The Foundations of Conceptual Programming

Roger T. Hartley
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. This paper presents an overview of an augmented version of John Sowa's conceptual graph theory; a recipe of necessary items for building a conceptual programming system; and an outline of technical difficulties in building the interpreter for the system. Conceptual programming is intended for work in knowledge engineering. A case is made, however, for its inclusion in the repertoire of general-purpose computing language systems.

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 MI 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 difficulties (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 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. Conceptual graphs and procedural knowledge

In his book "Conceptual Structures" (Sowa, 1984) John Sowa describes a knowledge representation scheme as a means of expression of a theory of cognitive functioning. The atoms of the scheme are concepts and conceptual relations which are arranged into *conceptual graphs* according to theoretical ideas about language and the way the world works . The graphs are formed according to a set of formation rules which govern their ultimate structure.

Presenting these rules axiomatically, Sowa gives the *canon* (graphs formed according to the rules are *canonical).* Its parts are:

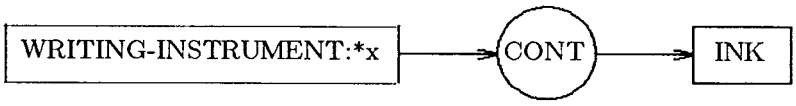
1. A type hierarchy which relates concepts according to the principles of generalization and specialization.
2. A set of individual markers which are the internal map of real-world objects.
3. A conformity relation which ensures that individual markers are tokens of the right concept type.
4. A finite "starter" set of graphs assumed to be canonical.
5. Four formation rules, essentially syntactic in nature with which new canonical graphs can be derived from existing ones. They are: copy, *restrict, join* and *simplify.*

Graphs acquire meaning through the notion of canonicality, and through definitional mechanisms for new concept types. Sowa's *semantic network* also contains links with psychological entities like percepts and emotions, as well as more concrete things like words, rules and procedures. Here, the type hierarchy is considered to be the internal mechanism which propagates meaning from concept to concept. Like all knowledge representation systems, a clear-cut method for determining

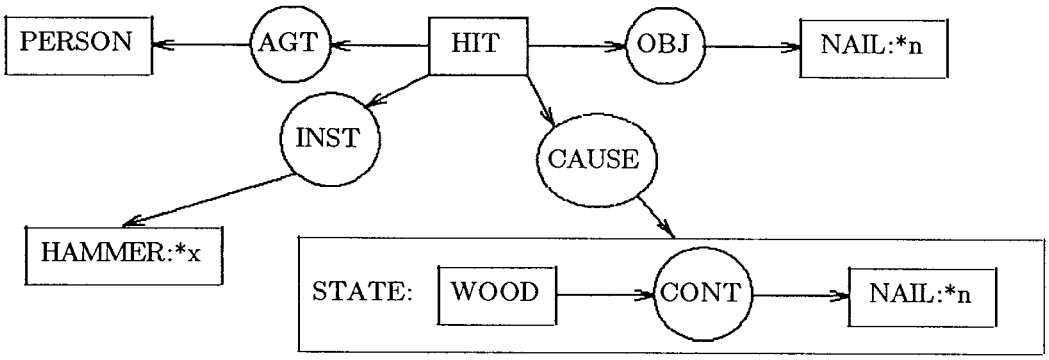
the denotations of the internal symbols is necessary. The individual markers (which are attached to real-world individuals by ostension) and the conformity relation ensure this.

Definitions can be formulated for types (Aristotelian), relations, aggregate individuals, schemata and prototypes. Types are defined through a set of necessary and sufficient conditions, expressed as a graph. On the other hand, a concept can be defined several different ways using less stringent conditions. Such a group of definitions is called a schematic cluster.

