J. Expt. Theor. Artif. Intell. 4(1992)85–94

Incremental planning using conceptual graphs

DANIEL ESHNER[1], JAMES HENDLER[2] and DANA NAU[3]

[1]*Computer Science Department,* [2]*Computer Science Department, Systems Research Center, and Institute for Advanced Computer Studies, and* [3]*Computer Science Department, Systems Research Center, and Institute for Advanced Computer Studies, University of Maryland, USA*
email:[1]eshner@cs.umd.edu; [2]hendler@cs.umd.edu; [3]nau@cs.umd.edu

Abstract. When performing a planning or design task in many domains it is often difficult to specify in advance what the precise goals are. It is therefore useful to have a system in which the planning process is performed interactively, with the solution approaching the users' intent incrementally through iterations of the planning process. A planning system intended to function in this way must be able to take goal specifications interactively rather than all at once at the beginning of the planning process. The planning process then becomes one of satisfying new goals as they are given by the user, modifying as little as possible the results of previous planning work. Incremental planning is an approach to interactive planning problems that allows a system to create a plan incrementally, modifying a previous plan to satisfy new or more precise goal specifications. In this paper we present an incremental planning system called the general constraint system (GCS) that is based on the conceptual programming environment (CP) developed at New Mexico State University and we show an example of the use of the system for a simple civil engineering design problem.

Received 4 December 1991

1. Introduction

The conceptual programming (CP) environment (Hartley 1986, Pfeiffer 1986, Hartley and Coombs 1990) is a complete knowledge representation environment based on John Sowa's conceptual structures (Sowa 1984, 1990) that is designed for use in both static and dynamic problem solving systems. In this paper we present a dynamic, incremental planning system called the general constraint system (GCS) that uses the CP environment as its representational basis. Of particular importance to GCS is the feasibility-runtime domain overlays to the basic conceptual graph theory. This level gives GCS the ability to propagate changes through a graph that represents a design or an incremental plan.

In many design situations, although a designer may have an intuitive idea of what should be achieved, it may not be clear in advance how to specify formally what those intentions are. In domains ranging from engineering design to computer programming, one often starts out with a rather vague and abstract set of ideas, and refines them as one goes along. It may not always be entirely clear, even to the designer, what all of the goals are until the entire design is complete. Therefore, it is useful in many domains to have an interactive system for design, in which a solution to the design problem is found through an interactive process of creating solutions and modifying them in order to clarify and specify the designer's intent. The system must be able to create solutions both interactively and incrementally.

The process of creating a finished design can be thought of as an *incremental planning* problem, in which an existing plan is incrementally modified to satisfy new or changing goals. The designer specifies goals to the design system, and the design system constructs a design representation that satisfies those goals. To produce the next iteration of the design, the designer specifies new goals, and the design system modifies the existing design to satisfy those new goals. However, there is one significant difference between this notion of incremental planning and incremental planning in the traditional sense. Since the goals stated by the designer do not necessarily correspond to his ultimate intent, in order to produce the next iteration of the design it may be necessary to modify the existing design in ways that violate the goals that led to the existing design. The designer cares less about what particular goals and steps produced the current design than what the current design is, and how it differs from his intentions.

The existing design can be thought of as a 'final state' produced by the execution of some plan created to achieve the goals of that design. However, for the purpose of producing the next design iteration, what those goals and that plan are, are not important, because we do not know how or whether they corresponded to the designer's intent. What does matter is that in an attempt to make clearer his intent, the designer is now stating some new goals—and in an attempt to achieve his intent, we need to modify the design in a minimal manner to achieve these new goals.

Most approaches to incremental design are based on derivational analogy (Mostow 1989, Mostow and Barley 1987, Steinberg 1987, Wile 1983). This process is one of modifying the *process* rather than the *product* of design by *replaying* a design plan used to solve a previous problem and modifying the plan when necessary (Carbonell 1986, Mostow 1985). It is important to note that these approaches concentrate on the plan that gives rise to a design rather than the design itself. Most classical planning systems (see Tate *et al.* 1990) and incremental planning systems (Elkan 1990, Hayes-Roth *et al.* 1979, Kambhampati and Hendler 1992 also concentrate on the plans rather than the world states that result from these plans.

