Argo: An Analogical Reasoning System for Solving Design Problems

Michael N. Huhns and Ramón D. Acosta

Microelectronics and Computer Technology Corporation
Advanced Computing Technology and VLSI CAD Programs
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. Closely integrated into Argo's analogical reasoning facilities are modules for the acquisition, storage, retrieval, evaluation, and application of previous experience. Problem-solving experience is acquired in the form of problem-solving plans represented as rule-dependency graphs. From increasingly abstract versions of these graphs, Argo calculates sets of macrorules. These macrorules are partially ordered according to an abstraction relation for plans, 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 are yielding insights into 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 [4, 5, 22, 29, 30, 31, 39]. These systems are particularly suited to situations in which heuristic expert knowledge must be employed because algorithmic techniques are unavailable or prohibitively expensive. Restrictions on the types of problems or domains handled by such design systems are progressively being eased. 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 the means for remembering the relevant parts of previous design efforts and be able to employ this accumulated experience in solving future design problems. Learning from experience is a powerful technique used by humans to improve their problem-solving ability. 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 these training examples in order to obtain more widely applicable results. The STRIPS [11, 12] problem-solving system incorporates a technique for generalizing plans and their preconditions based on the formation of macro-operators (MACROPs). In this technique, an existing plan, consisting of a sequence of operators whose execution yields a goal state, is stored in a data structure called a triangle table. This table represents the preconditions and postconditions for each operator in the plan. The plan is generalized by replacing all precondition constants by distinct parameters, and then correcting for overgeneralization by substituting for inconsistent parameters. The resultant generalized plan, a MACROP, is stored and later used as either a plan, a set of subplans, or an execution monitor.

A better procedure for generalization, developed in the context of learning from examples, uses a proof-based explanation (or verification) mechanism [9, 23, 25, 27, 28], often termed explanation-based generalization (EBG). It is an improvement over the use of a triangle table in that it does not require any heuristics to compensate for possible overgeneralizations. The proof employed comprises information about why a training example satisfies a particular goal. The procedure involves first a modified regression of the goal through the proof structure [33, 42], whereby sufficient constraints on the domain of training examples for which the proof holds are computed. These constraints are based on the codomain of goals allowed. The second stage of the procedure is to reapply the proof structure to the resultant generalized domain to obtain a generalized codomain. In the terminology used above, a plan is like a proof, a plan precondition is the domain for the proof, and the resultant design is the codomain.

For design problems, 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 [14], is accomplished in ABSTRIPS [36] by ignoring operator preconditions considered to be details. Criticality values are attached to the preconditions of operators to determine their importance. These values are computed based on the presumed difficulty of satisfying each precondition if it is not already satisfied. 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 [32, 38]. 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 the stored plan and subproblems of a partial design is difficult to establish automatically.

The transfer of experience from previous problem-solving efforts to new problems has also been accomplished via analogical reasoning methods [7, 19, 43, 44]. Analogical reasoning is a mapping from a base domain to a target domain that allows the sharing of features between these domains. With respect to problem-solving, many of the previously reported methods are limited by their requirements that either new problems be very similar to previously solved ones or analogies be supplied by a user and match perfectly.

1.2 Argo

The primary objective of our work has been to develop a robust and domain-independent system for applying analogical reasoning to solving search-intensive problems, such as those in the domain of design [1]. Search efficiency is increased by using past experience captured in the form of design plans: a new design problem can be solved by applying a design plan from an analogous old problem. Two problems are considered exactly analogous if the same design plan can be applied to solve either. Past experience, however, should also be useful for solving a new problem when a stored design plan "almost" applies. A fundamental hypothesis employed is that inexact analogies at one level of abstraction become exact analogies at a higher level of abstraction. Thus, techniques have been developed for automatically computing and storing increasingly abstract versions of design plans and subsequently employing them in solving new problems. These analogical reasoning and learning mechanisms are integrated into Argo, a tool for building knowledge-based systems.

In Argo, a design plan is represented using a rule-dependency graph. Abstractions of this design plan are obtained by deleting rules from this graph. Macrorules for a plan and its abstractions are calculated using an explanation-based scheme and inserted into a partial order of rules according to an abstraction relation. This allows the system to retrieve efficiently the most specific plan applicable to solving a new problem. The use of abstraction allows Argo to apply its previous problem-solving experience to problems that are analogous to those it has solved previously.

Section 2 of this paper contains a discussion of reasoning and learning by analogy and its application to the process of design. Section 3 describes the functional characteristics of Argo, as well as an application to VLSI digital circuit synthesis (Argo-V). Descriptions of the fundamental mechanisms in Argo that allow it to reason and learn efficiently are presented in Section 4. Section 5 lists and discusses some experimental results obtained using the Argo-V application. Finally, conclusions and directions for future research are presented in Section 6.

2 Analogical Reasoning and Learning

Analogy involves a transfer of information from a base domain to a target domain. Research into how people are able to reason and solve problems by analogy reveals that the relationships among domain objects, not the attributes of these objects, are what is transferred between the base and target domains. The particular relationships transferred are those obeying a systematicity principle [15], having a causal link [43, 44], or sharing a similar purpose [20]. Difficulties in adding analogical reasoning to knowledge-based systems stem from the formalization of the systematicity principle, identification of causal links, or specification of the purpose of a base domain example.

Two important issues in overcoming these difficulties for problem-solving are analogy recognition and analogical transformation. This section contains a discussion of these issues and their relation to the domain of design, as well as Argo's use of analogical reasoning with learned design experience.

2.1 The Use of Analogy for Solving Design Problems

Using analogy for solving problems is something that people appear to do very well and that machines currently do poorly [16, 17, 35]. After several attempts [2, 3, 21], it has begun to provide limited assistance in automatically proving mathematical theorems. Why should analogy be successful in design, a domain with a weaker underlying theory?

1. Design problems are typically represented by functional specifications that provide constraints on admissible solutions. These specifications are typically incomplete, allowing many "correct" solutions and, consequently, making the search for one of these easier.

2. Because the design domain is essentially hierarchical, transformations and decompositions that can be applied to functional specifications lead to increasingly simpler, and often independent, subproblems. Thus, even if a design system lacks the knowledge to completely solve a problem, partial designs generated by the system might be easily patched to yield a complete design.

3. Rules in the design system can be viewed as complex special-purpose inference procedures that match special structures in the functional specifications and construct instances of one or more design artifacts. By using these larger chunks of knowledge about design and problem decomposition, the complexity of design by analogy is reduced [37].

The domain of design thus provides an appropriate environment for the development and use of analogical reasoning.

We classify analogies as being either *exact* or *inexact*. Where there is an exact match between a past experience and a new problem-solving situation, an exact analogy exists and the new problem can be solved either by executing the old plan or by directly using the old results. Where an exact match does not exist, the two problems that arise are 1) *analogy recognition*: finding the most similar past experience, and 2) *analogical transformation*: adapting this experience to the new problem situation. Several techniques have been suggested for recognizing the most similar past experience.

1. Develop an analytical similarity measure [7]: this requires that the domain have a metric, which is not typical for design domains.

2. Find a past experience whose first stage is identical to the current problem situation [8]: this technique fails to find past experiences that differ by only an initial detail and requires that the initial reasoning be done without assistance from past experiences.

3. Find a past experience that has the same causal connections among its components as does the current problem [6, 15, 43, 44]: since there may be many causal networks that describe each past experience, all of which have to be computed and stored, this technique may discover too many analogous situations and be unable to determine the most appropriate one. More importantly, this makes analogy too dependent on a proper problem representation.

4. Find a past experience that has the same purpose as does the current problem situation [20]: this requires an advance enumeration of possible purposes for which an experience can be analogically employed, a difficult task to conduct automatically.

The second problem, the adaptation of old experiences to new problem situations, has been attempted previously by employing heuristically-guided, incremental perturbations according to primitive transformation steps [7]. 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 [19] and user intervention [32].