type PEN(x) is



schema HAMMER(x) **is**



schema HAMMER(x) **is**

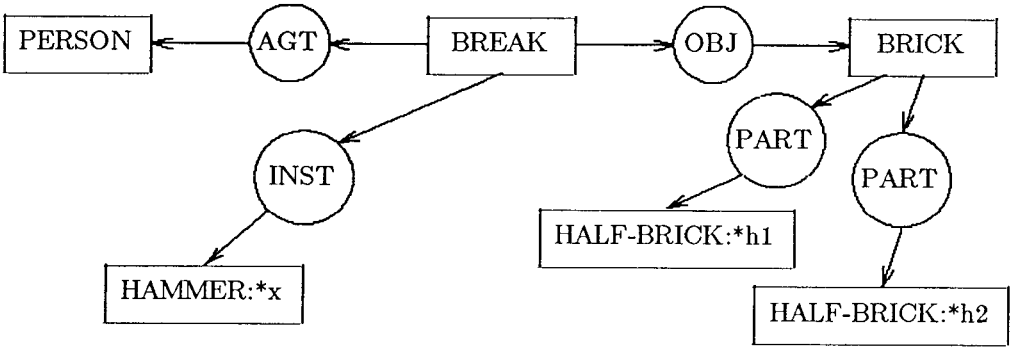


Figure 1. Examples of type and schema definitions.

Sowa also gives ways for handling logical expressions using C.S. Peirce's existential graph notation and for handling deductions in conceptual graphs. These will not concern us in conceptual programming, since it is our intention to make formal logic and its rules of inference a part of the expertise we wish to represent. Moreover, we wish to represent all procedural knowledge in graph form. An extension has been proposed which greatly expands Sowa's use of actors (Hartley, 1985).

Conceptual graphs, as briefly described above, 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. They operate, when their inputs are completely determined, by supplying referents for their output concepts (see figure 2b).

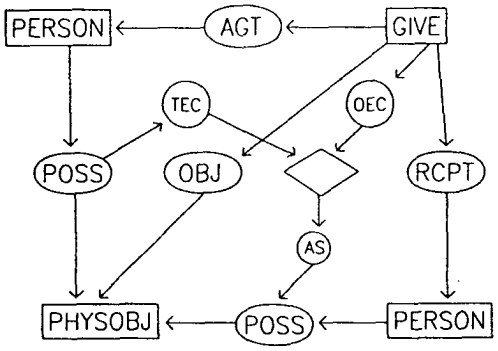


Figure 2a. An example of an actor.

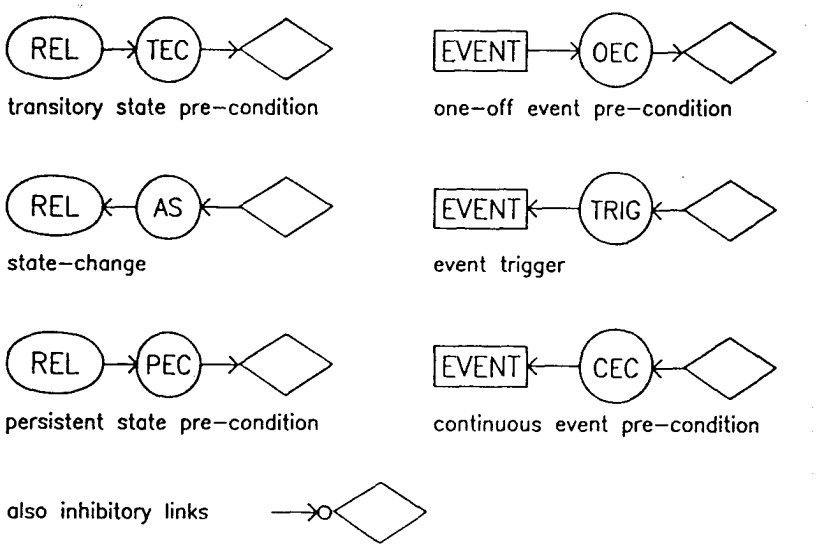


Figure 2b. Actor inputs and outputs.

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. The only difference from conceptual programming is that there is no other relationship between two functions or predicates other than the one of control (one calls the other). However, in 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 flavor 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 (e.g. the time the program takes to execute). 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 prob*lem solver* in the expression of the problem. Most language systems express 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.

4. Comparison with Prolog

A useful way to describe CP is to compare it with Prolog and to show that Prolog is just one special case allowed by it. Operationally, Prolog is a theorem prover over a closed world consisting of a set of a restricted sort of logical expression. 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. 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 *typed* propositions. These are not just assigned true or false as attached values, but are typed according to their method creation. Types include FACT (known to be true), BELIEF, NECESSITY, POSS1131LITY and their negations. Other types may be included by definition. 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 accepting it as a belief, or possibility. The strength of the link between the assumptions and the conclusion in the query is important here because the conclusion may be accepted if, for instance, the query is of the form 'assuming P1, P2, P3... is it *possible* that C'. On the other hand if the link is one of *necessity* then the assertion may not be made.