In contrast to the classical planning paradigm in which the focus is on the sequence of operators that make up a plan, in GCS the focus is instead on the *world state* that represents the current design. We call this world state a *model*. The approach to incremental planning in GCS is one of modifying a model in a minimal way to satisfy a new goal specification. Since the representation system in GCS is based on conceptual graphs, the modifications to the model are given in terms of graph operations. Specifically, the planning operators in GCS are specifications of the graphical changes to a model graph. We can thus use syntactic measures in terms of graph structures to measure minimality.

As mentioned in Mostow (1989), concentrating on the designs themselves rather than the design plans is problematic in that a single change in the problem may require a multitute of patches and mess up the structure of the design. In GCS the propagation of change in a model is controlled by a system of *constraints* associated with a model. These constraints are part of the feasibility-runtime level of the CP representational system. The use of CP with constraints allows GCS to represent the changes to models throughout an interactive planning process. Using a representation based on conceptual graphs also allows us to use well defined methods for joining graphs and for determining relevance of operators (Eshner and Pfeiffer 1990).

2. Incremental planning

Following Kambhampati and Hendler (1992), we define the *plan modification problem* as follows: given a planning problem P^n (specified by a partial description of an initial state and a goal state), and an existing plan R^0, along with the planning problem P^0 it was produced to solve, the objective of the system is to produce a new plan R^n or P^n by minimally modifying R^0. Intuitively, the aim is to modify an existing plan for a similar problem in a minimal way so that it solves the current problem. We define *incremental planning* as a subcase of the plan modification problem in which the given existing plan R^0 was produced to solve the current planning problem. The objective of the system is thus to modify an existing plan for the current planning problem such that it satisfies a new set of goals. This definition of incremental planning differs slightly from that used in some previous planning systems (Elkan 1990, Hayes-Roth *et al.* 1979) where incremental planning is used to describe a process of plan refinement.

An incremental planning system must have mechanisms for controlling modifications to plans and for approximating the extent of modifications. It is also important to have a notion of consistency or correctness in a plan as it is modified. In addition to logical inconsistencies there might also be domain-specific inconsistencies that result from a single modification. An incremental planning system must be able to catch these inconsistencies and correct them or disallow the modification.

Kambhampati and Hendler (1990) developed a plan modification system called PRIAR that addresses these issues for plans generated by a hierarchical nonlinear planner. In PRIAR, the first step in the plan modification process is to retrieve an existing plan and to map the objects in that plan onto objects in the current problem. Incremental planning is a subcase of this problem in which the system does not have to choose a retrieval candidate, since the plan to be modified is that which is already being used for the current problem. Further, the mapping problem is minimized since changes are generally performed on already defined objects.

The approach taken in PRIAR relies heavily on the assumption that the given plans were constructed by a hierarchical nonlinear planner (for a good discussion of classical planning that includes hierarchical and nonlinear planning, see Tate *et al.* 1990). Briefly, the hierarchical development of a plan in a hierarchical planner is captured by its *hierarchical task network* (HTN). Each task in the HTN has effects and conditions. A *validation* is a link between one task's effect and another's condition. This link corresponds to a protection interval that is maintained during planning. PRIAR keeps a finite set of these validations in what is called a *validation structure*. This structure can be viewed as an internal dependency representation of hierarchically generated plans. Plan modification in PRIAR is seen as a process of repairing the inconsistencies in the validation structure of an existing plan when that plan is interpreted in a new problem situation.

3. Incremental planning using conceptual graphs

As an alternative to using validation structures, we present an approach to incremental planning that centers on using conceptual graphs. The approach relies heavily on addition of quantitative constraints to the basic theory of conceptual graphs, and so our primary focus here is on these constraints. We have incorporated the notion of constraint overlays from CP into the representation environment used in GCS. This representation environment, called PRE (plan representation environment), is a general purpose environment that allows a planning system to use the power and flexibility of conceptual graphs and constraints. In this section we give a brief overview of PRE and show how it is used in GCS. For a complete discussion of the additions to the basic conceptual graph theory found in CP, see Pfeiffer and Hartley, pp 167–182; Eshner and Hartley (1988b), Eshner (1989), Pfeiffer and Hartley (1989). For an argument for the use of conceptual graphs as the representational framework for planning see Eshner and Hartley (1988a).

