Argo: A System for Design by Analogy

Michael N. Huhns and Ramón D. Acosta

Microelectronics and Computer Technology Corporation
Artificial Intelligence Laboratory
3500 West Balcones Center Drive
Austin, TX 78759

Abstract

The static and predetermined capabilities of many knowledge-based design systems prevent them from acquiring design experience for future use. To overcome this limitation, techniques for reasoning and learning by analogy that can aid the design process have been developed. These techniques, along with a nonmonotonic reasoning capability, have been incorporated into Argo, a tool for building knowledge-based systems that improve with use. Argo acquires problem-solving experience in the form of problem-solving plans represented by rule-dependency graphs. From increasingly abstract versions of these graphs, Argo calculates sets of macrorules. These macrorules are organized according to an abstraction relation for plans into a partial order, from which the system can efficiently retrieve the most specific plan applicable for solving a new problem. Knowledge-based applications written in Argo can use these plan abstractions to solve problems that are not necessarily identical, but just analogous to those solved previously. Experiments with an application for designing VLSI digital circuits demonstrate how design tools can improve their capabilities as they are used.

1 Introduction and Background

A number of knowledge-based systems for design have been developed recently [2,12,18,19]. These systems are particularly suited to situations in which heuristic expert knowledge must be employed because algorithmic techniques are unavailable or prohibitively expensive. Unfortunately, the knowledge embodied in many of these systems is static: it fails to capture the iterative aspects of the design process that involve solving new problems by building upon the experience of previous design efforts. Given the same problem ten times, these systems will solve it the same way each time, taking as long for the tenth as for the first.

The work reported here is based on the contention that a truly intelligent design system should improve as it is used, *i.e.*, it should have a means for remembering the relevant parts of previous design efforts and be able to employ this accumulated experience in solving future design problems. For a design tool, the remembered experience should consist of 1) design results, 2) design plans, and 3) preferences among these results and plans. These constitute different aspects of previous design efforts that the design tool can use as training examples.

1.1 Learning from Experience

Existing approaches to learning from experience attempt to generalize training examples in order to obtain more widely applicable results. The STRIPS [6] problem-solving system incorporates a technique for generalizing plans and their preconditions based on the formation of macro-operators (MACROPs). A better procedure for generalization, developed in the context of learning from examples, uses a proof-based explanation mechanism [5,13,16,17], often termed explanation-based generalization (EBG). It is an improvement over MACROPs in that it does not require any heuristics to compensate for possible overgeneralizations. For design problems, however, EBG-like generalizations are limited in that they arbitrarily give equal weight to all portions of the examples, without regard to whether each portion is relevant or important to solving future problems. More abstract generalizations can be obtained by taking this factor into account. Abstract planning, *i.e.*, choosing a partial sequence of operators to reach a goal, is accomplished in ABSTRIPS [22] by ignoring operator preconditions considered to be details. Only if a plan succeeds at an abstract level is it expanded by the addition of subplans to handle the details at a subsequent level.

Another technique for reusing past design experience is to replay a previously recorded plan, or design history [20,23]. This approach is interesting in its flexibility with respect to replaying portions of a stored plan to solve, or at least partially solve, a new problem. Unfortunately, the correspondence between a stored plan and subproblems of a design is difficult to establish automatically.

1.2 Analogical Reasoning and Learning

The transfer of experience from previous problem-solving efforts to new problems has also been accomplished via analogical reasoning methods. Analogical reasoning is a mapping from a base domain to a target domain that allows the sharing of features between these domains. The two problems that arise in this mapping are 1) *analogy recognition*: finding the most similar past experience, and 2) *analogical transformation*: adapting this experience to the new problem situation.

Aside from user guidance, several techniques have been suggested for automatically recognizing the most similar past experience.

- Develop an analytical similarity measure [3]
- Find a past experience whose first stage is identical to the current problem situation [4]
- Find a past experience that has the same causal connections among its components as does the current problem [7,25]
- Find a past experience that has the same purpose as does the current problem situation [11].

The disadvantages of these techniques are described in [10].

The second problem, analogical transformation, has been attempted previously by employing heuristically-guided, incremental perturbations according to primitive transformation steps [3]. These steps are generally problem and domain specific and are not amenable to automation. If used properly, however, differences between the old and new situations can guide the analogical transformation. Other approaches to analogical transformation

include heuristic-based analogical inference [9] and user intervention [20].