2.2 Analogical Reasoning in Argo Using Design Experience

The primary motivation for using analogical reasoning in Argo is to allow previous design experience to aid in solving new problems, even when there is only a partial match between old and new problems. Three questions that must be resolved in developing such a system are

1. In what form should the past experience be stored?

2. How can the most similar and relevant past experience among all those that are stored be found?

3. How can this past experience be used in solving the new problem?

For the second of these questions there are two possible approaches. First, an old solved problem that is "most similar" to a new unsolved problem can be located by comparing the new problem's description to a description of each old problem. Unfortunately, a metric or similarity measure for the the domain of design problem descriptions is difficult to quantify. The second approach is to compare a description of the new problem to a calculated precondition for a plan to solve each old problem. The precondition for a plan constitutes exactly those aspects of the problem that are relevant to its solution.

This last approach is the one employed by Argo. Hence, answers are provided to the other two questions: 1) past experience is stored in the form of design plans with their corresponding preconditions and postconditions, and 2) when a new problem satisfies the precondition for a plan, the postcondition of the plan can be directly executed to solve, at least partially, the new problem.

It has been postulated that humans discover and comprehend analogies at an abstract level, a level where both the base and target domains are identical [16]. Based on this notion, Argo generates increasingly abstract plans by deleting rules in a design plan—those corresponding to details of the problem solution. Because abstract plans apply to broader classes of design problems, this approach leads to the solution of problems that are analogous, but not necessarily identical to the original problem. As explained in Section 4, Argo's analogical reasoning capability has been developed with these concepts in mind.

3 The Argo Development Environment

Argo is a generic development environment for the use of analogical reasoning and learning in solving problems, particularly in design domains. It is a derivative of the Proteus expert system tool [34], which includes a rich set of knowledge representation techniques and inferencing strategies.

3.1 Knowledge Representation and Inference in Argo

Knowledge is represented in Argo using a combination of predicate logic and frames. Data can consist of ground assertions, general assertions, forward rules, backward rules, or slot values in frames. Each datum is included in a justification-based truth-maintenance system (JTMS), and as such, has a set of justifications and a belief status of IN or OUT [10]. 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, forward rules allow Lisp functions to appear in both their antecedents and consequents. Antecedents may also include the second-order predicates *forget* (used to prevent the inclusion of antecedents in the justifications for the consequents of a rule) and *unless* (which implements negation-by-failure and is used for nonmonotonic reasoning). Consequents may include the second-order predicate *erase* (used to invalidate existing justifications of a datum, enabling an emulation of rewrite rules). Backward rules allow Lisp functions and *unless* to appear in their antecedents.

The primary 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 (forward rule instance), its antecedents must be provable by either the explicit existence of assertions or slot values in the database, or by proving their implicit existence in the database through backward chaining. Consequently, backward rules are usually used as support rules for searching, computation, and parsing.

3.2 Argo Control Strategy for Design

A variety of artificial intelligence techniques is available for solving design problems. The following control strategy for solving a problem P is based on classifying these techniques according to the amount and specificity of domain knowledge they require [8]:

1. If knowledge of an artifact that satisfies P is available, then this solution is directly instantiated.

2. If a specific plan for transforming and decomposing P is available, then it is directly executed.

3. If a plan for solving P' is available, where P' is "similar" to P, then it is analogically transformed to synthesize an artifact for P.

4. If past experience is unavailable, then weak methods such as heuristic search or means-ends analysis are employed.

Argo executes this problem-solving strategy by means of the algorithm shown 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 [13]): it executes a production system by performing a sequence of rule-firing operations, called the *recognize-act cycle*. 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 4.4) 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). Valid rule instances are then placed in a conflict set. This set consists of those rules 1) whose antecedents are satisfied, 2) that have not previously fired on the same data, 3) that are not more abstract than another rule in the conflict set, and 4) for which there is no justifying datum that itself is justified by a forward rule which is a justification for this rule. This last restriction prevents intermediate rules, which may have contributed to the formation of a more specific rule during learning, from firing on the data generated by the firing of that macrorule.

After a conflict set is computed, a user may interact with the system by

```
Read a database of domain knowledge, consisting of forward
   rules, backward rules, frames, and assertions;
Insert forward rules into a partial order based on "abstraction";
Problem:  Loop until Quit asserted
          Read a problem specification into working memory;
          Compute conflict set;
          Design:  Loop until conflict set empty or Halt asserted
                   Resolve Conflicts--Select one forward
                        rule, R, from the conflict set
                        (interactively or syntactically);
                   Act--Perform the consequents of R and
                        update the JTMS justification network;
                   Match--Find the conflict set of most specific
                        applicable rules 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 connected subgraphs of RDG;
                   Insert macrorules into partial order;
                   Abstract RDG (by deleting leaf rules);
          Initialize working memory;
Store updated database;
End.
```

Figure 1: Argo Control Strategy

1. firing one or more actions,

2. asking why a particular action is eligible to fire,

3. asserting preferences about rules in order to control the selection of an action from the conflict set for firing, *e.g.*, asserting (*prefer* R_i R_j) indicates that rule R_i should be selected over rule R_j if both rules are in the conflict set,

4. asserting new facts or rules, which will cause a new (and potentially different) conflict set to be computed, and

5. contradicting a result in order to initiate dependency-directed backtracking.

In addition to *prefer*, rule firings can be controlled by means of priorities (a number from 1–99) assigned statically to rules. Conflict resolution then involves selecting one rule for which no other rule in the conflict set has a higher priority, and for which no other rule with the same priority is preferred.

This model for forward chaining has the following characteristics and advantages for analogical reasoning:

1. Abstract rules do not have to be considered once a more specific rule has been found to be satisfied on the current database; of course, these rules are reconsidered if a consequent of the more specific rule is contradicted.

2. The results obtained are independent of the order in which data are asserted, allowing new rules to be added at arbitrary times. However, as in Prolog, the order among facts and backward rules is significant.

3. Metalevel reasoning is easier in that an explicit conflict set clearly indicates which rules are competing to fire and, thus, which rules require further reasoning steps. For example, two rules that are meant to fire consecutively should not appear in the conflict set at the same time. If they do, then there must be some missing control knowledge, such as additional antecedents and consequents, triggering rules, or preferences.

In Figure 1, the learning phase is outside of the control loop of Argo's production-system interpreter, and as such, it 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 while executing the problem-solving design phase. These intermediate results are stored in the system as a list of asserted actions. The actions have justifications so that they can be managed by the truth-maintenance system. If the problem-solving system backtracks, causing actions to have a belief status of OUT, then their corresponding rule instances are not included in the rule-dependency graph used for representing plans. Thus, the plans that are learned do not incorporate failed lines of reasoning.

3.3 Argo Application: VLSI Circuit Design

As noted, Argo is a generic environment for the use of analogical reasoning and learning in problem-solving systems. 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 building and 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 a digital logic circuit. This set of assertions

- describes the *behavior* of the circuit
- 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
- may be a superset of the set describing the specification.
- is less "abstract" than the set describing the specification

The assertions and frames are based on VHDL (VHSIC Hardware Description Language) [40, 41].[1] 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:

Frame Definitions: defining structures for VHDL modules (*e.g.*, entities, interface bodies, and architectural bodies).

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

Assertions: library component slot values and other general knowledge.

Forward Rules: primary design rules.

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

[1] The system does not explicitly use VHDL syntax; rather, it employs a one-to-one translation of VHDL statements into Argo frames and assertions.

Argo-V's design knowledge currently consists of 32 frames, 38 frame instances, 23 backward rules, and 34 forward rules. Figure 2 shows the hierarchical relationship among some of the frame definitions and instances. 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.

Most of the design rules in Argo-V either transform, instantiate, or decompose. A transformation rule is used to convert one or more signal assignment statements of an architectural body 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 are used for grouping logically related signal assignments so that they can be treated as independent subproblems. Figures 3–5 contain examples of each of these types of rules.