3.1. *The plan representation environment*

The plan representation environment (PRE) (Eshner 1989) is a representational environment based on the CP environment. There are two additions to the CP environment: the first is an extension to the quantitative constraint overlays that allows for different types of constraints to be determined by the syntactic definition in a graph, and the second is an ability to store and manipulate multiple plans or designs.

There are two levels in the PRE system. The base level is the declarative level at which we represent facts and schemata as single graphs. Facts correspond in traditional planners to predicates that make up world states. Schemata correspond to definitions of objects in the world and actions that operate on them. Each action is described by a schema that shows the relationships between itself and objects in the world. Figure 1 shows the base schema for a beam in a building.

The second level in PRE is the constraint level in which functions are attached to schemas much like procedural attachment in a frame system. Each schema has associated with it a set (possibly empty) of constraint overlays. Functions in the constraint overlays are either procedures or constraints. Procedures are those

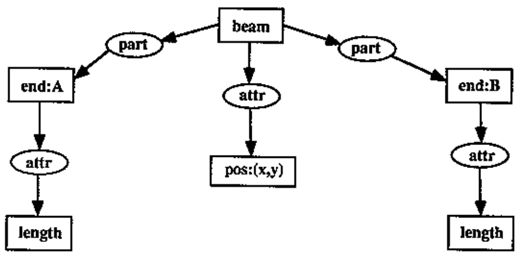

Figure 1. The base scheme for building beam.

functions in which there is clearly defined inputs to the function and a single output. Constraints are those functions in which all parameters are both input and output and the single parameter that is not provided is computed and returned. Procedures are used to fill in values of uninstantiated concepts when adding new bits of graph to a model. Constraints are used to show the relationships among objects in a system. For example, in a building design a joint connects a beam and a column. There is a constraint on those objects such that the location of the joint is the same as the location of one of the ends of the beam and column. All of the constraints in a design specification are represented by constraints.

Operators in PRE describe the changes to a model in terms of what bits of graph are taken out and what bits are added, much like the add and delete lists in classical planning systems. The most simple operators only change the value of a concept node, the most complex take out a subgraph in the model and replace it with a new graph. Figure 2 shows an operator that moves a building joint that connects beams and columns from one position to another.

Along with each operator are various constraint overlays. Constraints in the operators are always part of a graph that is being added to a model and are not evaluated but rather joined into the model along with the rest of the graph to be added. Procedures are used to instantiate new concept nodes being added to the graph and are executed as the operator is being applied to the model. As an example, Figure 3 shows a more complicated operator with both a procedure and a constraint that will be added to a model. The operator lengthens an existing beam. The procedure is one that fills in the length attribute of a beam given the position of the two ends of the beam. The constraint is a constraint on the maximum length of a beam.

Since all graphs can have associated constraint overlays, these overlays provide a method to attach constraints to objects and collections of objects that make up world states. As a reasoning system builds models of the world, a constraint network is automatically generated that shows how the different objects in the world are constrained to interact. Also, as the model changes, operators that change the model will add and delete constraints to and from this constraint network. Perhaps most importantly, as any concept in the model that is also part of the constraint network is changed, that change is reflected in the constraint network and the change is propagated through the constraint network to other parts of the model. This is the mechanism by which a single change to the model that is introduced by an operator can be propagated.

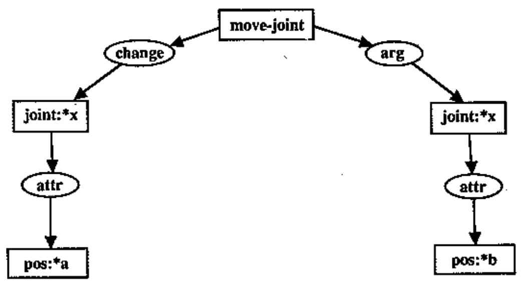

Figure 2. An operator for moving a building joint.

