MGR:

An Architecture for Problem Solving in Unstructured Environments

C. A. Fields, M. J. Coombs, and R. T. Hartley

Knowledge Systems Group
Computing Research Laboratory
New Mexico State University
Las Cruces, NM 88003-0001

505-646-5466

1. Introduction.

Model Generative Reasoning (MGR) is a problem solving architecture designed to support AI applications in *unstructured* task environments, i.e. in task environments in which the statistical quality, completeness, and relevance of the data that will be available to the problem solver at any particular time cannot be predicted reliably (Hartley et al., 1987; Coombs and Hartley, *in press* a; Coombs et al., *submitted*). MGR was originally developed to solve problems in decision support for process control and fault diagnosis (Coombs and Hartley, *in press* b). Current application areas include scenario-based meteorological data integration (Coombs et al., 1988) and automated nucleic-acid and protein sequence data analysis.

In the present paper, we motivate the design of the MGR architecture by considering the software engineering problems posed by unstructured environments, and the assumptions that must be made in order to address these problems using conventional approaches to automated problem solving. We describe the MGR architecture formally, and show that it is capable, in principle, of addressing these problems. We then describe a dynamic control system being developed to run on the MGR architecture, and show that it provides the ability to generate novel problem solving strategies in response to real-time features of the task environment. We conclude with a discussion of work in progress.

2. Problem Solving in Unstructured Environments.

Many task environments in which intelligent, autonomous systems are needed for operational use are characterized by noisy data, incompletely specified or otherwise ill-posed problems, and a high potential for novelty. Examples of such task environments include visual scene analysis (Poggio et al., 1985), decision support for operators of complex physical plants (Woods, 1986), meteorological data fusion (Coombs et al., 1988), and intelligence data integration and situation analysis (Thompson et al., 1986). In these task environments, and in many others, the data available at any one time may be statistically uncertain, fragmentary, overspecialized or overgeneralized, mutually incoherent, or unrelated to the task at hand. The structure of the data that will be available to a problem solver in such a task environment at any given time cannot be predicted reliably; hence such task environments are unstructured.

A problem solver functioning in an unstructured task environment must be capable of solving problems in the face of noise, incomplete specifications, and novelty. Data relevant to a single problem may, moreover, arrive at different times, and the data available at any one time may be relevant to a number of distinct problems; a problem solver functioning in such a task environment must, therefore, not only cope

with the problems of noise, incomplete specification, and uncertain relevance, but must do so in a data environment that changes in real time.

The problem of noise has traditionally been viewed by AI researchers in terms of statistical uncertainty (e.g. Shortliffe and Buchanan, 1984). In standard treatments of statistical uncertainty, it is assumed that certainty factors are assigned, by the knowledge engineer, to alternative *interpretations* of the raw data (e.g. to antecedent clauses of rules). In an unstructured environment, however, the knowledge engineer may be unable to predict either the range of variation of the uncertainty of the raw data, or the correct interpretation of a given noisy signal when it is removed from its real-time context, which may include other signals that allow it to be disambiguated. A treatment of noise that requires certainty factors to be assigned in advance by the knowledge engineer is unsatisfactorily in such an environment; what is needed is a procedure by which the problem solver itself can interpret noisy data, and assign certainty factors to its interpretations.

Incomplete problem specification is typically dealt with using a case-based (e.g. Kolodner et al., 1985; Hammond, 1986) or default (e.g. McCarthy, 1980; Reiter, 1980; Yager, 1987) reasoning strategy. This approach is adequate in task environments in which the data may be unreliable, provided that the set of data that may be relevant to the problem, and that may therefore influence the choice of case or default strategy to execute, can be circumscribed. The broader problem of uncertain relevance has previously been dealt with by incorporating a nonmonotonic relevance logic into the control structure of the problem solver; this is the approach taken, for example, in TMSs (Doyle, 1979; de Kleer, 1986; Reiter and de Kleer, 1987). Such a relevance logic enables the problem solver to select alternative sets of relevance relations depending on its input; the problem solver is allowed, in effect, to choose an alternative model of the world with a different set of relevance relations if its current model has been shown to be false. In this approach, the problem of determining the relevance of available data or knowledge to the particular problem currently being addressed is viewed as identical to the problem of determining the *true* set of relevance relations, i.e. the true circumscription of relevant data for the problem being solved. The system models used in the fault diagnosis systems of Reiter (1987) and de Kleer and Williams (1987), for example, can be thought of in terms of alternative sets of relevance relations in this way. The problem of relevance is, in these systems, viewed as a problem of determining the reliability of the system model, i.e. of the knowledge structure that specifies the relevance relations.

The problem of determining whether a datum is relevant to a given problem is conflated with the problem of determining the reliability of the data in a database or knowledge structure as a consequence of the ‘‘hand-crafting’’ of systems to solve only problems of a single, prespecified type. In this situation, the software engineer can determine, through constraints on acceptable input, the types of data that will be treated as relevant, and the types of knowledge that will be employed by the problem solver. The problem solver itself cannot alter either relevance assignment in any way. It is, however, to be expected that systems designed in this fashion will become progressively more brittle as their task environments become less structured. Outside of the system’s predefined operating range, it will be unable to distinguish between unexpected but relevant data that is indicative of some important novel situation and expected but irrelevant or misleading data.

In an unstructured task environment, the problem of relevance cannot be reduced to a problem of reliability. In an open world, there may be no single, well-defined set of relevance relations; i.e. circumscription may be impossible in principle (Hewitt, 1985). Even if circumscription is possible in principle (the philosophical realist position), it may be impossible from the point of view of practical software engineering. If this is the case, relevance relations will have to be *constructed* by the problem solver itself on the basis of the structure of the available data in real time.