5. The components of a conceptual programming system

A modern programming environment consists of: an *editor*, for program creation and modification; a *browser* or *inspector*, for viewing the program's parts and their relationships; an *interpreter*, for test execution; a *compiler* for production programs (and efficiency); and an *execution monitor* for understanding faulty programs (debugging etc.).

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, prototypes and schematic clusters of concepts, relations and actors. Second is the more problemspecific 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. Assertions from 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*. Some of these programs will contain actors with no pre-conditions; they will execute immediately. When actors fire they cause assertions to be made (and possibly de-assertions) which in turn enable further actors. Eventually no actor will have enabling pre-conditions; the program then terminates. The

similarity of this to a production system is not accidental. The differences come in two areas. Firstly, production systems usually operate rules either forwards of backwards, whereas conceptual programming can do both naturally. Secondly, production systems have a syntactic patternmatcher for matching assertions to rules. The conceptual join is a much more powerful mechanism (although similar in effect) since it preserves canonicality. 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 really starts to win 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 reformulation of the implementation of an algorithm, or the tinkering with declarative/procedural specifications in either Prolog or a formal KR system.

6. Representation of strategy and the conceptual programming interpreter

One of the most difficult issues for the technology of knowledge engineering is that of the representation of problem-solving strategies. It is no longer acceptable to present a knowledge engineer with a system which presents only one strategy, and an unchangeable one at that. This was true of early KE systems like EMYCIN, KAS, EXPERT etc. Moreover, the provision of forward or backward chaining rules in KEE or MRS is also not the end of the story. Experts can utilize many problem-solving strategies, sometimes locally, within the same problem and the successful and natural expression of these is important for new KE systems. Also the representation and use of general-purpose (common-sense or "heuristic") knowledge is clearly something which distinguishes experts from common expert systems. Here conceptual programming can help by being flexible in the ways in which actors can represent rules and other strategic devices.

Task-specific rules can be represented to run either forwards for deduction or prediction and backwards for abduction or induction. The only subtelty needed here is the representation of propositional types in episodic memory. Propositions can be *known* (through deduction), *possible* (through prediction), *hypothetical* (through abduction), or *generalizable* (through induction). Other types can be incorporated through additions to the canonical basis, or by definition as sub-types. These rules are represented as actors and incorporated into type definitions. An example of their use will be given in section 7.

More difficult, however, is the meta-level activity generally associated with experts who continually monitor their own problem-solving and switch strategies in order to improve performance, or to find a solution where none is forthcoming with the current line of thinking. This can happen when a program falls to produce a model at all (i.e. it contains no fully enabled actors). This is where the workings of the conceptual programming interpreter is important. If we assume that all possible strategies are followed in parallel so that switching of a "line-of-thought" is made easy, then the interpreter clearly has a lot of work to do in keeping track. This does not sound plausible either psychologically or technologically. A better solution is to have a strategy for switching strategies. This is roughly, the "meta-knowledge" approach where meta-rules decide which group of rules (e.g forward or backward) to use next. The meta-rules are considered more important by the interpreter (or they might never get used) so they are examined first on each cycle, assuming a production system is driving them. This does have the advantage of being more sound, but runs the risk of infinite regression (what strategy controls the meta-rules?). Moreover, the idea of rules which reference other rules, rather than just containing task-specific references is psychologically unsound. What is needed is the ability to recognize the applicability of generalized rules to the current situation i.e. to make them task-specific. It is possible that this can be done by specialization, but more likely is that they are triggered by the placing of "reflective" facts (the result of monitoring the problem-solving process) in episodic memory. Here "reflective" is used as in the phrase "quiet reflection". The conceptual programming interpreter achieves this through the ability to trigger more than one actor at a time. Some actors will further task-specific model generation. Some will trigger monitoring actors whose results can then control the problem-solving

actors. In this way explicit control over strategies can be maintained without resort to artificial devices.
The necessity for a monitoring activity to control strategy points to an interpreter capable of supporting parallelism. At any given time many actors will be ready to be triggered and several may be triggered at the same time. This problem has been addressed before (e.g. SNePS: Shapiro et al.) but conceptual programming is different in that the elements of parallelism are components of the representation, not adjuncts to it.

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 axe 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.

