CIS-TR 85-04

Building Control Strategies in a Rule-Based System

Stephen Fickas, David Novick, Rob Reesor

Department of Computer and Information Science
University of Oregon

Building Control Strategies in a Rule-Based System

by

Stephen Fickas
David Novick
Rob Reesor

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

January 1985

Abstract

Rule-based systems in which conflict resolution occurs require a control strategy. Many systems use domain-independent, fixed control strategies. This paper shows that effective conflict resolution often requires domain-specific knowledge. Moreover, defining control knowledge may be as difficult as defining domain knowledge. We present the ORBS model and implementation of a rule-based systems environment which supports complex and dynamic control strategies, and the interactive development of these strategies. ORBS and earlier systems are compared with respect to language, control, and environmental support for development of control. Four examples of control using ORBS are discussed: emulating a conventional system, separating domain knowledge from scheduling knowledge, using an agenda-based strategy, and dynamically changing a strategy during execution. The ORBS environment provides machine-aided construction of scheduling strategies through a catalog of scheduling functions from which strategies can be constructed, a skeleton scheduler, a scheduling editor, a scheduling tracer, and a break package. Finally, the paper discusses a model for system implementation of control from user-provided examples.

Topic Area: Expert Systems

Key Words: rule-based systems, control knowledge, interactive environments

Word Count: 5498

Contact: Stephen Fickas, (503) 686-4408, csnet fickas@uoregon

1. Introduction

This paper describes a control model for rule-based systems. Although it is part of a particular expert system language, we argue that it is general enough to be applied to any rule-based system where conflict resolution must occur. The model is based on two propositions:

(1) Conflict resolution is a complex process that often requires domain-specific knowledge to be effective. A fixed 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.

In this paper, our main focus is on the problem of control knowledge posed by the first proposition. In particular, our model separates domain knowledge from control knowledge, and allows conflict resolution to be tailored to the problem domain. The expert system writer may define, through a formal interface, the components of his or her strategy. These components may be selected from a catalog of predefined primitives or built from scratch.

The last two sections of the paper describe our progress on the problem of support posed by the second proposition. Our model supports strategy construction by supplying editing, tracing, simulation and break-point tools. Future work lies in allowing further automation of the construction process.

2. Control Issues for Conflict Resolution

Given a set of activations[1] competing for control, a rule-based system must choose which activation to apply. This process presents at least two kinds of control decisions, one dealing with competing solutions, and 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 a rule for every different component choice. In certain situations, more than one may be a candidate. The system should choose the one that best meets constraints and performance standards.

(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.

Different expert systems have proposed various approaches to these two 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 be represented separately from domain knowledge. Does the system facilitate separation of domain and control knowledge?
- Dynamic control. A system may deduce control knowledge while executing. Can the system take advantage of such information?

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

---

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

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 a system that performs silicon compilation. The rule, shown in figure 1, 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.