2 Argo

The primary objective of the work reported here has been to develop a robust and domain-independent system, Argo, for applying analogical reasoning to solving search-intensive problems, such as those in the domain of design [1,10]. This section describes the functional characteristics of Argo, as well as an application to VLSI digital circuit synthesis. Descriptions of the mechanisms in Argo that allow it to reason and learn efficiently are presented in Section 3. Section 4 lists and discusses some experimental results obtained using the Argo-V application. Conclusions are presented in Section 5.

2.1 Knowledge Representation and Inference

Argo is a generic development environment, derived from the Proteus expert system tool [21]. Knowledge is represented in Argo using a combination of predicate logic and frames. Data consist of ground assertions, general assertions, forward rules, backward rules, and slot values in frames. Each datum is included in a justification-based truth-maintenance system (JTMS). Frames are organized into an inheritance lattice, enabling multiple inheritance for slot values. The slots may be single-valued or multiple-valued. Rules primarily deal with relations, which may be either predicates or slots. In addition, rules allow Lisp functions in their antecedents and consequents.

The inference mechanisms available to Argo are forward chaining, backward chaining, inheritance through the frame system, truth maintenance, and contradiction resolution. Forward chaining is typically used as the strategy for design: the system applies forward rules deductively to hierarchically transform and decompose specifications or partial designs. In order for a forward rule to be eligible for firing as an action, its antecedents must either explicitly exist as assertions or slot values in the database, or implicitly be provable through backward chaining.

2.2 Argo Control Strategy for Design

Argo executes the problem-solving strategy depicted in Figure 1. This algorithm has two major phases: a problem-solving design phase followed by a learning phase. The basic control strategy for the design phase conforms to that of a standard production-system interpreter (*cf.* OPS5). It is modified for analogical reasoning by requiring that only the most specific rules from a partial order of forward rules (based on the *abstraction* relation defined in Section 3.4) be matched and considered for execution. A new cycle can be triggered interactively by a user or automatically by any JTMS adjustments to the database.

During each cycle, forward rules are considered for execution by attempting to prove their antecedents using all available data (assertions and backward rules). The least abstract (see Sections 3.4 and 3.5) valid rule instances are then placed in a conflict set. At this point, a user may interact with the system by ordering rule instances in the conflict set, firing instances, asserting new data, or initiating dependency-directed backtracking.

The learning phase is outside of the control loop of Argo's production-system interpreter, and as such, can be executed as a background task of the problem-solving system. This improves the system's problem-solving efficiency; it does not have to pause to learn in the middle of a design session. It also prevents the learning of results that might be subsequently invalidated due to nonmonotonic reasoning triggered by dependency-directed backtracking. Further, the plans that are learned do not incorporate failed lines of reasoning.

```
Read a database of domain knowledge, consisting of
   rules, frames, and assertions;
Partially order forward rules by "abstraction";
Read problem specification into memory;
Compute initial conflict set, CS;
Design: Loop until CS empty or Halt asserted
        Resolve Conflicts - Select a forward rule
              instance, Ri, from CS;
        Act - Perform the consequents of Ri and
              update the JTMS justification network;
        Match - Find CS of most specific applicable
              rule instances that have not previously
              fired on the same data;
Evaluate design - If unacceptable,
        Assert contradiction;
        Do dependency-directed backtracking;
        Go design loop;
Construct rule-dependency graph (RDG);
Learn:  Loop while nodes(RDG) > 1
        Compute macrorules from subgraphs of RDG;
        Insert macrorules into partial order;
        Abstract RDG (by deleting leaf rules);
Store updated database.
```

Figure 1: Argo Control Strategy

2.3 Application to VLSI Circuit Design

Argo is customized for a particular application by building a knowledge base of rules, assertions, and frames. The primary application that has been used for testing Argo is a system for VLSI digital circuit design. Design problems have been a motivation and justification for the approach to analogical reasoning described above because of the large search space by which they are typically characterized—a space consisting of both incomplete and complete design solutions.

