Conceptual Programming: Foundations of Problem-Solving

Roger T. Hartley and Michael J. Coombs

Computing Research Laboratory and Department of Computer Science BOX 3CU New Mexico State University Las Cruces, NM 88003

ABSTRACT

Conceptual Programming is a term meant to convey a similar idea to that of Logic Programming, but at a higher level of representation. Programming with concepts, as presented here, has all the advantages that many knowledge representation schemes have in dealing with declarative knowledge i.e. explicitness, naturalness, expressibility, and transparency. It also gives procedural knowledge these same features, using the same representational technology, thus providing uniformity of expression for both forms of knowledge. We present an augmented conceptual graph theory, the building blocks of a conceptual programming system, and a description of the interpreter for the system. Conceptual programming is intended for work in knowledge engineering.

1. Introduction: Programming systems for knowledge engineering.

Knowledge Engineering is about the engineering of better computer systems to handle tasks involving cognitive expertise, human decision making, heuristic reasoning and complex, empirically learned associations. The three main phases of a KE exercise all involve knowledge. They are: elicitation, representation and acquisition. The three are necessarily intertwined, but also have a degree of independence in that they involve different active participants. Elicitation of knowledge involves a human expert (or his writings); representation involves the knowledge engineer; acquisition involves the machine and its constraints. It is these constraints which are the focus of this paper.

The central task, representation, is often a compromise between the needs of the knowledge engineer to cope with imprecision (or at least with the more informal, high-level descriptions necessary in representing expertise), and the desire for formally correct inference procedures. With this goal in mind, representation at the level of concepts, with good epistemological underpinnings (Brachman, 1979) gives the most flexibility.

Along with the level of representation, the handling of procedural and declarative knowledge is also important for KE. Expertise is the application of appropriate problem-solving techniques to a situation-specific body of facts. Strategies for problem-solving need expression in the same terms as these facts in order to avoid loss of generality (Chandrasekaran, 1984). If possible, procedural knowledge should enter into the same mechanisms of abstraction and generalization that are common in declarative knowledge (as in many knowledge representation systems such as KL-ONE). Commonly there is no

choice of strategy provided or strategies are expressed in a different language from the rest of the knowledge base. This paper describes a uniform way of including procedural knowledge and factual information in both the assertional and terminological components of the representation system. The paradigm behind these methods is called *conceptual programming*.

Until a few years ago the best available tools for building expert systems were the ‘‘empty’’ systems, typified by EMYCIN (van Melle, 1980). A spate of EMYCIN look-alikes are now commercially available (e.g. Teknowledge’s M1 and TI’s Personal Consultant). Expert system technology has become equated with the rule-based paradigm (syntactic pattern-matching, forward or back chaining), even though such systems have been criticized for their methodological and technological shortcomings (Clancey, 1983; Hartley, 1984; Chandrasekaran, 1983). The reported inadequacies of these systems led to the development of *hybrid* systems which include IntelliCorp’s KEE and Inference’s ART. Among the ‘‘goodies’’ they include are frames, rules, procedures, demons (or ‘‘active values’’), viewpoints and truth maintenance, all in the same package. However, the mere provision of facilities does not make it easier to build expert systems since there is precious little help in choosing appropriately among them.

At the other end of the scale is Prolog, whose foundations are rooted in formal logic, thus giving an extremely elegant system, but which turns out to be inefficient in execution (in its pure form) and limited in scope. In between these two extremes there are few expert system tools which are expressive enough for ease of use and applicability and yet have formal underpinnings. A notable exception is Omega, best described as a description logic (Attardi et al., 1984) which caters for frame-like expression of declarative knowledge yet has a formally complete set of inference rules.

We might wish for a language system in which arbitrary expressions of knowledge can be handled, and yet the arbitrariness is restricted to capturing real relationships in the world, not in the methods and procedures used to manipulate them. We might hope that there would be no Lisp code to write and no choice to be made between seemingly equivalent pieces of technology. It is partially this lack of orthogonality in the hybrid systems which makes them so hard to work with. Moreover, a system which explicitly separates domain-specific information from all other types *and* in which everything that is not domain-specific is minimized and formalized is preferable. Conceptual Programming (hereafter ‘CP’) is just such a system since it expresses the content of knowledge, both declarative and procedural, using concepts, and its interpreter follows a formally correct methodology based on hypothesis generation and testing.

2. Extended conceptual graphs and procedural knowledge

Conceptual graphs, as described in Conceptual Structures, express declarative knowledge. An *assertional* mechanism built using such graphs would allow a collection of graphs to represent a state of knowledge of an agent. If rules governing the assertion and de-assertion of graphs can be expressed within the *terminological* component, then procedural knowledge can also be incorporated. Sowa has actors behaving like functions embedded within a graph to provide concept referents (i.e. for instantiation). Our extension allows actors to be much more like ‘‘active concepts’’ which accept states as preconditions and events as triggers. Having been triggered, they assert states and enable further acts as by-products of their activity. These actors may be used to express *causality*, involving states

and events, and *inferences*, involving propositions. Since each actor of this sort is completely specified by its inputs and outputs, there is no need to label the actor box in the graph. In effect, the actor merely acts as a confluence node for its inputs and outputs, not as a function as in Sowa's original formulation.

