FORLOG: A Logic-based Architecture for Design

Thomas G. Dietterich
Department of Computer Science

David G. Ullman, P.E.
Department of Mechanical Engineering
Oregon State University
Corvallis, OR 97331

Abstract

It is difficult to build intelligent computer-aided design (ICAD) programs using available expert system shells and AI programming languages. To build ICAD programs, tools are needed that support (a) generative search of design spaces, (b) deep search of design spaces to evaluate alternative designs, (c) simultaneous exploration of alternative designs to compare designs, (d) constraint posting and propagation, (e) knowledge-based control of inference, and (f) the representation of complex mechanical and electronic devices. Existing shells and programming languages either do not support these activities or provide only ad hoc and inefficient supporting mechanisms. We have constructed a logic programming system called FORLOG (FORward-chaining LOGic Programming) that provides well-integrated support for all of these activities. This paper presents the architecture of FORLOG and provides some simple examples of how FORLOG can be applied to constructing ICAD systems.

1 Introduction

Intelligent computer-aided design (ICAD) tools provide support for the intellectual aspects of engineering design, rather than simply supporting the process of producing engineering drawings, which has been the primary application of existing CAD tools. The intellectual aspects of design include specification acquisition and validation, conceptual design, qualitative analysis, detailed design, and quantitative analysis. The goal of our research is to develop AI-based tools for all of these aspects of design.

To attain that goal, we are pursuing a two-pronged research strategy. The first line of research has as its goal the identification of opportunities and requirements for ICAD tools in mechanical design. The second line of research seeks to develop software tools that in turn support the construction of ICAD systems. In this paper, we will focus on the second research effort, in which we have developed the FORLOG logic programming system. First, however, we will give an overview of the first line of research. This overview will identify six basic requirements for software tools for ICAD. Following the overview, we will describe the FORLOG system and show how it meets these six basic requirements.

2 Empirical Research on the Design Process

To identify opportunities and requirements for intelligent CAD systems, we are conducting an empirical study of how mechanical engineers think as they solve design problems. We are applying the method of *protocol analysis* (Ericsson & Simon, 1985), which has been developed and refined by cognitive psychologists. We have carefully developed two mechanical design problems, each of which requires roughly ten hours to solve. These problems have been presented to expert mechanical designers, who were instructed to "think out loud" as they solved them. Their verbal protocols (and also their drawings, body movements, etc.) were recorded on video tape, which is currently being transcribed and analyzed. The result of this analysis will be a detailed account of the various phases of the design process including (a) the state of the current design or designs at each point in the process, (b) the design decisions made at each point, (c) the domain knowledge that was applied to make those decisions, and (d) the overall strategies that guided the design process.

The two problems that we have developed were selected with several goals in mind. First, we wanted to observe all phases of the design process. Hence, we provided our subjects with incomplete, high level specifications, and we followed their progress until they had produced detailed working drawings for at least some parts of the final design. Second, we wanted to explore the difference between *product* designs and *one-off* designs, since it is clear to most engineers that designing for a product is different than designing a one-of-a-kind device. We achieved this goal by developing one product-oriented problem and one "one-off" problem. Third, we wanted to explore the relationship between the engineer's knowledge and skills and the requirements of the problem. To achieve this, we have taken protocols of graduate students as well as experienced mechanical designers. We have also selected problems for which our expert subjects can be expected to have a high degree of expertise and experience.

Here is a brief description of the two design problems that we have developed. The first problem, which we call the "flipper-dipper," involves designing a machine to grasp a thin aluminum plate and position it onto the surface of a water bath. The machine must dip both sides of the plate, one at a time. This problem requires simple knowledge of kinematics and some control technology, such as pneumatics or small electro-mechanical transducers. It is a "one-off" problem, since only three of these machines are to be constructed. This problem is based on a consulting contract completed

a number of years ago by the second author.

The second problem is product-oriented. It involves designing the battery contacts for the batteries of a portable computer. The contacts must be designed so that a robot can easily install the contacts in the computer during assembly. This problem requires knowledge of metal springs, molded plastic materials, and robot assembly constraints. Over the expected lifetime of the product, approximately 1.8 million units will be produced. This problem was developed with the cooperation of a major computer manufacturer.

At the time of writing, we are still analyzing the protocols. Our preliminary analysis of the data has already demonstrated several requirements for any ICAD system (see Ullman & Dietterich, 1986; Ullman, Stauffer, & Dietterich, forthcoming). We now discuss these requirements.

First, *an ICAD system must be capable of conducting a* generative *search of a large space of possible designs.* A generative search of a design space operates by *constructing* possible designs from subcomponents.[1] This is quite different from most current expert systems, which only consider a fixed, pre-enumerated set of possible solutions. Clancey (1985) has called these more traditional systems "heuristic classification systems," since their task is to classify a given situation or object into one or more classes. Systems that support heuristic classification, such as EMYCIN and its descendants (e.g., Teknowledge's S.1 and M.1 products, most personal computer-based expert system tools, etc.), are very difficult to apply to design tasks.

Second, *an ICAD system must be capable of conducting moderately deep searches* in order to evaluate alternative designs. Given only the overall "design concept" for a design, it is usually difficult to assess its cost or feasibility. Proper evaluation must be postponed until the "concept" is further developed and refined. Hence, an ICAD system must provide facilities for conducting a (moderately) deep search to explore the consequences of design decisions.

Existing design and configuration expert systems (e.g., R1 (McDermott, 1982), PRIDE (Mittal, Dym, & Morjaria, 1986), HI-RISE (Maher, 1984)) provide basic facilities for satisfying these first two requirements. These systems perform generative searches of fairly large design spaces, and they are capable of conducting at least some further search in order to evaluate particular design decisions.