The Argo VLSI design application, Argo-V, refines circuit specifications to synthesize circuits in terms of elementary digital components. A design problem specification is a set of assertions in first-order logic describing the *behavior* of a digital logic circuit. This set of assertions is organized into a lattice of frames. A solution to a design problem is also a set of assertions in first-order logic that describes the *structure* of the digital logic circuit and is less "abstract" than the set describing the specification.

The assertions and frames are based on VHDL (VHSIC Hardware Description Language) [24]. Since VHDL is designed to deal with abstraction, its declarative facilities provide a natural medium for describing design hierarchies. An *entity* in VHDL corresponds to a component that is described by an interface body and one or more architectural bodies. The *interface body* is used to define externally visible ports and parameters of an entity. The *architectural bodies* are used for describing entities in terms of behavior and/or structure. The two primary types of statements used in architectural bodies are 1) *signal assignment statements* (behavioral), which assign waveforms to signals, and 2) *component instantiation statements* (structural), which instantiate substructure components.

The design knowledge base in Argo-V is structured as follows:

Frames: VHDL modules(*e.g.*, interface bodies).

Frame Instances: primitive library components (*e.g.*, logic gates, transistors, and inverter loop memory cells).

Assertions: component slot values and general knowledge.

Forward Rules: design rules.

Backward Rules: rules for parsing signal assignments and computing ports.

A design problem's specification is entered into the system by instantiating frames for its top level VHDL modules and asserting slot values for its internal features, including signal assignment statements and signal declarations.

The design rules in Argo-V either transform, instantiate, or decompose. A transformation rule is used to convert one or more signal assignment statements into other signal assignment statements having a simpler or more convenient form. An instantiation rule converts one or more signal assignment statements into statements specifying library components. A decomposition rule removes one or more signal assignment statements from an architectural body and associates them with newly-built entities that are instantiated from the architectural body. Decomposition rules allow groups of logically-related signal assignments to be treated as independent subproblems.

Figure 2 contains an example of a rule for instantiating a pass-transistor component. In this rule, a signal assignment statement is matched with an antecedent of the form

```
(signal-assignment ?body
                   (?lhs (?signal ?delay ?condition)
                         (?signal2 ?delay2)))
```

where `?body` is the architectural body of an entity and `?lhs` is assigned the value of `?signal` after a delay of `?delay` if `?condition` is satisfied, or `?signal2` after `?delay2` if `?condition` is not satisfied.

```
;;;; Rule for Instantiating a Pass Transistor
;;;;
((architecture ?e:entity ?b:architectural-body)
 (unless (type ?e hardware-module))
 (signal-assignment
        ?b (?lhs (?s:signal ?delay1 ?c:signal)
                 (hi-z ?delay2)))
 -45->
 (erase (signal-assignment
                ?b (?lhs (?s ?delay1 ?c)
                         (hi-z ?delay2))))
 (component ?b (PASS-TRANSISTOR (?s ?c ?lhs))))
```

Figure 2: Example of a component instantiation rule

3 Analogy Mechanisms in Argo

This section presents the representations and techniques used in Argo for analogical reasoning. These enable Argo to improve its problem-solving performance with repeated use.

3.1 Rule-Dependency Graphs

To implement the problem-solving strategy of Section 2.2, Argo must be capable of formulating, remembering, and executing problem-solving plans. A plan in Argo is a directed acyclic graph having nodes corresponding to forward rules and edges indicating dependencies between the rules. Thus, the terms plan and rule dependency graph (RDG) are used interchangeably throughout this paper. Because a plan is implicitly represented by the justifications maintained by the JTMS, no overhead is incurred by Argo's inference engine for plan maintenance.

One example from Argo-V involves the design of a content-addressable-memory (CAM) cell, similar to the one used in [18]. The specification for the CAM-cell design problem can be seen in Figure 3. Note that the entity *CAM-Cell* has one interface, *CAM-Interface*, and one architectural body, *CAM-Architecture*. *CAM-Architecture* is a behavioral problem specification because it only makes use of signal assignment statements.