In the example in figure 2a the declarative component of the graph expresses an assertion which in English can be written: ''A person x is the agent of an act of giving a physical object, which is in x's possession, to a person y''. The addition of an actor can express the change of possession from x to y. The actor is linked to its inputs and outputs via special conceptual relations which collectively express a factorization of the combination of states and events typically found in a theory of action (cf. Rieger's similar treatment in common-sense algorithms: Rieger, 1976). In our case, a state is defined canonically as a specialization of the graph [T] $\rightarrow$ (REL) $\rightarrow$ [T] and an event as an arbitrary number of joins (on ACT) of specializations of the graph [ACT] $\rightarrow$ (CASE-REL) $\rightarrow$ [CASE-TYPE] where CASE-REL encompasses any of the standard set of case relations (AGT, PTNT, INST, SRCE etc.) and CASE-TYPE is restricted to conform to the corresponding case. For example, the AGT relation is, canonically, [ACT] $\rightarrow$ (AGT) $\rightarrow$ [ANIMATE] and SRCE is [ACT] $\rightarrow$ (SRCE) $\rightarrow$ [LOCATION]. In the example, two inputs are needed: the state of possession of a physical object by person x (via the relation 'TEC') and the event involving x giving it to y, via the relation 'AS', and the de-assertion of the input state. 'TEC' denotes this transitory state. The graphs involving actors are actually abbreviated from graphs containing embedded graphs of type STATE and EVENT. However, the embedded level has been removed, unambiguously, since the nature of actor inputs and outputs are restricted. The complete graph is given in figure 2b. It can be seen that figure 2a is cleaner, as long as Sowa's original notation is extended to allow relations embedded in states to be connected to actor relations. The actor relations are necessary to type inputs and outputs according to their temporal relationships.

The epistemology of actors, together with their inputs and outputs is meant to be as simple as possible while still maintaining adequate expressiveness. States are to be thought of as intermediate between events; roughly speaking, states enable events and events cause states. Any actor linked to a relation (via one of the actor relations shown in figure 2c) has a state as input or output; an actor linked to a concept has an event as input or output. Input states can either be triggering, or enabling, and transitory or persistent, in all combinations. Triggering states are ones which causes events directly and the existence of the state is enough to start the event. For example, a switch being 'on' causes current to flow. On the other hand, enabling states have no direct causal link. To continue the electricity example, the presence of a voltage source enables the current to flow, but does not trigger it under normal circumstances. States can also be classified as transitory, meaning that the state disappears when some event stops it, or persistent, when an event has no effect. Thus the presence of a voltage source is persistent with respect to the current flow, but transitory with respect to the failure of the source (say a battery running down). It is also possible that the absence of a state can trigger or enable an event. In this way it is possible to handle negated conditions. For example, a solenoid may be magnetized thus inhibiting a switch from closing. A soon as the solenoid becomes unmagnetized, the switch closure event can begin.

Events are either continuous, when their effects continue after the event ceases, or one-off when the effects terminate with the event. The flowing of a current in a wire causes a

magnetic field to surround it, but as soon as the current stops, so does the field. This is one-off behavior. On the other hand the act of closing a switch causes the current to flow, but it remains flowing after the act has terminated. This is continuous behavior. As with states causing or inhibiting events, events can cause the presence or absence of a state to occur. It is also possible to represent a ''lagged response'' which occurs on some forms of causality. In fact the current/magnetic field example is one of these, where the field takes a finite time to build, and cannot be thought of as coextensive in time with the current flowing. In particular, after the current is switched off the field takes a small amount of time to collapse again.

Figure 2c shows all the possible combinations of state and event causality, together with their canonical time charts. These simple diagrams show the temporal extent of states and events and their interrelationships. Time proceeds to the right, along the horizontal straight lines. A small vertical bar indicates a definite start or stop instant, whereas an arrow head indicates an indefinite instant. The vertical relationship between the instants (i,e, before, after or equal) characterizes the different relationships between the intervals. The full meaning and use of the actor relations in figure 2c are described in (Hartley, 1987), along with their translation into Allen's temporal relations (as in Allen, 1983). The example to be presented in section 7 shows how actors can express the procedural components of concept definitions, and can be 'compiled' into declarative forms suitable for answering queries about the sequencing of states and events.

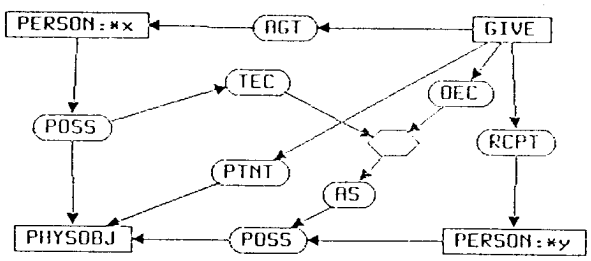

Figure 2a. An example of an actor.