We have developed the MGR architecture in an attempt to address the problems posed by unstructured task environments directly. Our goal is to develop an architecture that can be used to design and develop autonomous problem solvers for task environments in which the availabilty, recognizability, perishability, and relevance of data cannot be specified in advance. Our approach focusses on designing for *plasticity*; a problem solver in an unstructured environment must be capable of responding appropriately to unanticipated problems the first time that they occur (Fields and Dietrich, 1987a). This focus on immediate response distinguishes the MGR approach from learning approaches to the brittleness problem (e.g. Holland, 1986; Rumelhart et al., 1986); learning systems require either experience in the task environment to

learn appropriate responses, or training by a knowledge engineer who can foresee the structure of the task environment.

MGR systems are capable of autonomously selecting a subset of the available data to treat as relevant in a particular problem solving situation, generating a model of the situation to cover the selected data by adding information from memory, and evaluating the model for its explanatory power. Data that are not selected in a particular context are regarded as irrelevant. These relevance assignments for data are independent of the system's assignments of uncertainty factors to the data items. Similarly, stored knowledge is regarded as relevant if and only if it covers data that are regarded as relevant. Relevance is thus defined operationally in response to the real-time structure of the data, and independently of the certainty assigned to either data or knowledge. Certainty factors are employed only in the evaluation of possible solutions. In MGR, filtering out noise from input data and developing hypotheses which explain these data are two aspects of the same problem, and are solved together.

3. The MGR Architecture.

Overview.

The objective of an MGR problem solver is to explain facts, derived from observations of the world, using stored knowledge. Explanations take the form of descriptive structures - *models* - that provide coherent interpretations - *covers* - of subsets of the available facts. The problem solver is assumed to generate, evaluate, and refine its models continuously, driven by new input from its environment. MGR problem solvers are not goal driven in the traditional sense, in that they have no fixed expectations that determine the structures of solutions.

The MGR architecture is shown in block-diagram form in Fig. 1. The architecture comprises two processing modules, an *analysis* module that has access to facts, and an *interpretation* module that has access to stored knowledge in the form of definitions of the terms in the language used to describe facts. Models cycle between these two processing modules. The analysis module selects a subset of the available facts to treat as initial models in the current problem solving cycle. These models are specialized by the interpretation module, which joins the facts to definitions of terms that are used in their descriptions. These specialized models are passed back to the analysis module, where they are used to select additional facts for interpretation. This cycle continues until an acceptable model of the available facts is generated, or until the system can no longer satisfy the input requirements of any of its operators.

Fig. 1. Block diagram of the MGR architecture. The data ingestion module is assumed to be an independent processor that generates descriptions of facts from raw signals.

The MGR cycle illustrated in Fig. 1 is motivated by the view that robust problem solving is a process of *equilibration* between the perhaps conflicting demands of observations and expectations. In an unstructured environment, it is not possible to establish relevance relations between data (facts) and knowledge (definitions) ahead of time. The relevance of a datum can only be determined by its usefullness as a seed for model generation, and its reliability can only be determined by the coherence of the models that it seeds. Similarly, the relevance of a knowledge item can only be determined by its usefullness in covering data, and its reliability can only be determined by its coherence with the data. A problem solver operating in such an environment must, therefore, balance its knowledge-based expectations of the environment's structure against the data that it actually receives when it observes the world. In MGR, expectations drive the selection of facts to treat as relevant, and hence limit the range of facts that are considered on any cycle. Observations drive the selection of knowledge to treat as relevant, and hence limit the range of knowledge

that is drawn on to explain the facts in any cycle. These processes will equilibrate, and hence converge on a solution, as long as a significant subset of the facts cohere with a significant subset of the available knowledge.

This view of problem solving as a process of equilibration is motivated by a number of considerations, including general systems theoretic (Ashby, 1952; Pask, 1975), developmental (Piaget, 1977), and simulated neural network (Grossberg, 1980; 1987) analyses of learning. The control structure that defines ''equilibration'' operationally is specified as a symbolic algorithm - the *MGR algorithm* - in the current implementation of the MGR architecture (Hartley et al, 1987; Coombs and Hartley, *in press* a); our current research focusses on the development of a dynamic control structure in which equilibration will be a continuous process (Sect. 4).

Representation language, operators, and control.

The representation language employed in MGR is based on the *conceptual graphs* formalism developed by Sowa (1984). The language has two components, a set **G** of conceptual graphs, and an associated type hierarchy **T**. A conceptual graph is a finite, simple, connected, bipartite digraph; that is: i) no two nodes of a conceptual graph are connected by more than one arc, ii) no nodes are isolated, and iii) the set of nodes is the union of two disjoint subsets **O** and **R**, such that every arc connects an element of **O** with an element of **R**. Elements of **O** and **R** are interpreted as *objects* and *relations*, respectively. Every node has a label taken from a finite set L(**G**). Two nodes of a single conceptual graph can have the same label; but the set L(**O**) of labels of object nodes is disjoint from the set L(**R**) of labels of relation nodes (i.e. conceptual graphs are *colored*).

Conceptual graphs may contain object nodes that are interpreted as *actors*, in which case the graph is referred to as a *procedural overlay*. We have extended the actor formalism developed by (Sowa, 1984) to allow actors to function as ''active concepts'' that accept states as preconditions and events as triggers (Hartley et al., *in press*). Actors are executable, and provide a representation for processes in the conceptual graph formalism; procedural overlays are, therefore, interpreted as procedural definitions of terms that refer to processes. When an actor is executed, the actor node and the relation nodes adjacent to it are replaced, as a unit, by a relation node labeled with the duration (if any) of the process (Allen, 1983), or with a nontemporal dependence relation such as [CAUSES]. Formally, the actor nodes form a proper subset of the object nodes. All actor nodes have the same label (which is taken to be **blank**). An example conceptual graph containing object, relation, and actor nodes is shown in Fig. 2.

