CIS-TR 85-05

Control Knowledge in Expert Systems: Relaxing Restrictive Assumptions

Stephen Fickas, David Novick

Department of Computer and Information Science
University of Oregon

Control Knowledge in Expert Systems: Relaxing Restrictive Assumptions

by

Stephen Fickas
David Novick

Computer and Information Science Department
University of Oregon
Eugene, Oregon 97403 USA

Telephone (503) 686-4408
csnet fickas@uoregon

February 1985

Abstract

Many expert systems use fixed, domain-independent control strategies. These systems often embody assumptions about control and control knowledge which limit the utility of application programs. This paper shows that effective control often requires domain-specific knowledge. We present the ORBS model and implementation of a rule-based systems environment which attempts to expose, and give users access to, hidden control points. ORBS provides a language for defining control strategies, and supports the interactive development of these strategies. We discuss the process of explicating control knowledge through a process of refining an application rule.

1. Introduction

This paper presents our recent work in control of expert systems. In particular, our interests lie in the definition and construction of control strategies for problems requiring sophisticated use of human and machine resources. Our research suggests that traditional expert systems, which rely on a fixed, non-analyzable control model are well suited for only limited problem domains, and make it difficult to address problems of termination and higher-order knowledge.

Over the past five years, we have built various expert systems using languages with fixed control strategies. We have come to learn that these systems embody and thus require adherence to many assumptions about the nature of control. These assumptions are often tied to some particular problem domain, e.g., modeling human cognition or solving problems in predicate logic. While it is sometimes possible to override these assumptions by clever encoding of control knowledge in domain knowledge terms, the resulting system is cumbersome to develop, debug, and maintain. In general, an entirely new control model must be simulated atop the old. Though some languages make this simulation process easier than others, we argue that this is the wrong approach.

Instead, we have attempted 1) to explicate the implicit control decision points in a rule-based system, and 2) to make all control points accessible to and modifiable by the user or application program. Specifically, our experience with building rule-based systems in languages such as OPS5 [9], YAPS [1], PROLOG [4], and Hearsay III [6] led us to conclude that

(1) Conflict resolution is a complex process that often requires domain-specific knowledge to be effective. A "hard-wired" system control strategy shown to be useful in one domain may be ill-suited in another.

(2) Defining control knowledge may be just as difficult a task as defining domain knowledge. Hence, the construction, debugging, and maintenance of control knowledge should be well supported by the system.

We present a model that separates domain knowledge from control knowledge, and allows a control strategy to be tailored to the problem domain. We argue that this degree of flexibility is only possible if all control decisions are made explicit and accessible to the application program. Although the model we describe is part of a particular expert system language, we argue that it is general enough to be applied to any rule-based system where control is an issue.

The next section discusses the particular control decisions an expert system must face. Section 3 introduces the ORBS control model. Section 4 presents the gradual evolution of domain and control knowledge in a small portion of a VLSI design system. In this example, we point out how the transformation of implicit control to explicit control allows us to tailor control knowledge to our design problem.

2. Control Issues in Expert Systems

Given a set of activations[1] competing for control, a rule-based system must choose which, if any, activation to apply. This process presents at least two types of primitive or first-order control decisions, one dealing with competing solutions, one dealing with organization or tasking:

(1) The system must choose between alternative solutions. For instance, in a system dealing with the choice of circuit components, there might exist an activation for every different component choice. The system should choose the one that best meets constraints and performance standards.

[1] An *activation* represents a unique match of the left hand side clauses of a rule against a data base of facts.

(2) The system must order a set of tasks. For example, the choice of components may be one subtask out of many, e.g., define interfaces, optimize area. The system must insure that dependent tasks are taken in the right order.

In addition to these first-order concerns, the system must also deal with termination and higher-orders of control:

(1) The system must decide when to halt. Reasons for halting may vary from a) finding a solution to b) giving up when all attempts at solution have proved futile.

(2) The system must assess whether the current control strategy is still applicable, or possibly may choose between competing strategies. Can the control strategy be examined and changed by the application program?

