE Software, Jan. 1986.