3. Conceptual Programming: a programmer's point of view.

Writing a conceptual program is very similar to many other forms of top-down programming. In particular the similarity with Lisp and Prolog is striking. In both of these languages one writes a large number of hierarchically related definitions, expressing, in Lisp's case, procedurality and in Prolog's case, a mixture of logical relationships and procedurality. The program is then executed by handing control to one of the defined objects in some pre-determined way. Functions or predicates are only related through control (one calls another) and through the binding of values in parameters. However, in

```
[GIVE] -
  -> (AGNT) -> [PERSON:*x] -> (POSS:*q) -> [PHYSOBJ:*z]
  -> (PTNT) -> [PHYSOBJ:*z]
  -> (RCPT) -> [PERSON:*y] -> (POSS:*p) -> [PHYSOBJ:*z]
  -> (OEC) -> <> -
                -> (AS) -> (POSS:*p)
                <- (TEC) <- (POSS:*q).
```

Figure 2b. The expanded form of 2a in conventional notation.

conceptual programming, definitions are structured according to their intended semantics. Their relationship in the type hierarchy is part of their meaning. Languages such as Smalltalk and the Flavors subsystem of some dialects of Lisp can express some of the desired structure, but they are still *logical-level* languages. Conceptual programming is only possible using defined, meaningful concepts and relations; it is not possible to express arbitrary concepts or relations which are not derivable from the canonical basis.

The programmer should also shift his or her thinking away from making temporary definitions with a limited lifetime (the phenomenon of the throw-away type definition). Instead the programmer should think towards the augmentation of an already existing base of knowledge by task-specific additions. A conceptual programming environment (see next section) will be capable of solving many problems, not just one; the integration of multiple task-specific and general-purpose knowledge is a trademark of a human expert - so it is in conceptual programming.

Most computer languages are decidedly procedural in nature. They invite the programmer to think in terms of operations on various primitive data structures as the answer to the satisfaction of requirements. The movement towards object-centered programming has redressed the balance somewhat between procedural and declarative representations, but at the expense of increased confusion as to what is an appropriate solution to a given programming problem. Since all primitives in conceptual programming are either naturally meaningful i.e. the canonical basis, or derivable from it, programming becomes less of a choice between equally bad (or good) alternatives, and more of a natural expression of the problem area itself.

Another possible benefit of conceptual programming concerns the possibility of including the problem solver in the expression of the problem. Most language systems express

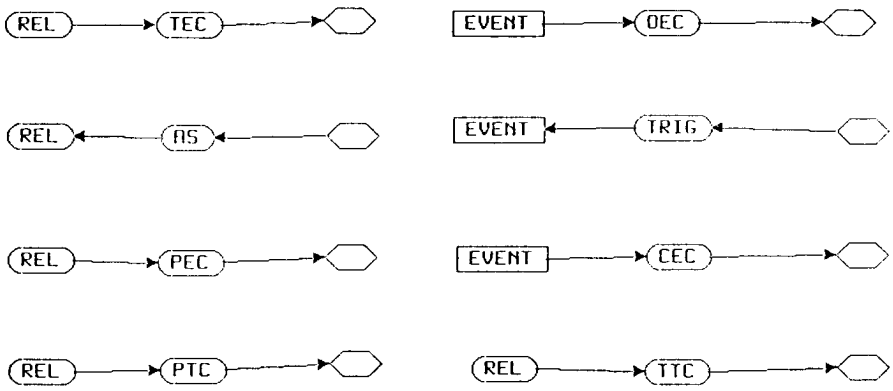

Figure 2c. Actor inputs and outputs.

procedurality through commands - they are imperative in nature. However, conceptual programming actors are more naturally expressed as being triggered by events involving agents. Many such instances, especially involving the problem-solving strategy, can mention the problem-solver as agent. The programmer is much more inclined then to think of the program as a model of expertise (or a competence model - see Hartley 1981) than as an algorithm ‘‘telling’’ the computer how to solve the problem. These ideas are being incorporated in a theory of problem-solving strategies.

4. Closed-world and open-world reasoning

A useful way to describe CP is to compare its perceived model (i.e. the user’s view of the way in which the language computes) with that of pure Prolog and to show that Prolog is just one special case allowed by it. In these terms, Prolog is a theorem prover over a closed world consisting of a set of a restricted sort of logical expression (technically Horn clauses). The theorem is proved (assuming that it is not present already) by displaying a model which is consistent with the world. The world is equated with a set of facts assumed to be true. In order to generate a model, a ‘query’ is presented to the system. The presence of variables in the query indicates that the theorem (the variables are actually existentially quantified) can only be proved if instantiation of the variables is achieved. If no direct match with the world model is possible, Prolog back-chains

through a set of horn clauses (essentially implications) until either a match is achieved or failure occurs. The set of instantiations forms the model which satisfies the query. The mere success of the back-chaining and matching procedure is sufficient grounds (in a closed world) to state the query as a theorem. If no model was generated, then the 'negation as failure' rule (again a consequence of the closed-world assumption) is grounds to state the negation of the query as a theorem. 'Resatisfaction' may produce alternative models, but these merely further support the theorem.