CAM-Cell is an instance of ENTITY
- Type: *ABSTRACT-MODULE*
- Interface: *CAM-INTERFACE*
- Architecture: *CAM-ARCHITECTURE*

CAM-Interface is an instance of INTERFACE-BODY
- Input-Port: *COMPARE, PHI1, PHI2, LOAD, ENABLE, DATA-IN*
- Output-Port: *MATCH*

CAM-Architecture is an instance of ARCHITECTURAL-BODY
- Signal-Declaration: *PHI1-LOAD, STATE*
- Component: no known values
- Signal-Assignment:

(*MATCH* (PASSED-LOW 33NS
(BIT-AND (BIT-EQUAL *ENABLE* HIGH)
(BIT-NOTEQUAL *STATE COMPARE*)))
(HI-Z 0NS)),
(*STATE* (*DATA-IN* 50NS (BIT-EQUAL *PHI1-LOAD* HIGH))
(*STATE* 0NS)),
(*PHI1-LOAD* ((BIT-AND *PHI1 LOAD*) 15NS))

Figure 3: Behavioral specification for the CAM-cell problem

Argo-V solves circuit design problems by deductively applying rules that hierarchically refine behavioral specifications. In the process, Argo-V constructs a hierarchical design tree representing a partial solution. Each node of this design tree is an entity, or component, that is described in terms of its interface and one architectural body. A design is completed when all the statements in the design tree are instantiations of library components. The hierarchical design tree for the solution of the CAM-cell problem appears in Figure 4.

Once a design, or partial design, has been completed, the learning phase of Argo can be invoked. Its first task is to build an explicit plan representation according to the justifications for fired actions. The design plan for solving the CAM-cell problem, consisting of 19 forward rule instances, is shown in Figure 5.

3.2 Abstract Plans

The analogical reasoning model used by Argo comprises solving new problems by making use of plans for previous design experiences at appropriate levels of abstraction. In this vein, the primary function of the system's learning phase is to compute and store abstractions for the plan of a solved problem. This task is accomplished by computing macrorules for increasingly abstract versions of the plan and inserting these rules into a partial order.

A number of domain-dependent and domain-independent techniques for automatically generating plan abstractions are possible. These include deleting rules from the plan, replacing a rule by a more general rule that refers to fewer details of a problem (such as is done in ABSTRIPS [22]), and computing and generalizing a macrorule for the plan without reference to the components of the original plan.

Currently, Argo abstracts a plan by deleting all of its leaf rules, which are those having no outgoing dependency edges. For many design domains, the leaf rules trimmed from a plan tend to be those that deal with design details at the plan's level of abstraction. Increasingly abstract versions of a plan are obtained by iteratively trimming it (see Figure 1). A sequence of abstractions for the CAM-cell example, generated by this technique, appears in Figure 5. Note that all of the rules deleted by trimming one level from the original plan are rules that handle the details of instantiating library components.

A possible drawback of Argo's automatic abstraction scheme is that deleting all leaf rules might eliminate useful abstract plans in which only some of the leaf rules should be deleted. Except for small plans, however, it is not practicable to generate macrorules

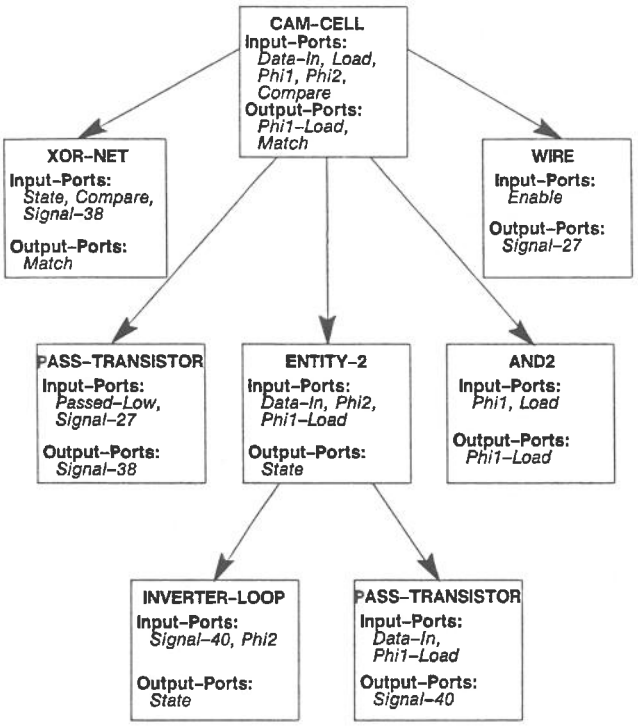
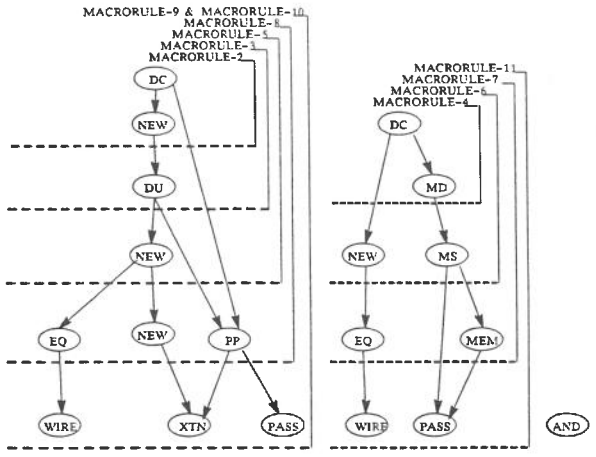