Third, *an ICAD system must be capable of reasoning about several alternative designs simultaneously.* Design usually involves optimization—the designer seeks the most efficient and least expensive solution to the design problem. Sometimes, design problems can be converted into numerical optimization problems, and strong mathematical methods can be applied. However, most design problems also involve "structural" design decisions for which mathematical techniques are lacking. In these cases, engineers typically investigate a few promising alternative designs, compare them with one another, and choose the best one. If ICAD systems are to assist engineers in this process, they must also be capable of pursuing more than one alternative design at a time. Only a few expert systems shells (e.g., ART (Trademark of Inference Corporation) and KEE (Trademark of Intellicorp)) are able to do this at present.

The fourth important requirement for ICAD systems is that *they must be capable of performing constraint propagation, and more generally, forward reasoning.* Design often involves the posting and propagation of constraints (see Stefik, 1981; Stallman & Sussman, 1977). When a design decision is made concerning one component of a device, this decision may have ramifications for other device components, some of which may be quite distantly related to the component in question. Constraint propagation is the process of inferring these ramifications by tracing them through sequences of immediately adjacent components. More generally, constraint propagation is a kind of

[1] This construction process can either work "bottom-up," constructing larger components out of subcomponents, or "top-down," decomposing large components into subcomponents.

forward reasoning to infer the consequences of a particular design decision. Any tool for constructing ICAD systems must provide this kind of reasoning.

The fifth requirement is that *ICAD systems must include a capability for knowledge-based control of inference.* For heuristic classification systems, control of inference is not a critical issue because the space of possible inferences (i.e., the set of given classes that require investigation) is fixed and relatively small. Indeed, most expert systems shells conduct exhaustive searches! However, in mechanical design, the space of possible devices that could satisfy the specifications is infinite. It is impossible to conduct an exhaustive search of this space. Instead, the search must be carefully guided by domain knowledge so that only feasible and cost-effective designs are considered.

Finally, the sixth requirement for ICAD systems is that they must be able to *represent complex mechanical and electronic devices.* Most expert systems shells are limited to representing objects (patients, situations, etc.) in terms of a fixed set of variables. These variables themselves are usually restricted to have only a small set of numeric or symbolic values. Such systems are incapable of representing devices whose number of components and interconnections are variable and unknown. Even structured object systems of the type provided by KEE and LOOPS (Stefik, Bobrow, Mittal & Conway, 1983) make it inconvenient to describe complex interconnected devices. ICAD systems require the full expressive power of first-order logic to represent not only the final design, but also the intermediate states of the design in which the device is only partially specified.

This concludes our discussion of the important requirements for ICAD systems. The remainder of the paper is organized as follows. First, we describe the paradigm of logic programming and evaluate its suitability for building ICAD systems in light of these requirements. Second, we describe the FORLOG system, and show that it overcomes the disadvantages of previous logic programming approaches. Finally, we conclude with a small example design problem that has been solved using FORLOG.

3 Logic Programming and Design

Logic programming is a relatively new programming paradigm in which the program is written as a set of logical assertions (i.e., facts and rules), and program execution consists of deriving logical consequences from these assertions. Hence, the "code" in a logic program has two simultaneous interpretations or meanings. First, it can be interpreted as a set of logical facts that describe objects and their interrelations. Second, it can be interpreted as a set of instructions to a computer to cause it to carry out a sequence of logical inferences. The chief advantage of logic programming is that the correctness of the second interpretation can be proved by considering the first interpretation.

The most popular logic programming language is Prolog (Clocksin & Mellish, 1984; Lloyd, 1984), although there are several other logic programming systems (e.g., MRS (Russell, 1985), DUCK (McDermott, 1985), and EQLOG (Goguen & Meseguer, 1984)). All of these existing systems operate primarily by a form of reasoning called *backward chaining.* Briefly, in this form of reasoning, all computation occurs in response to a query. Suppose, for example, that the following facts have been given to Prolog (capital letters denote variables):

```
[1] metal(steel).
[2] metal(aluminum).
[3] metal(copper).

[4] strength(steel,10).
[5] strength(aluminum,5).
[6] strength(copper,3).
```

```
[7] conducts-electricity(M) :- metal(M).

[8] acceptable(X) :- conducts-electricity(X), strength(X,S), S>4.
```

The first three lines state that steel, aluminum, and copper are metals. The second three lines provide the relative strengths of these materials.

Line 7 gives the rule that all metals conduct electricity (the symbol `:-` should be read as "if" and the comma should be read as "and"). And the last line states that a material is acceptable if it conducts electricity and has a strength larger than 4.

To perform a computation in Prolog, the user issues a query, such as `acceptable(X)`. This is interpreted to mean, "find all values for the variable `X` such that they are `acceptable`." Prolog answers this query by reasoning as follows:

> To find something (`X`) that is acceptable, according to my last rule I must find something that conducts electricity and has a strength greater than 4. To find things that conduct electricity, according to line 7 I must find things that are metal. According to the first line of the program, steel is a metal, hence, steel conducts electricity, hence, I must determine if steel has a strength greater than 4. To determine the strength of steel, I look at line 4. It says that the strength of steel is 10. Finally, let me check whether 10 is greater than 4. It is. Hence, a possible value for `X` is steel.

This is called backward chaining, because Prolog works backward from the given goal (`acceptable(X)`), chaining through intermediate rules (such as `conducts-electricity(M) :- metal(M)`), until it finds simple facts that it can use (such as `metal(steel)`). Notice that the answer returned by Prolog is simply a value (called a "binding") for the variable `X` that makes the initial goal true.

The value returned as an answer can be much more complex than a simple constant such as `steel`. Instead, it can be a general expression called a *term*. Consider, for example, the term `car(ford, e212, red)`. This term might represent a Ford automobile with engine-type `e212`, whose body is painted red. Terms can have substructure. Hence, the term `car(ford, engine(v6,212,ohc), red)` might describe the same car, but instead of simply giving the engine type, it would use a term (recursively) to describe the engine as a 212 cu. in. V6 with an overhead-camshaft. In general, terms can be employed to describe any kind of hierarchical or tree-structured objects.