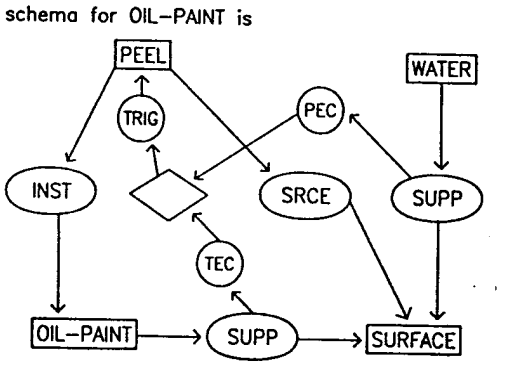


Figure 3. Definition of oil paint.

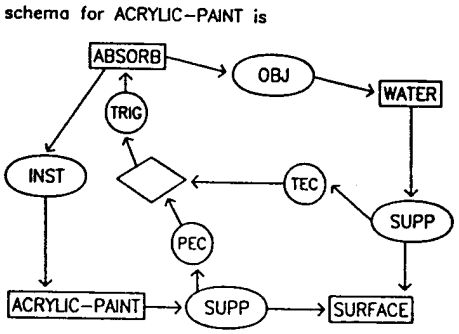


Figure 4. Definition of acrylic paint.

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

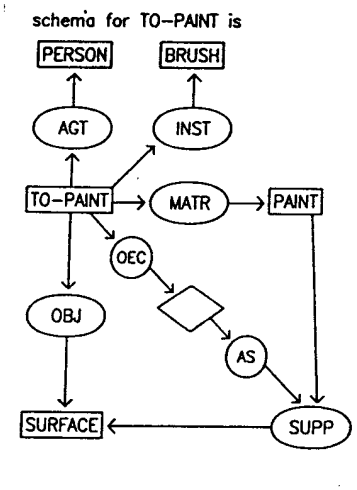


Figure 5. Definition of painting a surface.

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 necessary proposition namely "a painted surface". 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. The propositional mode of the question (necessity, possibility etc.) will determine the style of evaluation. The query may be represented linearly as:

[QUERY] -
(PART) ⟶ [ASSUMPTION: [SURFACE: * s] -
(SUPP) ⟵ [WATER
(OBJ) ⟵ [TO-PAINT],
(PART) [NECESS ITY: [PAINT] ⟶ (SUPP) ⟶ [SURFACE: * s]].

The first step is to interpret the query in the light of any assumptions which may need to be made. By joining the assumption referent to the definition of TO-PAINT we get figure 6. Subsequent separate joins to the definitions of OIL-PAINT and ACRYLIC-PAINT produce the interpretations in figure 7. The shaded boxes indicate the newly joined graphs.

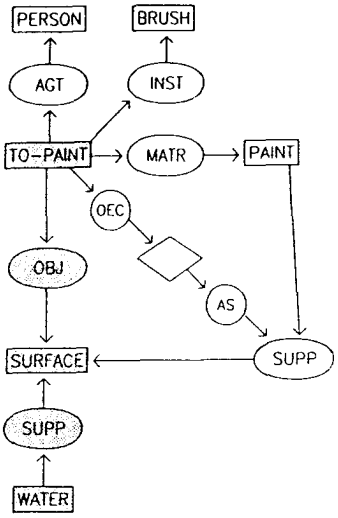


Figure 6. Assumption join.

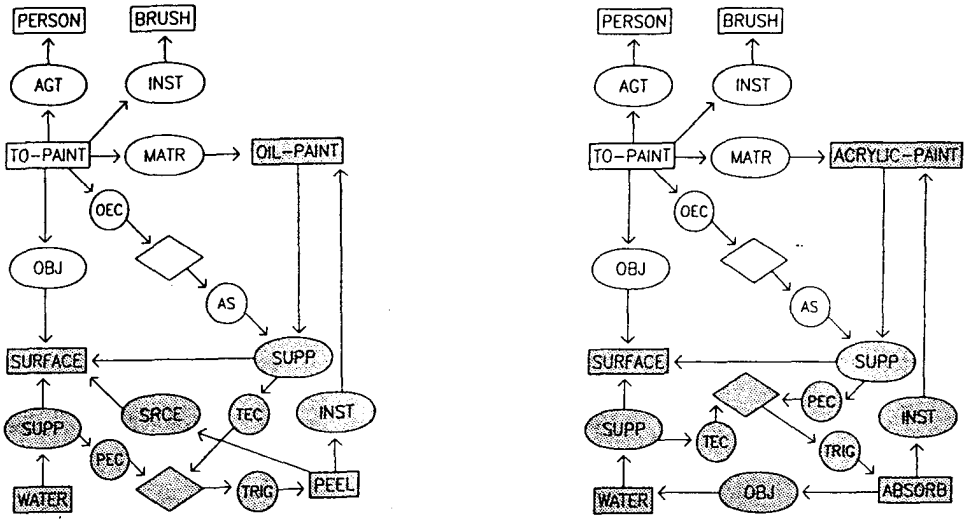


Figure 7. The interpretations for OIL-PAINT and ACRYLIC-PAINT.

