Representation of Procedural Knowledge for Expert Systems

Roger T. Hartley
Department of Computer Science
Fairchild Hall
Kansas State University
Manhattan, KS 66506

ABSTRACT

Many of the knowledge representation schemes developed in the past have concentrated on declarative knowledge. Many have provided assertional mechanisms for deductive retrieval and some give terminological mechanisms for classification and abstraction. However, none present the system developer with similar sophistication, at a high level of representation, in procedural knowledge. Often the developer is left to write attached procedures or demons using the base language and embedded logical functions or procedures. This paper presents an attempt to elevate procedural knowledge to the same representational level as declarative knowledge, by using a uniform declarative notation. The notation, based on conceptual modelling, has a procedural component giving it the flavor of conceptual programming. It enables complete specification, at a conceptual level, of all aspects of expert knowledge. In particular strategies, which in some systems are represented through "control knowledge", are incorporated naturally into assertional and terminological components. An extended example from shallow fault diagnosis is presented to illustrate the technology.

1. Introduction: Uniformity in Knowledge Representation

A knowledge representation (hereafter KR) scheme has two jobs to do. Its main task is to enable a computer to carry a machine representation of a person's understanding of some aspect of the world. The computer, being capable of controlled dynamic change, can also mimic such happenings in the real world. This is the performance aspect of KR. However, KR schemes are not arbitrary systems. They purport to display regularities and structures, not only of the world, but of our systematization of it. To this end, a KR scheme should be a formal system, capable of description in formal terms independently of the machine Such systematization serves to make explicit our understanding of the world, in terms which be communicated to others. This second aspect, the *communication* aspect, is the main focus of this paper.

As an example of these two aspects, consider a simple change of of state in that part, of the world involving John and Mary. John has a book, which he gives to Mary. The act, of giving results in two state changes. Firstly, John no longer has the book. Secondly, Mary now has it. We may to choose to represent the sequence linguistically as:

John has a book.
John gives the book to Mary.

However, to make the correct inference that Mary now has the book, we need to know about acts of giving. We can do this by representing the meaning of these sentences in a conceptual model. John having the book is a state of possession. Following Sowa's notation [1] this is:

[PERSON:John] - > (POSS) - > [BOOK:#327]

where "John" is the name of an individual of type PERSON, and the book is number 327 of type BOOK. A case analysis of "give" reveals an agent, John, a recipient, Mary, and an object, the book. In the conceptual notation this is:

```
[GIVE:#22]-
   (AGNT) -> [PERSON:John]
   (RCPT) - > [PERSON:Mary]
   (OBJ) -> [BOOK:#327]
```

where indentation denotes a multi-way (here three-way) relationship. The relations POSS, AGNT, RCPT, and OBJ are assumed to be implicitly defined by examining several sentences and their respective conceptual models. The types GIVE, PERSON, BOOK are assumed to be defined in a type lattice of sub and super-type relationships.

Whereas the above two conceptual graphs capture some of the meaning (and indeed more than is explicit) of the two sentences, they do not capture the dynamic change of possession of the book. We could add:

[PERSON:Mary] -> (POSS) -> [BOOK:#327]

to the sequence, but there is still not enough information to make the change apparent.

If we regard the sequence of graphs as a program, we can envisage an interpreter executing the graphs and producing the desired state transition. What additional information would the interpreter need in order to function? States such as possession would be represented in a "program state vector" much as in normal programming languages. However, the interpreter would have to handle "give" (as well as hosts of other acts) in an ad hoe way to take the book away from John and give it to Mary. The state vector must reflect the change, and the interpreter must do it, but where does it get its information from? If we generalize from this example, most KR schemes handle declarative knowledge, such as the above facts, in similar ways. Whether the scheme is frames semantic networks, logic or conceptual models, the KR system can only represent, states. Some of these states represent states-of-affairs of the world, such as John possessing the book. Some, however, only represent. role relationships between acts and entities. For example. John has the agent role in give #22, and Mary has the role of' recipient. Nowhere does it say *how* the giving takes place, in terms of its enabling and resultant states. This *procedural* knowledge is often handled in a completely different spirit from the declarative representations. Frames have attached procedures (sometimes called demons or active values) which are triggered by access to a slot value. The procedure then has the responsibility of producing the appropriate state change. In deductive retrieval systems an implication rule is invoked to produce additions to the state. In other hybrid systems, production rules are used in addition to the declarative mechanims to produce changes in the state vector.