The elements of L(**G**) are organized into a type hierarchy **T**. All elements of L(**G**), including **blank**, are included in **T**. **T** is capable of supporting vertical inheritance (Sowa, 1984); vertical inheritance is not, however, used in MGR.

A data flow diagram of the MGR architecture is shown in Fig. 3. MGR accepts input from two databases, the *fact* database F and the *definition* database D. The contents of these databases are sets **F** and **D** of conceptual graphs representing facts and definitions, respectively, such that $\mathbf{F} \cap \mathbf{D} = \varnothing$ and $\mathbf{F} \cup \mathbf{D} = \mathbf{G}$. Two restrictions apply: i) for every label $l \in$ L(**G**), there must be at least one graph in **D** containing a node labelled *l*, and ii) no graph in **F** can contain actor nodes. In addition, there is a store M containing a set **M** of *models*, which are conceptual graphs generated by the execution of operators.

Sowa (1984) defines join and project operators on conceptual graphs by analogy to the database join and project operators. We define **join** and **project** operators in terms of those of Sowa as follows. **join** takes as input two conceptual graphs g_i and g_j, and returns their maximal join with respect to **T**. This maximal join is computed by replacing every pair of labels l_i and l_j of nodes n_i and n_j of g_i and g_j, respectively, with their greatest common specialization (GCS), provided that their GCS exists and is not identical to **Bottom**, and then identifying n_i and n_j. **project** takes as input two conceptual graphs g_i and g_j, and returns their maximal projection with respect to **T**. This maximal projection is computed by replacing every pair of labels l_i and l_j of nodes n_i and n_j of g_i and g_j, respectively, with their least common generalization (LCG), and then identifying n_i and n_j. Both of these operators are generally nondeterministic, especially in the case in which either input graph contains two or more nodes with a single label. These definitions can be straightforwardly extended to the case of n inputs.

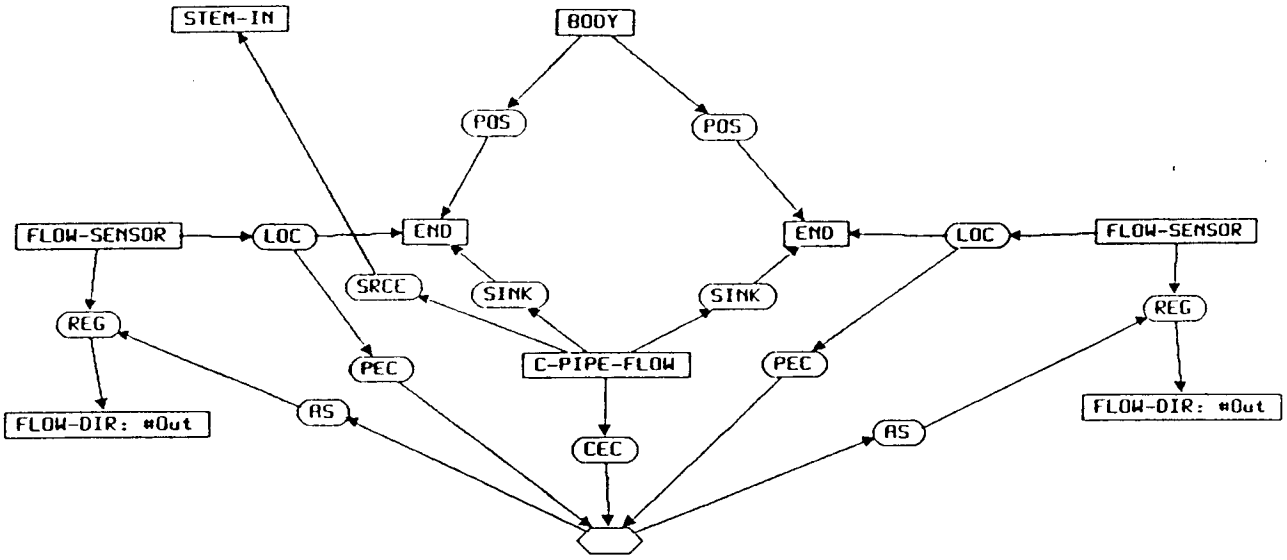

Fig. 2: Example conceptual graph. Object nodes (other than actors) are represented as rectangles, relation nodes as ovals, and actor nodes as diamonds. Relations are read in the directions indicated by the arrows. Relations into actors represent enabling (EC) conditions; relations out of actors represent assertions (AS) (Coombs and Hartley, *in press* a). The conceptual graph shown represents a pipe with an input T-junction and two flow-sensors, each of which reads ''out'' (from Coombs and Hartley, *in press* a).

MGR employs four operators: *Generalize* **Gn**, *Merge* **Mr**, *Specialize* **Sp**, and *Fragment* **Fr**. These operators are implemented using the **join** and **project** operators defined above. The operators are defined as mappings over the sets **F**, **D**, and **M** as follows:

Gn: PS(**M**) $\rightarrow$ **M**

Mr: PS(**M**) $\rightarrow$ **M**

Sp: **M** $\times$ PS(**D**) $\rightarrow$ PS(**M**)

Fr: **M** $\times$ PS(**F**) $\rightarrow$ PS(**M**)

The notation PS(**X**) denotes the power set of the set **X**.