Now that we have described the basic capabilities of Prolog, let us evaluate the extent to which it meets the six requirements for ICAD systems. The first two requirements present no problems. Prolog is capable of conducting generative searches and it is capable of conducting deep searches. However, the remaining four requirements are not well-satisfied by Prolog.

The third requirement—the ability to consider alternative designs simultaneously—is not met by Prolog because Prolog only considers one alternative design at a time. Prolog conducts its search for solutions in a purely depth-first fashion, constructing one entire solution and returning it before going on and constructing the next solution.

The fourth requirement—that ICAD systems perform constraint propagation—is also difficult to accomplish in Prolog. This is because Prolog operates by backward-chaining, and constraint propagation is a kind of forward-chaining.

Prolog also fails to satisfy the fifth requirement, because it has no capability for knowledge-based control of inference. Its inference strategy is fixed: depth-first backward chaining.

Finally, and perhaps most importantly, Prolog fails the sixth requirement, because it is unable to represent complex mechanical and eletronic devices. This last point may surprise the reader, since

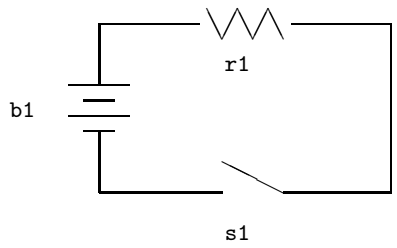

Figure 1: A circuit that cannot be represented as a term

we have just described how terms (and recursive subterms) can be used to capture the hierarchical structure of devices. The problem is that Prolog can *only* represent devices as terms if the devices *are* hierarchical. Unfortunately, most electronic and mechanical devices are not hierarchical (Sussman & Steele, 1980). For example, suppose we tried to represent the circuit in Figure 1 as a term. We could say something like `battery(b1, resistor(r1), switch(s1))` to state that the battery is connected to both the resistor and the switch, but there is no way to talk about the connection *between* the resistor and the switch, because that connection violates the hierarchy.

We can see from this brief review that the basic facilities provided by Prolog do not supply all of the support needed for ICAD systems. This does not mean, of course, that Prolog could not be used to develop such systems. It only means that before ICAD systems could be developed, these additional facilities would need to be implemented within Prolog.

4 Logic Programming in FORLOG

The FORLOG logic programming language was designed to address these shortcomings of Prolog. There are four fundamental differences between FORLOG and Prolog:

- Answers are represented as assertions. Instead of terms, answers can be represented as collections of assertions. This permits FORLOG to represent any kind of complex device or system. For example, by using assertions, FORLOG can describe the circuit in Figure 1 as follows: `wire(b1,r1) & wire(r1,s1) & wire(s1,b1)`. This says there is a wire connecting `b1` with `r1`, a wire from `r1` to `s1`, and a wire from `s1` back to the battery `b1`.
- Forward-chaining reasoning. All reasoning in FORLOG is done by forward chaining instead of backward chaining, thus permitting FORLOG to perform constraint propagation.
- Order-independent search. In FORLOG, rules can be applied in *any* order. The implementation provides complete flexibility to explore multiple alternatives in parallel.
- Knowledge-based control of search. Control rules can decide in what order to explore the search space.

In the remainder of this section, we will describe each of these four differences in turn.

4.1 Representing designs as assertions

To show how FORLOG represents designs as collections of assertions, let us begin with a circuit design example. Suppose we want to write down rules for designing simple electrical circuits. The

following FORLOG rules describe how to design a direct-current voltage source for a circuit (the `->` should be read as "implies", and comma should be read as "and"):

```
[1] dcsource(S,V) ->
        [V <= 12, N=ceiling(V/1.5), series(pos(S),neg(S),N)] or
        [V > 12, actodc(pos(S),neg(S),V)].

[2] series(P1,P2,N) ->
        battery(B,1.5), wire(P1,pos(B)),
        [N=1, wire(neg(B),P2)] or
        [N>1, series(neg(B),P2,N-1)].
```

The first rule says that if `S` is a DC voltage source of `V` volts, then there are two ways of implementing it. If `V` is 12 volts or less, then `N` 1.5 volt batteries should be connected in series from the positive output of `S` to the negative output of `S`. On the other hand, if `V` is greater than 12 volts, the voltage source should be constructed by converting an AC power source to direct current.

The second rule says that to connect `N` batteries in series from point `P1` to point `P2`, take a 1.5 volt battery, `B`, and connect its positive terminal to `P1`. Then if `N` is greater than 1, wire the negative terminal to a string of `N-1` batteries connected in series. If `N` is equal to 1, then we are done—simply connect the negative terminal of the battery to `P2`.

To design a voltage source in FORLOG, we *assert* that one exists, and FORLOG computes the consequences of that assertion. For example, to design a 4.5 volt DC source (named `s3`), we would assert the following:

```
dcsource(s3,4.5).
```

This causes the first rule to trigger, because its left-hand-side is matched. FORLOG considers the two separate possibilities on the right-hand-side of the rule (i.e., the two branches joined by `or`). The second branch is not satisfied, because `V` is not greater than 12. Hence, the first branch is the only possibility. FORLOG computes `N` to be 3, and asserts

```
series(pos(s3),neg(s3),3).
```

This causes the second rule to trigger, so FORLOG begins to execute the right-hand-side of the rule. It invents names for all new variables appearing on the right-hand-side, in this case, the battery `b23`. FORLOG then asserts

```
battery(b23, 1.5).
wire(pos(s3),pos(b23)).
```

Now, FORLOG comes to two branches joined by `or`. The first branch is discarded, because `N` is not 1. Hence, the `series` assertion in the second branch is made:

```
series(neg(b23),neg(s3),2).
```