Figure 4: Hierarchical description of the final design for the CAM-cell



DC:	Decompose conditional-signal-assignment statements
NEW:	Construct new signal-assignment statements from decomposed statements
DU:	Decompose unconditional-signal-assignment statements
EQ:	Transform a statement containing an equality into a simpler statement
WIRE:	Instantiate a connection between two components
PP:	Transform a set of signal assignments to represent a cascade of pass-transistor nets
XTN:	Instantiate an exclusive-OR pass-transistor network
PASS:	Instantiate a pass transistor
MD:	Decompose an entity into ones containing memoried and combinational logic
MS:	Complete the specification of an entity containing memoried statements
MEM:	Instantiate an inverter loop for a one-bit memory
AND:	Instantiate an AND gate

Figure 5: Design plan (rule-dependency graph) for the CAM-cell

for all possible subgraphs of the RDG, even though these would be valid and potentially useful. Instead, Argo begins with a plan's precomputed abstraction, followed by instantiations of some of its trimmed rules, to obtain an appropriate plan for solving a new problem.

3.3 Macrorules

During the learning phase, the design plan or abstract plans for a solved problem are not explicitly learned by the system. Instead, the rule instances of each plan are compiled into a set of macrorules that embody the relevant preconditions and postconditions of the plan. These macrorules are built by regressing through the component rules of the plan using a variant of explanation-based generalization [5,16]. This scheme involves computing macrorules for each edge in the plan, followed by a merging operation in which macrorules for connected subgraphs of each abstraction level of the plan are calculated for all sets of compatible edge macrorules. The antecedents and consequents of these macrorules can be viewed, respectively, as "variabilized" problem specifications and design solutions.

Using this scheme for compiling and storing plans has some important advantages:

- only relevant antecedents and consequents of the RDG are preserved,
- because macrorules are independently computed for connected subgraphs of the RDG, which correspond to independent subproblems of a design solution, greater flexibility is available in applying subplans to future design problems,
- greater efficiency is obtained by applying a single macrorule for a given plan than by individually applying each of its component rules,
- correspondence between the parts of a problem and a candidate plan for solving it is automatically maintained by the variable bindings of the plan's macrorule,[1]
- as long as the original domain theory is correct, the resultant macrorules are provably correct because they lie within the deductive closure of the system, and
- increasingly abstract macrorules, obtained by deleting leaf rules from an RDG, satisfy the *abstraction* relation, so they can be organized into a partial order which can be efficiently searched during problem-solving.

Argo's use of rule-dependency graphs contrasts with the explanation-based learning mechanism in [16], which uses proof trees having edges between individual antecedents and consequents of dependent rules. While only one macrorule is computed for the technique presented in [16], Argo computes a set of one or more macrorules for a given explanation. Although the macrorules are harder to compute, they can be applied to situations differing structurally from the original problem.

The justification for a macrorule in Argo's truth-maintenance system is a list of its component rules. If any of these component rules is invalidated by being given an OUT status, the macrorule is also invalidated. This, in effect, gives Argo a nonmonotonic learning capability [14].

In the CAM-cell example discussed previously, a total of ten macrorules are generated for the various abstract plans in Figure 5. These are then inserted into the system's partial order of forward rules.