The MGR operators are implemented as follows. **Gn** takes as input a subset **M´** of **M**, and returns the **project** of the models in **M´**. **Mr** takes as input a subset **M´** of **M**, and returns the **join** of the models in **M´**. **Sp** takes as input a subset **D´** of **D** and a model $m \in$ **M**, and returns a set **M´** of models, each of which is a **join** of m with all of the elements of **D´** with which it is joinable. **Fr** takes as input a subset **F´** of **F** and a model $m \in$ **M**. **Fr** then computes the **join** f of the elements of **F´**. If m has a set of disconnected subgraphs g_i ... g_k such that **project**(g_i ... g_k) = f, then **Fr** returns $\{g_i,...,g_k\}$.

Both **Fr** and **Sp** are defined for the special case in which the $m \in$ **M** that they take as input is the null graph. In this case, **Fr** and **Sp** return the input subsets **F´** and **D´**, respectively. This special case provides a route for introducing facts and definitions, respectively, directly into **M**.

Informally, **Sp** takes a model as input, and generates a set of larger, more specialized models by adding definitions. The role of **Sp** is, therefore, to ''glue'' knowledge to facts to create models that cover the facts. **Fr** opposes **Sp** by breaking models into fragments in a way that preserves the information contained in facts, but may destroy information obtained from definitions. **Fr** thus generalizes definitional

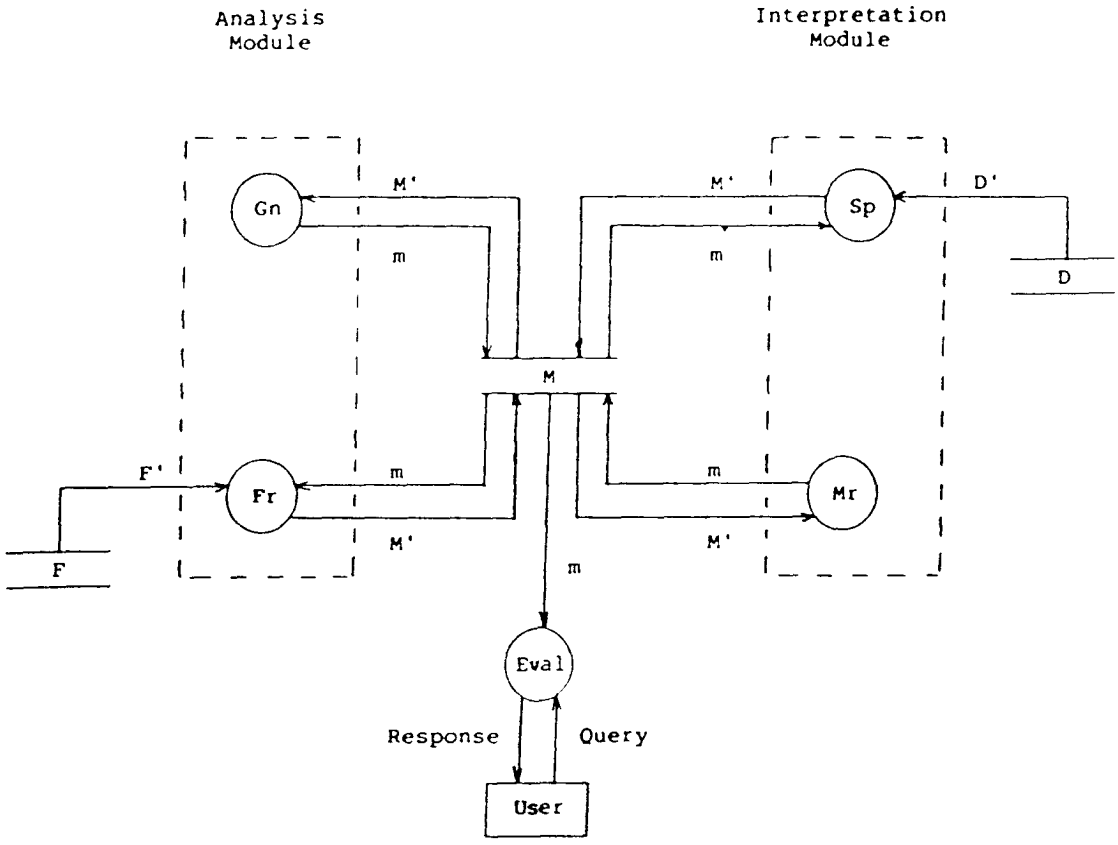

Fig. 2: Data flow diagram of the MGR architecture, using the notation of Yourdan and Constantine (1979). The *analysis* and *interpretation* modules are as in Fig. 1; they are interpreted as maintaining coherence between the model population in **M** and the fact and definition populations, respectively. Users interact with MGR through an *evaluation* module, which samples the model population for models that cover a user *query*, which is a conceptual graph not in **G**, containing no actors, and labelled entirely with labels in L(**G**).

information, but not factual information. The role of **Fr** is to break apart models that do not cohere with all of the available the facts in order to generate fragments that can be recombined. **Gn** and **Mr** take subsets of models as input, and generate single models as output which are, respectively, less or more specialized than the models from which they were generated. **Gn** is capable of generalizing both factual and definitional information; its role is to maintain coherence with the facts by removing over-specializations. **Mr** merges models whenever possible; its role is to generate models that have the greatest possible covering power. All of the operators write over the model population; the set of models available for operating on, therefore, changes on every cycle.

MGR is logically a shared data concurrent machine, in which each operator is a single-instruction, multiple-data (SIMD) processor. MGR is, therefore, most naturally implemented in a concurrent setting. A *control function* for the MGR architecture is a function that regulates the execution of the MGR operators, by regulating their firing order, their input requirements, or their rates of execution. All of the MGR operators are capable of destroying information as well as creating it, and if any one operator is allowed to run unhindered, it will remove most if not all of the information present in **M**. For example, **Gn** can generalize all models to **Top**, while **Mr** can create a single model in which all distinctions between facts and

definitions are lost. The key to control in MGR is to balance the operators against each other in a way that results in the overall addition of information to **M**, and hence the generation of models with explanatory power.