In these rules, signal assignment statements are matched with antecedents of the form

```
(signal-assignment ?body (?lhs (?signal1 ?delay1 ?condition)
                                (?signal2 ?delay2)))
```

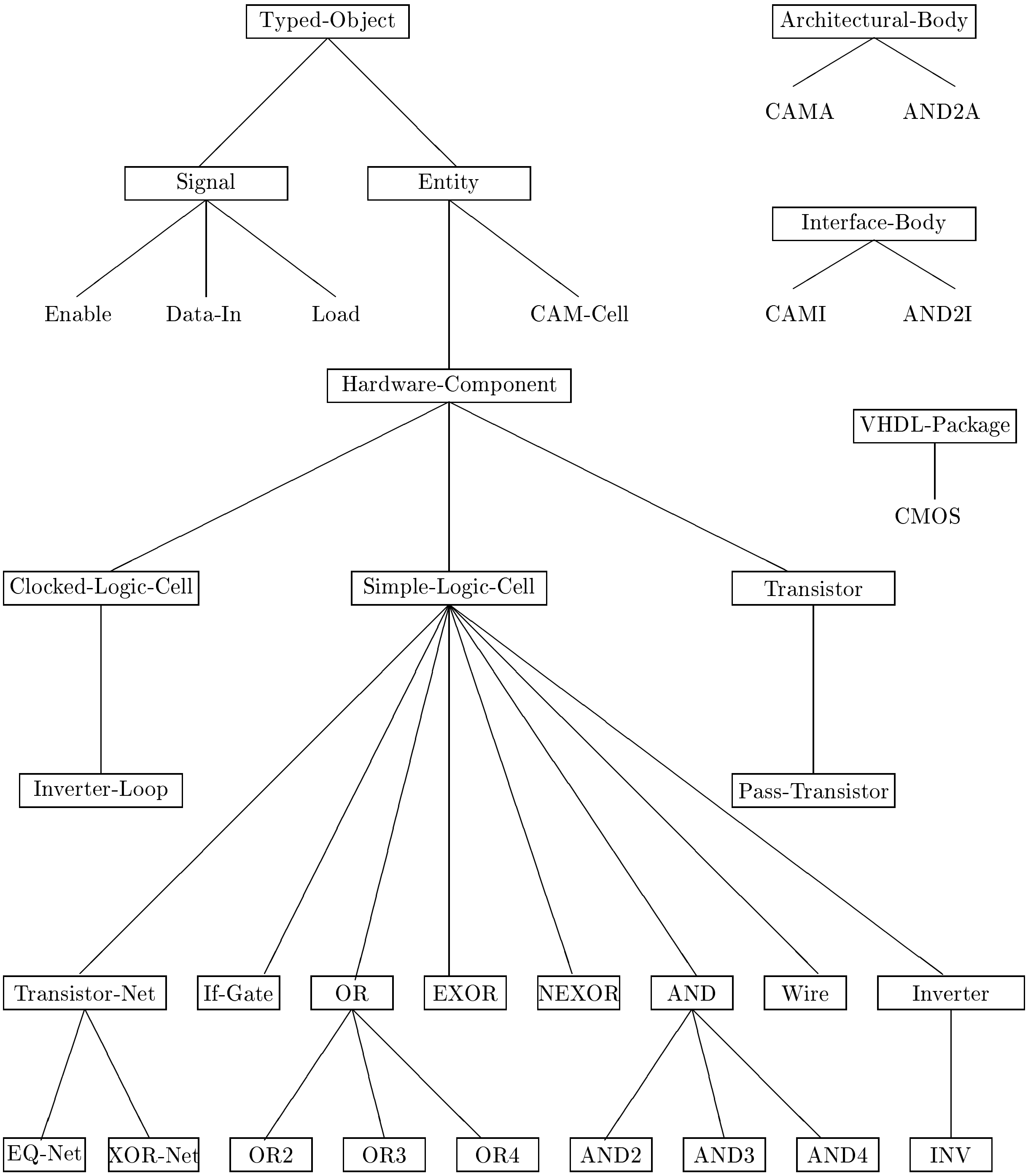
where `?body` is the architectural body of an entity, `(?signal1 ?delay1 ?condition)` means that `?lhs` is assigned the value of `?signal1` after a delay of `?delay1` if `?condition` is satisfied, and `(?signal2 ?delay2)` is the default assignment of `?lhs` if `?condition` is not satisfied. The Pass-Partition rule, shown in Figure 3, describes how the ANDed conditions of a signal assignment statement can be transformed into a cascade of pass transistor networks. Components for pass transistors and exclusive-OR transistor networks are instantiated using the rules in Figure 4. A decomposition rule for statements requiring memoried components due to signal feedback appears in Figure 5.

4 Analogy Mechanisms in Argo

The five facets of Argo's analogical reasoning mechanism involve the acquisition, storage, retrieval, evaluation, and application of previous experience. A plan is acquired by the system as it solves a problem and is represented using a graph of rule dependencies. Increasingly abstract versions of the plan are obtained by deleting rules from it. Macrorules, consisting of relevant preconditions and postconditions, are computed for each plan and stored in a partial order according to the *abstraction* relation. The partial order facilitates retrieval and evaluation of the most specific plan to apply en route to solving an analogous problem.

4.1 Rule-Dependency Graphs

To implement the problem-solving strategy of Section 3.2, 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



Note: Boxes indicate classes and unboxed items indicate class instances.

Figure 2: Hierarchy of classes for digital circuit design in Argo-V

```
;;;; Pass-Partition Transformation Rule (PP)
;;;;
((architecture ?entity:entity ?body:architectural-body)
 (unless (type ?entity hardware-module))
 (signal-assignment ?body
                    (?output (?input1 ?delay1 ?signal) (hi-z ?delay2)))
 (signal-assignment ?body
                    (?signal ((BIT-AND ?input2 ?input3) ?delay3)))
 -50->
 (= ?new-signal (new signal))
 (erase (signal-assignment ?body
                    (?output (?input1 ?delay1 ?signal) (hi-z ?delay2))))
 (erase (signal-assignment ?body
                    (?signal ((BIT-AND ?input2 ?input3) ?delay3))))
 (type ?new-signal bit-passive)
 (signal-declaration ?body ?new-signal)
 (signal-assignment ?body
                    (?new-signal (?input1 ?delay3 ?input2)
                                 (hi-z ?delay3)))
 (signal-assignment ?body
                    (?output (?new-signal ?delay1 ?input3)
                             (hi-z ?delay2))))
```

Figure 3: Example of a transformation rule in Argo-V

```
;;;; Rule for Instantiating an Exclusive-OR Transistor Net (XTN)
;;;;
((architecture ?entity:entity ?body:architectural-body)
 (unless (type ?entity hardware-module))
 (signal-assignment ?body
                    (?output (?input1 ?delay1 ?signal) (hi-z ?delay2)))
 (signal-assignment ?body
                    (?signal ((BIT-NOTEQUAL ?input2 ?input3) ?delay3)))
 -45->
 (erase (signal-assignment ?body
                    (?output (?input1 ?delay1 ?signal) (hi-z ?delay2))))
 (erase (signal-assignment ?body
                    (?signal ((BIT-NOTEQUAL ?input2 ?input3) ?delay3))))
 (component ?body (XOR-NET (?input1 ?input2 ?input3 ?output))))
```

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

Figure 4: Examples of instantiation rules in Argo-V

```
;;;; Decomposition Rule for Memoried vs. Combinational Logic (MD)
;;;;
((architecture ?entity:entity ?body:architectural-body)
 (type ?entity abstract-module)
 (signal-assignment ?body (?lhs . ?rhs))
 (is-contained-in ?lhs ?rhs)
 -50->
 (erase (signal-assignment ?body (?lhs . ?rhs)))
 (= ?new-entity (new entity))
 (= ?new-interface (new interface-body))         ;for new entity
 (= ?new-architecture (new architectural-body))  ;for new entity
 ;; assert attribute values for new entity
 (predecessor ?new-entity ?entity)
 (type ?new-entity memory-module)
 (interface ?new-entity ?new-interface)
 (architecture ?new-entity ?new-architecture)
 (signal-assignment ?new-architecture (?lhs . ?rhs))
 ;; add the proper ports and signals to this entity
 (assign-input-ports ?new-entity)
 (assign-output-ports ?new-entity ?entity)
 (assign-local-signals ?new-entity)
 ;; add the new entity to the structure of its predecessor entity
 (component ?body (?new-entity (ports ?rhs) . ?lhs)))
```

Figure 5: Example of a decomposition rule in Argo-V

interchangeably throughout this discussion. 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. Only when the learning phase is invoked is an explicit representation of the plan built.