Open-world reasoning, on the other hand, is forced to relax these requirements and assumptions. Provability in a strict logical sense is impossible when both queries and conclusions can be questioned. To replace it a notion of 'coherence' is needed which involves an evaluative procedure. The basic architecture of CP is however the same as the Prolog case. We still have a query and still present it to the world for testing. Now, however, the query is subject to interpretation, according to alternative meanings of the terms used, as supplied through the their definitions in a schematic cluster. A query is an arbitrary proposition (just as in the Prolog case), but it may be of the form 'assuming p1, p2, p3... is c?' where the ps and c are all propositions. The world (or assertional component) consists of a set of independent propositions. They are not all assumed to be true (as in most logic-based systems), but only coherent when taken in appropriate subsets. The system needs a starting subset (the p's in the query above) which provide an initial guess as to the appropriate context for c the query's conclusion part. Further confirmation of disconfirmation of this guess is achieved through interpretation of the query. The set of interpretations form *contexts* in which the query makes sense. Since queries are often incompletely specified or are ambiguous, a context needs to be established before the query can be answered.

The next step is to instantiate the alternative interpretations in an attempt to generate models which support the query. Models are alternative pictures of the real world generated in accord with the constraints that the real world provides. The complexity of the form of these models is in stark contrast to the truth theoretic denotations of logical expressions in a closed-world system. All of these models may support the query in the sense that the query has in it a projection of the model. In this case the query may be asserted as a contingent fact and incorporated into the world (cf. Prolog's rules). Note that Prolog does not usually incorporate proved theorems into the system, although it is easy to do so if required. The models may however contradict the query and form grounds for asserting the negation of the query. Between these two extremes lie cases where some models support and some contradict the query. There must be therefore an evaluation carried out of the models in the light of the query to see whether there are grounds for asserting the query, its negation or refining them further.

5. The components of a conceptual programming system

The conceptual programming environment consists of: an *editor*, for program creation and modification; a *browser* or *inspector*, for viewing the system's parts and their relationships; and an *interpreter*, for execution. In fact, CP is being used as a test-bed for work in model-generative reasoning, or MGR (Coombs, Hartley, 1987). A series of interpreters incorporating these ideas is being implemented. The example to be presented in section 7 illustrates some of these ideas.

In conceptual programming, it is the editor's job to maintain canonicality of definitions. It is better, therefore, to think of it as a knowledge acquisition utility for the terminological component. The notion here is similar to Nikl's classifier, with the added constraint of canonicality (Schmolze and Lipkis, 1983).

There are many structures for the browser to keep track of. Firstly, the terminological component needs to be displayed at coarse and fine levels of detail. This includes definition through types and schematic clusters of concepts, relations and their procedural components. Second is the more problem-specific knowledge which changes through assertion and de-assertion, the assertional component. These are canonical graphs which contain specializations of type definitions joined in appropriate ways. They express something like the episodic memory of the problem-solver, whereas the type hierarchy is an expression of semantic or universal memory. Thirdly comes the representation of state i.e the pattern of enabled and triggered actors. The browser can report on which actors have been triggered (the program trace) and which are partially or completely enabled but not yet triggered.

The interpreter for conceptual programs is the subject of the next section, but here it can be said that its basic control mechanism is the conceptual join. Facts selected by the user are joined to the terminological component to produce interpretations (the contexts referred to previously). The procedural content of these (from the actors built into definitions) form *programs*. Prolog's unifier is the logical-level counterpart of the conceptual join, but until Prolog accepts types (and semantic types, not just arbitrarily complex data structures) it will remain as such.

In the area of debugging, conceptual programming has a potential advantage over conventional languages. instead of having to deal with low-level idiosyncracies of the language, the programmer can interact with the program at the level of the original problem formulation. the alteration of concept definitions is a much more meaningful activity then the re-formulation of the implementation of an algorithm, or the tinkering with declarative/procedural specifications in either Prolog or a formal KR system.

6. Algorithms in the CP interpreter

CP is not a programming language in the normal sense. It does not compute with assignment to a store as in Pascal, Ada or Modula. nor does it use variable binding as in Lisp and Prolog. All of these are logical-level languages in that they are epistemologically neutral. CP, however, is not neutral; it presents a particular view of how knowledge is best represented and manipulated to perform tasks of problem-solving and reasoning. Conceptual graphs and their associated operations (join and project especially), together with the CP routines for handling actors, form the logical-level underpinnings and can be said to be the basis of computation.

At the epistemological level, computations based on these primitives are made on facts, contexts, definitions, procedural overlays, programs and models. all of these categories of knowledge can be mapped to structures, all of which are conceptual graphs at base. Functionally speaking, the following four algorithms are involved:

interpret: $A \times D' \rightarrow C$

proc-overlay: $C \times PO \rightarrow P$

execute: $P \rightarrow M$

evaluate: $M \times F' \rightarrow E$

The abbreviations denote sets of the epistemological entities as follows: A = assumptions, and $A \subset F$, the facts; $D' \subset D$, the concept definitions; C = contexts; PO = procedural overlays; P = programs; M = models; $F' \subset F$; E = explanations.