It is my contention in this paper that such hybrid or ad hoc methods are unnecessary and even confusing. It is preferable, I believe, to have a uniform representation for both factual knowledge and knowledge of how the world works. Although it is clear that there are two main kinds of knowledge, declarative and procedural, it is also clear that they are inextricably entwined with respect to any well-defined domain. A KR scheme would do well, therefore, to allow a uniform representational mechanism across both types of knowledge, so that the close linkage between them is made explicit.

2. Acquisition and Representation of Knowledge for Expert Systems

Any tool which helps in the design and implementation of an expert system must present the same two attributes to its user as a programming language. It must give enough expressive- to enable the user to specify the system adequately. It must also give the user the pragmatic utility to create, test and modify the system, once specified. It is clear that, from the user's point of view, the highest possible level of representation is an important goal to aim for when designing a KR system. It is agreed by many that a level where concepts and conceptual relations are the building blocks is preferable to a more formal, but more arbitrary (and hence less expressive) system such as a general-purpose programming language. The ideal conceptual modelling system is one in which the labels used have the obvious connotation, one step removed from their use in language. Hence the use of a type label "GIVE" is meant to denote all acts of giving (there may be additional constraints brought about by the use of certain case roles). A similar use of a label in (say) Lisp, would have no such meaning. It is very natural and also correct to use the concept "GIVE" wherever the word "give" may be used in a sentence (there are, of course, differences of sense to be catered for, but conceptual modelling can make these explicit). The expressiveness of a conceptual modelling notation can thus be an important aid to an expert system developer.

The acquisition of knowledge by the development tool puts different, but related constraints on the choice of a KR scheme. If high-level means complex, then high-level is no good. Pragmatic constraints point to simple systems. with easy-to-learn syntax and little possibility for semantic confusion. However, there is a difference between complex and hard-to-use. They do not necessarily go together, as the rise of intelligent programming environments has proved. In fact the Lisp environments being offered now are more complex, but the general concensus is that they are easier to use than the old ones. They did not have the sorts of editors, debuggers and source control systems that are available now. The aim, therefore, is to produce complex high-level programming systems, but to give the user lots of help in the programming task. With an expert system tool, the level of representation should be the highest, thus demanding even smarter environments to keep the utility high.

3. Conceptual Programming

The basis of conceptual programming as presented here, is Sowa's conceptual graph notation [1]. The elements are concept8 sented as tipper-case type labels surrounded by square brackets, and relations sented as upper-case relation labels surrounded by parentheses. qtates are represented as type restrictions on canonical graphs of the form:

[ENTITY] -> (RELATION) ->[ATTRIBUTE]

Actions are represented as multi-way case relations. For instance, acts with a direct object (e.g. lilt, bend, stroke, push) have the following canonical form:

```
[ACT]-
   (AGNT)->[ANIMATE]
   (OBJ)->[PHYS-OBJ]
```

Comprehensive details can be found in Sowa's book, and an example of their use in [2].

In order to represent procedural knowledge in the same notation, a third element is introduced - the actor. Sowa only used these to compute referents (usually numeric) in a data-flow sense, but here the actor is included in the general scheme as an element with equivalent status. It is intended that actors can be defined in the same way as concept types and relations. They can compute referents or produce state changes.

Actors are notated as upper-case labels enclosed in diamond brackets, thus: <ACTOR>. They are activated via an assertional mechanism in the system, whereas conceptual models are purely descriptive in nature and come into play merely through their existence. This assertional mechanism takes a descriptive graph and asserts it as either reflecting a state-of-affairs in the present, or as an act potentially producing changes of state. The similarities between this assertional mechanism and an interpreter for a programming language are plain to see.

Each actor is linked on its input side to zero or more concepts, representing the enabling pre-conditions for the actor to fire. The actor is triggered by one or more events (perhaps concurrently) to produce assertions of new states, or the existence of individuals (conceptual referents) on its one or more outputs. Those actors which have enabling states as inputs and which are triggered by acts can be called inferential actors, while those producing referents can be called computational actors.