One example from Argo-V involves the design of a content-addressable-memory (CAM) cell, very similar to the one used in [29]. The specification for the CAM-cell design problem can be seen in Figure 6. 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.

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 7. Figure 8 contains a circuit diagram for this solution.

Once a design, or partial design, has been completed, the learning phase of Argo can be invoked. Its first task is to build a plan 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 9.

4.2 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. The issue of how to formulate plan abstractions is discussed in this section. Macrorule computation and the partial order of rules are presented in succeeding sections.

A number of domain-dependent and domain-independent techniques for automatically generating plan abstractions are possible. For a given plan, these techniques include:

1. Deleting a rule having no outgoing edges, *i.e.*, one upon which no other rule in the plan is dependent. For design domains, such rules typically instantiate details; it seems plausible that deleting these rules will yield plan abstractions because the resultant plans will make fewer commitments to implementation details.

2. Replacing a rule by a more general rule that refers to fewer details of a problem. A more general rule might be one having fewer antecedents or consequents, fewer constants, more variables, more general domain constants, etc. As achieved in ABSTRIPS [36], these rules can be generated from the initial domain knowledge using criticality measures.

3. Replacing a sequence of rules or a subgraph by a single, more general rule.

```
CAM-CELL is an instance of ENTITY
    Type:                 ABSTRACT-MODULE
    Predecessor:          NONE
    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
    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))
    Component:            no known values

ENABLE is an instance of SIGNAL
    Type:                 BIT-ACTIVE
    Object-Class:         none
    Static-Expression:    none
    Characteristic-Indication: none
    Constraint:           none

MATCH is an instance of SIGNAL
    Type:                 BIT-PASSIVE
    Object-Class:         none
    Static-Expression:    none
    Characteristic-Indication: none
    Constraint:           none
```