[1] In some systems for design, a design plan is a tree of rules that have been applied chronologically to a design component in order to yield a design for it. Because some rules decompose components into subcomponents, a problem arises in determining correspondence between parts of the plan and the subcomponents to which they should apply [20].

3.4 Abstraction

A collection of plans, which are represented by their corresponding macrorules, can be organized into a partial order based on a relation called *abstraction*. A plan P_i is a mapping from a domain D_i, determined by the antecedents of the macrorule for P_i, to a range R_i, determined by the consequents of the macrorule for P_i. Intuitively, one plan is more abstract than another if it applies to more situations and if its execution results in fewer commitments. More precisely,

$$P_i \sqsupset P_j \Leftrightarrow (D_i \succ D_j) \wedge (R_i \succ R_j)$$

where $\sqsupset$, the *abstraction* relation, is to be read "is an abstraction of," and where

Definition 1 $S_i \succ S_j \Leftrightarrow$ *the set of possible worlds in which* S_j *is true is a subset of the set of possible worlds in which* S_i *is true.*

This is not a computational definition because of the large number of possible worlds which would exist in a typical application. A simpler and sufficient definition that has been implemented in Argo is

Definition 2 $S_i \succ S_j \Leftrightarrow$ *(one-way-unify* S_i S_j*).*

As defined here, *abstraction* is a transitive, reflexive, and antisymmetric relation: it thus induces a partial order on a set of rules.

3.5 Redesign

After incorporating learned macrorules into its partial order of forward rules, the system is ready for solving a new problem. If a specification is given to the system that is analogous to the CAM-cell specification, then the system follows specialization paths in the partial order of forward rules in order to choose the least abstract macrorule that is applicable, *i.e.*, one that instantiates the largest number of details without making incorrect design commitments. By successively selecting the least abstract rules, the system will typically find the shortest path to a valid design.

4 Results

Table 1 shows measurements of the effort expended in designing several circuits similar to the CAM-cell example, both with and without the experience of designing the original CAM-cell. Circuit 3 and Circuit 6 are exactly analogous to the original problem, differing only in the values for several constants. Thus, the same design plan applies to all three. After the system has been trained on the CAM-cell example, these three circuits can each be solved by executing just three rules, the least abstract macrorules learned by designing the CAM-cell. Circuit 2 utilizes Macrorule-4 and Macrorule-10, Circuit 4 utilizes Macrorule-1 and Macrorule-10, and Circuit 5 utilizes Macrorule-3 and Macrorule-10: these design problems are inexactly analogous to the original example and so use abstractions of the original design plan. Additional rules, which primarily instantiate details, have been located and fired to complete their designs. Circuit 7 is exactly analogous to a subproblem of the CAM-cell, so just one of the calculated macrorules, Macrorule-10, is needed to solve it completely. In all cases, learning resulted in improved design times.

Although the designs generated before learning occurred are correct, they are not optimal in terms of a minimum number of transistors. After being trained by a designer to find an optimal design for the CAM-cell, Argo is able to apply this knowledge to the other circuit design problems and derive better quality designs for them. The improvements, shown as *Design Quality* in Table 1, are substantial.

Table 1: Effects of Learning on VLSI Design

Design	Before Learning/After Learning		
	Time (seconds)	Rules Fired	Design Quality (transistors)
CAM-cell	66.5/59.6	17/3	30/20
Circuit 2	68.8/51.6	17/7	30/26
Circuit 3	66.1/34.9	17/3	30/20
Circuit 4	48.7/33.3	11/5	25/22
Circuit 5	61.7/45.6	15/5	32/17
Circuit 6	64.3/43.2	17/3	30/20
Circuit 7	23.3/19.6	10/1	16/9

Note: Timings were made on a Symbolics 3600.

In this experiment, Argo possessed sufficient metaknowledge, in the form of static priorities on rules, dynamic preferences about rules, and selective erases of assertions, to achieve a correct design without ever having to backtrack. It is unrealistic to expect that for large applications a design system will have enough metaknowledge to guarantee correct designs without search. If Argo possessed none of the above metaknowledge, then it would have to explore many possible paths leading to a design solution in order to locate a correct and complete design. Exhaustively exploring these paths is also unrealistic, but it emphasizes the importance of finding ways to reduce the size of the design space. Macrorules and their abstractions provide just such a capability.