This new `series` assertion is a *recursive* case. It calls for placing 2 batteries in series. It causes a similar sequence of rule triggerings and assertions, and the result is the following:

```
battery(b24, 1.5).
wire(neg(b23),pos(b24)).
series(neg(b24),neg(s3),1).
```

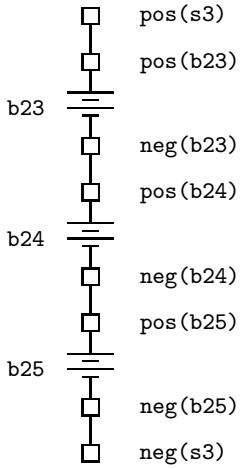

Figure 2: A simple DC voltage source

A second battery, `b24` has been created, and connected to the negative terminal of `b23`. The last `series` assertion will again trigger rule `[2]`. This time, however, the recursion will halt, because the second branch is ruled out (`N` is now 1). FORLOG makes the following assertions:

```
battery(b25, 1.5).
wire(neg(b24),pos(b25)).
wire(neg(b25),neg(s3)).
```

Figure 2 shows the resulting design.

This example demonstrates several important points concerning FORLOG.

First, in FORLOG, because it is a forward-chaining system, all reasoning takes place in response to an assertion. We tell FORLOG to design a DC voltage source by *asserting* that one exists, and letting FORLOG determine the consequences of that assertion. If we were to try to write a similar set of rules in Prolog, the implication arrows (i.e., the `->`) would point in the opposite direction. Such rules could not be used to *design* a DC voltage source, but they could be used to *recognize* one. We could, for example, enter into Prolog a set of assertions, such as `wire(pos(b22),neg(b23))` or `battery(b23,1.5)` and then *ask* whether they constituted a 4.5 volt DC source.

Second, unlike Prolog, FORLOG makes no distinction between goals and assertions. A goal (such as `dcsource(s3,4.5)`) is simply an assertion that causes implementation rules to fire. This handling of goals makes it possible to employ deKleer's assumption-based truth maintenance system (deKleer, 1986a, b, c) to manage FORLOG's search processes. It is critical to permitting FORLOG to pursue alternative designs in parallel.

Third, each of the two rules above illustrate how FORLOG handles design alternatives. The alternatives are placed on the right-hand-side of the rule, separated by `or`. When FORLOG is faced with a choice, it attempts to rule out each alternative immediately. If this cannot be done, it then investigates each alternative in parallel (examples of this will be shown below).

Finally, the example shows how FORLOG makes it natural to represent designs as collections of assertions instead of terms. Each assertion made during the execution of the program is added to the FORLOG database. Hence, the database provides a record of the design decisions that have been made so far.

4.2 Constraint propagation in FORLOG

The second major difference between FORLOG and Prolog is the support that FORLOG provides for constraint propagation. Suppose, for example, that we are inserting three batteries into a battery holder. We are told that the batteries belong in series, but we don't know whether the positive end of the series is supposed to go in the top or the bottom of the holder. The following rules represent this situation:

```
[3] series(pos(s1),neg(s1),3).
[4] series(X,Y,N) ->
        location(X,L1),
        location(Y,L2),
        opposite-ends(L1,L2).

[5] opposite-ends(L1,L2) ->
        [L1=top(H), L2=bottom(H)] or
        [L1=bottom(H), L2=top(H)].

[6] not(top(X)=bottom(X)).
```

Statement 3 simply asserts that `s1` is a series of three batteries, whose positive terminal is `pos(s1)` and whose negative terminal is `neg(s1)`. Statement 4 says that any series with positive terminal `X` and negative terminal `Y` must be positioned in a battery holder so that its two terminals are at opposite ends. Statement 5 defines what it means for two locations to be at opposite ends of the battery holder. Finally, statement 6 says that the top of the battery holder is never the same location as the bottom of the battery holder.

If FORLOG is permitted to run at this point, rule 4 will trigger and make up names (e.g., `l1` and `l2`) for the locations of the two terminals. It will then assert that these two terminals are at opposite ends of the battery holder:

```
[7] location(pos(s1),l1).
[8] location(neg(s1),l2).
[9] opposite-ends(l1,l2).
```

Suppose someone interrupts FORLOG at this point and tells it that `l1` is actually the bottom of the battery holder. This can be asserted as `l1=bottom(h1)` (where `h1` is the name of this particular battery holder). This assertion, combined with the assertion that `opposite-ends(l1,l2)` causes the final rule to trigger. FORLOG considers the two alternatives, but it rejects the first branch, because `l1` is known not to be the top of the battery holder. Hence, the second branch is chosen, and FORLOG concludes that `l2=top(h1)`.

This is an example of constraint propagation, because it shows how a constraint affecting one part of the design—the location of `l1`—is propagated to determine some other part of the design—the location of `l2`. In larger designs, this propagation can be very valuable in computing the consequences of making a particular design decision.

4.3 Investigating design alternatives in parallel

Suppose in the previous example, that FORLOG had not been told the correct location for `l1`. What would have happened then? Rule 5 would still have triggered, and FORLOG would face the two alternatives

```
[l1=top(h1), l2=bottom(h1)] or [l1=bottom(h1), l2=top(h1)].
```

When this situation arises, FORLOG investigates the two alternatives in parallel.

One approach to investigating alternatives is simply to mix all of the alternatives together in the same database. Hence, FORLOG could simply make the four separate assertions

```
[10] l1=top(h1).
[11] l2=bottom(h1).
[12] l1=bottom(h1).
[13] l2=top(h1).
```

Such an approach will not work at all, because FORLOG will infer by the rules of equality and statements 10 and 12 that

```
[14] top(h1)=bottom(h1).
```

Statement 6 can then be applied, and it declares this to be a contradiction. FORLOG represents this as

```
[15] CONTRADICTION.
```

This result should not be surprising. After all, the whole point of alternative design decisions is that they disagree about how the design is to be accomplished. If FORLOG were to mix the two alternatives together, that would be equivalent to attempting to do both of them at the same time.

