ROUGH OUTLINE OF A DATA STRUCTURE

William Bricken
May 1985

These ideas are derived from a declarative approach. An object oriented representation is easily converted into the declarative data structures.

NOTE: no domain accuracy here, just representation ideas.

SOME ADVANTAGES

1. Connects directly to inference engines (MRS, YAPS, LOSP); no process code necessary. Easy to integrate production rules.

2. Easy to understand; facilitates explanation.

3. Facilitates formal analysis; consists of objects, relations, and logic.

4. Easy to conceptualize analysis processes such as unification, hierarchical interfaces, constraints, hierarchical planning.

BASIC IDEAS

1. Spend most of the programming effort in describing the domain; let a simple, generic inference engine do the processing work.

2. One representation for all processing tasks.

3. Define primitive elements and their (primitive) relations. Build all complex structures from these primitives.

4. To change the state of the world, incorporate a temporal operator. We may not need this.

5. What follows is LOGICAL PROGRAMMING, and is documented in many books on the subject. Kowalski's LOGIC FOR PROBLEM SOLVING is a classic.

So, as they say on the Peter Pan ride at Disneyland, "Here we goooooo..."

PROCESSING OVERVIEW; SKIP THIS IF YOU KNOW HOW LOGIC PROGRAMMING WORKS

1. The state of the computation is stored in one (or several) databases. Thus, the functional effect of computation has a historical record.

2. Programming is accomplished by mathematical *description* of the world. Knowledge engineering of the domain identifies OBJECTS and RELATIONS, which define a mathematical STRUCTURE. This vocabulary is used to express the starting point and the ending point (goal) of a computation. LOGICAL operators determine the sequence of computation.

3. A generic inference engine moves the computation from start to finish. This engine is independent of the programmer. Its facilities include

automatic MATCHING (unification)
automatic search through the space of possible solutions (proof)
semi-automatic control of the computational process.

A "query variable", such as a question mark, in the place of an argument instructs the inference engine to search the data base for instantiations (matches) of the argument. Inferential rules (if-then) instruct the engine about acceptable paths of search for a solution. Logical operators are used to build complex process descriptions.

ISOMORPHISMS

The representations of Predicate Calculus, semantic networks, and object-oriented frames are isomorphic. It is useful to distinguish between the representation of a problem and the computation of an answer. Basically, if you can represent the world, in whatever form is convenient to the user, you have solved the computational problem, since the computation over the various representations is identical.

This point is of interest to those who model the domain by drawing nodes and arcs. The nodes are objects with attributes. The arcs are relations. If the node-and-arc diagram is fully labeled, then the procedural aspects of a program are effectively written.

My own view is that knowledge engineering (naming the object attributes and their relational connections) is mandatory. Why invest extra effort in procedural programming; it is free if the knowledge engineering is expressed within a logic programming language.

APPLICATION PRIMATIVES

ASSUME: non-probabilistic model

(A lot of what follows is condensed and simplified.)

OBJECT:
ATTRIBUTES: Name
Location
State
Time
History

History refers to a list of objects with the same Name and ordered Time attributes.

RELATION: Event

(event <name> <state> <time>)

Event is constructed from the attributes of the object, and is the only primitive relation in this model.

Note that it is easy to go from the object-attribute structure to the relational structure.

FACTS:

The only facts are TRUE event relations. Facts are determined from measurements, and true facts are added to a current data base.

In principle, it is easy to move from a factual data base to a probabilistic data base by using the standard concept of probabilistic truth. A CONFIDENCE field must be added to all facts, and a probability calculus is added to the inference mechanism.

QUERIES:

The existence of a fact is determined by matching a query structure to a data base entry. The form of a QUERY is something like:

(event <object-12> ? 3)

which returns the state of <object-12> at time 3, or NIL.

BEHAVIORS:

A Behavior is a ordered sequence of events. An example of a Behavior

template (definition):

```
(event delta-? ready ?t1)
& (event delta-? transit ?t3)
& (sequence ?t1 ?t3)
```

which defines a sequence of behaviors called delta. Note the type constraint on the query variable (delta-?), and the indexing of similar time-query variables. When a query variable occurs more than once in a template, the first occurrence is bound to a data base entry, and the second occurrence gets the same binding. So query variables that are mentioned more than once are CONSTRAINTS on the template.

To extend the model to include time sequences, a new relation of SEQUENCE is added. Since the sequence information is included in the History attribute, an alternative representation is to define time as history:

EVENT: (event <name> <state>)

HISTORY: a sequence of events with the same name.

and

```
BEHAVIOR:    delta

(ordered-members (event delta-? ready) (event delta-? transit) )
```

The relation ORDERED-MEMBERS takes any number of event arguments, and determines if they are in an ordered sequence in the History attribute. This representation is more efficient than the previous one and can handle continuous time.

Finally, we want the results of the behavior calculation to be added to the data base, so we add the BEHAVIOR rule:

```
(if
(ordered-members (event delta-? ready) (event delta-? transit) )
      then add-to-data-base
(behavior delta-? delta <start-time> <finish-time>)" )
```

The ADD-TO-DATA-BASE meta-variable causes the structure that follows to be added to the data-base. I haven't specified how to put in the time delimiters of the new fact, but it would be something like counting the position of the fact in the history, or relying explicitly on stored time facts.