The *initial* state of MGR is the state **M** = ∅. MGR is initialized from **F** or, if **F** is empty, from **D**, through the use of **Fr** or **Sp**, respectively. MGR then runs until a *halt* state is reached. MGR halts when either: i) no operator that is allowed to execute by the control function has input, or ii) a **halt** instruction is received from the evaluation module, signifying that the user's query has been answered. As pointed out above, query evaluation in MGR is completely passive; MGR is driven by its inputs from **F** and **D**, not by user-specified goals.

A *symbolic* control function for MGR is a conditional preference order for the execution of operators. Such a preference order is employed as a control function in the current MGR implementation (see below). A symbolic control function must also specify the conditions under which **Fr** and **Sp** can accept the null graph as input, the criteria by which subsets of **F** and **D** are selected for input to **Fr** and **Sp** respectively, and the evaluation function to be executed by the evaluation module.

Different preference orders are appropriate in task environments having different characteristics. In an application task environment such as DNA sequence data analysis, in which the data are all equally specialized, and are of similar quality, but may form mutually incoherent subsets, **Fr** is used to break apart models that fail evaluation, and **Gn** is turned off. In a task environment such as diagnosis in which the data may be overspecialized, **Gn** is needed to introduce generalizations of the data items that may more accurately reflect the state of the world.

As an alternative to generating symbolic control functions for each different task environment, we are developing a general, *dynamic* control function for MGR. The dynamic control function specifies the *rates* of execution of the four operators in a concurrent setting. This function is described below (Sect. 4).

We have proved elsewhere that the MGR architecture is Turing equivalent (Fields et al., *submitted*). With its abilities to select relevant data, generate, decompose, and recombine the fragments of models, and employ flexible evaluation criteria, MGR provides a powerful and very flexible architecture for developing AI applications for unstructured task environments.

Current implementations of MGR and CP.

The MGR architecture has been implemented in a serial, symbolic reasoning system that supports user querying (Coombs and Hartley, *in press* a). Both MGR and the associated Conceptual Programming (CP) environment (Hartley and Coombs, *in press*) are implemented in Symbolics Common Lisp, and run on Symbolics 3600 series computers. Versions of both systems in Ibuki Common Lisp for the Sun 3 are planned.

Conceptual graphs representing facts and definitions are created using the CP environment (Hartley, 1986; Hartley and Coombs, *in press*). CP implements the minimum number of structures required by the current MGR algorithm. Given the importance in MGR of model decomposition and recombination, schemata have been selected as the foundation for all definitions, since they impose the weakest semantic constraints needed for controlling coherent specialization and generalization of concepts. However, the conceptual graphs notation is as expressive as any of the other advanced semantic network or frame-based knowledge representation systems, without having the disadvantage of being optimized for one particular definitional structure. It is capable of supporting a rich variety of definitional mechanisms, including logical definition through Aristotelian types, contextual definition through schematic types and default definition through prototypes. The system may also be readily expanded to express KL-ONE (Brachman and Schmolze, 1985) conceptual roles and role restrictions, in addition to supporting a rich system of quantification and individuation. The conceptual graphs notation has, moreover, a sound formal foundation in lattice theory (Sowa, 1984). CP can, therefore, be readily expanded to accommodate the representational needs of new applications or theoretical research.

The distinction between declarative and procedural definitions currently requires the MGR operators to be implemented as sequences of graph operators. For example, an execution of **Sp** requires three steps in the current implementation. First, the input model is joined to one or more declarative definitions to form a *context*. The context is then completed by joining it to one or more procedural overlays to form a *program*.

When executed, the program results in a *model*, in which the procedural information is replaced by a time chart, which is similar in form to the time maps used by Dean and McDermott (1987) to represent the patterns of coexisting objects and processes.

A *covering* criterion is used to constrain model generation in the current implementation; the set **D´** selected for input into **Sp** is required to contain definitions that provide a minimal - hence parsimonious - cover of the model chosen for specialization. Models are, moreover, evaluated on every cycle for their ability to cover facts not represented in the set of models. If a model covers at least one such additional fact, it is labeled as a *candidate explanation*. In the absense of user interaction, these candidate explanations are the system's output. If a query is present, the candidate explanations are passed to the evaluation module, which tests whether they cover the user's query. Candidate explanations that cover the query are produced as output *explanations* of the query. A detailed example of an MGR problem-solving session is provided in Coombs and Hartley (*in press* a).

Control is defined in the current implementation by the MGR algorithm, which specifies the following preference order for the four operators: **Sp** $\rightarrow$ **Mr** $\rightarrow$ **Fr** (Coombs and Hartley, *in press* a; **Fr** is referred to as *Generalize* in this paper). The current MGR system has four of the basic requirements for reasoning in unstructured environments. These include:

i. the separation of the data analysis and data interpretation functions;

ii. the representation of solutions as coherent conceptual covers of selected data that achieve some externally defined explanatory criterion;

iii. the generation of alternative competing domain models during processing, which are constructed from an amalgam of data and schematic definitions;

iv. an ability to create new knowledge structures (i.e. not trivially covered by existing schemata) by using **Fr** and **Gn** to generate new models from an incoherent model, and **Mr** to create a new complete model from sets of partial models.

The current design requires the presence of a user to seed the system with an initial set of facts, and thus direct attention to certain aspects of the environment. The system is being reimplemented as a shared memory system in which the operators are implemented as separate software modules; the requirement for user seeding will be removed in this new implementation.