Different expert systems have proposed various approaches to these three control problems. The approach used will often determine system characteristics in at least three important respects:

- Flexibility. Different applications may require different control strategies. Is the system flexible enough to permit different kinds of strategies to be matched to different domains?
- Representation. Control knowledge is an important component of a rule-based system. It should have representational and environmental support on a par with domain knowledge. Does the system provide an explicit representation of control knowledge?
- Accessibility. Can the application program access control points? In particular, we suggest that the black-box model of control is inadequate for sophisticated expert systems.

A system which performs well in some of these aspects may perform poorly with respect to others. Our goal is to provide a control model which is sufficiently powerful to perform all types of control decisions while maintaining desirable system characteristics.

3. The ORBS System

Our control model is one piece of a larger environment for building rule-based systems. The environment is called ORBS (Oregon Rule Based System), and is discussed in detail in [7]. In this paper, we are interested in both the ORBS control mechanism and parts of the environment that support control.

ORBS traces its direct lineage to two ancestors: Hearsay III [6] and YAPS [1]. An ORBS system contains a data base of relational n-tuples called facts that may include both Lisp objects and Flavor [11] objects, a set of forward-chaining rules that trigger on those facts, and a set of scheduling functions that determine which triggered rule to apply. To illustrate these ideas, we use throughout this paper an ORBS rule taken from an interactive system that assists a silicon compiler. The rule shown in Figure 1, written in the YAPS style of control, represents the following piece of knowledge: "If you are building a ring oscillator that must be both small and fast, then use inverter-type-3 as the basic building block."

```
(defrule choose-inverter-3
      (goal (choose-inverter))                    ; left-hand side (LHS)
      (ring-osc -ro)                              ; -ro is a pattern variable
 test (eq (send -ro 'speed) 'fast)
      (eq (send -ro 'area) 'small)
->
      (remove 1)                                  ; right-hand side (RHS)
      (fact ro-cell inverter-3)
:::
      status: active                              ; attributes
      author: simoudis)
```

Figure 1

The LHS is made up of two patterns that will match data base relations of type goal and ring-osc. The goal relation is used for control, providing a means of stepping through a set of tasks. For the rule to match, we must be in the "choose-inverter" task; i.e., some other process in the system must have inserted the goal relation into the data base. (Figure 3 in section 4 shows how this type of control knowledge can be removed from a rule's LHS.)

The optional keyword "test" marks the beginning of the filtering predicates. In this case, the two predicates ask the ring-oscillator (i.e., the flavor instance representing the ring-oscillator, bound to -ro) for its speed and area by sending the appropriate messages. Section 4 discusses why we might want to represent these two predicates as separate pieces of control knowledge.

The "->" marks the beginning of the RHS actions, in this case two pre-defined Lisp functions: "remove" removes the fact matching the ith LHS pattern from the data base; "fact" adds a relation to the data base. In general, any evaluable Lisp expression may appear on the RHS of a rule. In this way, an ORBS rule is more akin to a Hearsay Knowledge Source than a traditional production which allows only certain restricted data base actions to appear as actions. While such productions can often be run either forward or backward, an ORBS rule is limited to forward-chaining.

The ":::" marks the beginning of further rule attributes: "status" is an ORBS-defined attribute, and "author" a user-defined attribute. An ORBS rule may be extended by zero or more user-defined attributes. Using an ORBS rule-extension declaration, the attribute "author" was added as follows:

```
(add-rule-attribute 'author)
```

ORBS gives the rest of the system machinery for accessing the author field. For instance, we could define a piece of control knowledge that chooses one rule over another depending on the confidence in the rule author. Section 4 provides a further example of rule extension.