Figure 6: Behavioral specification for the CAM-cell design problem

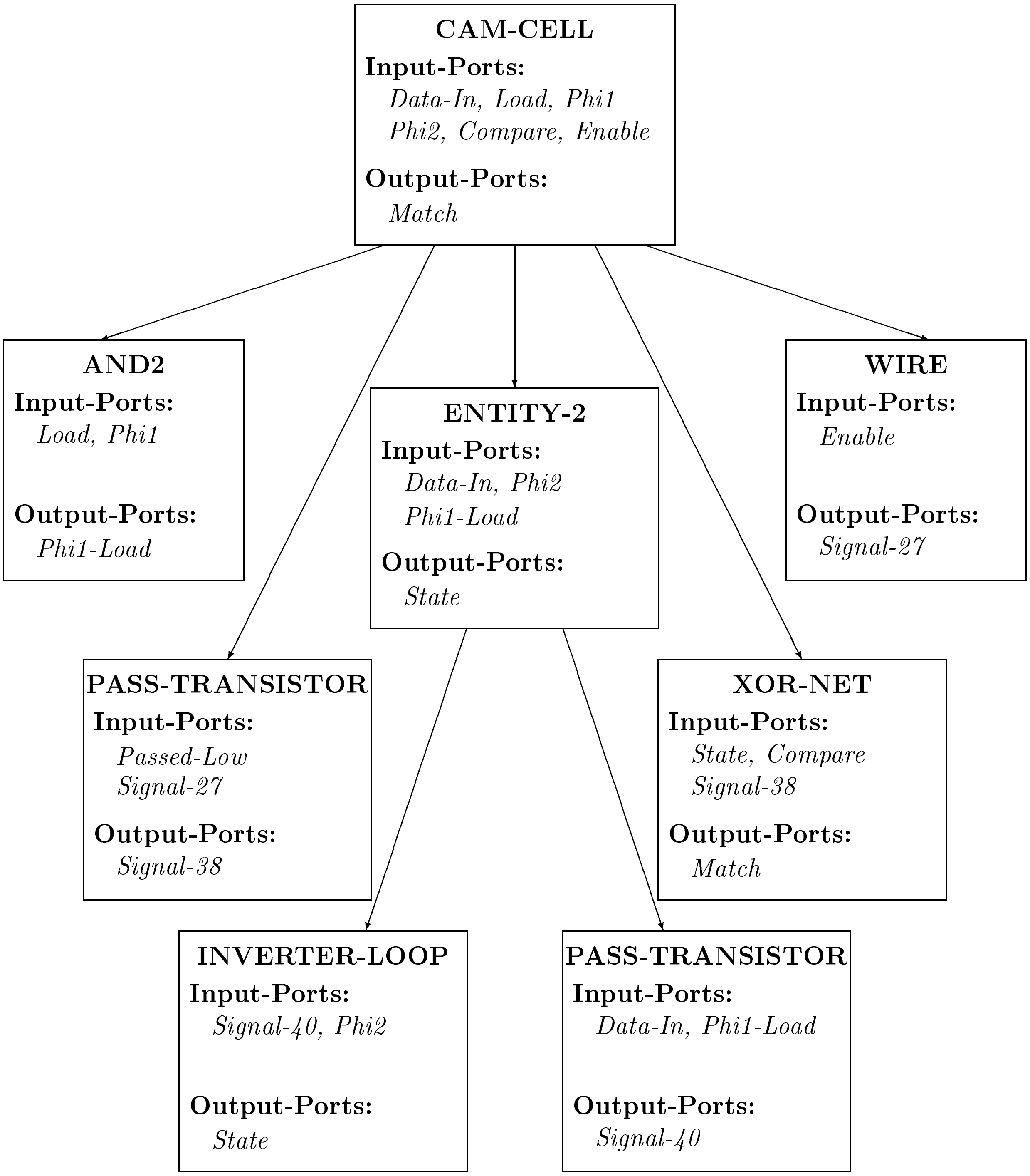


Figure 7: Hierarchical description of the final design for the CAM-cell

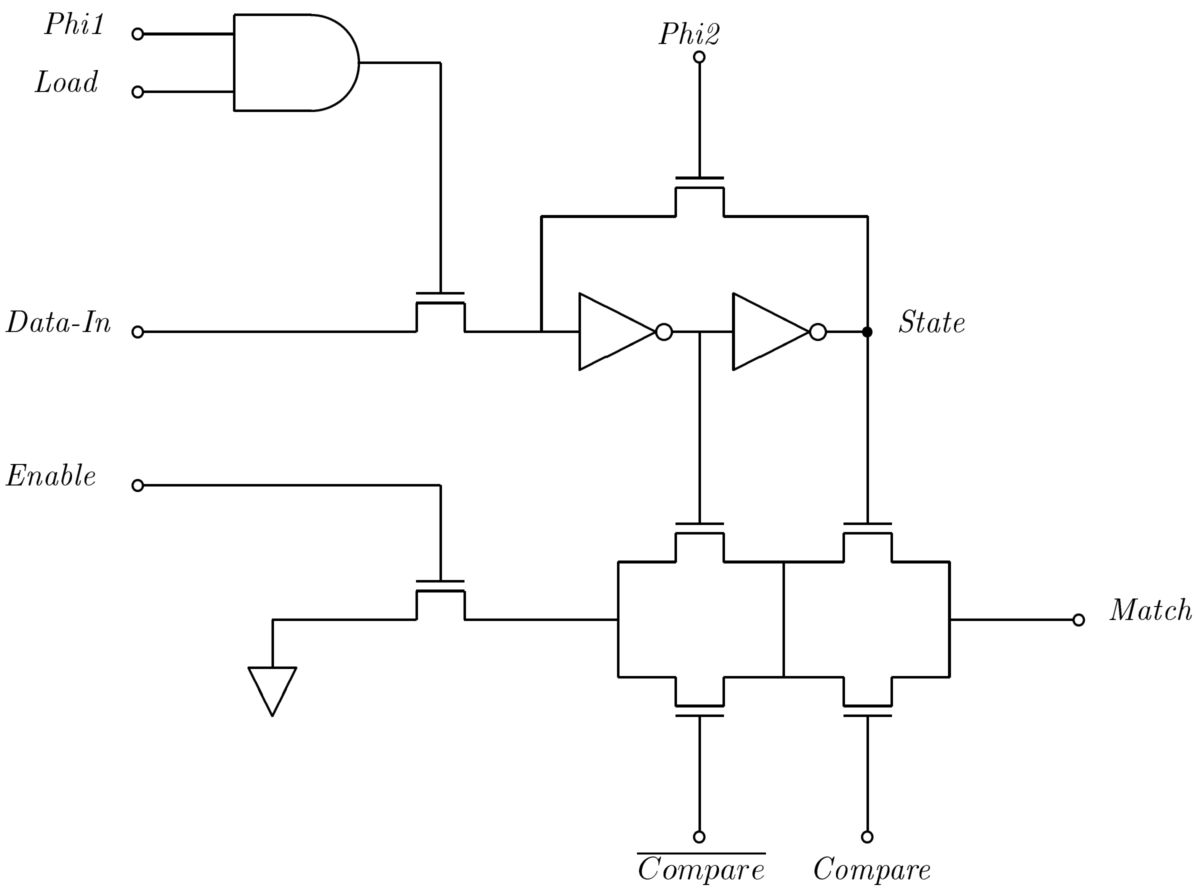
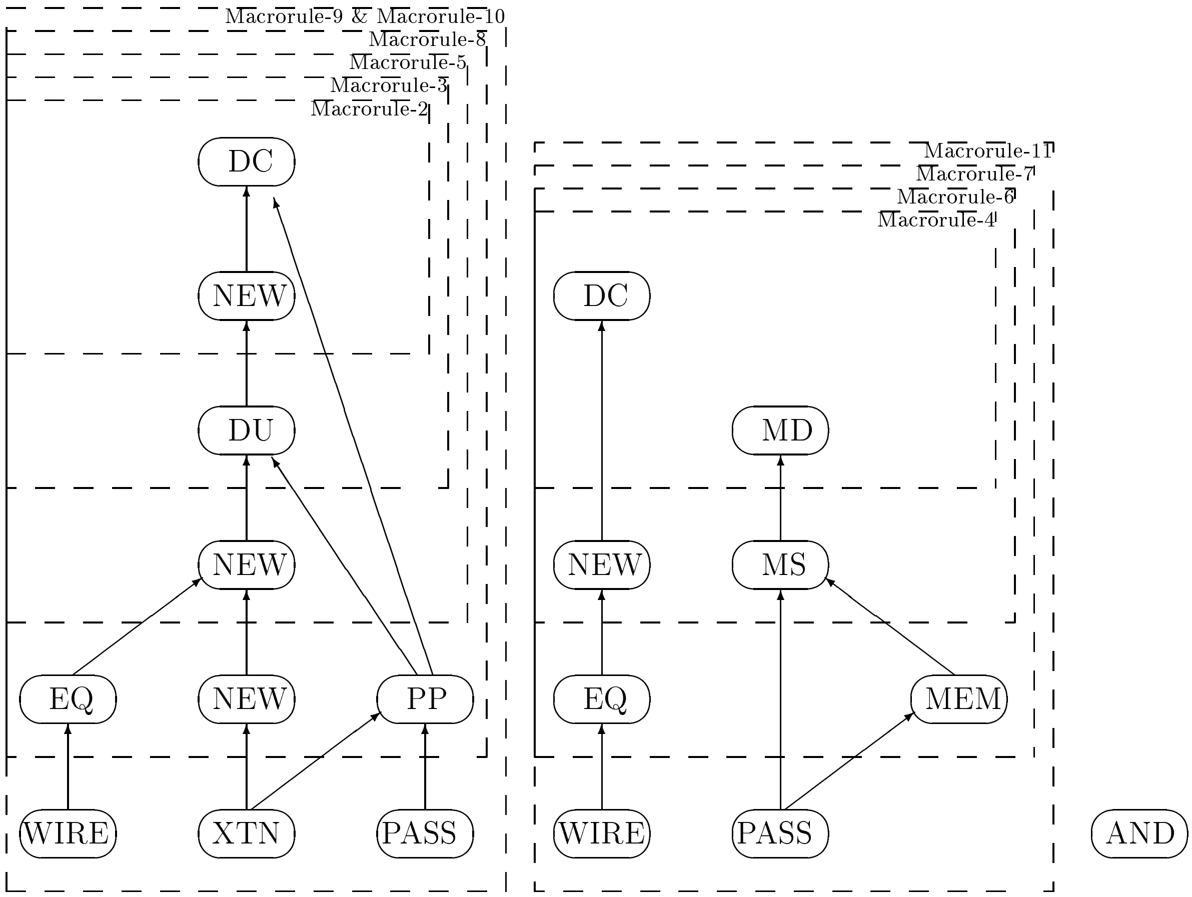


Figure 8: Schematic diagram for the CAM-cell



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 block of signal-assignment statements into ones representing a cascade of pass-transistor networks

XTN: Instantiate an exclusive-OR pass-transistor network

PASS: Instantiate a pass transistor

MD: Decompose an entity into one containing memoried statements and one containing 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 9: Design plan (rule-dependency graph) for the CAM-cell design problem

4. Computing and generalizing a macrorule for the plan without reference to the components of the original plan.

Some of the questions that must be carefully considered in choosing an appropriate abstraction scheme include the following:

1. How independent is the technique from the application domain?
2. Are the abstractions generated within the deductive closure of the system?
3. How automatic is the technique?
4. How easy is it to implement?
5. How useful are the abstractions for solving analogous problems?

In keeping with these guidelines, the abstraction scheme currently employed in Argo is a variation of the first option listed above. It involves automatically formulating a plan abstraction 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. Thus, increasingly abstract versions of a plan are obtained by iteratively trimming it until either one or zero nodes remain (see Figure 1). A sequence of abstractions for the CAM-cell example, generated by this technique, appears in Figure 9. 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.

One possible drawback of Argo's automatic abstraction scheme is that deleting all leaf rules might eliminate potentially useful abstract plans in which only part of the leaf rules should be deleted. Except for the very smallest plans, however, it is clearly not practicable to automatically generate macrorules for all possible subgraphs of the RDG, although these would be valid and potentially useful. In addition, although additional forward chaining might be required, it is always possible for the system to start with a plan's previously computed abstraction, followed by instantiations of the relevant trimmed rules, to obtain the appropriate "abstraction" required to solve a new problem.

Argo computes abstractions during its learning phase—after a problem is solved. In contrast, it is possible to save a plan and only compute abstractions when necessary, *i.e.*, when solving new problems in which an abstract version of an original plan is applicable. There are difficulties with using this approach, including identification of the most suitable previous plan using some type of partial match procedure and analogical transformation of the selected plan based upon the partial match results. Consequently, Argo uses an *a priori* approach to generating abstractions.

4.3 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 [9, 27]. The antecedents and consequents of these macrorules can be viewed, respectively, as "variabilized" problem specifications and design solutions.[2]

The Argo generalization 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. This merging is accomplished by incrementally merging each set of edge macrorules into a set of cumulative macrorules for previously merged edges. When an edge and a cumulative macrorule are merged, constraints on variable bindings are maintained by modifying a cumulative substitution list, rather than explicitly making substitutions in the individual antecedents and consequents of the cumulative macrorule. Substitution lists are also used for efficiently detecting incompatibilities between macrorules. This scheme does not require that a plan be restricted to being a tree or a connected graph.

A brief example of this procedure is illustrated with the plan in Figure 10, consisting of an instance of the Pass-Partition (PP) rule followed by the rules for instantiating an exclusive-OR transistor network (XTN) and a pass transistor (PASS). (Notice that this RDG is a subgraph of the plan for the CAM-cell design problem in Figure 9.) Macrorules for this RDG are obtained by independently computing macrorules for the two edges in the graph, PP → XTN and PP → PASS. These edge macrorules are subsequently merged, yielding the two macrorules in Figure 11. Two macrorules are computed because, depending on the specification, the resultant design can have the pass transistor appear before or after the exclusive-OR transistor network.