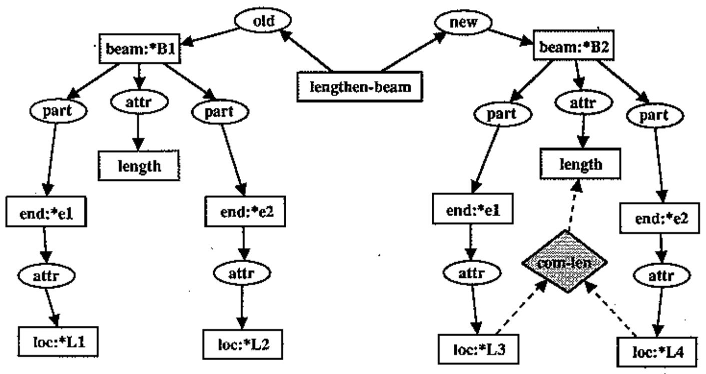

Figure 3. An operator for lengthening a building beam.

3.2. *Modifications and relevance*

Our approach to incremental planning is one of minimizing the amount of change in a world model that occurs through propagation in the constraint network as operators are applied to the model. These operators are precisely those actions that are considered relevant to solving the current goal that has been given to the system. The key to this approach is finding the set of relevant operators and choosing the operators that result in the minimum amount of change to the model.

There are two ways in which a search for relevant operators is triggered. The first is if there is a goal that is not satisfied, i.e. there is a portion of the model that does not match a desired goal, then the system must choose from among all possible operators a subset of operators that are relevant to the current goal(s) and model. This is done by isolating the portion of the graph that does not match and performing a graph *cover* of that graph portion by the set of operators. This operation will return a set of relevant operators (for a more complete discussion of the graph cover operation see Eshner 1990). The second way in which a search is triggered is when a constraint is violated. Given a constraint that has been violated, all the concepts associated with the constraint that have not already been changed are used to isolate a portion of the model on which to *focus*. This focus is a portion of the model that is to be covered by a set of operators as discussed previously.

Given a set of relevant operators that are applicable to a model, the goal of the system is to choose the subset of those operators that will minimize the number of changes to the model while still satisfying the goals. There are potentially many alternative combinations of operators that will be able to achieve this criteria. This implies that the system must search through a space of combinations of operators to find a subset that is minimal in the number of changes to the model while still satisfying the goals of the system. The heuristic used for the search is the *change* in the number of constraints in the model due to the application of an operator. This is because the change in the number of constraints is a good estimate of the amount of *potential* change in a model that can result from the application of an operator to the model.

4. An example

The domain in which GCS is currently being applied is taken from the field of civil engineering structural design. More specifically, we assume a designer is interactively specifying a building design while the system propagates all changes through the model and attempts to stabilize the model given all current constraints in the constraint network.

To illustrate the incremental design process used in GCS, consider the simple one-story one-bay steel structure shown in Figure 4. It has two identical columns (C1 and C2) that are 20 feet high. Initially, the single beam at FLOOR 1 is 20 feet long. Joint elements (J3 and J4) are used to connect/transfer forces among local structural (beam, column and truss) elements. Joints J1 and J2 transmit column actions to the structural foundation.

The model graph for this structure is shown below in Figure 5. Along with the graph is an example constraint, of which there are total of nine in the graph with only one shown here. In the example we have two columns, a beam and four joints. Each beam and column has two ends A and B that are attached to the joints. Each joint has a constraint on its position in relation to the beams and

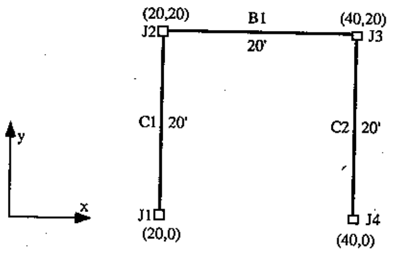

Figure 4. A simple building design.