The matching process produces zero or more activations. This is called the conflict set. Each cycle produces a new conflict set.[2] An activation represents the match of a rule's LHS against facts in the database. The activations in the conflict set may represent competing solutions or cooperating tasks to be completed. We may want to choose none, one, some, or all activations for invocation. In ORBS, the choice is made by the control strategy defined by the system developer. That is, ORBS has no built-in control strategy such as YAPS or PROLOG have. Rather, ORBS provides tools for constructing a control strategy. To build a control strategy in ORBS, the developer 1) defines a set of scheduling functions in Lisp or chooses from a catalog of pre-existing functions, and 2) declares how the functions are to be combined to form a control strategy. The latter is done by using a modified form of McDermott and Forgy's

[2]For sake of efficiency, ORBS incrementally maintains the conflict set from cycle to cycle.

scheduling expressions [10]. Here is an abstract example:

[F1] > (F2 arg1) > (F3 * F4) > [F5]

The initial conflict set CS is passed to the scheduling function F1. F1 returns a new, possibly empty, conflict set CS', which is a subset of CS. If CS' is non-empty, it is always passed on to the next function. If CS' is empty, we have two choices: 1) pass along the empty set (i.e., CS'), or 2) treat F1 as a no-operation and pass along the original conflict set CS. The brackets specify that CS' is to be passed along, empty or not. If a function lacks brackets, then a return value of the empty set will cause the function to be ignored.

The function F2 takes an argument. That is, when it is called, it will be passed both the current conflict set CS' and arg1.

The next two functions, F3 and F4, are combined using the intersection operator: the results of F2 (i.e., CS') are passed to both functions, and their results are intersected.

Finally, F5 is called with the results of the intersection. The conflict set returned by F5 becomes the execution set ES. The activations in EC are passed to the activation-invoker and their RHS actions are invoked. This ends the cycle.

Figure 2 provides a graphical view of the ORBS system.

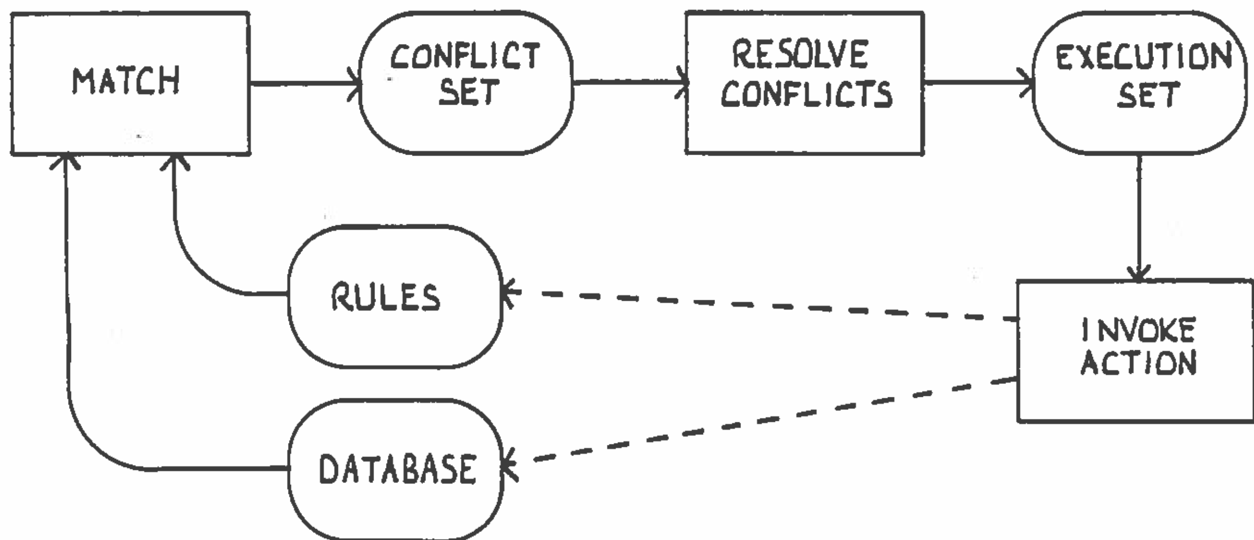

Figure 2

A complete cycle represents a match/resolve-conflicts/invoke-action process.