4. Concurrent Dynamic Control.

The current MGR algorithm encodes a particular problem solving strategy. While it has proved useful in process control domains (Coombs and Hartley, *in press* b), it may be far from optimal in other domains. In particular, the current MGR algorithm may prevent convergence to a solution in domains in which both **Fr** and **Gn** are needed to solve problems. The MGR algorithm amounts, moreover, to a constraint on the behavior of the problem solver which prevents it from exhibiting plastic behavior in the face of some forms of novelty or incomplete data. If plasticity is regarded as a measure of intelligence (Fields and Dietrich, 1987a), the use of a fixed strategy must be regarded as a deficit in a problem solver.

One approach to alleviating the limitations imposed by a fixed problem solving strategy is to design a strategy that allows the problem solver to emulate the behavior that would result from applying a task-appropriate method to each individual task. This is the general approach taken by Newell's group in the design of their ''Universal Weak Method'' (Laird and Newell, 1983; Laird et al., 1987). The approach that we have taken is similar; however, instead of designing a ''universal'' symbolic weak method that allows emulation of alternative strategies, we have designed a dynamic control structure that allows the arbitrary superposition of strategies within a single execution cycle. This structure allows the MGR system to, in effect, arbitrarily integrate alternative strategies into new methods customized to the particular problem at hand.

A dynamic control strategy is very natural in a concurrent environment. The graphs on which the operators act are viewed as composing three *populations*, the fact, definition, and model populations. The operators select individual graphs from these populations on each execution cycle, execute, and return their outputs to the model population. The control structure controls the behavior of the problem solver by regulating the rates at which the four concurrent operators act on their respective graph populations.

We begin by specifying the *state* of the graph populations. Let **X**(t) represent the state of the population of graphs available to the system at time t, i.e. **X**(t) = <**D**(t), **F**(t), **M**(t)>. This state is *represented* to the control structure as a state vector $\Psi(\mathbf{X})$. The components of $\Psi(\mathbf{X})$ are functions that measure properties of the three populations of graphs.

Control can be defined to be either *deterministic*, in which case it is defined over particular graphs, or *stochastic*, in which case it is defined over populations of graphs that are taken to behave as statistical ensembles. If control is defined deterministically, the result of a particular execution cycle will be predictable, up to the nondeterminism of the individual operators. If control is stochastic, the result of a particular execution cycle will not be predictable, but the average results of an ensemble of cycles of execution on the same graph populations will be predictable. Stochastic control increases the plasticity of the system in the face of novel or incompletely-specified problems (Fields and Dietrich, 1987a; b); however, deterministic control is preferable from a software engineering perspective if the task environment is well-enough structured to choose appropriate selection functions to specify the ways in which inputs are selected for each operator.

The deterministic - stochastic decision determines the state functions used as the components of the state vector. In the case of deterministic control, we define two state functions:

i. the *complexity* of a graph g_i, $\sigma(g_i)$ = [# of arcs in g_i]. This function provides a meaningful measure of complexity, since no two nodes in a graph constructed in CP can be connected by more than one arc.

and

ii. the *covering index* for the pair of graphs g_i and g_j, $\chi(g_i, g_j)$, is defined to be the number of labels, of either object nodes or relation nodes, shared by the graphs g_i and g_j.

In the case of stochastic control, we define four state functions:

i. the *entropy* of the j^{th} population of graphs $\nu(\mathbf{X}_j)$ = -ln [# of graphs in $\mathbf{X}_j$], which measures the relative likelihood that any particular graph in the population will be operated upon in any cycle,

ii. the *average complexity* of the j^{th} population of graphs $\sigma(\mathbf{X}_j)$ = [# of arcs in $\mathbf{X}_j$]/[# of graphs in $\mathbf{X}_j$].

iii. the *average covering index* $\bar{\chi}(\mathbf{X}_i, \mathbf{X}_j)$ for the i^{th} and j^{th} populations of graphs is the average of the covering indices $\chi(g_i, g_j)$, where g_i is in the i^{th} population and g_j is in the j^{th} population.

and

iv. the *maximum covering index*, $\chi_m(\mathbf{X}_i, \mathbf{X}_j)$ for the i^{th} and j^{th} populations of graphs is the maximum of the covering indices $\chi(g_i, g_j)$, where g_i is a member of the i^{th} population and g_j is a member of the j^{th} population.

These functions are clearly linearly independent. The state vector in the deterministic case is, therefore:

$$\Psi(g_i, g_j) = <\sigma(g_i), \sigma(g_j), \chi(g_i, g_j)>,$$

for graphs g_i and g_j. In the stochastic case, the state vector is:

$$\Psi(<\mathbf{X}_i>) = <\nu(\mathbf{X}_i), \sigma(\mathbf{X}_i), \bar{\chi}(\mathbf{X}_i, \mathbf{X}_j), \chi_m(\mathbf{X}_i, \mathbf{X}_j)>.$$

The control function itself is defined in terms of cost and response functions, which depend on the state functions. Each operation is assumed to require time proportional to the complexity of the graphs being operated upon. This *cost* is represented for operator *i* as a function $\zeta_i(\sigma(\oplus\mathbf{X}_i))$, where $\oplus\mathbf{X}_i$ represents the direct sum (disjoint union) of the graph populations on which the operator is defined, and $\sigma(\oplus\mathbf{X}_i)$ represents the sum of the complexities, or in the stochastic case, the average complexities of the graphs in these populations.

The MGR operators are correlated in their global effects on the populations of graphs. The correlation between operators *i* and *j* is represented by a function ξ_{ij}, which is assumed to act on the state vector Ψ. This correlation function represents the *response* of the correlated pair *ij* of operators to the current state of the system; the control function thus acts on correlated pairs of operators through the response functions, and on single operators through the cost functions.