Although the implementation of these algorithms is variable, depending on the particular problem-solving strategy employed, their specifications are currently as follows:

interpret:

Join the assumption set to the set D of definitions, so that the number of concepts in it that are covered by concepts in definitions is maximized and the excess concepts in these definitions (i.e. not mentioned in A) is minimized. a join can be rejected when either count is inadequate. The result of interpret is a set of contexts each covering the assumption concepts with a mutually exclusive set of definitions. The way in which interpret works is a major component of strategy.

proc-overlay:

Each definition can have, in general, many procedural overlays, each representing a different type of strategic component. For each context, there may therefore be several programs. Selection may be made on the basis of a chosen strategy (e.g. one which mentions particular attributes or other concepts).

execute:

Programs are essentially local rule-systems which express causality through a theory of action similar in spirit to Rieger's common-sense algorithms (Rieger, 1976) or any other sort of inference with the same epistemology. This means one involving independent, parallel actors, with typed pre- and post-conditions. The results of running a program are models in which causal components are replaced by a program 'trace' in the form of a time chart (cf. Dean and McDermott's temporal data-base, 1987).

evaluate:

Since models contain more concepts than the initial set A, they can be supported or contradicted by further facts. A join therefore between each model and the facts to achieve the same sort of cover as in interpret can determine the extent of this support. Contradictory facts may support different models and thus be separately true, while being conjointly inconsistent.

7. Execution of a CP program

In order to illustrate the operation of a CP interpreter a simple example will be presented. Although it displays many of the features of the conceptual programming methodology, it is a closed-world example which can be programmed in Prolog. The crucial aspects of open-world programming i.e. the evaluation of the models generated from query interpretations and the revision of the query in the light of it is too long to display here.

The example concerns a fragment of knowledge about the nature of paint and whether or not it will peel off a wet surface (the example is due to Mike Coombs). Only two types of paint will be represented: oil paint which does peel off and acrylic paint which does not, because it absorbs any surface water. Clearly there is scope here for uncertainty (how

much water can acrylic paint absorb? can oil paint tolerate a little moisture?) but this is not handled in the example in order to keep it simple. The definitions and interpretations are presented in graphical form since graphs are easier to read†.

Firstly, the definitions of oil paint and acrylic paint. Oil paint is defined as peeling off a wet surface (figure 3) and acrylic paint as absorbing surface water (figure 4). Note that the actors produce the correct event (peeling or absorbing) as long as their inputs are instantiated.

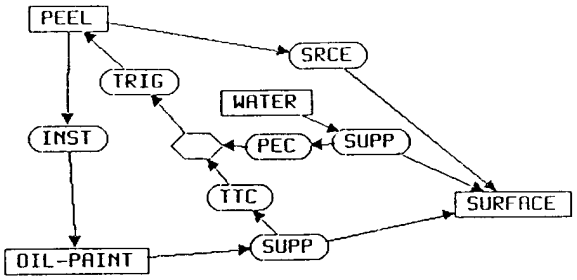

Figure 3. Definition of oil paint with procedural overlay.

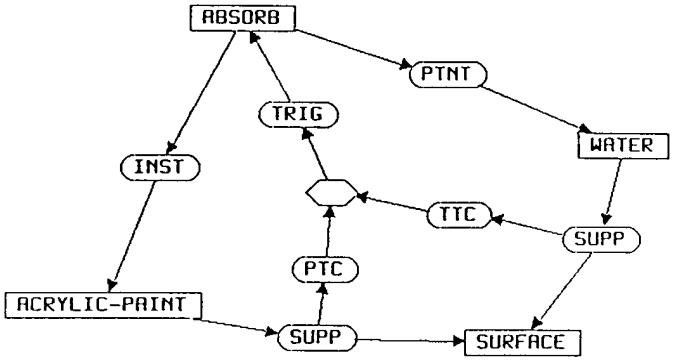

Figure 4. Definition of acrylic paint with procedural overlay.

† the graphs are hardcopy prints of the window of the graph definition system which forms the knowledge acquisition interface of CP.

The remaining definition is that of painting a surface. Here it is a simple schema which mentions a brush as an instrument (figure 5).

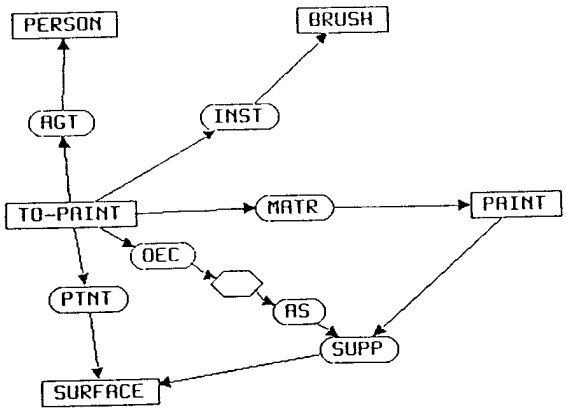

Figure 5. Definition of painting a surface with procedural overlay.

The query which will be presented to the system is ‘‘When painting a wet surface, will paint stay on it?’’ It is in two parts, an *assumption*, namely ‘‘painting a wet surface’’ and a *focus* ‘‘the surface is painted after painting has finished’’. The assumption will be used to form the query interpretations i.e. to put the query in some meaningful context. The actual question (will the surface remain painted?) will be used to evaluate the models generated from these interpretations. Figure 6 presents this information in the form of two graphs. Notice the inclusion of the temporal relation ‘‘STARTS’’ in the focus which is the correct translation of the English version. This is one of Allen’s temporal relations mentioned before.