To solve this problem, FORLOG incorporates deKleer's assumption-based truth maintenance system (ATMS) (see deKleer, 1986a, b, c). What the ATMS does is to attach *labels* to each fact and rule. These labels keep track of which facts belong to each alternative design. Each label is a *set* of atomic symbols, and each atomic symbol stands for a decision to believe a particular fact. In his papers, deKleer calls these symbols *assumptions* or *choices*. In this case, FORLOG might call the first design choice (where `l1=top(h1)`) `D1` and the second design choice `D2`. Suppose that the label `B` is given to all basic facts. The resulting database looks like this:

```
[6] not(top(X)=bottom(X)).          {B}
[10] l1=top(h1).                    {D1}
[11] l2=bottom(h1).                 {D1}
[12] l1=bottom(h1).                 {D2}
[13] l2=top(h1).                    {D2}
```

Intuitively, these labels say the following. If you believe all of the basic facts (symbolized by "choice" B), then you must also believe statement 6. If you believe choice D1, then you must believe statements 10 and 11. If you believe choice D2, then you must believe statements 12 and 13.

Now, when FORLOG applies the rules of equality, it will infer

```
[14] top(h1)=bottom(h1)             {D1,D2}
```

This is exactly the same reasoning that it performed before, but this time, the fact has a label (`{D1,D2}`) attached to it. The label on any derived fact is computed by taking the *set union* of the labels attached to all of the antecedent facts. For example, fact 14 was inferred from facts 10 and 12. These are labelled with `{D1}` and `{D2}`, hence fact 14 gets the label `{D1,D2}`. In English, this says "If you believe choices D1 and D2, then you must also believe fact 14."

Of course, FORLOG will also find the same contradiction as before,

```
[15] CONTRADICTION                    {B,D1,D2}
```

but now the contradiction also has a label attached to it. Line 15 can be interpreted as saying "You cannot believe choices B, D1, and D2 simultaneously." It is still ok to believe B and D1 (alone) or to believe B and D2 (alone), but you can't believe all three choices.[2] This is exactly the behavior we want. There are two separate designs, and FORLOG has simply discovered that the two designs can't be combined without deriving a contradiction.

Now that FORLOG has derived these additional facts, suppose we want to query the database to examine them. When we want to ask a question of the database, we must supply both the question and a set of choices. For example, if we ask `top(h1)=L` (i.e., what is located at the top of the battery holder?) FORLOG can't give a unique answer, because it might be `l1` or `l2` depending on which design we are talking about. Hence, we must also provide a set of choices when we ask the question: `top(h1)=L {B,D1}`, in which case FORLOG can answer `L=l1`.

In general, a set of choices C selects a set of facts in the database, namely those facts whose labels are subsets of C. For example, if C is the set {B}, then the set of facts contains only statement 6 and any statements that have the empty set as a label. If C is the set {B,D2}, then the corresponding set of facts is expanded to include statements 12 and 13. In effect, the ATMS says "tell me what choices you currently want to believe, and I'll give you a consistent view of the database." This database architecture makes it easy to switch among different sets of assumptions. It also makes it possible to share facts (such as statement 6) between otherwise mutually inconsistent views of the database.

This completes the discussion of the basic ATMS concepts. However, to understand the ATMS completely, it is important to note several additional aspects of the system.

First, contradictions require special handling. Statement 15 above is a special "fact"—the contradiction fact. Whenever `CONTRADICTION` is given a label, this label is immediately recorded in a separate database, called the *nogood database*. In this case, FORLOG records the statement `nogood{B,D1,D2}`. After the nogood is recorded, the ATMS scans through the fact database and deletes from the database all labels that are supersets of this new nogood label. In this specific case, the label {B,D1,D2} will be simply be deleted from fact 15.

Whenever a query (with an associated set C) is issued to FORLOG, it is checked to see if C is a superset of any recorded nogood. If so, C contains a contradiction, and there are no corresponding facts in the database. Hence, if C were the set of choices {B,D1,D2}, FORLOG would not produce any corresponding facts. Without this nogood check, a simple-minded approach would produce all of the facts in the database, because all of their labels are subsets of {B,D1,D2}!

A second important point to note is that there is an important distinction between a fact that has no label and a fact that has the empty set as a label. A fact with no label can never be believed, because it doesn't have a label that is a subset of the given set C of choices. After the {B,D1,D2} label is deleted from fact 15, it has no label, and it will never be provided as the answer to any query. This is quite different, however, from a fact that has the empty set {} as a label. Such facts are always believed, because this label is a subset of every possible choice set C. If we wanted to consider our basic facts to be absolutely true in every design, we could give them the empty label rather than the label {B}.

A third important point is that along with an attached label, each fact in the database contains a record of *how* the fact was inferred. These records are called *dependencies*. For example, since fact number 14 (`top(h1)=bottom(h1) {D1,D2}`) was derived from facts 10 and 12, it contains a

[2] It is even possible to believe D1 and D2, but to do this, you must give up believing that `top(h1)` and `bottom(h1)` are always different locations.

dependency record listing those two facts as the facts it depends upon. This is also true, of course, for the special `CONTRADICTION` fact. Its dependencies tell how all of the nogoods were discovered.

A fourth point is that it is possible for a fact to have more than one label. For example, we might be considering a third alternative design, called `D3`, in which `l1` is again located at the top of the battery holder, just as it was in design `D1`. We can represent this by giving an additional label to fact 10:

```
[10] l1=top(h1)                    {D1}{D3}
```

When we provide this new label to fact 10, the ATMS automatically updates the labels of all facts that depend on fact 10. In this case, the ATMS will add a second label to fact 14:

```
[14] top(h1)=bottom(h1)            {D1,D2}{D2,D3}
```

This label-propagation process is one of the key services that the ATMS provides. In effect, the ATMS has memorized (as a dependency record) every inference made by FORLOG. When new labels are added to the antecedents, the ATMS consults its memorized dependency information and propagates these labels to the appropriate consequents. This saves FORLOG the work of having to re-infer those consequences.