Given these definitions, we specify the change per unit time of the *firing rate* $\mathbf{R}_i$ of the i^{th} operator by:

$$\partial \mathbf{R}_i(\mathbf{X})/\partial t = \zeta_i(\sigma(\oplus \mathbf{X}_i))\ \Sigma\{[\delta_i^\alpha + \delta_i^\beta]\ \xi_{\alpha\beta}\ (\Psi(\mathbf{X}_i))\},$$

where δ_i^α is the Kronecker delta function, i.e. $\Sigma\ \delta_i^\alpha\ \phi_\alpha = \phi_i$. The firing rate $\mathbf{R}_i$ of operator *i* is the time integral of this expression. The *rate vector* of the system is the vector $\mathbf{R}(t) = \langle \mathbf{R}_i(t) \rangle$ of these operator firing rates. This vector describes the behavior of the system through time, and is taken to define the control structure of concurrent dynamic MGR.

The above expression can be simplified considerably if the correlation functions $\xi_{\alpha\beta}$ are assumed to be linear. In this case, $\xi_{\alpha\beta}\ (\Psi)$ can be represented as the inner product $\xi_{\alpha\beta} * \Psi$. The sum of the correlation vectors involving the i^{th} operator, $\Sigma\ [\delta_i^\alpha + \delta_i^\beta]\ \xi_{\alpha\beta}$, can however be represented as a single correlation vector ξ_i. The expression for the inner products of the correlation vectors with the state vector then reduces to $\Sigma\ \xi_i^\gamma\ \Psi_\gamma(\mathbf{X}_i)$, and the expression for the change in firing rate of the i^{th} operator becomes:

$$\partial \mathbf{R}_i(\mathbf{X})/\partial t = \zeta_i(\sigma(\oplus \mathbf{X}_i))\ \Sigma\ \xi_i^\gamma\ \Psi_\gamma(\mathbf{X}_i).$$

This expression is taken to define the control structure of *linear* concurrent dynamic MGR.

The above expression is semilinear, and is indeed isomorphic to the semilinear node functions that characterize most parallel distributed processing (PDP) systems (Rumelhart et al., 1986). The correlation vectors ξ_{ij} can, therefore, be regarded as the components of the weight matrix of a PDP network. The approach to dynamic control outlined above can thus be thought of, in the special case of linear correlation functions, as equivalent to using a PDP machine to calculate the rates of application for a set of concurrent symbolic operators. This fusion of symbolic and connectionist reasoning strategies in a single system represents a novel approach to problem solving architecture.

5. Current Research Issues.

Current research on the MGR architecture focusses on two issues: the development of the concurrent dynamic control structure, and the investigation of effective MGR problem solving strategies. The MGR architecture is, meanwhile, undergoing several enhancements to adapt it to solve problems efficiently in various application domains.

The adequacy of linear correlation functions for representing the correlations one would intuitively expect between the operators employed by MGR has yet to be determined. We expect, however, that nonlinear correlation functions will be required to produce certain desirable behaviors. With linear correlation functions, optimal values of the state variables can only occur at the extremes of their ranges. While this is intuitively reasonable for some state variables, such as the covering index between models and facts, it is not for others. One might, for example, intuitively expect that the optimal value of the average complexity variable for models would be somewhere in the middle of its range. Expressing this intuition requires nonlinear correlation functions, which correspond, in a PDP architecture, to nonlinear weights. Nonlinear weight functions have not been investigated in detail by PDP researchers; however, such functions may more adequately represent biological neural systems in which modulation plays an important role.

Incorporation of a simple learning algorithm into the semilinear dynamic control system will be straightforward. Learning at the level of adjustment of rates of concurrent operations has yet to be investigated in a symbolic problem solver; dynamic MGR provides an architecture in which the effects of rate adjustments can be studied in a controlled setting.

Problem solvers that are capable of arbitrarily constructing strategies have not been investigated previously. We believe that an architecture, such as dynamic MGR, that allows such construction is required both to generate adequate explanatory models of the behavior of autonomous agents in realistic task environments, and to design autonomous problem solvers for unstructured task environments. The software engineering issues that arise when strategy generation is possible have not, however, been investigated previously.

Our current work on strategy generation focusses on emulating semantic analogues of the weak methods (Laird et al., 1987) as emergent properties of concurrent MGR. Manipulation of the relative rates at which the four operators are applied changes the variability, complexity and explanatory power of the

models generated. Starting from blind search, faster **Sp**, for example, produces a depth-like pattern of search, while faster **Gn** is more breadth-like. Hill-climbing behavior appears to emerge from favoring **Mr** along with **Sp**, since **Mr** will tend to generate specialized models that explain large sets of data. Best-first search, which is more conservative, appears when MGR runs **Fr** as a relatively favored background operation. **Fr** ensures that models do not become overly constrained by intensional concepts at an early stage, and so gives maximum opportunity for a model with the greatest explanatory power to evolve. We are currently investigating the implications for strategy generation, and for the conditions under which convergence to a solution will occur, of various alternative input selection criteria for these operators.

References.

Allen, J.F. (1983). Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26, 832-843.

Ashby, W. Ross (1952). *Design for a Brain*. London: Chapman and Hall.

Brachman, R.J. and J.G. Schmolze (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9, 171-216.

Coombs, M. J., C. A. Fields, and G. McWilliams (1988). *Artificial Intelligence Methods for Optimizing the Use of Meteorological Databases: Recommendations for Implementing the MERCURY System*. Final Report, WAO 87-2.10-5, Contract # DAAD07-86-C-0034. US Army Atmospheric Sciences Laboratory.