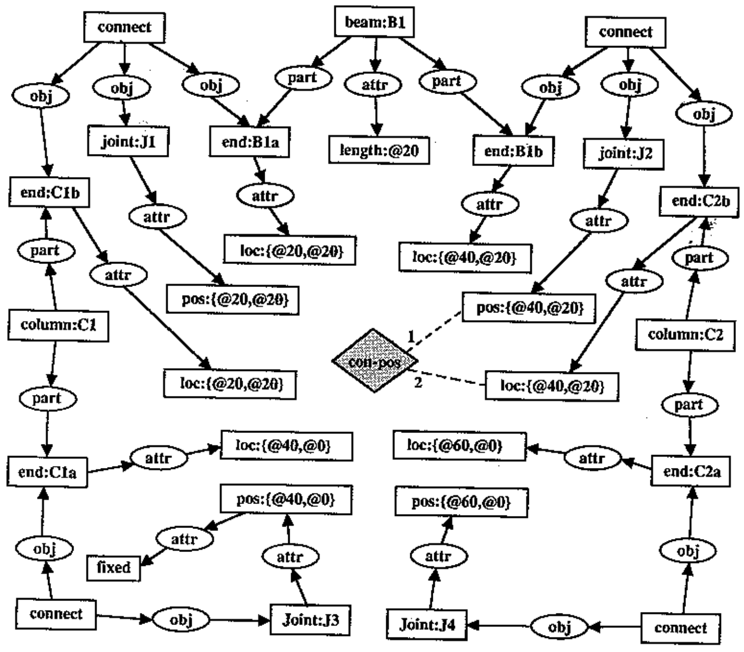

Figure 5. The model for the design in Figure 4.

columns it is connecting (the positions must be the same). There are also tags on some portions of the model that indicate the portion is not to be altered. In this example there is a tag on **joint:J3** that indicates it is not to be moved. All constraints are currently satisfied. The system thus has a model (plan) for a building that is stable. We now introduce a new goal to the system that will start the incremental planning process. The new goal is to have **joint:J4** at a new position of (60,0), moved 20 feet to the right.

The first step of the system is to match the goal to the model. The system searches the model for the subgraph that corresponds to the goal and isolates the concepts in the goal that are either not present in the model or are different. The concepts that are either missing or different become the *focus* for changing the model. The next step of the system is to search for operators that will potentially eliminate the focus by adding missing concepts or changing different concepts. Depending on the amount of information in the focus, there may be a very large number of potentially useful operators. In order to reduce the number of operators that can potentially eliminate the focus, it is important to include more information in the search than the focus alone. To this end the system searches backward through the directed edges of the graph as far as possible in order to place the focus in a *context*. Thus the context is simply the initial focus with surrounding concepts included.

In this example, the focus is originally the set {**position**} (the concept **position** alone) with the context being the set {**joint:J4**, **connect**}. Intuitively, the backward search in the example takes the attribute (**position**), finds the concept it modifies (**joint:J4**) and continues searching backwards in the model to find the concept (**connect**) that is related to the second concept (**joint:J4**). At this point the search stops since there are no more links to follow. The system then searches for operators that can eliminate the focus. In this case, the most relevant operator is the **move-joint** operator shown in Figure 2. This operator simply moves the position of a joint, in this case **joint:J4**. By applying this operator to the model we get a new position for **joint:J4**. The constraint for the joint is now violated since the position of **end:B** of **column:C2** (40,0) is not equal to the position of the joint (60,0).

When a constraint is violated, all the concepts associated with the constraint that have not already been changed are used to create a focus. In this case, the focus is the set {**location**} with the context being the set {**end:C2a**, **column:C2**, **connect**}. The operator search returns **move-column** as the only relevant operator to this context. The application of this operator results in new positions for both ends of **column:C2** (**end:C2a**=(0,60) and **end:C2b**=(20,60)). A connected constraint for **joint:J2** is now violated since the position of **joint:j2** (20,40) is not equal to the position of **end:C2a** of **column:C2** (20,60). The same process is repeated with the focus being {**pos**, **joint:J2**, **connect**} which is used to find the **move-joint** operator as the most relevant. This operator is applied resulting in a new position (20,60) for **joint:J2**. This violates a different connected constraint for **joint:J2** since the beam position (20,40) is now different from the position of the joint (60,20).