4. Explicit Control Representation in ORBS

In this section we take a closer look at the choose-inverter-3 rule from Figure 1. In particular, we will gradually refine it into a rule that better represents both our domain and control knowledge in the VLSI design problem. The refinement process as we present it here corresponds closely to the actual evolution of our ideas on control. That is, our initial VLSI system did little to represent control explicitly. As problems with this approach became apparent, we began to seek out hidden control decisions and make them

accessible to the application program. Each time this was done, new and powerful control strategies became apparent.

Our first attempt to build a VLSI system using ORBS was based on a control model we were familiar with—YAPS. Hence, we defined a control strategy in ORBS that modeled that of YAPS.

The YAPS Strategy

YAPS uses the following control strategy:

(1) Using a history list, eliminate activations that have been executed on any previous cycle.

(2) If one or more activations contain the keyword "goal", prefer the activations whose goal patterns match the goal relations most recently added to the data base. (Control knowledge, in the form of goal relations, appears throughout the domain rules for tasking purposes, as in the choose-inverter-3 rule of figure 1.)

(3) Reaching this step implies that in step (2) either a) there were no goal relations, or b)there were multiple goal relations. In either case, break ties by looking at non-goal patterns in the LHS, preferring the activations whose LHS clauses match with relations most recently added to the data base.

(4) If ties still exist, choose an arbitrary activation for execution.

(5) Halt the system if no activations remain.

To model the YAPS strategy, we use five ORBS scheduling functions: D2 eliminates already-applied activations;[3] R5K prefers the rule activations with the most recent "goal" fact; R5F prefers activations with the most recent non-goal fact; AD1Y arbitrarily selects among ties; HECS halts the system if the conflict set is empty. These functions, along with all the scheduling functions proposed by McDermott and Forgy [10], are part of the ORBS catalog of scheduling functions.

We now must inform ORBS that 1) the functions D2, R5K, R5F, AD1Y and HECS are to be used in scheduling, and 2) they are to be applied in a certain order. ORBS accepts an extended form of McDermott and Forgy's scheduling expressions [10] to accomplish this:

[D2] > (R5K goal) > R5F > AD1Y > HECS

This defines a control strategy that applies the function D2 to the conflict set and passes (as denoted by the ">") the resulting conflict set to R5KF. The square brackets indicate that if the D2 function produces an empty conflict set (i.e., all activations have been previously fired) then pass the empty conflict set along. If we remove the square brackets from D2, then if D2 produces an empty conflict set, the function is ignored, i.e., the original conflict set is passed along. The function R5K is applied to the conflict set using "goal" as the keyword. The resulting conflict set, if non-empty, is passed to R5F. R5F is applied with the resulting conflict set being passed to AD1Y. AD1Y arbitrarily selects one of the remaining activations by deleting all but the first activation in the conflict set. If HECS receives an empty conflict set, say for instance because D2 eliminates all activations, it halts the system. Note that if we take HECS out of the strategy, as we shall do shortly, the system will continue to cycle until some other process explicitly halts the system.

Making tasking information explicit

YAPS (like its predecessor, OPS5 [9]) does not support the explicit representation of control knowledge;

[3] ORBS keeps a list of all previously invoked activations.

domain and control knowledge are intermixed. Although McDermott and Forgy do lay out the beginning of a control vocabulary [10], the control model itself remains a black box: its components cannot be examined or changed. The YAPS form of the choose-inverter-3 rule would look generally the same as the ORBS rule in Figure 1 without the attribute fields.

Our first step separates out tasking (i.e., control) information from domain (i.e., LHS) information. To do this, we associate with each rule in our system a specific task. First we define a rule attribute "task":

```
(add-rule-attribute 'task)
```

As each rule is defined, we can fill its task field with the appropriate value. In the VLSI example, the tasks we have identified are start-up, choose-inverter, and choose-cell. We can now discard the goal patterns from the LHS of the rule in Figure 1 (and discard the goal relations in the ORBS database as well). Thus our new choose-inverter-3 rule is:

```
(defrule choose-inverter-3
          (ring-osc -ro)
  test    (eq (send -ro 'speed) 'fast)
          (eq (send -ro 'area) 'small)
  -->
          (fact ro-cell inverter-3)
  :::
          status: active
          author: simoudis
          task: choose-inverter)
```