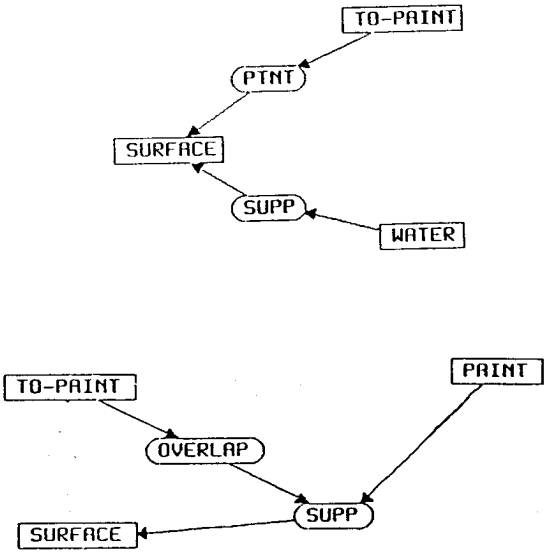

Figure 6. The initial assumption and the query focus.

The first step is to interpret the query in the light of any assumptions which may need to be made. Coverage of the assumption concepts (TO-PAINT, SURFACE, and WATER) needs joins to the definitions of TO-PAINT, and either OIL-PAINT or ACRYLIC-PAINT. The result of these interpretations is given in figure 7.

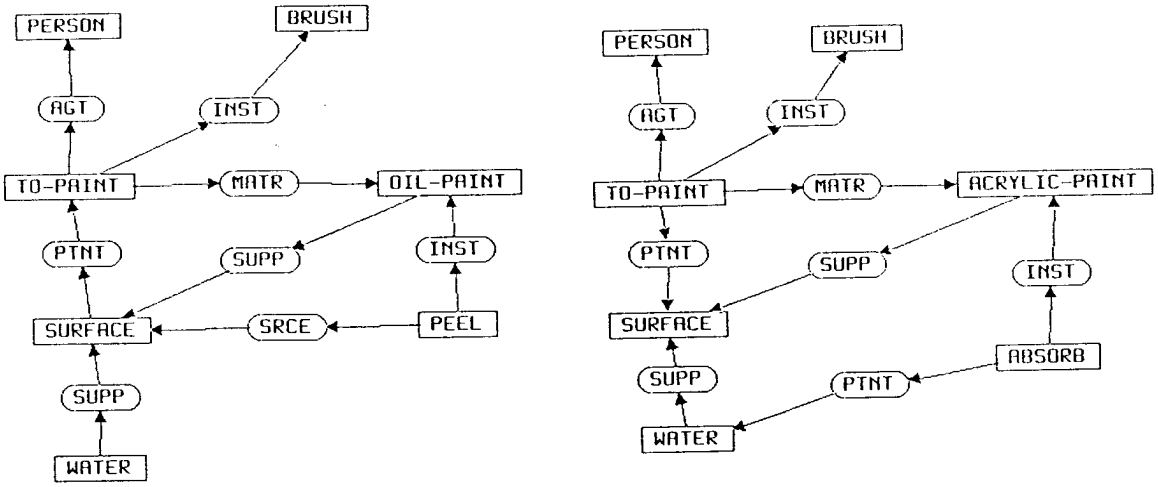

Figure 7. The interpretations for OIL-PAINT and ACRYLIC-PAINT.

The procedural overlays for the three definitions are then joined to each context to produce ‘programs’ which will generate models when the actors contained in them are initiated. The program for ACRYLIC-PAINT is shown in figure 8. The program for OIL-PAINT is similar. There is only one actor with no pre-conditions which are dependent on other actors. It immediately instantiates the state of a surface having water on it. This in

turn causes the absorbing actor to fire, and it ‘absorbs’ the water on the surface. The whole model is a declarative structure containing the temporal relationships between events and states, each having reference to surfaces, people, paints and brushes. The OIL-PAINT program goes through the same sequence, but the model shows how the paint on the surface peels off, allowing the water to remain. The model for ACRYLIC-PAINT is shown in figure 9. It is here shown as containing temporal relations, whereas in fact the system keeps a time chart for all states and events from which these relations may be compiled when necessary.

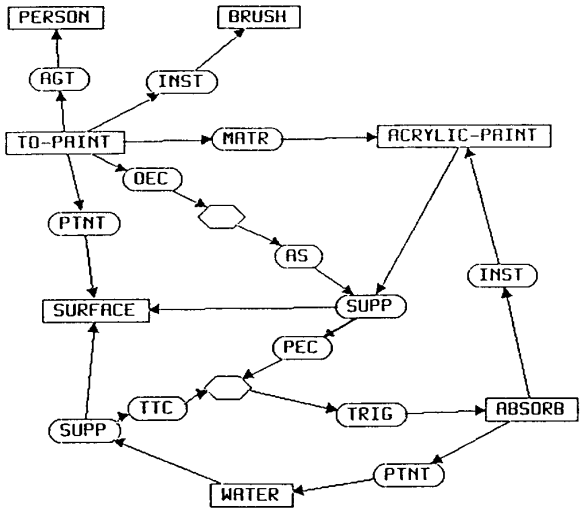

Figure 8. The program for ACRYLIC-PAINT.