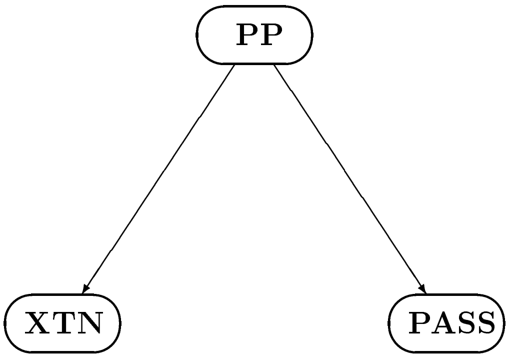


Figure 10: A subplan of the RDG for the CAM-cell, used to illustrate macrorule formation

The use of macrorule compilation for storing plans has some important advantages:

1. only relevant antecedents and consequents of the RDG are preserved,

2. 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,

[2]Note that if the rules of a plan are rewrite rules, then the macrorule is also a rewrite rule.

```
(Macrorule-A
 (architecture ?x0:entity ?x1:architectural-body)
 (unless (type ?x0 hardware-module))
 (signal-assignment ?x1 (?x2 (?x3 ?x4 ?x5) (hi-z ?x6)))
 (signal-assignment ?x1 (?x5 ((BIT-AND ?x7 ?x8:signal) ?x9)))
 (signal-assignment ?x1 (?x7 ((BIT-NOTEQUAL ?x12 ?x13) ?x14)))
 (= ?x10 (new signal))
  -50->
 (erase (signal-assignment ?x1 (?x2 (?x3 ?x4 ?x5) (hi-z ?x6))))
 (erase (signal-assignment ?x1 (?x5 ((BIT-AND ?x7 ?x8) ?x9))))
 (erase (signal-assignment ?x1 (?x7 ((BIT-NOTEQUAL ?x12 ?x13) ?x14))))
 (type ?x10 bit-passive)
 (signal-declaration ?x1 ?x10)
 (component ?x1 (XOR-NET (?x3 ?x12 ?x13 ?x10)))
 (component ?x1 (PASS-TRANSISTOR (?x10 ?x8 ?x2))))


(Macrorule-B
 (architecture ?x0:entity ?x1:architectural-body)
 (unless (type ?x0 hardware-module))
 (signal-assignment ?x1 (?x2 (?x3 ?x4 ?x5) (hi-z ?x6)))
 (signal-assignment ?x1 (?x5 ((BIT-AND ?x7:signal ?x8) ?x9)))
 (signal-assignment ?x1 (?x8 ((BIT-NOTEQUAL ?x12 ?x13) ?x14)))
 (= ?x10 (new signal))
  -50->
 (erase (signal-assignment ?x1 (?x2 (?x3 ?x4 ?x5) (hi-z ?x6))))
 (erase (signal-assignment ?x1 (?x5 ((BIT-AND ?x7 ?x8) ?x9))))
 (erase (signal-assignment ?x1 (?x8 ((BIT-NOTEQUAL ?x12 ?x13) ?x14))))
 (type ?x10 bit-passive)
 (signal-declaration ?x1 ?x10)
 (component ?x1 (XOR-NET (?x10 ?x12 ?x13 ?x2)))
 (component ?x1 (PASS-TRANSISTOR (?x3 ?x7 ?x10))))
```

Figure 11: Two macrorules computed from a subplan of the RDG for the CAM-cell

3. greater efficiency is obtained by applying a single macrorule for a given plan than by individually applying each of its component rules because a) variable bindings are preserved and do not have to be reestablished and b) rewritten assertions do not have to be made and then retracted,

4. 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,[3]

5. 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,

6. 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, and

7. the most general—*i.e.*, necessary and sufficient—precondition for a plan is obtained as a disjunction of the set of antecedents of all macrorules at a given level.

Argo's use of rule-dependency graphs contrasts with the explanation-based learning mechanism in [27], in which explanations consist of proof trees having edges between individual antecedents and consequents of dependent rules. While only one macrorule is computed for the technique presented in [27], 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 [24].

In the CAM-cell example discussed previously, a total of ten macrorules are generated for the various abstract plans in Figure 9. These are then inserted into the system's partial order of forward rules, as defined in the next section.

4.4 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

[3]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 [32].

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 that 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. Figure 12 shows the partially-ordered rules used in Argo-V for designing digital circuits. This figure includes the macrorules learned from the CAM-cell example.

4.5 Redesign

At this point, the system is ready for solving a new problem. If a specification is given to the system that is exactly analogous to the CAM-cell specification, then, depending upon the structure of the new problem, Macrorule-10, the AND-Gate Rule, and Macrorule-5 or Macrorule-6 in Figure 9 can be applied to completely solve it. Alternatively, if the specification is for a problem that is inexactly analogous to the CAM-cell, 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. It is expected that by successively selecting the least abstract rules, the system will typically find the shortest path to a valid design. The architectural body of an inexactly analogous CAM-cell variation, differing in how the *MATCH* output is computed, appears in Figure 13. Using Macrorule-1 and Macrorule-10, a solution to this CAM-cell variation, as shown in Figure 14, can be obtained with the plan illustrated in Figure 15.

5 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. This experience results in the calculation of ten macrorules. Specifications for the architectural bodies of the examples are shown in Figure 16. Circuit 3 and Circuit 6 are exactly analogous to the original design 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-8 and Macrorule-11, Circuit 4 utilizes Macrorule-2 and Macrorule-11, and Circuit 5 utilizes Macrorule-5 and Macrorule-11: these design problems are inexactly analogous to the original example and so can use only 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