Figure 3

Thus far we have removed the tasking information from the domain knowledge database and the rule's LHS. We still have to define a scheduler that can use the rule's task field to order tasking. We do this in ORBS by modifying our original strategy as follows:

```
[D2] > (L2 start-up) > (L2 choose-inverter) > (L2 choose-cell) > AD1Y > HECS
```

Figure 4

Here we have replaced the recency rule R5 with more explicit tasking rules. The scheduling function (L2 task) prefers activations of rules associated with the given task[4]. For example, L2 could be defined as follows[5]:

```
(defsched L2 (conflict-set task)
  (for activation in conflict-set
              when (eq task (get-attribute activation 'task))
              collect activation))
```

This process of refinement of control can be summarized in two steps. First, we removed the need to store

[4]We might instead replace the sequence of L2 functions with the single L1 function in ORBS's catalog which takes an ordered list of tasks, e.g., (set-strategy '([D2] > (L1 (start-up choose-inverter choose-cell)) > AD1 > HECS). L1 prefers activations of rules in the order of the tasks in the list.

[5]Scheduling functions are defined with the function *defsched*. The first argument to a scheduling function must be a list of activations, which, following the syntax proposed by McDermott and Forgy, is left implicit in the strategy. Subsequent arguments, such as the task in L2, are stated explicitly. The function returns zero or more activations.

tasking information (through goal relations) in the domain data base; this moves us towards our goal of explicit representation of control information. Second, we moved control information to a new field specifically defined to hold it. This field is readable and writable by other portions of the system, and hence allows an application program a certain amount of introspection and dynamic restructuring of its control strategy.

Multiple activations per cycle

By including AD1Y, our control strategy allows only one activation per cycle to be invoked. Many systems have such a hard-wired decision on the number of activations that can be invoked on any cycle. In the VLSI domain this is overly restrictive: we find that we can safely execute all activations of a particular task type. That is, all rule ordering is handled through the task field and the L2 functions. We can guarantee that invoking two rules of the same type during the same cycle will not interfere with each other. To allow multiple activations, we simply remove the AD1Y function:

[D2] > (L2 meta) > (L2 start-up) > (L2 choose-inverter) > (L2 choose-cell) > HECS

In general, a control strategy can specify one, some or all activations in a conflict set to be invoked by including the appropriate scheduling function.

Explicit representation of selection knowledge

The rule choose-inverter-3 uses the attributes "speed" and "area" as discriminators for choosing the components of a ring oscillator. As the VLSI system evolved, we discovered other factors that should be considered, e.g., dollar cost, reliability, and availability. Our first approach was to attach each to the rule through the rule's test slot as shown below:

```
(defrule choose-inverter-3
          (ring-osc -ro)
  test    (eq (send -ro 'speed) 'fast)
          (eq (send -ro 'area) 'small)
          (eq (send -ro 'dollar-cost) 'low)
          (eq (send -ro 'reliability) 'high)
          (eq (send -ro 'availability) 'ready)
  ->
          (fact ro-cell inverter-3)
  :::
          status: active
          author: simoudis
          task: choose-inverter)
```

This rule will choose a ring oscillator that exactly meets the test criteria. This, however, was too restrictive; there may not be a ring oscillator which exactly fits the test. For instance, we may be willing to accept a fast, small, highly reliable, readily available ring oscillator whose dollar cost is large if there is nothing better available.

Our solution uses an agenda-based approach where each scheduling function represents one of the attributes, such as speed, area, and cost. Each function adds a weight to the activation depending on its local merits (ORBS provides a pre-defined weight field for each activation). In this way, the value-testing predicates may be moved from the test portion of the choose-inverter-3 rule (and all other rules dealing with component selection) to the scheduling functions. Hence we will have a scheduling function for each significant attribute, e.g., AREA, SPEED, and DOLLAR-COST. To conclude the agenda strategy, we

select from the catalog of scheduling functions W1, a function that chooses the activation(s) with the greatest weight.