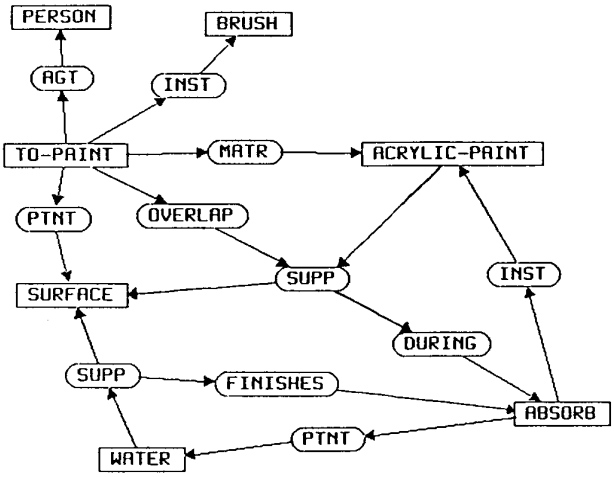

Figure 9. The model for ACRYLIC-PAINT.

The final step is to evaluate the available models and then determine whether the focus part of the query is a projection in any model. In the case of OIL-PAINT this check fails (indicating a contradictory model) whereas the ACRYLIC-PAINT contains the focus as a

projection, thus indicating a supporting model.

Since there is at least one supported model, the answer to the user is ‘yes’, the justification being the existence of one explanation. (the ACRYLIC-PAINT model).

8. The implementation of conceptual programming

An implementation of the prototype conceptual programming system exists on a Symbolics 3670, in Symbolics Common Lisp. Its architecture is shown in figure 10. GDS is the graph definition system, which enables the acquisition of facts, definitions, and procedural overlays in graphical form. A sample window during one such session is shown in figure 11. Since the syntax of conceptual graphs (including our actor extensions) is well-defined, it is impossible to draw graphs which are syntactically incorrect. In order to facilitate the acquisition and display of graphs too large for the screen, a method of overlaying graphs which have common nodes has been implemented. This also allows the display of graphs generated through possibly multiple joins, without having to redraw multiple graphs to produce a nice-looking display. GDS allows browsing of these structures, their creation and modification. In addition, the semantic network can be browsed, in order to show what definitions are currently available. Definitions are bound to a particular user and can be activated or de-activated selectively without the necessity of building a physically separate knowledge base for each new application.

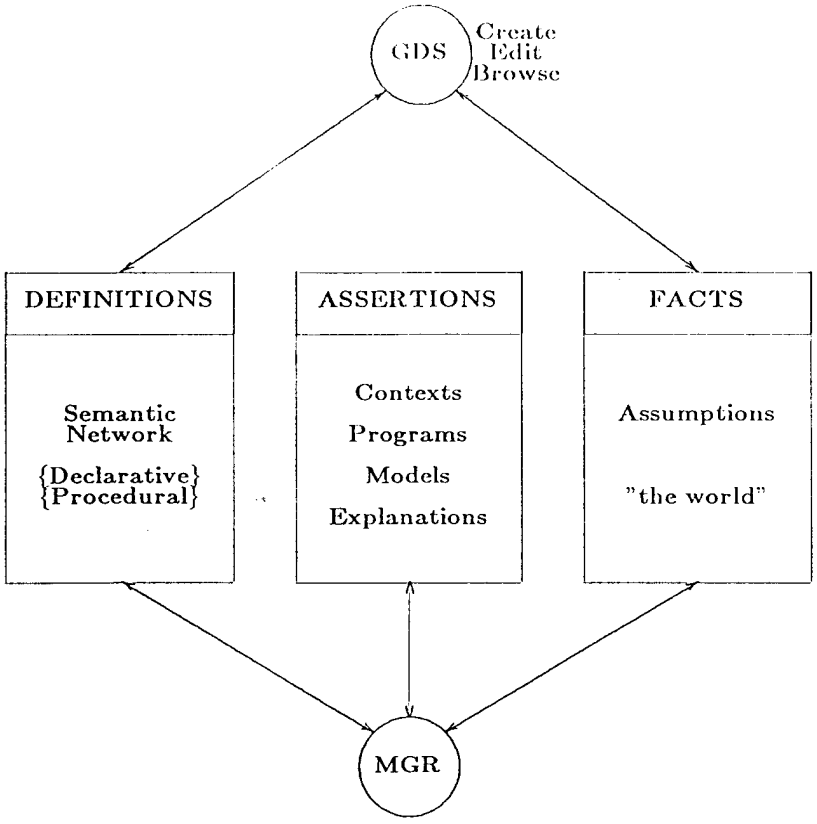

Figure 10. The architecture of CP.