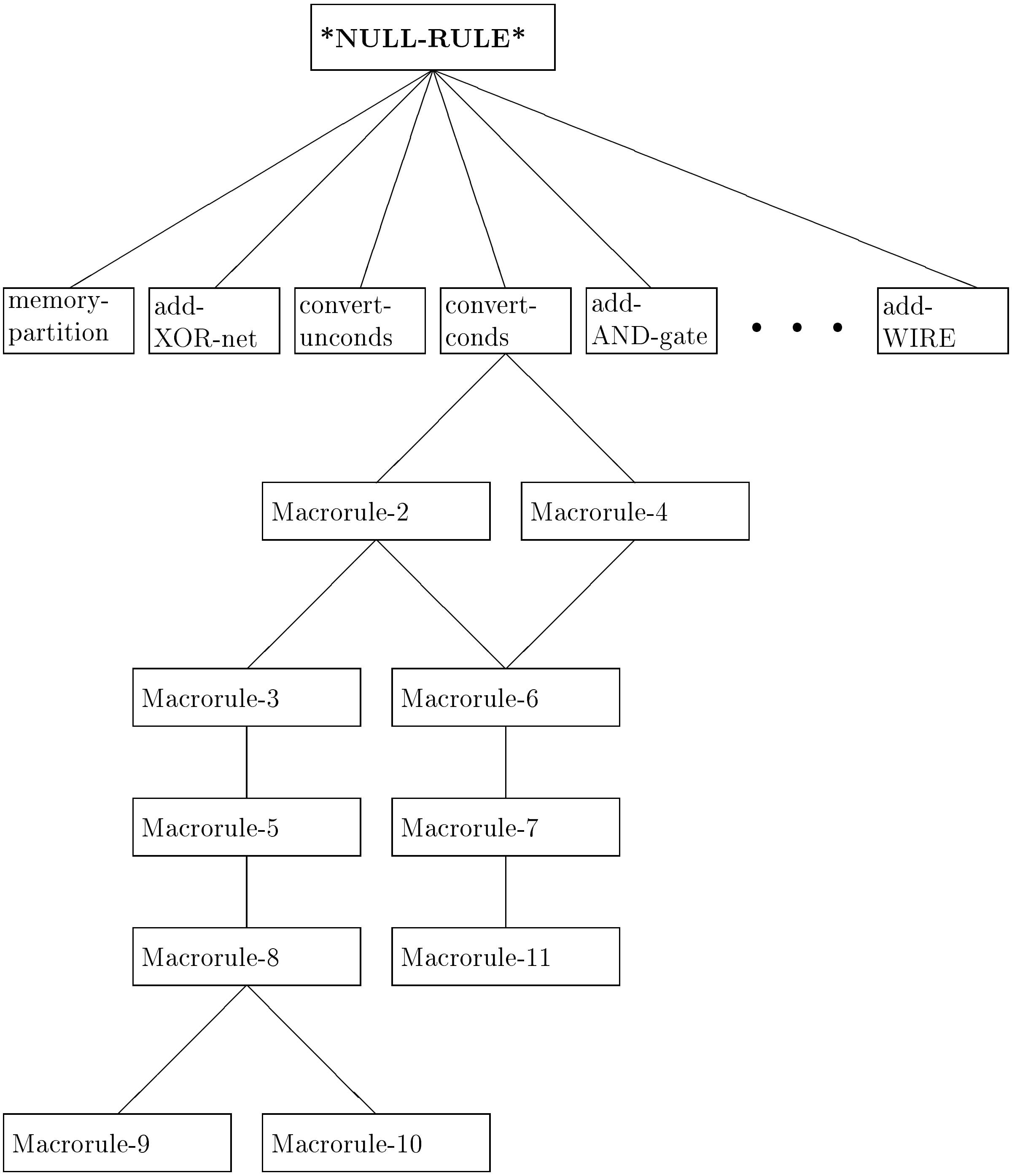


Figure 12: Partial order of forward rules in Argo-V after learning from the CAM-cell example

CIRCUIT-4 is an instance of **ENTITY**

Type:	*ABSTRACT-MODULE*
Predecessor:	NONE
Interface:	*CIRCUIT4-INTERFACE*
Architecture:	*CIRCUIT4-ARCHITECTURE*

CIRCUIT4-INTERFACE is an instance of **INTERFACE-BODY**

Input-Port:	*COMPARE, PHI1, PHI2, LOAD, DATA-IN*
Output-Port:	*MATCH*

CIRCUIT4-ARCHITECTURE is an instance of **ARCHITECTURAL-BODY**

```
Signal-Declaration:   PHI1-LOAD, STATE
Signal-Assignment:    (MATCH (PASSED-LOW 33NS
                                  (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))
Component:            no known values
```

Figure 13: Behavioral specification for an analogous design problem

just one of the calculated macrorules, Macrorule-9, 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 applies the knowledge it has learned to the other circuit design problems, resulting in better quality designs for them. The improvements, shown as *Design Quality* in Table 1, are substantial.

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. Even for the simple examples considered here, this would require a prohibitively long time (billions of years). However, the use of macrorules reduces this time by estimated factors[4] ranging from 24,000 to 3×10^{19}. This behavior of a system for VLSI design, and the possibility of exhaustively exploring such a huge space, are also unrealistic, but they emphasize the large size of a typical design space and the importance of finding ways to reduce the size of this space. Macrorules and their abstractions provide just such

[4]The estimates are based on using the average size of the conflict set as the branching factor of a search tree whose depth is the number of rules fired.

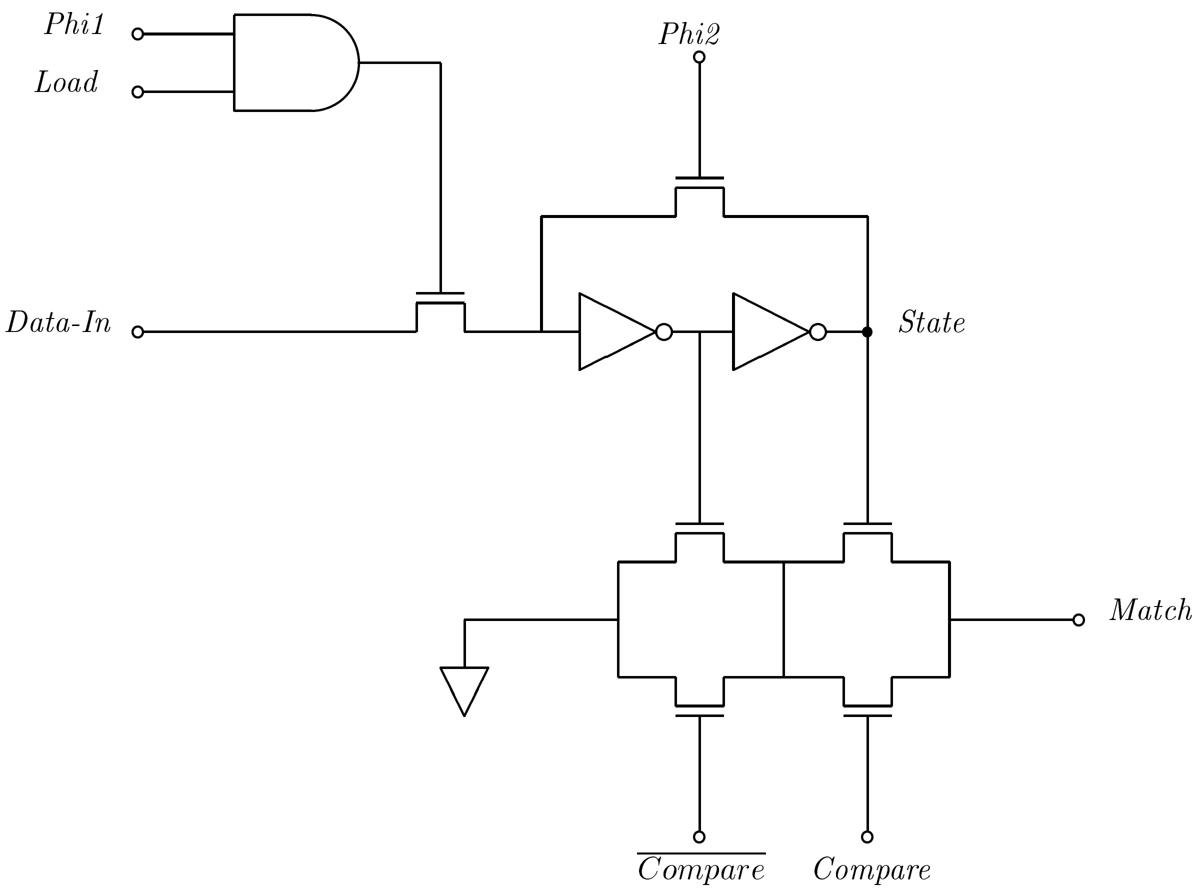
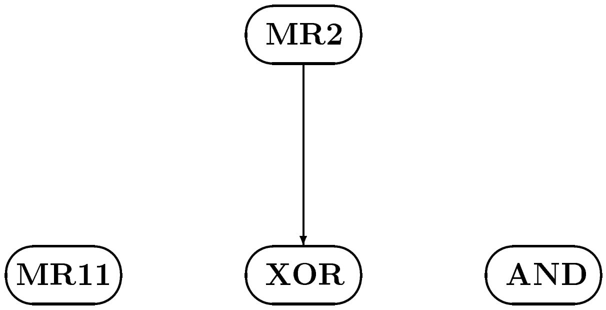


Figure 14: Schematic diagram for the analogous design problem



MR2: Macrorule-2

MR11: Macrorule-11

XOR: Instantiate an exclusive-OR gate

AND: Instantiate an AND gate

Figure 15: Rule-dependency graph for the analogous design problem

a capability.

Macrorules, however, are a supplement to, not a replacement for, the initial rules in an application system. 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 the system to run more slowly because it has more possibilities to consider at each problem-solving step (*i.e.*, it attempts to satisfy the antecedents of more rules) [26]. The slowdown has been apparent in other examples on which Argo-V was tested. In particular, one circuit was unable to use any of the macrorules learned from designing the CAM-cell. The time needed to find a design increased by 50% because the presence of macrorules increased the size of the rule base by 20%. 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.