The corresponding strategy (in three parts) is as follows:

[D2] > (L1 (start-up choose-inverter choose-cell)) >

As we have discussed earlier, D2 eliminates all previously fired activations, and L1 does tasking by preferring activations of rules in order of the tasks in the list. This reduces the conflict set to activations concerned with a single common task. The resulting conflict set is passed through one or more groups of agenda-weighting functions.

... > AREA > SPEED > DOLLAR-COST > RELIABILITY > AVAILABILITY > ... >

The ellipses represent other groups of functions for other tasks. The scheduling functions shown weight the activations based in turn on area, speed, dollar cost, reliability, and availability. The conflict set will be passed untouched through any weighting groups that do not pertain to the current scheduling task. The resulting conflict set is then passed to W1 which chooses the activations with the greatest weight.

W1 > HECS

Thus the new choose-inverter-3 rule is:

```
(defrule choose-inverter-3
        (ring-osc -ro)
 ->
        (fact ro-cell inverter-3)
 :::
        status: active
        author: simoudis
        task: choose-inverter)
```

Now, control knowledge relevant to choosing among alternatives is explicitly represented in the control strategy itself.

When should the system halt?

Until now our control strategy has included the scheduling function HECS. HECS causes the system to halt when the conflict set is nil by calling the ORBS function *halt-orbs*. In fact, this decision was "hard-wired" into the system initially. That is, the system would always halt when the control strategy produced an empty set of activations. Like the decision to allow only one activation invocation per cycle, we quickly ran into cases where this was overly restrictive. In particular, an empty conflict set in our VLSI system signals that we are stuck. In this case, we would like to ask the user to loosen some of the problem constraints before deciding to give up (i.e., halt). To bring this about, we replace HECS with VLSI-LC, a scheduling function that interacts with the user to add and remove facts from the database when the conflict set is nil.[6]

As an example, suppose that the task is choose-inverter and the conflict set is nil when VLSI-LC is called.

[6] Note that VLSI-LC was built from scratch for the VLSI application. The strategy in which it is used includes scheduling functions like W1, L1, and D2, which are all part of the ORBS catalog and can be used across application domains.

This means that rules like small-fast have all been trimmed from the conflict set because none was found to meet the user's constraints, as represented by scheduling rules such as AREA, SPEED, DOLLAR-COST, RELIABILITY, AVAILABILITY. At this point, the user is given the chance to loosen one or more constraints so that the system can try again. For instance, the user may set a higher cost limit. Once this is done, the system activation-invoker is called in the normal way. Since the conflict set is nil, it does nothing and a new match cycle is started.

When does the system halt? Strictly speaking, whenever the function halt-orbs is called. This function can be called any place a normal Lisp function can be called, such as from the RHS of a rule, from a break, or from a scheduling function. Its action is to halt after the current cycle is complete. A cycle is completed after the activation-invoker finishes. In our VLSI example, there are two types of system halts. The first occurs when a solution has been found. A rule monitors for this, and, among other actions on its RHS, it calls orbs-halt. The second occurs when the system is hopelessly stuck. A part of VLSI-LC checks to see if the user is willing to loosen the current set of constraints. If not, halt-orbs is called and the system gives up.

Higher levels of control

Other than our termination example above, we thus far have seen examples of what we might call first-order control knowledge--knowledge that deals directly with the conflict set. As others have noted (see for instance [5]), higher orders of control knowledge may exist. In the VLSI system, we have run into such a case: when a certain time threshold is reached, we wish to change from an agenda-based strategy to a quick-and-dirty control strategy. We will first describe how we implemented this control knowledge in ORBS, and then discuss its limitations.

We will represent our strategy-switching knowledge as a domain rule:

```
(defrule nearly-out-of-time                  ; change strategy if nearly out of time
        (time-remaining -tr)                 ; this fact maintained by system
 test   (< -tr critical-time)                ; critical-time is a global variable
 ->
        (set-strategy 'fast-search)
:::
        status: active
        task: change-strategy)
```

We want this rule to fire when the time we have remaining to find a solution falls below a given threshold. Further, an activation of this rule should be given top priority; strategy changing (second-order control) takes precedence over choosing among VLSI rules (first-order control). We have defined a new task change-strategy. We can place an L2 scheduling function in the current strategy such that nearly-out-of-time (and other strategy changing rules) will be given first priority:

[D2] > (L2 change-strategy) > (L2 start-up) > (L2 choose-component) ...