STATES WITH VARIABLE DURATION

Method 1

Specify the duration of a state as a fact, using the new relation of STATE-DURATION.

```
            (state-duration ready 6 10)
```

And store duration of a state as an event parameter. That is, events are no longer points in time, but are intervals of time delineated by state changes.

```
      (event <name> <state> <time-in-state>)
```

States using state-duration look like this:

```
      (event delta-? ?state ?duration)
            & (state-duration ?state ?shortest ?longest)
            & (greater-than ?duration ?shortest)
            & (less-than ?duration ?longest)
```

Behaviors look like this:

```
      (event delta-? ready ?duration)
            & (state-duration ready ?shortest ?longest)
            & (greater-than ?duration ?shortest)
            & (less-than ?duration ?longest)
            & ...
```

...similarly for the other states.

Method 2

Store all this in the History, so a behavior would be:

```
      (ordered-members
            (event delta-? ready ?ready-duration)
            (event delta-? transit ?transit-duration) )
      & (state-duration ready ?ready-shortest ?ready-longest)
      & (in-between ?ready-shortest ?ready-duration ?ready-longest)
      & ...
```

I've created a new relation IN-BETWEEN as a short way of saying that the duration is constrained.

Here's a good place to illustrate how constraints are implemented.

IN-BETWEEN is evaluated by a direct call to LISP:

```
(DEFUN IN-BETWEEN (x y z)
      (and (> y x) (< y z)) )
```

So basically a relational name has two faces: either it is a "real" relation whose truth can be determined only by looking in the data base, or it is a Boolean function that returns a truth value.

The top-level control keeps a list of the lisp functions and when the reader encounters one, control goes to LISP. Alternatively, the code can keep the meaning explicit by the META-RELATION EVAL. For example:

```
(ordered-members ...
...
& (EVAL (in-between ?ready-shortest ?ready-duration ?ready-longest)
```

IMPLEMENTATION STUFF

1. Matching is integral to the inference engine, and is activated by the ? syntax.

2. Constraints are expressed declaratively as illustrated above. They can be written in LISP if complex, and accessed by EVAL. They can be packaged up so that a single relational name activates a collection of constraints.

3. Hierarchical transfer is accomplished by similar mechanisms.

4. The value of a variable can be determined by a LISP call. If the call returns a non-Boolean result, for example (add1 count), it must be framed in a Boolean context. In the example, this might be (= (EVAL (add1 count)) ?total). The equality relation converts numerical calculation to a Boolean; EVAL is used to tell the parser that ADD1 is a function, not a relation; and the query variable ?TOTAL is a dummy that gets the result of the functional incrementation.

HIERARCHICAL EXAMPLE

(I have to make most of the content up.) Imagine a group of 12 objects, and a need to determine readiness from behaviors. Say that an object is ready if it is in delta behavior.

Add the READY relation to the concept structure:

```
(ready <name> delta <time-index>)
```

and the READY fact to the data-base:

```
      (ready delta-? delta now)
```

where "delta" refers to the behavior identified above.

If all the behaviors have been calculated, the data base will have facts that look like:

```
      (behavior <name> <behavior-type> <start-time> <finish-time>)
```

Add the READY rule:

```
      (if (behavior ?name delta ?start ?finish)
            then add-to-data-base
            (ready ?name delta ?finish) )
```

to create a listing of the ready facts. To determine object readiness, examine the data-base for ready facts, and add to the data-base ready-count facts:

```
      (if (ready ?name delta ?time) & (ready-count ?time ?number)
            then add-to-data-base
            (ready-count ?time (eval (add1 ?number))) )
```

Actually, this kind of accumulation is more intuitive if done directly in LISP:

```
(defun ready-count (data-base)
      (loop for item in data-base
            (if (eq (first item) 'ready)
                (add1 ready-items) ) ) )
```

PROJECTION INTO THE FUTURE

Roughly, when an behavior is to be projected into the future, a hypothetical data-base is created which contains the future facts. Future facts are generated by adding to the data base the desired events with a future time label.

THE MAIN POINT

With this kind of representation the majority of the time is spent describing the world and describing what we want. Rapid prototyping is easy cause there is no procedural code to rewrite. Objects, relations, and rules are easy to delete and to add. The key concept:

ENFORCED STEP-WISE MODULARITY

THE MAIN PROBLEM

New techniques require new skills. It may be a delicate decision to choose between the dysfunctionality of old habits and the dysfunctionality of having to learn new tricks (witness moving from LISP to LOSP).

A personal anecdote: I learned logic programming from Genesereth while I was learning LISP, and I did not have the experience or the wisdom to know that the two styles were different. Consequently I struggled a lot with both subjects. The struggle to move from a FORTRAN style (or a LISP style, if that exists) to a PROLOG style is not the main issue, the two styles are complementary. What is important is to select the style that is most appropriate for the problem at hand.

And here's my bias: most AI programming needs (rapid prototype, explanation, severe modularity, inference base, hierarchical structure, etc.) are directly accommodated by the logic programming style.

Bias number two: Logic programming is state-of-the-art. Our research obligation is to think in state-of-the-art concepts.