Macrorules, however, are a supplement to, not a replacement for, the initial rules in an application. The initial rules apply in many situations when macrorules do not. For applications requiring little or no search, the presence of macrorules may actually cause slower execution because more possibilities are considered at each problem-solving step [15]. However, when an application requires a search through many alternative paths, macrorules, constituting compiled paths that have proven to be successful in the past, provide dramatic improvements in efficiency.

5 Conclusions

The work reported here is based on developing the fundamental methodology for a system, Argo [1,10], that reasons and learns by analogy for solving search-intensive problems, such as those in design. This methodology includes the use of design plans to effect the analogical transfer of knowledge from a base problem to a target problem, the use of *abstract* plans to allow the transfer of experience to inexactly analogous target problems, an algorithm for calculating macrorules for a design plan that allows the plan to be retrieved and applied efficiently, and the definition of an abstraction relation for partially ordering plans. A fundamental hypothesis employed is that inexact analogies at one level of abstraction become exact analogies at a higher level of abstraction.

If design is viewed as state-space problem solving, then the knowledge in any knowledge-based system for design can be categorized into three fundamental types: design knowledge, control knowledge, and patching knowledge. Based on these categorizations, Argo learns control knowledge. This knowledge is implicit in the design plans and their corresponding macrorules. It is biased by user preferences when Argo is guided interactively to a solution and by rule priorities and preferences when Argo searches automatically. Because Argo stores plans as rule-dependency graphs, the control knowledge preserves user and system choices based on logical, not temporal, precedence.

Argo is typically used as follows: a designer trains an application system on a set of representative examples by making choices as to which solution paths to pursue and manually controlling its backtracking, essentially producing acceptable plans for achieving correct designs. Argo compiles these plans, at various levels of abstraction, into a set of macrorules and maintains these macrorules in the justification network of its JTMS. A knowledge-based contradiction-resolution mechanism is used to revise and

update this network. Given a specification for a new design, Argo attempts to find and apply the least abstract macrorule that is appropriate. Note that less abstract macrorules require fewer additional rules in order to complete a design. Using macrorules in this manner, Argo drastically reduces the amount of automatic search required for new design problems while still producing a correct design.

Argo's use of automatic-but-rigid versus manual-but-flexible mechanisms limits it in several ways. As with other systems employing explanation-based generalization, it cannot learn to design anything outside of the deductive closure of its rule base, because plans are built from an application's domain rules. Its scheme for abstracting plans is inflexible not only in its uniform deletion of all leaf rules, but in preventing Argo from making use of arbitrary parts of a plan. The system does, however, gain leverage by independently computing macrorules for connected subgraphs of the rule-dependency graph (corresponding to design solutions for independent subproblems). Also, alternative procedures for formulating plan abstractions and constructing plan hierarchies are being considered. Other work in progress includes studying and implementing design system architectures embodying analogical reasoning along with more explicit representations of goals, plans, constraints, and contradictions [8].

References

[1] R. D. Acosta, M. N. Huhns, and S. Liuh, "Analogical Reasoning for Digital System Synthesis," *Proceedings of the IEEE International Conference on Computer-Aided Design*, Santa Clara, CA, November 1986, pp. 173-176.

[2] D. C. Brown, *Expert Systems for Design Problem-Solving Using Design Refinement with Plan Selection and Redesign*, Ph.D. Dissertation, Department of Computer Science, The Ohio State University, Columbus, OH, 1984.

[3] J. G. Carbonell, "Learning by Analogy: Formulating and Generalizing Plans from Past Experience," in *Machine Learning, An Artificial Intelligence Approach, Vol. I*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, eds., Tioga Press, Palo Alto, CA, 1983, pp. 137-161.

[4] J. G. Carbonell, "Derivational Analogy: A Theory of Reconstructive Problem Solving and Expertise Acquisition," in *Machine Learning: An Artificial Intelligence Approach, Vol. II*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, eds., Morgan Kaufmann, Los Altos, CA, 1986, pp. 371-392.

[5] G. DeJong and R. Mooney, "Explanation-Based Learning: An Alternative View," *Machine Learning*, vol. 1, no. 2, 1986, pp. 145-176.