As an example of an inferential actor, we shall return to John giving Mary a book. The necessary actor is here called 'I' for 'inferential'. It takes in one pre-condition, namely John having the book, and is triggered by the event involving giving. Its output is the state of Mary having the book. Canonically this is:

```
<I>-
  (TEC) <- [STATE]
  (TRIG) <- [EVENT)
  (AS) <- [STATE]
```

TEC means "temporary enabling condition". Some states are removed by the occurrence of the event, whereas some persist (these would have a PEC link). The existence of individuals may also serve as pre-conditions via the IEC link. TRIG is the trigger relation with the event. AS means "assert state". This state (possibly many) is produced by the act. Individuals may be created with an AI (assert individual) link. We are now in a position to generalize our particular act of giving to a schema definition which contains information about one possible way (there may be others) an act of giving may occur. Incorporating the I actor as above, we arrive at a definition which describes the relations involved and also provides enough fuel for an interpreter to execute the act, producing the necessary state change:

schema for GIVE(x) *is*

```
<I>-
  (TEC)<-[STATE:[PERSON:*p]-> (POSS) -> [PHYS-OBJ:*o]]
  (TRIG) <-[EVENT:[GIVE:*x]-
              (AGNT) -> [PERSON:*p]
              (RCPT) ->[PERSON:*q]
              (OBJ) -> [PHYS-OBJ:*o]]
  (AS) -> [STATE: [PER SON *q]- >(POSS) -> [PHYS-OBJ:*o]].
```

Note the use of' an asterisk followed by a lower-case identifier as a notation for a variable. The definition says *that* giving involves an actor and a recipient who are both people, and a physical object The inclusion of the inferential actor shows *how* the act produces the transfer of the object from one person (the agent) to the recipient.

This definition would be used in the following way. Firstly, the fact of John having the book would have to be asserted. This could come about either directly from the user (as a "line" in a conceptual program) or as the result of the previous triggering of another actor. Perhaps John bought the book, and another instance of the I actor produced the state of John possessing the book. Secondly the occurrence of the act would also have to be asserted. This would immediately trigger the actor, producing the new state. The conceptual program to do this might be (it is a single graph, or a "one-line" program because of the direct referent in BOOK:

```
[PERSON:John] -> (POSS) -> [BOOK:#372].
[GIVE]-
   (AGNT) -> [PERSON:John]
   (RCPT) - > [PERSON:Mary]
   (OBJ) - > [BOOK:#372].
```

The interpreter produces the necessary type restrictions (through conceptual joins) to activate the GIVE schema and also works the actor mechanism.

Sequences of otherwise unconnected acts or events are connected by a "next" or "N" actor. Without this the intepreter would execute one graph and stop. For instance, the sequence "John looked at Mary. A car passed by." would be represented as:

```
<N>-
  (TRIG) <-[EVENT:[LOOK]-
              (AGNT) -> [PERSON:John]
              (EXPR) -> [PERSON:Mary]]
  (THEN)-> [EVENT:[PASS]-
              (INST) -> [CAR]].
```

The relation 'then' is the procedural counterpart of a descriptive successor relation, not merely signalling the passage of time, but reproducing it, through procedural interpretation. There are many other undefined actors which are primitive to the system. Some of them are illustrated in the example which follows.

4. An Extended Example: A Shallow Fault Diagnosis System

To show how conceptual programming can be put to use, the problem-solving strategy of the fault-finding system CRIB will be reexamined and represented in the new notation, using primitive actors to express the procedurality in the technique. See [3] for an overview of CRIB.