At this point the focus is {**loc**, **end:B1b**, **beam:B1**} which is used to find relevant operators. The operator search returns a set of operators {**move-beam**, **lengthen-beam**} that are all equally relevant. The search heuristic currently used is one that minimizes the change in the amount of graph structure altered. The heuristic is used to create a total ordering (if two operators are equivalent with respect to the heuristic the ordering is arbitrary) of the set of operators returned as relevant. In this case **move-beam** is a simple operator that changes the graph only very slightly (changing the value of a concept) while the lengthen-beam operator modifies the graph structure slightly and so move-beam is chosen first. This operator is used to create a branch in the search tree. The model that results from the application of **move-beam** has a new position for both ends of the beam. This violates the connected constraint for **joint:J3**, triggering an operator search that returns **move-joint**. At this point the tag on **joint:J3** that indicates it is not to be moved causes the search to be assigned a prohibitively high cost. This path in the search is therefore suspended.

The system now takes the second possible operator, **lengthen-beam**, to satisfy the connected constraint for **joint:J2**. This begins another path in the search tree as the operator is applied to the model in which the constraint is violated. After the application of the operator, the beam has a length of 40 with **end:B1a** having its original position (20,20) and **end:B1b** having the same location as **joint:J2** (20, 60). This violates a constraint on each beam that states that a beams length must be less than or equal to 25 feet. This initiates another operator search with focus {**length, beam:B1**} that returns **split-beam.**

Split-beam operator is a more complex operator that changes the structure of the model and adds more constraints, thus it has a relatively high cost assigned by the search heuristic. This operator splits the beam near the middle and adds a brace. After the application of this operator, all constraints are satisfied and the model is once again stable. The structure that is created as a solution is shown in Figure 6.

It is important to note that at every step in the search process the most simple and direct solution is chosen, although this stage is highly dependent upon the search heuristic. We are currently seeking the solution that modifies the model as little as possible, as stated in the problem description.

5. Conclusion

The ability to interactively specify goals enables users to incrementally specify their intent in a design. A planning system that can modify solutions incrementally to match the users' changing intentions allows the system to be used in domains in which it is difficult to specify the goals of the user in advance. The system presented in this paper takes goals from a user interactively and changes the current model to satisfy these goals. The changes to the model are controlled through propagation in a constraint network, thus keeping the model consistent. The system currently uses a notion of minimal change to insure that the current change affects as few of the users previous intentions as possible. In this way the

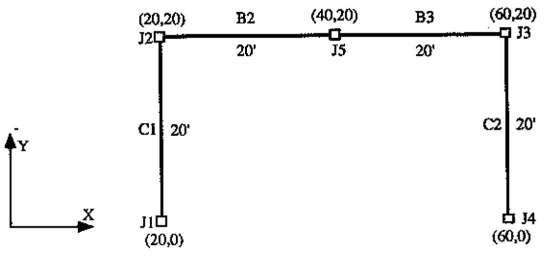

Figure 6. The updated building design.

system allows a designer to incrementally modify a design such that it achieves their intent. The use of a representation system based on a combination of the conceptual programming environment developed at New Mexico State University and the plan representation environment developed at both NMSU and the University of Maryland allows GCS to specify operations and heuristics in graph theoretic terms. The system therefore has a precise, formal base on which to build the operations that enable incremental planning.

Acknowledgements
This work was supported in part by NSF Grant NSFD CDR-88003012 to the University of Maryland Systems Research Center, NSF grant IRI-8907890 for Dr. Nau and Dr. Hendler, and ONR grant N00014-88-K-0560 for Dr. Hendler.