[6] R. E. Fikes, P. Hart, and N. J. Nilsson, "Learning and Executing Generalized Robot Plans," *Artificial Intelligence*, vol. 3, 1972, pp. 251-288.

[7] D. Gentner, "Structure Mapping: A Theoretical Framework for Analogy," *Cognitive Science*, vol. 7, no. 2, April 1983, pp. 155-170.

[8] M. A. Gray, "Implementing an Intelligent Design Machine in a TMS-Based Inferencing System," *Proc. 1987 IEEE International Conference on Systems, Man, and Cybernetics*, Alexandria, VA, October 1987, pp. 163-172.

[9] R. Greiner, *Learning by Understanding Analogies*, Ph.D. Dissertation, Stanford University, Technical Report STAN-CS-1071, Palo Alto, CA, September 1985.

[10] M. N. Huhns and R. D. Acosta, "Argo: An Analogical Reasoning System for Solving Design Problems," MCC Technical Report No. AI/CAD-092-87, Microelectronics and Computer Technology Corporation, Austin, TX, April 1987.

[11] S. Kedar-Cabelli, "Purpose-Directed Analogy," Technical Report ML-TR-1, Laboratory for Computer Science Research, Rutgers University, New Brunswick, NJ, August 1985.

[12] T. J. Kowalski, *An Artificial Intelligence Approach to VLSI Design*, Kluwer Academic Publishers, Hingam, MA, 1985.

[13] J. E. Laird, P. S. Rosenbloom, and A. Newell, "Chunking in Soar: The Anatomy of a General Learning Mechanism," *Machine Learning*, vol. 1, no. 1, 1986, pp. 11-46.

[14] S. Liuh and M. N. Huhns, "Using a TMS for EBG," MCC Technical Report No. AI-445-86, Microelectronics and Computer Technology Corporation, Austin, TX, December 1986.

[15] S. Minton, "Selectively Generalizing Plans for Problem-Solving," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August 1985, pp. 596-599.

[16] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli, "Explanation-Based Generalization: A Unifying View," *Machine Learning*, vol. 1, no. 1, 1986, pp. 47-80.

[17] T. M. Mitchell, S. Mahadevan, and L. I. Steinberg, "LEAP: A Learning Apprentice for VLSI Design," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August 1985, pp. 573-580.

[18] T. M. Mitchell, L. I. Steinberg, and J. S. Shulman, "A Knowledge-Based Approach to Design," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-7, no. 5, September 1985, pp. 502-510.

[19] S. Mittal and A. Araya, "A Knowledge-Based Framework for Design," *Proceedings AAAI-86*, Philadelphia, PA, August 1986, pp. 856-865.

[20] J. Mostow and M. Barley, "Automated Reuse of Design Plans," *International Conference on Engineering Design*, Boston, MA, August 1987.

[21] C. J. Petrie, D. M. Russinoff, and D. D. Steiner, "PROTEUS: A Default Reasoning Perspective," *Proceedings of the 5th Generation Computer Conference*, National Institute for Software, Washington, D.C., October 1986.

[22] E. D. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence*, vol. 5, no. 2, 1974, pp. 115-135.

[23] L. I. Steinberg and T. M. Mitchell, "The Redesign System: A Knowledge-Based Approach to VLSI CAD," *IEEE Design and Test*, vol. 2, no. 1, February 1985, pp. 45-54.

[24] "VHDL: The VHSIC Hardware Description Language," *IEEE Design and Test of Computers*, vol. 3, no. 2, April 1986.

[25] P. H. Winston, "Learning by Augmenting Rules and Accumulating Censors," in *Machine Learning, An Artificial Intelligence Approach, Vol. II*, Morgan Kaufman, Los Altos, CA, 1985.

Michael N. Huhns

Computer Society Order Number 837
Library of Congress Number 87-83478
IEEE Catalog Number 88CH2552-8
ISBN 0-8186-0837-4
SAN 264-620X

Proceedings
The Fourth Conference on Artificial Intelligence Applications

Sponsored by the Computer Society of the IEEE
In cooperation with the American Association of Artificial Intelligence

Sheraton Harbor Island Hotel, San Diego, California • March 14-18, 1988

THE COMPUTER SOCIETY OF THE IEEE

IEEE THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

IEEE COMPUTER SOCIETY PRESS