In addition to computing a new label for fact 14, the ATMS will also note that the set of choices {`B,D2,D3`} is contradictory. This is because the `CONTRADICTION` fact will receive this new label during the label updating process.

This completes our discussion of the ATMS and the methods that FORLOG employs to keep track of alternative designs. Now let us consider how FORLOG carries out the search process.

In Prolog, whenever the system encounters a choice, one of the alternatives is selected, and the remaining alternatives are placed on a backtrack list to be considered if the first alternative fails. This approach forces Prolog to consider only one alternative at a time. In FORLOG, when a choice is encountered, new choice names (e.g., D1, D2, ...) are invented to stand for each of the alternatives. FORLOG then pursues each alternative in parallel, adding the appropriate choice name to the label of each alternative. This is how new choice names are introduced into the ATMS.

An important goal in constructing FORLOG was to provide complete flexibility to the programmer over the order in which the design space is searched. To accomplish this, FORLOG employs a standard agenda-based architecture. Figure 3 shows the overall architecture of the FORLOG system. A program is given to FORLOG as a set of facts and rules and a body of control knowledge. Simple facts, such as `metal(steel)` are recorded in the fact database. Rules, such as `metal(X) -> conducts-electricity(X).` (i.e., "all metals conduct electricity"), are recorded in the fact database and also converted into rule demons. A rule demon is an active process that watches the fact database for facts that should trigger reasoning. For example, the rule demon for the rule "all metals conduct electricity" monitors the fact database for facts of the form `metal(X)`. Whenever it sees a new fact of this form, such as `metal(steel)`, it creates a new *reasoning job* that says "assert `conducts-electricity(steel)`" and places this job on the agenda.

The use of an agenda is critical for providing search flexibility. Rather than simply executing the reasoning jobs in the order that they are created, FORLOG applies user-supplied control rules to select the best reasoning job, and it is then executed. The result of executing a reasoning job is to add more facts to the fact database, which in turn may cause more demons to "fire" and create more jobs. Reasoning continues in this loop until the control knowledge dictates that a solution has been found.

Each reasoning job can be thought of as an assertion waiting to be made. Like an assertion, each reasoning job has an ATMS label and a list of justifications. This label and justification will

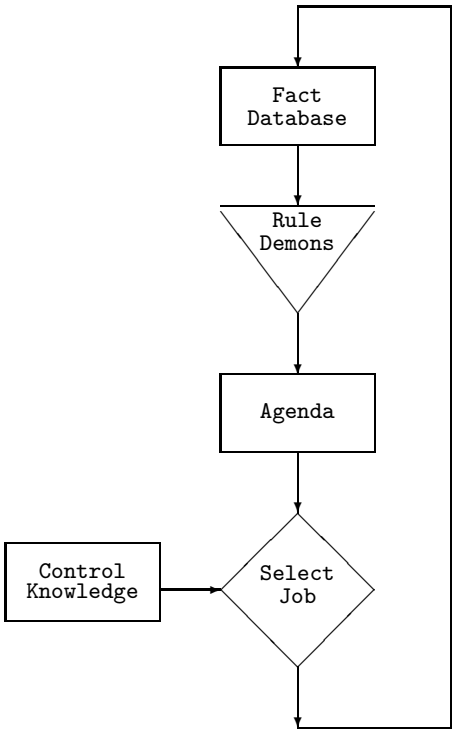

Figure 3: Overall architecture of FORLOG

be the label and justification that the assertion will receive when it is eventually inserted into the database. An important use for this information on the agenda is to assist the control rules in selecting the next job to execute.

4.4 Meta-level control of reasoning

FORLOG has a set of general, domain-independent deliberation criteria for selecting jobs from the agenda. We are also developing a method for allowing the user to specify domain-specific deliberation criteria, which will automatically be incorporated into the deliberation process. The general criteria are as follows.

First, never run a job that has no ATMS labels. Such jobs can arise when a contradiction is detected after the job has been placed on the agenda (but before it has been executed). These jobs, if they were to be run, would create assertions that would not be currently believed.

Second, give highest priority to jobs that are going to assert contradictions. The sooner a contradiction is detected, the less work is wasted exploring a contradictory set of design assumptions.

Third, give high priority to jobs that *may* assert contradictions. Some jobs contain conditional code that will be evaluated when the job is executed. If this conditional code returns "true," then the job will assert a contradiction. Such jobs are given high priority.

Finally, give low priority to jobs that will create new choices (i.e., new assumptions). If new assumptions are created indiscriminately, combinatorial explosion usually results, because too many combinations of choices must be explored.

In addition to these general criteria, the programmer can specify that certain sets of choices should be explored before others. This is the primary way that domain-specific knowledge can be used to control inference.

It is important to note that no jobs are ever removed from the agenda unless they are run. This is because a job, once it is created, will never be created by the rule demons again. Even jobs that currently have no ATMS labels cannot be deleted, because subsequent inference might give them new labels (via label updating). This general problem-solver organization is based on deKleer's "consumer architecture." In deKleer's terms, a reasoning job is called a consumer. In a sense, each reasoning job is created only once and executed only once. The results of its execution are cached in the fact database, where they are maintained by the ATMS routines.

4.5 Summary

This section has described the FORLOG logic programming system and demonstrated that it meets all six requirements for ICAD systems, including the four that Prolog does not satisfy. In particular, FORLOG makes it easy to (a) explore alternative designs simultaneously, (b) perform constraint propagation from one part of a design to another, (c) apply knowledge to control the reasoning process, and (d) represent complex mechanical and electronic devices.