The next step is to instantiate the interpretations to produce models. In order to do this a strategy has to be employed. The strategy needed here is simply one of making sure to create a wet surface before painting it. This is the definition in figure 8.

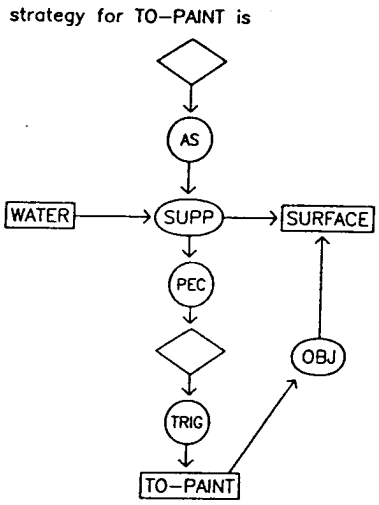


Figure 8. The strategy for painting.

The strategy is joined to each interpretation to produce a 'program' which will generate models when the actors contained in it are initiated. The join for OIL-PAINT is shown in figure 9. The one for ACRYLIC-PAINT is similar. There is only one actor with no pre-conditions. It immediately instantiates the state of a surface having water on it. This in turn starts up the painting actor which then paints the surface. This in turn causes the peeling actor to fire, and it 'removes' the paint from the surface. The whole model is a sequence of events and states each having reference to concrete surfaces, people, paints and brushes. The ACRYLIC-PAINT program goes through the same sequence, but the model shows how the water on the surface is absorbed, allowing the paint to stick. These models are (briefly):

For OIL-PAINT:

- SURFACE s (has) WATER w
- PERSON p TO-PAINT t SURFACE s (using) BRUSH b (with) OIL-PAINT o
- OIL-PAINT o PEEL (from) SURFACE s
- SURFACE s (has) WATER w

For ACRYLIC-PAINT:

- SURFACE s (has) WATER w
- PERSON p TO-PAINT t SUFRACE s (using) BRUSH b (with) ACRYLIC-PAINT a
- ACRYLIC-PAINT a ABSORB ab WATER w (on) SURFACE s
- ACRYLIC-PAINT a (on) SURFACE s

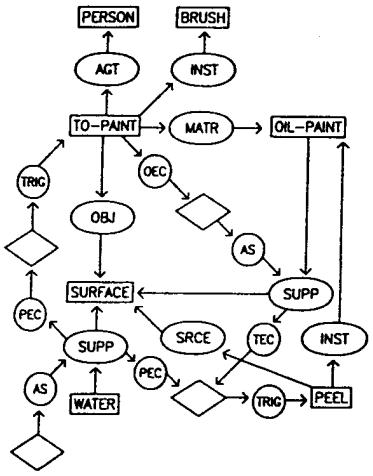


Figure 9. The program for OIL-PAINT.

The final step is to 'answer the question' by joining the available models to the question part of the query. In the case of OIL-PAINT this produces the empty graph (indicating a contradictory model) whereas the ACRYLIC-PAINT model produces :

[ACRYL IC- PAINT: a] -, (SUPP) -> [SURFACE: s

indicating a supporting model. Since the requirement is one of necessity, the answer is effectively 'no' (cf. the Prolog 'no'). However, if the requirement was one of possibility, the answer would be 'yes'. This last stage of model evaluation gives CP great scope for open-world computations.

8. The implementation of conceptual programming

The implementation of a prototype conceptual programming system is proceeding with the help of an Explorer running Common Lisp. This is an ideal environment for such an endeavor, since there is available a very full range of high-level constructs. Most notable are the ability to handle multiple processes, which will help with the parallel nature of actors, and the graphics interface which can case the implementation of the browser. The first test-bed version of conceptual programming is expected-by the end of 1986.

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 enables 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**

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.

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.

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.