CRIB's strategy is based on ,ill easy-to-follow technique of "divide and conquer". Diagnostic tests are performed with the purpose of splitting that, part, of the machine assumed to be faulty into faulty and non-faulty parts. In this way progress is made towards isolating the fault in a field- replacable unit (FRU). The breakdown of the machine and remains fixed. It is useful to think of the machine as a hierarchy of functions, where each function is known to reside wholly in a physical component or assernbly , here called a 'unit'. Knowledge about faults is represented through groupings of observed test symptomswhich have occurred in previous successful fault investigations. How these groupings are formed is not our concern but every function in the hierarchy is assumed to have associated with it a set, or sets of symptoms which, if observed would locate the fault in the physical unit where the function resides. There is an implied loop in the above procedure, as progress is made down the hierarchy. The sequence inside the loop is , linguistically:

1. Choose a test which discriminates best between the current function's sub-functions. (The current function is the one assumed to reside in the faulty unit).
2. Perform the test, and observe the resultant symptoms.
3. Add the observed symptoms to the current symptom set.
4. Analyse the current function using the current symptoms. (If an FRU is found to be faulty, stop).

The sequence may be represented as a schema for the act 'diagnose' which makes explicit the above sequence:

```
schema for DIAGNOSE(x) is
 [EVENT: [DIAGNOSE: *x]
         (OBJ)-> [FUNCTION:system]
                 (LOC)- > [UNIT:computer]- > (ATTR)- > [FAULTY]
                 (ATTR)-> [CURRENT]]-
  (TRIG)-> <N>-
         (IPC)<-[FUNCTION:system]
         (THEN)- >
 [EVENT:* ch= [CHOOSE]-
           (OBJ)-> [TEST: *t]
                 (INST) < - [DISCRIMINATE]-
                         (MANR)- > [BEST]
                         (OBJ)-> [FUNCTION]-
                                 (SUBSUMES) < - [FUNCTION: * f]
                                         (ATTR)- > [CURRENT]]-
           (TRIG)-> <N>-
              (IPC)<-[TEST:*t]
             (THEN)- >
 [EVENT: [PERFORM]
        (OBJ)->[TEST:*t]
        (RSLT)->[OBSERVED-SYMPTOM:*os={*}]]-
  (TRIG)-> <N>-
          (lPC)<-[OBSERVED-SYMPTOM:*os]
          (THEN)- >
 [EVENT:[ADD]-
        (OBJ)- > [OBSERVED-SYMPTOM: *os]
        (DEST)- > [CURRENT-SYMPTOM: {*I}]
        (RSLT)->[CURRENT-SYMPTOM:*os={*}]]-
  (TRIG)- > < N >-
        (IPC) < - [CURRENT- SYMPTOM]:*cs]
          (THEN)->
  [EVENT:[ANALYSE]-
        (INST)->[CURRENT-SYMPTOM:*cs]
        (OBJ)- > [FUNCTION: *f]]-
    (TRIG)- > < N > - > [EVENT:*ch].
```

Note the loop control in the last N actor, taking control back to (he CHOOSE event. Also note the use of variables to handle back and forward references to previously mentioned concepts (e.g.. the FUNCTION f in both the CHOOSE and ANALYSE portions of the schema.

The heart of the system is in the schemata for ANALYSE. Each grouping of symptoms which points to a particular function being faulty has a separate schema for ANALYSE, since it is assumed that ANALYSE only gets its meaning in these different but similar contexts. It includes the inferential actor I which links the particular set of symptoms to its associated function and hence faulty unit. It is this association which is learned by the technician through experience in diagnosing the same faults again and again. One such schema for SYMPTOMs p, q and r pointing to FUNCTION f is:

```
schema for ANALYSE(x) is
<I>-
(TRIG) <-[EVENT: [ANALYSE:*x]-
                (INST)-> [SYMPTOM: {p,q,r)]- > (IMPLIES)- > [FUNCTION:*g]
                (OBJ)- >[FUNCTION: *f]-
                          (ATTR)->[CURRENT]
                          (LOC)-> [UNIT: *u]- > (ATTR)-> > [FAULTY]
                          (SUBSUMES)->
                        [FUNCTION:*g]-
                         (LOC)- >[UNIT: *v]]
 (TEC) <-[STATE: [FUNCTION: *f]- > (ATTR)-> > (CURRENT]]
 (TEC) <-[STATE: UNIT:*u]- > (ATTR)- > [FAULTY]]
(AS)-> [STATE:[FUNCTION:-g]->(ATTR)-> [CURRENT]]
(AS)-> [STATE: [UNIT: *v]- > (ATTR)- > [FAULTY]].
```