When and if rule nearly-out-of-time is invoked, it causes the above scheduling strategy to be changed to the one below:

[D2] > (L2 start-up) > ((L2 choose-component) * (COMP-TYPE CMOS)) ...

The scheduling function COMP-TYPE is one specifically tailored to the VLSI application. It represents knowledge that a CMOS implementation is always possible. While it might not always be the best, we will use it when time runs out. By accessing appropriate fields in an activation, COMP-TYPE determines

if the proposed component uses CMOS technology. If not, it trims the activation from the conflict set.[7]

Second-order control knowledge presents problems for ruled-based systems in which control is represented through relations in the LHS. In such systems, changing the control strategy requires redefining the rules. In ORBS, we have extracted control from domain knowledge, and so avoid the problems of redefining rules. However, the implementation of second-order control as a *domain* rule points out other problems. First, we have reintroduced control knowledge into the domain portion of our system, as a rule now rather than as a goal relation. Second, we have mixed second-order control knowledge with first-order control knowledge by placing our second-order change-strategy function among the first-order tasking and component selection functions. Thus we are back to simulating a more general control model on top of the language. We view the problem as another manifestation of the same problem we initially set out to solve: a control model resting on implicit assumptions. For instance, we currently assume that the only interesting conflict sets are empty and non-empty ones. In some cases, however, we might be interested in the size of the set. For instance, we may wish to halt when the number of competing alternatives is two or less. This type of control knowledge is best stated explicitly and only once. In the current ORBS system, it would require placing a scheduling function F after each scheduling function that removed alternatives. F would have to check the size of the current conflict set, and halt the system if it became less than three. Without an explicit representation of the two-candidate rule, maintenance of the control strategy becomes problematic.

One of our current projects is the definition of a control vocabulary that allows a richer statement of policy. In it, we could explicitly state control knowledge such as

Whenever the conflict set contains only two alternatives, halt.

If the time remaining is less than THRSH then switch to a quick-and-dirty control strategy.

Whenever the conflict set contains more than N activations use alternate strategy S.

Among other things, it appears that this will require a "control database" separate from the domain database, one similar to Hearsay III's scheduling blackboard [6].

5. Conclusions

We have shown through examples that a fixed control strategy can lead to problems when dealing with a domain that requires sophisticated use of both machine and human resources. In particular, fixed strategies embody assumptions that often lead to awkward simulation of better suited control models. We have illustrated four assumptions that should rightfully be made explicit, but are generally buried in domain knowledge or system code: tasking, invoking multiple activations, selection criteria, and halting. The ORBS control model represents each of these four as control decisions with an explicit representation and under supervision of the application program. By combining these decisions in various ways, the system developer can tailor a control model to the application at hand. In general, we have found this flexibility has allowed us to build bigger, more complex systems in less time. More important, it has allowed us to maintain them reliably over time.

Is ORBS the last word in control? No, for several reasons. First, we lack a representation of control beyond the first-order. As we saw in the last example of section 4, we currently must model second-order knowledge on top of domain and first-order knowledge. As we have pointed out, a better solution to this problem will require us to make our first-order assumptions explicit.

A second problem with the ORBS model is lack of an explicit representation of backtracking control. There are two parts to this problem: choice points and strategy.

---

[7]Read (A • B) as the intersection of activations returned by A and B. ORBS provides union (+) and weighting (&) scheduling operators as well.

5.1. Backtracking Choice Points

An underlying mechanism is needed to record choice points, and return to them when necessary. Hearsay III provides this through its Hypothetical Reasoning facility. In Hearsay III, the system can be told that a choice is being made between competing alternatives in state S. The system will spawn a new problem solving state S' for the choice. If the choice eventually turns out to be bad, it is possible to return to state S and try another alternative, which spawns another new state (a sibling of S'). ORBS currently has no analogous facility.