The description in this section has been informal, and it has focused on the suitability of FORLOG for developing ICAD systems. A more theoretical description of FORLOG can be found in Dietterich, Corpron, and Flann (forthcoming), where a number of results are proved concerning FORLOG, its relationship to Prolog, and its facilities for programming with assertions instead of with terms. The main theorm is that a subset of FORLOG, called mini-FORLOG, is procedurally isomorphic to Prolog. In other words, it is possible to transform any Prolog program into an equivalent mini-FORLOG program that, when executed by FORLOG, will compute the same answers (via exactly analogous steps) as Prolog. Hence, FORLOG retains all of the advantages of Prolog, while overcoming many of Prolog's shortcomings.

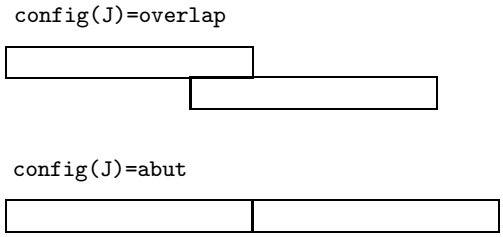

Figure 4: Joint design problem

5 FORLOG and Design

Now that we have described in detail the architecture of FORLOG, let us present an example of a simple design problem that has been solved using FORLOG. Suppose that we are designing a joint `J` between two sheets of material, `M1` and `M2`. We will consider two possible configurations for the joint: overlapping and abutting (see Figure 4). The materials can be either metal, plastic, or wood, and the bond between the materials can use either a weld, some bolts, or an adhesive. The following rules describe the problem and encode some expertise about joints, configurations, and bonding methods.

```
[A] joint(J,M1,M2) -> config(J)=overlap or config(J)=abut.
[B] joint(J,M1,M2) -> bond(J)=weld or bond(J)=adhesive or bond(J)=bolt.
[C] joint(J,M1,M2) -> material(M1), material(M2).
[D] material(M) -> type(M)=metal or type(M)=wood or type(M)=plastic.
[E] joint(J,M1,M2) -> [type(M1)=type(M2), same(J), type(J)=type(M1)] or
                      [not(type(M1)=type(M2)), different(J)].
[F] config(J)=abut -> not(bond(J)=bolt).
[G] different(J) -> not(bond(J)=weld).
[H] same(J), not(type(J)=metal) -> not(bond(J)=weld).
[I] same(J), type(J)=metal -> not(bond(J)=adhesive).
```

The predicate `joint(J,M1,M2)` says that there are two materials, `M1` and `M2`, that form a joint whose name is `J`. Rules A and B list the alternative configurations and bonding methods for the joint. Rule D describes the available materials (rule C causes rule D to be applied to each of the two materials in the joint). Rule E determines whether the materials on the two plates are the same, and it requires some explanation. If the two materials are identical, then the predicate `same(J)` is asserted. In addition, we say that the joint has a "type" as well. In other words, if both materials involved in a joint are the same (e.g., wood), then we say that the joint is a wood joint (`type(J)=wood`). If the two materials are not the same, the predicate `different(J)` is asserted.

Rules F, G, H, and I are typical of the kind of rules that encode expertise in this domain. For example, Rule F says it is impossible to bolt materials that abut. Rule G says it is impossible to weld two different materials. Rule H says that you can only weld metal-metal joints (we are ignoring plastic welding in this example). Finally, Rule I says that you should not use adhesives to bind two metals to each other. All of these rules have exceptions of course, but they will suffice for this example.

At the start of the design process, suppose that we have only determined that there will be a joint, `j1`, and that one of the two materials `m1` is wood (recall that these lower-case symbols refer to a specific joint and material). We can tell FORLOG this by assertions [a] and [b]:

```
[a] joint(j1,m1,m2). {}
[b] type(m1)=wood. {}
```

Notice that these are each given the empty set as a label, which tells FORLOG that these are facts that should always be believed.

FORLOG will now begin to investigate the search space. It considers the rules in the order shown. First, Rule A fires (based on assertion [a]) and asserts the two possible geometries. ATMS choices `A1` and `A2` are created to keep track of these alternatives:

```
[c] config(j1)=overlap {A1}
[d] config(j1)=abut {A2}
[e] choose{A1,A2}
```

The notation `choose{A1,A2}` is also created by FORLOG, and it tells the ATMS that one of the two choices `A1` or `A2` must be included in any final solution.

Now Rule B is triggered (also by [a]), and it asserts three possible bonding methods:

```
[f] bond(j1)=weld {B1}
[g] bond(j1)=adhesive {B2}
[h] bond(j1)=bolt {B3}
[i] choose{B1,B2,B3}
```

Rule C is triggered, and it simply makes assertions [j] and [k]. These in turn cause Rule D to be fired. When Rule D is fired from fact [j], FORLOG is able to apply fact [b] (`type(m1)=wood`) to discard both the `type(m1)=metal` branch and the `type(m1)=plastic` branch. Hence, nothing new is inferred. However, when Rule D is fired from fact [k], three alternative materials for `m2` are identified:

```
[j] material(m1).  {}
[k] material(m2).  {}
[l] type(m2)=metal. {D1}
[m] type(m2)=wood.  {D2}
[n] type(m2)=plastic. {D3}
[o] choose{D1,D2,D3}
```

So far, FORLOG has encountered three different design decisions that it needs to make. It is pursuing each of them in parallel. Now, FORLOG starts considering the rules that contain knowledge about how these design decisions interact. Rule E draws different conclusions under different sets of assumptions. Statement [q] (i.e., that `j1` is a wood-wood joint) is valid under assumption {`D2`}, because according to that assumption, the type of `m2` is wood, which matches the given type of `m1`. Statement [r] is valid in either {`D1`} or {`D3`}, because under those assumptions, `m2` has a type different from wood.

```
[p] same(j1) {D2}
[q] type(j1)=wood {D2}
[r] different(j1) {D1}{D3}
```