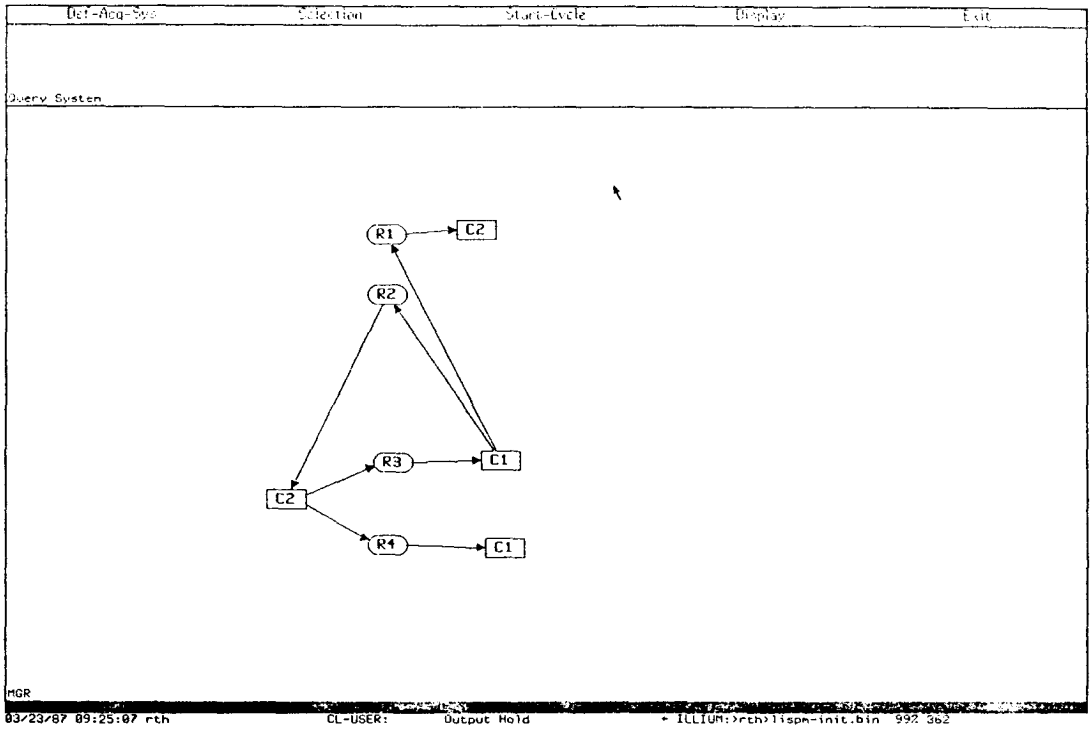

Figure 11. The graph-definition interaction window.

MGR is not one problem-solving system, but is a test-bed for a family of interpreters based on CP, each one eventually illustrating different aspects of the reasoning-by-explanation paradigm. The one described here, the 'Prolog' interpreter, is the first, but already planned are interpreters where the simple interpretation/evaluation sequence is embedded in a cycle, and a more sophisticated one which will focus on the evolution of models, again in a cyclic form. These cyclic interpreters are meant to capture the naturalistic reasoning employed by scientists and idealized in the scientific method of hypothesis generation and testing. More details of this approach can be found in (Coombs and Hartley, 1987).

9. Conclusions

Conceptual programming is a methodology for engineering expert systems. Its underlying representation is conceptual graph theory, augmented to include inferential and state-change actors for procedural knowledge. Expert systems are seen as models of problem-solving including both task-specific and reflective modes of reasoning. The problem-solver can be modeled explicitly.

The conceptual programming environment includes a browser, editor, interpreter and execution aids. It is designed to facilitate the complete production of an expert system from requirements to validation.

Since actors can be defined at the same high level as declarative knowledge, and become part of the generalization hierarchy it is safe to envisage the system as having general purpose capabilities, as with standard procedural languages. The advantage is that concepts may be represented only fractionally below the level at which human beings think, and inconsistencies may be avoided through the constraints of canonicality, maintained through well-defined graph operations.

10. References

Allen, J.F. (1983). Maintaining Knowledge about Temporal Intervals. Comm. ACM 26(11), pp. 832-843.

Attardi, G., Corradini, A., De Cecco, M., and Simi, M. (1984). The Omega Primer. Tech. Report ESP/85/8, Delphi SpA.

Brachman, R.J. (1979). On the epistemological status of semantic networks. In Associative Networks, N.V. Findler (Ed.), Academic Press: New York.

Chandrasekaran, B., (1984). Expert systems: Matching techniques to tasks. New York University Symposium on Artificial Intelligence Applications for Business, 18-20, May 1983. also in Artificial Intelligence Applications for Business. Reitman, W. (Ed.) Ablex.

Clancey, W. J., (1983). The Advantages of Abstract Control Knowledge in Expert System design. Proceedings of the 3rd National Conference on Artificial Intelligence (Washington, D.C.), pp. 74-78.

Dean, T.L., and McDermott, D.V. (1987). Temporal Data Base Management. AI Journal 32(1).

Hartley, R.T. (1981). How expert should an expert system be? Proc. IJCAI-81, Vancouver, pp. 862-867, 1981.

Hartley, R.T. (1984). Expert Systems - A Conceptual Analysis. Int. J. of Systems Research and Information Technology, 1(1).

Hartley, R.T. (1985). Representation of procedural knowledge in expert systems. Proc. of Second IEEE conference on AI applications, Miami.

Hartley, R.T. (1987). Integrating theories of action and temporal relations for the CP environment. CRL monograph. XXXXXXX.

Rieger, Chuck. (1976). An Organization of Knowledge for Problem Solving and Language Comprehension. Artificial Intelligence 7, pp. 89-127.

Schmolze and Lipkis, (1983). Classification in the KL-ONE knowledge representation system. Proc. of IJCAI-83, pp. 330-332.

Sowa, J.F. (1984). Conceptual Structures. Reading, Mass.: Addison Wesley.

van Melle, W., (1980). System Aids in Constructing Consultation Programs. Ann Arbor, Michigan: UMI Research Press, 1981.