Coombs, M. J. and R. T. Hartley (*in press* a). The MGR algorithm and its application to the generation of explanations for novel events. *International Journal of Man-Machine Studies*.

Coombs, M. J. and R. T. Hartley (*in press* b). Explaining novel events in process control through model generative reasoning. *International Journal of Expert Systems*.

Coombs, M. J., R. T. Hartley, and C. A. Fields (*submitted*). The Model Generative Reasoning approach to problem solving in unstructured environments. (AAAI-88).

Dean, T.L and D.V. McDermott (1987). Temporal data base management. *Artificial Intelligence*, 32, 1-55.

de Kleer, J. (1986). An assumption-based TMS. *Artificial Intelligence* 28, 127-162.

de Kleer, J. and B. C. Williams (1987) Diagnosing multiple faults. *Artificial Intelligence* 32, 97-130.

Doyle, J. (1979) A truth maintenance system. *Artificial Intelligence* 12, 231-272.

Fields, C.A. and E. Dietrich (1987a). Multi-domain problem solving: A test case for computational theories of intelligence. *Proceedings of the Second Rocky Mountain Conference on AI*, University of Colorado, 205-223.

Fields, C. and E. Dietrich (1987b). A stochastic computing architecture for multi-domain problem solving. *Proceedings of the Second International Symposium on Methodologies for Intelligent Systems*. Colloquim Volume, Oak Ridge National Laboratory (ORNL-6417), 227-238.

Fields, C. A., M. J. Coombs, and R. T. Hartley (*submitted*). The MGR architecture is Turing equivalent. (AAAI-88).

Grossberg, S. (1980). How does the brain build a cognitive code? *Psychological Review*, 87, 1-51.

Grossberg, S. (1987). Competitive learning: From interactive activation to adaptive resonance. *Cognitive Science*, 11, 23-63.

Hammond, K. J. (1986). CHEF: A model of case-based planning. *Proc. AAAI-86.* Los Altos, CA: Kaufmann. 267-271.

Hartley, R. T. (1986). The foundations of conceptual programming. *Proceedings of the First Rocky Mountain Conf. on AI*, University of Colorado, 3-15.

Hartley, R. T., M. J. Coombs, and E. Dietrich (1987). An algorithm for open-world reasoning using model generation. *Proceedings of the Second Rocky Mountain Conference on AI*, University of Colorado, 193-203.

Hartley, R. T. and M. J. Coombs (*in press*). Conceptual programming: Foundations of problem solving. In J. Sowa, N. Foo, and P. Rao (eds), *Conceptual Graphs: Theory and Applications*. Reading, MA: Addison-Wesley.

Hewitt, C. (1985). The challenge of open systems. *Byte* 10, 223-242.

Holland, J. H. (1986). Escaping brittleness: The possibilities of general-purpose machine learning algorithms applied to parallel rule-based systems. In: R. Michalski, J. Carbonell, and T. Mitchell (Eds.) *Machine Learning: An Artificial Intelligence Approach*, Vol. 2. Los Altos, CA: Kaufmann.

Kolodner, J. L., R. L. Simpson, and K. Sycara (1985). A process model of case-based reasoning in problem solving. *Proc. IJCAI-85.* Los Altos, CA; Kaufmann. 284-290.

Laird, J. E. and A. Newell (1983). A universal weak method: Summary of results. *Proceedings of IJCAI-83*, 771-773.

Laird, J. E., A. Newell, and P. S. Rosenbloom (1987). SOAR: An architecture for general intelligence. *Artificial Intelligence*, 33, 1-64.

McCarthy, J. (1980). Circumscription - A form of nonmonotonic reasoning. *Artificial Intelligence* 13, 27-39.

Pask, G. (1975). *Conversation, Cognition, and Learning*. Amsterdam: Elsevier.

Piaget, J. (1977). *The Development of Thought: Equilibration of Cognitive Structures*. New York: Viking.

Poggio, T., V. Torre, and C. Koch (1985). Computational vision and regularization theory. *Nature* 317, 314-319.

Reiter, R. (1980). A logic for default reasoning. *Artificial Intelligence* 13, 81-132.

Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence* 32, 57-95.

Reiter, R. and J. de Kleer (1987). Foundations of assumption-based truth maintenance systems: Preliminary report. *Proc. AAAI-87.* Los Altos, CA: Kaufmann. 183-188.

Rumelhart, D. E., G. E. Hinton, and J. L. McClelland (1986). A general framework for parallel distributed processing. In D. Rumelhart and J. McClelland (Eds.), *Parallel Distributed Processing*. Vol. 1. Cambridge, MA: MIT/Bradford, 45-76.

Shortliffe, E. H. and B. G. Buchanan (1984). A model of inexact reasoning in medicine. In: B. Buchanan and E. Shortliffe (Eds.) *Rule-Based Expert Systems.* Reading, MA: Addison-Wesley. 223-262.

Sowa, J. (1984). *Conceptual Structures*. Reading, MA: Addison-Wesley.

Thompson, J., R. Trout, and B. Landee-Thompson. (1986). *Artificial Intelligence Applications for Sensor Data Fusion*. Perceptronics and Knowledge Systems Concepts, Report A002, Rome Air Development Center.

Woods, D.D. (1986). Paradigms for intelligent decision support. In E. Hollnagel, G. Mancini and D.D. Woods (eds), *Intelligent Decision Support in Process Control*. Heidelberg: Springer-Verlag, 153-174.

Yager, R. (1987). Using approximate reasoning to represent default knowledge. *Artificial Intelligence* 31, 99-112.

Yourdan, E. and L. Constantine (1979). *Structured Design.* Englewood Cliffs, NJ: Prentice-Hall.