Some schemata for ANALYSE will involve FRUs. In these cases the state of the FRU being faulty will enable a HALT actor, also triggered by ANALYSE. Leaving out the detail (which remains the same) this is:

```
schema for ANALYSE(x) is
  <I>
  (TRIG)< -[EVENT: [ANALYSE:*x]
          .
          .
          .
           ]]->(TRIG)->
      <HALT>-
           (PEC)< -[STATE: [UNIT: *v]- > (ATTR)- > [FAULTY].
```

The schemata for PERFORM and ADD are relatively simple, but each involves a new actor. ASK is an actor which prompts the user for input, producing the information typed in as output. UNION is one of a family of set operations provided to compute referents involving sets. Sets are the conceptual mechanism provided for handling aggregations of individuals.

schema for PERFORM(x) *is*
<ASK>-
(TRIG) <-[EVENT: [PERFORM: *x]-
(OBJ)->[TEST:*t]
(RSLT)- > [OBSERVED-SYMPTOM:*os=(*)]
(IEC) <-[TEST: *t]
(AI)-> [OBSERVED-SYMPTOM: *os].

schema for ADD(x) is
[EVENT: [ADD: *x]-
(OBJ)-> [OBSERVED-SYMPTOM:*os={*}]
(DEST)-> [CURRENT-SYMPTOM:*cs={*}]
(RSLT)- > [CURRENT-SYMPTOM:{*)]-
(AI)<-<UNION>-
(IEC) <-[OBSERVED-SYMPTOM:*os]
(IEC) < -[CURRENT-SYMPTOM:*cs].

The final schema, that for CHOOSE, gives the heuristic strategy for choosing an appropriate test to try next. Here an over-simplified version is given which just involves choosing the test which takes the least time to perform from all those which might yield symptoms in partially matched groups. Here several computational actors are used to manipulate sets and one called FIND which locates the appropriate test in an unordered set.

schema for CHOOSE(x) *is*
<FIND>-
(TRIG) <-[EVENT: [CHOOSE: *x]
(OBJ)-> [TEST: *t]
(INST) <-[DISCRIMINATE]
(MANR)- > [BEST]
(OBJ)- > [FUNCTION:Resp
(SUBSUMES) <-[FUNCTION]
(IMPLIES)- > [SYMPTOMS: ss=Resp{**}]
(IEC) <-[TEST: *ts=Resp {*)]-
(PRODUCES)- > [SYMPTOM:Resp{*
(AI)<-<DIFF>-
(IEC) <- [SYMPTOM:{*}]-
(AI) < - < JOIN > <-(IEC) <-[SYMTPOMS:*ss]
(IEC) <- [CURRENT-SYMPTOM:{*}]
(IEC) <-[TIME:Resp:{ *}]-
(DUR) <-[TEST:* ts]
(IEC)- >< M IN>->(AI)- [TIME:*TM] <- (DUR)<-[TEST: * T]
(IEC) <-[TIME: * tm].

The actor DIFF is the operation of set difference the actor JOIN is the grand union of a set of sets; the actor MIN is the operation taking the minimum of a set of numbers. The actor FIND takes two sets (here TESTs mid corresponding TIMEs), searches them in parallel, one element at a time (looking for an element in either set equal to its third input. If the element is found, the corresponding one in the other set Is produced as output. If the element is not, found, the output is not produced at all. CHOOSE call be seen to be an operation of taking all the symptom sets which imply the current function's immediate constituents and finding those symptoms in these sets which have not yet been observed. The test which has the least time of those which might. produce. all element of the difference set is the one chosen.

It should be stressed that these schemata presented here are intended to demonstrate the expressiveness of the notation, and are not meant to reflect the workings of CRIB accurately. However, the spirit of the system has been preserved. Even so, much has been left out, most notably the description of the computer system as a hierarchy of functions each residing in a physical unit. The system is activated via the assertional mechanism, with a command (an act without an agent) of the form "diagnose the system" or conceptually:

[DIAGNOSE]->(OBJ)-> [FUNCTION:system]

The interpreter will join this to the schema for diagnose, thus activating the sequence of steps inside it. These in turn will be joined to their corresponding schemata, eventually producing a HALT when an FRU is found to be faulty, or a "system halt" occurs when CHOOSE fails to instantiate a test to perform.