Rule F says that bolts cannot be used when the two materials abut. The effect of this is to conclude that the set of assumptions {A2,B3} leads to a contradiction, because A2 is the assumption that the materials abut and B3 is the assumption that they should be bolted together. Statement [s] encodes this "nogood" combination of design decisions. Similarly, Rule G yields statement [t], Rule H yields statement [u], and Rule I yields statement [v]. Statements [t], [u], and [v] can be simplified to conclude that B1, welding, cannot be used in this problem. This is captured by statement [w].

```
[s] nogood{A2,B3}
[t] nogood{D1,B1}
[u] nogood{D3,B1}
[v] nogood{D2,B1}
[w] nogood{B1}
```

At this point, FORLOG has gone as far as it can. It has ruled out welding, but it turns out that there are still 9 possible designs at this point. FORLOG has not enumerated each of these designs individually, but they could be enumerated by considering all combinations of choices that are not "nogood."

Suppose that we decide that the two plates will abut. We tell this to FORLOG in statement [x]. This assertion already exists in the database as statement [d] with label {A2}. The effect of statement [x] is to rule-in A2 and rule-out A1. Hence, FORLOG concludes that nogood{A1} in assertion [y]. Furthermore, since A2 is now known to be correct, FORLOG can conclude from [s], that B3 must be "nogood" all by itself. This is recorded in [z].

```
[x] config(j1)=abut.  {}
[y] nogood{A1}
[z] nogood{B3}
```

This eliminates both B1 and B3 and leaves only choice B2 as a possible bonding technique. In other words, bond(j1)=adhesive is the only remaining possibility. Note that FORLOG has still not determined a unique material for m2—it can keep its options open until more information becomes available.

From this example, we can see how FORLOG is able to explore multiple design decisions in parallel without generating each of the 54 possible combinations of materials, configurations, and bonding methods. The ATMS permits FORLOG to keep track of each alternative design decision separately, and yet combine them when necessary. The result is a kind of *least-committment* design strategy in which FORLOG avoids committing to any particular material, configuration, or bonding method until the decision is dictated by other parts of the design problem. This strategy, first explored by Stefik (1981), results in a very efficient design process.

6 Concluding Remarks

Although there are many interesting and powerful expert systems for intelligent computer-aided design, virtually all of them have been constructed using inadequate tools. The requirements for ICAD that have emerged from these early design systems and from our protocol-analysis studies have provided the impetus for developing the FORLOG architecture. FORLOG improves and extends the paradigm of logic programming to support programming with assertions (instead of terms), complete search flexibility, and constraint propagation. FORLOG provides a clean and elegant programming language for building ICAD systems.

An interpreter and compiler for FORLOG have been implemented in Interlisp-D on the Xerox 1100 series lisp machines.

Acknowledgments

This research was supported in part by the National Science Foundation under grants DMC-8514949 and IST-8519926 and by a contract from Tektronix, Inc. We thank Colin Gerety, Nicholas Flann, Dan Corpron, and Ritchey Ruff for their contributions to the design and implementation of FORLOG.

Bibliography

Clancey, W. J. 1985. Heuristic classification, *Artificial Intelligence,* 27 (3), 289–350.

Clocksin, W. F., and Mellish, C. S., 1984. *Programming in Prolog.* Berlin: Springer-Verlag.

deKleer, J. 1986a. An Assumption-based TMS. *Artificial Intelligence,* 28 (2) 127–162.

deKleer, J. 1986b. Extending the ATMS. *Artificial Intelligence,* 28 (2) 163–196.

deKleer, J. 1986c. Problem-solving with the ATMS. *Artificial Intelligence,* 28 (2) 197–224.

Dietterich, T. G., Corpron, D. R., Flann, N. S. Forthcoming. Forward-chaining logic programming in FORLOG.

Ericsson and Simon, *Protocol Analysis*, 1985.

Goguen, J. A., and Meseguer, J. 1984. Equality, types, modules and (why not?) generics for logic programming. *Journal of Logic Programming,* 2, 179–210.

Lloyd, J. W., 1984. *Foundations of Logic Programming.* Berlin: Springer-Verlag.

Maher, M. 1984. HI-RISE: An expert system for the preliminary structural design of high rise buildings. Doctoral dissertation. Department of Civil Engineering, Carnegie-Mellon University.

McDermott, D. 1985. The DUCK manual. Rep. No. 399, Department of Computer Science, Yale University.

McDermott, J. 1982. R1: A rule-based configurer of computer systems. *Artificial Intelligence,* 19(1), 39–88.

Mittal, S., Dym, C., Morjaria, M. 1986. PRIDE: An expert system for the design of paper handling systems. To appear in *IEEE Computer* Special Issue on Expert Systems for Engineering Applications.

Russell, S., 1985. The compleat guide to MRS. Rep. No. KSL-85-12. Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

Stallman, R. M., and Sussman, G. J., 1977. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence,* 9, 135–196.

Stefik, M. 1981. Planning with constraints, *Artificial Intelligence,* 16 (2), 111–139.

Stefik, M., Bobrow, D. G., Mittal, S., and Conway, L. 1983. Knowledge programming in LOOPS: Report on an experimental course. *AI Magazine,* 4 (3), 3–13.

Sussman, G. J. and Steele, G. L., Jr. 1980. CONSTRAINTS—A language for expressing almost hierarchical descriptions. *Artificial Intelligence,* 14, 1–39.

Ullman, D. G., and Dietterich, T. G., 1986. Mechanical design methodology: Implications on future developments of computer-aided design and knowledge-based systems. *Proceedings of the 1986 ASME International Computers in Engineering Conference.* American Society of Mechanical Engineers. Chicago, Illinois. Volume I.

Ullman, D. G., Stauffer, L. A., Dietterich, T. G. Forthcoming. Preliminary results of an experimental study of the mechanical design process.