5.2. Backtracking Strategy

The second part of the backtracking control problem is defining a strategy. PROLOG provides a fixed, depth-first search strategy. As we have argued with respect to other models of control, this is too inflexible for many problems. In Hearsay III, while the mechanism is there, there is no explicit means to represent it. That is, while the system developer is free to define any backtracking strategy he or she sees fit, all of this must be done within Lisp code; Hearsay III lacks an explicit representation of a control strategy, backtracking or otherwise. In ORBS, we have so far resisted the temptation of adding a Hearsay III style backtracking mechanism *without* an accompanying representation, i.e., a language for specifying backtracking strategies. While this seems a particularly tough nut to crack in general, we are encouraged by others making progress on more limited forms of the problem [3].

In summary, we view the current ORBS system as an intermediate point along the route to a flexible, extensible, and explicit model of control. We have flushed out a number of control decisions that were hidden within domain knowledge or code in earlier systems. We have also shown that other assumptions remain to be explicated.

6. Status

The ORBS system runs on a VAX using the Maryland extension to Franzlisp [2], and on a Symbolics 3600. Problem solving control is one of several problems we are working on in ORBS. Others include

(1) The construction of an interactive, graphics environment for building and debugging expert systems.

(2) The use of Software Engineering techniques in developing expert systems. For example, we are experimenting with a specification language, KATE [8], for specifying components of a problem suited to an expert system implementation. KATE will eventually be mapped into ORBS objects, rules, and scheduling functions.

(3) Automating strategy construction. ORBS currently keeps a history of conflict resolution. The state of the data base, the conflict set, the action of the scheduling rules, and the activations chosen to fire are recorded for each cycle. We are experimenting with this data to allow the system to learn what control strategies are applicable to what problems.

Acknowledgments

Other past and current members of the ORBS project include Allen Brooks, Kim Dannewitz, Michael Hennessy, Rob Reesor and Bill Robinson from the University of Oregon, and Bill Bregar from Oregon State University.

This work was partially supported under National Science Foundation grant DCR-8312578.

References

[1] Allen, E.
YAPS: Yet Another Production System,
TR 1146, Computer Science Dept., University of Maryland, 12/83

[2] Allen, E., Trigg, R., Wood, R.
Maryland Franzlisp Environment,
TR 1226, Computer Science Dept., University of Maryland, 11/83

[3] Clancey, W., Bock, C.,
MRS/NEOMYCIN: Representing Meta Control in Predicate Calculus
HPP Memo 82-31, Computer Science Dept., Stanford, 11/82

[4] Clocksin, W., Mellish, C.
Programming in Prolog,
Springer-Verlag, 1981

[5] Davis, R.
Applications of Meta Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases,
In *Knowledge-based Systems in Artificial Intelligence*,
Davis & Lenant (Eds.), McGraw-Hill, New York, 1982

[6] Erman, L., London, P., Fickas, S.
The design and example use of Hearsay III,
In *7th International Joint Conference on AI*, Vancouver, 1981

[7] Fickas, S.,
An Introduction to ORBS
Tech Report CIS-TR-84-02, Computer Science Dept., University of Oregon, 2/84

[8] Fickas, S., Laursen, D., Laursen, J.,
Knowledge-based Software Specification,
In *Workshop on Knowledge Based Design*, Rutgers Univ., 1984

[9] Forgy, C.,
OPS5 User's Manual,
Tech Report, Computer Science Dept., CMU, 1981

[10] McDermott, J., Forgy, C.
Production system conflict resolution strategies,
In *Pattern-Directed Inference Systems, Academic Press, 1978*

[11] Weinreb, D, Moon, D.
Objects, Message Passing, and Flavors,
Lisp Machine Manual, Ch. 20, Symbolics Inc., 1981

Note of Omission

Mr. Evangelos Simoudis was a valuable first user of our system. We neglected to list him in the acknowledgements section, but do so here. His contribution was appreciated.