5. Comparison with other work

There is little that is new in the notations of knowledge representation schemes. This paper is no exception. Conceptual models are a species of semantic network with the terminological component straightened out. Similar work is Shapiro's SNePS, especially the assertional component SNIP [4]. Also noteworthy is the work of MyIopoulos et al. [5). The updated work on KL-ONE, called NIKL (New Implementation of KL-ONE) is similar in that it is being proposed as the basis of exert system development, especially by Swartout et al. at USC/ISI [6]. The commercial hybrid tools KEE and ART (7], (8] represent the state-of-the-art in expert System development. On the academic side the Stanford efforts MRS and the earlier work on RLL are in the same spirit as the thinking in this paper, but using different KR schemes as their basis [9), [101. This paper presents a new departure in that one KR scheme (conceptual modelling) is put forward as being capable of use for expert system development. For an argument against (in particular) production rules as the sole basis of expert system development see [11].

6. Conclusion

A knowledge representation scheme has been advanced aimed at elevating procedural knowledge to the same level of representation as declarative knowledge. Sowa's conceptual graph notation has been adapted, with an interpreter in mind, to provide dynamic performance, by giving actors equivalent status with concepts. In addition, actors can have conceptual relations linking them to concepts, to capture the variety of input and output options. This notation enables specification of all aspects of knowledge. In particular problem-solving strategies (as exemplified by the strategy from CRIB) can be represented by conceptual programming techniques. The high level of representation offered, as well as its uniformity of expression, makes the technique highly suitable for expert system development.

Problems exist with the practical aspects of conceptual modelling. It might be preferable to elim inat e the linear style of expression used in this paper and instead go for graphic expres,,slon in nodes and arcs. Both methods, however, have their weaknesses. Lines (arcs) tend to wander and cross each other in graphical forms of expression. Conversely, the use of variables for intra-graph reference in the linear form can sometimes look confusing, and repetition is unavoidable. The choice between a structure editor for the linear form and a graphics editor for the graph form is also not an easy one to make. These problems should not be minimized because, as mentioned in the introduction, pragmatic utility is Just as important as expressiveness.

Work is going on currently to develop a library or actors and to allow definition or complex actors in terms of simpler ones. The uniformity of expression allows this to take place far easier than if levels of representation had been mixed As a learning exercise, several other expert systems are being re-specified in conceptual terms In this way, a library of explicitly defined strategies call be built, leading to better understanding of tile technology of expert system development tools.

REFERENCES

[1] Sowa, J.F. Conceptual Structures. Addison Wesley. 1984.
[2] Sowa, J.F. Using a Lexicon of Canonical Graphs in a Conceptual Parser. (Draft) IBM Systems Research Institute, New York, 1985.
[3] Hartley, R.T. CRIB: Computer Fault-finding through Knowledge Engineering. IEEE Computer. March 1984.
[4] Shapiro, S. The SNePS Semantic Network Processing System. in Associative Networks, Findler, N. (Ed.) Academic Press , 1979.
[5] Levesque, H. and Mylopoulos J. A Procedural Semantics for Semantic Networks. in Associative Networks, Findler, N. (Ed.) Academic Press, 1979.
[6] Moser, M.G. An Overview of NIKL, the New Implementation of KL-ONE. Tech. Report 5421, Bolt, Beranek and Newman Inc. 1985.
[7] IntelliCorp. KEE Reference Manual. IntelliCorp, Menlo Park, California, 1985.
[8] Williams, Chuck. ART - Conceptual Overview. Inference Corp. Los Angeles, 1985.
19] Genesereth, M. An Overview of MRS for AI Experts. Memo HPP-82-27. Stanford Heuristic Programming Project, Stanford University, 1982.
110] Greiner, R. RLL-1: A Representation Language Language. Working Paper HPP-80-9. Stanford Heuristic Programming Project, Stanford University, 1980.
[11] Hartley, R.T. The Competely Generic Expert System Builder. Proc. The 1985 Conference on Intelligent Systems and Machines. Oakland University, April 1985.