References

Carbonell, J. G. (1986) Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In: Michalski, R.S., Carbonell, J. G. and Mitchell, T. M. (eds) *Machine Learning: An Artificial Intelligence Approach 2* (Los Altos:Morgan Kaufmann) 371–392.
Chapman, D. (1987) Planning for conjunctive goals, *Artificial Intelligence*, **32**, 333–337.
Elkan, C. (1990) Incremental, approximate planning, *Proceedings of the Eighth AAAI*, Boston, MA, 145–150.
Eshner, D. P. (1990) A graph theoretic basis for problem solving. *Proceedings of the Fifth Rocky Mountain Conference on Artificial Intelligence*, Las Cruces, NM, 169–174.
Eshner, D. P. (1989) A general representation environment for artificial intelligence planning unpublished masters thesis, New Mexico State University.
Eshner, D. P. and Hartley, R. T. (1988a) A unified approach to plan representation, *Proceedings of the Third Rocky Mountain Conference on Artificial Intelligence*, Denver CO, 34–45.
Eshner, D. P. and Hartley, R. T. (1988b) Conceptual programming with constraints, *Proceedings of the Third Annual Workshop on Conceptual Graphs*, AAAI-88, St. Paul, MN, 3.1.2. 1–6.
Hartley, R. T. (1986) The foundations of conceptual programming, *First Rocky Mountain Conference on AI*, Boulder, CO, 3–15.
Hartley, R. T. (1991) A uniform representation for time and space and their mutual constraints. In Lehmann, F. (ed.) *Special Edition on Semantic Networks in Artificial Intelligence, Computers and Mathematics with Applications* (in press).
Hartley, R. T. and Coombs, M. J. (1989) Reasoning with graph operations. In: Sowa, J. (ed.) *Formal Aspects of Semantic Networks* (New York: Addison-Wesley) (in press).
Hartley, R. T. and Coombs, M. J. (1990) Conceptual programming: foundations of problem solving. In Sowa, J., Foo, N. and Rao, P. (eds) *Conceptual Graphs for Knowledge Systems* (New York: Addison-Wesley) (in press).
Hayes-Roth, B., Hayes-Roth, F., Rosenschein, S. and Cammarata, S. (1979) Modeling planning as an incremental, opportunistic process, *Procedings IJCAI-79*. 375–383.
Kambhampati, S. (1990) A theory of plan modification, *Proceedings of the Eighth AAAI*, Boston, MA, 176–182.
Kambhampati, S. and Hendler, J. (1992) A validation structure based theory of plan modification and reuse, *Artifical Intelligence* (in press).
Mostow, J. (1985) Toward better models of the design process, *AI Magazine*, **6**(1), 44–57.
Mostow, J. (1989) Design by derivational analogy, *Artificial Intelligence*, **40**: 119–184.
Mostow, J. and Barley, M. (1987) Automated use of design plans. In Eder, W. E. (ed.) *Proceedings 1987 International Conference on Engineering Design (ICED87)*, Boston, MA 632–647.
Pfeiffer, H. D. (1986) Graph definition system, *Proceedings of the Second New Mexico Computer Science Conference*, Las Cruces, NM, 97–103.
Pfeiffer, H. D. and Hartley, R. T. (1989) Semantic additions to conceptual programming, *Proceedings Fourth Annual Conceptual Graphs Workshop*, Detroit, MI, 6.07-1–6.07-8.
Pfeiffer, H. D. and Hartley, R. T. (1990) Set representation and processing: additions to conceptual programming, *Proceedings 1990 Conceptual Graphs Workshop*, Boston, MA, 176–182.
Sowa, J. S. (1984) *Conceptual Structures* (Reading, MA: Addison Wesley).
Steinberg, L. (1987) Design as refinement plus constraint propagation: The VEXED experience, *Proceedings AAAI-87*, Seattle, WA 830–835.
Tate, A., Hendler, J. and Drummond, M. (1990) A review of AI planning techniques. In Allen, J., Hendler, J. and Tate, A. (eds) *Readings in Planning* (San Mateo: Morgan Kaufman) 26–49.
Wile, D. S. (1983) Program developments: Formal explanations of implementations, *Commun ACM* **26**(11), 902–911.