6 Conclusions and Future Research Directions

The work reported here is based on developing the fundamental methodology for a system, Argo, that reasons and learns by analogy for solving problems 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 formal definition of an abstraction relation for partially ordering plans.

If design is viewed as state-space problem solving, then the knowledge in any knowledge-

CIRCUIT2-ARCHITECTURE is an instance of **ARCHITECTURAL-BODY**

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

CIRCUIT3-ARCHITECTURE is an instance of **ARCHITECTURAL-BODY**

```
    Signal-Assignment:    (MATCH (HI-Z12NS
                                     (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))
```

CIRCUIT5-ARCHITECTURE is an instance of **ARCHITECTURAL-BODY**

```
    Signal-Assignment:    (MATCH (PASSED-LOW 33NS
                                     (BIT-AND (BIT-EQUAL ENABLE LOW)
                                              (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))
```

CIRCUIT6-ARCHITECTURE is an instance of **ARCHITECTURAL-BODY**

```
    Signal-Assignment:    (MATCH (PASSED-HIGH 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))
```

CIRCUIT7-ARCHITECTURE is an instance of **ARCHITECTURAL-BODY**

```
    Signal-Assignment:    (OUTPUT (PASSED-LOW 33NS
                                      (BIT-AND (BIT-EQUAL INPUT1 HIGH)
                                               (BIT-NOTEQUAL INPUT2 INPUT3)))
                                  (HI-Z 0NS)),
```

Figure 16: Partial behavioral specifications for analogous design examples

Table 1: Effects of Learning on VLSI Design

Design	Before Learning/After Learning		
	Time (seconds)	Rules Fired	Design Quality (transistors)
CAM-cell	73.7/32.5	19/3	30/20
Circuit 2	73.6/57.8	19/6	30/20
Circuit 3	83.3/34.9	19/3	30/20
Circuit 4	40.5/32.7	13/4	25/19
Circuit 5	74.8/42.6	18/7	32/15
Circuit 6	81.3/29.4	19/3	30/20
Circuit 7	25.0/ 8.7	10/1	16/9

Note: Timings were made on a Symbolics 3600.

based system for design can be categorized into three fundamental types: design knowledge, control knowledge, and patching knowledge. Design knowledge is used to refine a particular state of the problem-solving process by means of transformations, decompositions, and instantiations. Patching knowledge is used to move to a new state, when refinements to the current state are evaluated as being unacceptable. Control knowledge is used to 1) evaluate a given state and determine whether to halt, backtrack to a previous state, refine the current state, or patch the current state, and 2) choose among possible alternatives for refining or patching the current state. 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. Note that as macrorules get more abstract, the number of details that must be filled in to solve a given problem increases. Hence, given a specification for a new design, Argo attempts to find and apply the least abstract macrorule that is appropriate. Using macrorules in this manner, Argo drastically reduces the amount of automatic search required for new design problems while still producing a correct design.

In order to provide an environment for developing and evaluating Argo, a knowledge-based system has been implemented for designing VLSI digital circuits. Argo-V refines VHDL behavioral specifications into structural modules by building hierarchical design trees. This refinement is accomplished by executing plans composed of transformation, instantiation, and decomposition rules. With use, the system accumulates design knowledge resulting in an increase of both its performance and its ability to synthesize digital circuits. At present, Argo-V has rules for designing circuits comprised of elementary digital components, including transistors, logic gates, and memory cells. A previous version of the system has designed switch-level circuits of up to several hundred transistors [1].

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 (explanations) 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).

The overall goal of our work has been to develop techniques for reasoning and learning by analogy and incorporate these techniques into a knowledge-based system for design. To overcome the limitations cited above, alternative procedures for formulating plan abstractions and constructing plan hierarchies are being investigated. We are conducting larger experiments in order to assess Argo's potential for continuous learning. 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 [18].

Acknowledgements

We would like to thank Tom M. Mitchell for supplying many of the ideas and insights upon which this research is based. Michael A. Gray, Shiuh-li Liuh, and Charles J. Petrie, Jr. provided numerous helpful suggestions during the course of our work. We would also like to thank Jack Mostow for extensively commenting on an earlier draft of this paper and helping us relate Argo to his own work on Bogart.

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] W. W. Bledsoe, "The Use of Analogy in Automatic Proof Discovery (Preliminary Report)," MCC Technical Report No. AI-158-86, Microelectronics and Computer Technology Corporation, Austin, TX, May 1986.

[3] B. Brock, S. Cooper, and W. Pierce, "Some Experiments with Analogy in Proof Discovery (Preliminary Report)," MCC Technical Report No. AI-347-86, Microelectronics and Computer Technology Corporation, Austin, TX, October 1986.

[4] 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.

[5] H. Brown, C. Tong, and G. Foyster, "Palladio: An Exploratory Environment for Circuit Design," *Computer*, vol. 16, no. 12, December 1983, pp. 41-56.

[6] M. H. Burstein, "A Model of Learning by Incremental Analogical Reasoning and Debugging," *Proceedings AAAI-83*, Washington, D.C., August 1983, pp. 45-48.

[7] 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.

[8] 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.

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

[10] J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, vol. 12, 1979, pp. 231-272.

[11] R. E. Fikes and N. J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, vol. 2, 1971, pp. 189-208.

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

[13] C. L. Forgy, *OPS5 User's Manual*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, July 1981.

[14] P. E. Friedland and Y. Iwasaki, "The Concept and Implementation of Skeletal Plans," *Journal of Automated Reasoning*, vol. 1, 1985, pp. 161-208.

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

[16] M. L. Gick and K. J. Holyoak, "Schema Induction and Analogical Transfer," *Cognitive Psychology*, vol. 15, 1983, pp. 1-38.

[17] M. L. Gick and K. J. Holyoak, "Analogical Problem Solving," *Cognitive Psychology*, vol. 12, 1980, pp. 306-355.

[18] M. A. Gray, "Implementing an Intelligent Design Machine in a TMS-Based Inferencing System," MCC Technical Report No. AI-045-87, Microelectronics and Computer Technology Corporation, Austin, TX, January 1987.

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

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

[21] R. E. Kling, *Reasoning by Analogy with Application to Heuristic Problem-Solving: A Case Study*, Ph.D. Dissertation, Department of Computer Science, Stanford University, Palo Alto, CA, 1971.

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

[23] 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.

[24] 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.

[25] S. Mahadevan, "Verification-Based Learning: A Generalization Strategy for Inferring Problem-Reduction Methods," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, Los Angeles, CA, August 1985, pp. 616-623.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

[31] S. Mittal, C. L. Dym, and M. Mojaria, "PRIDE: An Expert System for the Design of Paper Handling Systems," *Computer*, vol. 19, no. 7, July 1986.

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

[33] N. J. Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Co., Palo Alto, CA, 1980.

[34] 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, (also available as MCC Technical Report no. AI-352-86, Microelectronics and Computer Technology Corporation, Austin, TX).

[35] D. E. Rumelhart and A. A. Abrahamson, "A Model for Analogical Reasoning," *Cognitive Psychology*, vol. 5, 1973, pp. 1-28.

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

[37] D. R. Smith, "Top-Down Synthesis of Divide-and-Conquer Algorithms," *Artificial Intelligence*, vol. 27, no. 1, September 1985, pp. 43-96.

[38] 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.

[39] C. Tong, "A Framework for Circuit Design," *Proceedings of IEEE Compcon*, San Francisco, CA, Spring 1984.

[40] "VHDL Language Reference Manual, Version 7.2," Technical Report IR-MD-045-2, Intermetrics, Inc., Bethesda, MD, August 1985.

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

[42] R. Waldinger, "Achieving Several Goals Simultaneously," in E. Eliork and D. Mitchie, eds., *Machine Intelligence 8: Machine Representations of Knowledge*, Ellis Horwood, Chichester, England, 1979, pp. 94-136.

[43] P. H. Winston, "Learning New Principles from Precedents and Exercises," *Artificial Intelligence*, vol. 19, no. 3, November 1982, pp. 321-350.

[44] 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.