We have chosen this rule not because it is particularly profound, but because it illustrates the points we wish to make. In particular, we will show in section 5 how the initial form of the rule in figure 1 can be modified to represent better the control of the application domain.

```
(defrule small-fast
      (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., something else must have inserted the goal relation into the data base. (Figure 3 in section 5.2 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 5.3 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. 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 author attribute may have been added as follows:

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

ORBS gives the rest of the system machinery for accessing the author field. For instance, we could define a piece of scheduling knowledge that chooses one rule over another depending on the confidence in the rule author. Section 5.2 provides a further example of rule extension.

4. Two Example Control Models

In this section we look at two control models, those of YAPS and Hearsay III, which represent widely different points of view. The control model used by YAPS is one found in many traditional rule-based systems: a fixed, black box model. The Hearsay III control model, on the other hand, relies on a tailored scheduling strategy being constructed for each new problem. Also, the scheduling process itself is open to inspection and modification by the application program.

To restate the two control problems introduced in section 2, we have a conflict set containing competing activations. Competitors may represent 1) alternative solution choices, or 2) alternative tasks to be performed.

YAPS

In YAPS, the two control problems are intermixed. First, activations that have been chosen previously for invocation are eliminated.[2] Next, the remaining activation that contains the most recent "tasking" fact in the data base will be chosen. Thus, the LHS of each rule typically has a task clause along with domain clauses (see the small-fast rule in figure 1 of section 2.2). For tie-breaking (i.e., where multiple activations address the same task), non-tasking facts are checked; the activation with the most recent non-tasking fact is chosen. Any remaining ties are broken arbitrarily.

We argue that YAPS (and its predecessor, OPS5 [8]) lack the three desirable characteristics of control that we specified in section 2. In particular, the YAPS control model cannot be changed to suit the problem domain (it is fixed for all time).[3] It does not support the explicit representation of control knowledge (it does not separate competence from performance knowledge). Although McDermott and Forgy do layout the beginning of a control vocabulary [9], the control model itself remains a black box (its components cannot be examined or changed).

Hearsay III

One of Hearsay III's design goals was the promotion of control knowledge (also known as scheduling knowledge) to first-class citizenship. When a Hearsay III Knowledge Source (KS) triggers, it creates an activation, which is placed on a *scheduling blackboard* (SBB). The user may define *scheduling knowledge sources* (SKS) that trigger on this placement. These in turn cause activations to be created and placed on the scheduling blackboard (the process stops here: an SKS is not allowed to trigger on the placement of an activation of another SKS). SKS are generally used to organize and order KS activations on the scheduling blackboard. Once ordered by SKS, the activations of KS are invoked by a domain scheduler (a piece of user-defined Interlisp code). It is this scheduler along with the SKS that determines the structure of the SBB, e.g., a priority queue, an ordered agenda.

---

[2] In many rule-based systems, Hearsay III being one exception, the same set of facts may match a rule on every cycle, thus causing the same activation to be generated from one cycle to the next.

[3] OPS5 (but not YAPS) does allow the user to change the built-in strategy by specifying whether either the first clause or any clause is to be designated as dominant.

To illustrate the Hearsay III approach better, we define a Knowledge Source that corresponds to our small-fast rule in section 3:

```
(DECLARE-KS small-fast (ro x y z)
    (APAND
        (ring-osc ro)
        (ROLE-OF timing ro z)
        (ROLE-OF x-dimension ro x)
        (ROLE-OF y-dimension ro y)
        (EQ (quantify-speed z) 'fast)
        (EQ (quantify-area x y) 'small))
    'choose-inverter
    (@ (ro-cell inverter-3)))
```

Figure 2

The pattern variables of this knowledge source are ro, x, y, z; no hyphen is needed to distinguish them since they are formally declared. The "APAND" marks the beginning of the trigger, which consists of four patterns matching the relations ring-osc and ROLE-OF on the blackboard. The trigger also contains two filtering predicates using EQ. "choose-inverter" is a scheduling level, which we will discuss shortly. The body of the knowledge source consists of a function "@" that places an instance of the relation ro-cell onto the blackboard.

Looking more closely at the scheduling level concept, we might structure the SBB around priority levels. "choose-inverter" would be one such level. It would have levels above it and below it, e.g., "start-up", "order-cells". When a KS activation is created, it is placed on the level determined by its scheduling-level field. We could define a Hearsay III scheduler that simply moves down levels looking for activations. If one is found, it is fired, and its activation is removed from the SBB. In this way, all "start-up" activations are executed before all "choose-inverter" activations, which are executed before all "order-cells" activations, and so on. Note that all competing activations are placed on the SBB. This, in conjunction with Hearsay III's ability to spawn new problem solving contexts, makes possible dependency-directed backtracking and most of its variants.

The Hearsay III control model provides flexibility, a separate representation for control knowledge, and can be changed dynamically. However, our experience[4] shows it has two major flaws: 1) over complexity, compounded by 2) lack of environmental support. For each new application, a scheduling strategy must be constructed from scratch with barely minimal debugging tools. The representation of control as two disjoint components -- SKS and a Lisp scheduler -- makes it clumsy to build and maintain scheduling strategies. As we shall see in the next section, ORBS attempts to remedy both of these problems.

Finally, note that YAPS and Hearsay III bound a wide spectrum of control models. While space constraints prohibit us from a comprehensive survey, general discussions can be found in [4, 5, 9].

5. Developing Control Strategies in ORBS

The ORBS control model attempts to build on both YAPS and Hearsay III. From YAPS it borrows the notion of successive filtration. This provides a framework that is uniform and simple. This in turn makes it easy to build scheduling editors, tracers, and break routines. From Hearsay III it borrows the notion of control as a first-class citizen. Thus, ORBS treats the scheduling process as something that has its own representation, and is analyzable and modifiable.

---

[4] One of the authors, Fickas, has developed various large and small Hearsay III systems over the last five years.

To build a scheduler in ORBS, the user must 1) define a set of scheduling functions in Lisp or choose from a catalog of pre-existing functions, and 2) declare how the functions are to be combined to form a scheduling strategy. In this section, we show four examples of how control strategies may be developed in ORBS. The examples demonstrate progressive removal of control knowledge from a rule.

5.1. The YAPS Strategy

We begin with an example in which control knowledge and domain knowledge are intermixed. Specifically, we will build a scheduling strategy that emulates YAPS. Control knowledge, in the form of goal relations, appears throughout the domain rules for tasking purposes, as in the small-fast rule of figure 1.

YAPS uses the following control strategy:

(1) Using a history list, eliminate any activations that have been executed on a previous cycle. If all activations are eliminated, the system halts.

(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.

(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.

To model this strategy in ORBS, we use four scheduling functions: D2 eliminates already-applied activations using ORBS' activation history list; R5K prefers the rule activation(s) with the most recent "goal" fact; R5F prefers activation(s) with the most recent non-goal fact; and AD1Y arbitrarily selects among ties[5]. These functions, with all the scheduling functions proposed by McDermott and Forgy [9], are contained in the ORBS catalog.

We now must inform ORBS that 1) the functions D2, R5K, R5F, and AD1Y 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 [9] to accomplish this:

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

This defines a scheduling strategy that applies the function D2 to the conflict set and passes (as denoted by the ">") the resulting non-empty 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 halt the system. The function R5K is applied to the conflict set using "goal" as the keyword, the resulting conflict set 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.

5.2. The Hearsay III Strategy

The second example shows how we can begin removing tasking information from the LHS of ORBS rules, leaving only domain knowledge (see [4] for a related discussion). In figure 2, we defined a Hearsay III rule

[5]The "arbitrary" choice made in YAPS turns out always to be the first activation in the conflict set. This behavior is modeled by AD1Y. ORBS's catalog also provides a "true" pseudo-random, arbitrary function AD1, as defined by McDermott and Forgy.

that used "tasking levels" to represent the order that rules should be run. It did this by associating with each rule a specific task level; hence the firing of an activation of a rule can be considered carrying out part of its associated task. We can achieve similar control in ORBS through the use of user-defined rule attributes. As a start, we add a new attribute "sched-level" for all rules:

```
(add-rule-atts 'sched-level)
```

As each rule is defined, we can fill its sched-level field with the appropriate value. In the VLSI example, the levels we have chosen are start-up, choose-inverter, and choose-cell. We can now discard the goal patterns from the LHS of the ORBS rules (and can also discard the goal relations in the ORBS database). Thus our new small-fast rule is:

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

Figure 3

We're now half way home. We still have to define a scheduler that can use the sched-level field to order tasking. In Hearsay III, this was accomplished by writing code that would process the scheduling blackboard levels in the right order. We can do the same thing in ORBS by defining an appropriate scheduling strategy, as shown in figure 4.

```
[D2] > (L2 start-up) > (L2 choose-inverter) > (L2 choose-cell) > AD1
```

Figure 4

The scheduling function (L2 level) prefers activations of rules at the given level[6]. For example, L2 could be defined as follows[7]:

```
(defsched L2 (conflict-set level)
  (for activation in conflict-set
                  when (eq level (level-of activation))
                  collect activation))
```

A sophisticated system may be able to glean new control information while it is running. For example, the system might detect that it frequently encounters a situation where the work done by one rule is partly undone by a second rule. It would be advantageous, then, to be able to change the level of the first

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

[7]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 [9], is left implicit in the strategy. Subsequent arguments, such as a level, are stated explicitly. The function returns a list of activations, or nil if, for instance, no activation is produced for any rule at a given level.

rule to give it preference over the second rule. ORBS makes it possible to do this because a rule's level is simply an attribute which may be set or re-set at any time. Moreover, ORBS keeps track of these changes so that the system can perform at least partial backtracking. Therefore, in ORBS the level of a rule may be changed dynamically.

We could include a "meta" level in our list of rule levels in the VLSI example in figure 3, resulting in a new strategy:

[D2] > (L2 meta) > (L2 start-up) > (L2 choose-inverter) > (L2 choose-cell) > AD1

The meta level shows how the system could change the level of a rule dynamically. Rules which change control aspects of the system are designated as meta level; activations of these rules are preferred by the new strategy. Accordingly, we could define a rule:

```
(defrule change-rule-level
        (change-rule-level -r -newlevel -reason)        ; a fact has been entered into the database
                                                        ; suggesting that the level of rule -r be
                                                        ; reset to -newlevel due to reason -reason.
 test   (not-equal -r change-rule-level)
 ->
        (rule-set -rule 'level -newlevel))
 :::
        sched-level: meta                               ; level of this rule is meta because
        status: active)                                 ; it contains system control knowledge
```

Rule change-rule-level is triggered when a fact suggesting a change of rule level has been entered into the database. When rule change-rule-level fires, its RHS changes the level of the chosen domain rule to the suggested new level. Conceivably, change-rule-level could change its own level, but this is prevented by the test clause.

5.3. An Agenda-Based Strategy

The rule small-fast (figure 3, section 5.2) uses the attributes "speed" and "area" as discriminators for choosing the components of a ring oscillator. As the system evolves, we may discover other factors that should be considered, e.g., dollar cost, reliability, availability. We could attempt to attach them to the rule through the test slot as shown below:

```
(defrule choose-ro
        (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
        sched-level: choose-inverter)
```

This rule will choose a ring oscillator that exactly meets the test criteria. This, however, may be 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 is to use 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 defines a weight field for each activation). In this way, the value-testing predicates may be moved from the test portion of the small-fast rule (and all other rules dealing with component selection) to the scheduling functions. Hence we will have a scheduling function for each significant attribute—AREA, SPEED, and DOLLAR-COST, for example. Finally, we select from the catalog of scheduling functions 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 in sections 5.1 and 5.2, D2 eliminates all previously fired activations, and L1 does tasking by preferring activations of rules in order of the levels 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 functions above 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 level. The resulting conflict set is then passed to functions that choose the activation(s) with the greatest weight and, if necessary, choose an arbitrary activation.

W1 > AD1Y

The final small-fast rule is:

```
(defrule small-fast
        (ring-osc -ro)
-->
        (fact ro-cell inverter-3)
:::
        status: active
        author: simoudis
        sched-level: choose-inverter)
```

Thus as a result of the agenda-based strategy, control has been abstracted into the scheduling functions, and all the scheduling filters have been eliminated from the rule.

5.4. Changing the Strategy Dynamically

Dynamic control presents problems for ruled-based systems in which control knowledge is represented through relations in the LHS. In such systems, changing the control strategy requires redefining the rules. Abstracting control knowledge from the application facilitates dynamic control, and ORBS provides tools for doing this. In section 5.2, we saw an example of dynamic change to system control by changing the user-defined rule attribute "sched-level"; the conflict resolution strategy, however, remained unchanged. We now look at how the scheduling strategy itself may be changed dynamically in ORBS. That is, the strategy used to resolve the conflict set may be changed while a program is running, either by action of a rule or by the user at a break.

Here is a simple example showing a switch from an optimal-solution best-first search to a heuristic-pruning search in the face of time constraints. This would be useful if the expert system were interactive and a less-than-optimal solution would be acceptable when an optimal solution cannot be found within a reasonable response time.

In addition to the regular domain rules, we add the rule nearly-out-of-time:

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

In this example, the initial strategy is

[D2] > (L2 change-strategy) > ASTAR > W2 > AD1

[D2] prunes all previously-fired activations, and causes the system to halt when all activations have been fired. (L2 change-strategy) prefers rules at level "change-strategy" to domain rules. The test clause checks whether the remaining time -tr is less than a preset amount of time considered critical for the proper performance of the system. When time remaining is less than this critical level, then the rule's test is true and an activation will be produced. The activation of rule nearly-out-of-time will be selected over activations of all other rules in the conflict set because the L2 function precedes other functions in the initial strategy. ASTAR is a user-defined function implementing the A* algorithm (see [10]). ASTAR sets the weight attribute of each activation directly, based on cost and a conservative heuristic. Once ASTAR has set the weight attribute of all activations, scheduling function W2 selects the activations with the lowest weight. If the lowest weight in the conflict set is contained in more than one activation, AD1 selects one of these arbitrarily. Given that functions cost-so-far and cost-remaining are defined appropriately, the scheduling function ASTAR is defined as follows:

```
(defsched ASTAR (conflict-set)
        (for activation in conflict-set do
                (set-activation-attribute activation 'weight
                        (plus (cost-so-far activation)
                              (cost-remaining activation)))))
```

The new strategy fast-search could be set to

[D2] > (KW2 CMOS) > R5 > AD1

This strategy, in effect, searches using heuristic pruning. In this example, the new strategy is based on the idea that we know that a solution state exists using CMOS, although it may not be the best solution. Given the time constraint, however, we choose to pursue a possibly non-optimal solution. Note that the new strategy no longer contains the scheduling function (L2 change-strategy), which is now unnecessary because the rule it prefers has already been fired and need not be fired again.

6. Machine-Aided Construction of Scheduling Strategies

We have argued that the ORBS model of control gives users the flexibility to build complex control strategies, tailored to specific domain constraints. We have also seen how a catalog of scheduling primitives

can provide building blocks for defining new strategies (see [4] for a related discussion). In experimenting with ORBS, we have found that this is not yet enough support. Typically, a scheduling strategy evolves over time, just as the corresponding domain knowledge evolves. Indeed, one of the major problems we encountered building non-trivial systems in Hearsay III, which also provides a flexible scheduling model, was the lack of support in the construction, testing, debugging, and maintenance of performance knowledge. Tools exist for the incremental construction of ORBS' *domain* knowledge, e.g., editors, a break package, tracing routines. Why not provide the same support for the construction of scheduling knowledge?

Our answer in ORBS is to view the construction of domain and scheduling knowledge as co-equal tasks. Hence, we have defined a set of tools for the incremental construction of a scheduling strategy. The tools include

(1) A skeleton scheduler. Before any scheduling functions are defined, the user may use a skeleton scheduler to view, trim, and select activations from the conflict set interactively; that is, the user becomes the scheduler. Thus, the user can separate the task of defining domain knowledge from that of defining scheduling knowledge.

(2) A scheduling editor. Scheduling functions may be added to and deleted from the skeleton scheduler incrementally. Thus, partial strategies are possible; the user simulates the missing pieces.

(3) A scheduling tracer. The user may trace one, some, or all of the scheduling functions.

(4) A break package. The user may select one, some, or all of the scheduling functions as break points. When a broken rule is applied, the ORBS break package is called. From this point, the user can view system objects (e.g., the data base, rules, the conflict set), modify the current state (e.g., add a fact to the data base, call the strategy editor), or take over scheduling him or herself using the skeleton scheduler.

Our model of system construction in ORBS, then, is 1) define a prototype domain component incrementally, simulating scheduling functions for the time being; 2) gradually fill in scheduling functions as they become apparent and when the domain component becomes stable; and 3) incrementally tune both domain and scheduling components to the desired state. From our experience with Hearsay III and ORBS, we strongly believe that the separation of the two complex tasks into separately derivable pieces leads to the tackling of larger problems, faster implementation, and better systems overall.

7. Automating Strategy Construction

The environmental support of strategy construction provided by the current system can be described as "modern programming tools applied to the scheduling problem". While these tools have been useful, the construction process remains relatively unautomated. In this section we describe our recent efforts to remedy this.

The skeleton scheduler is a tool for stubbing out portions of a system while other sections are being built. It allows users to take over scheduling while defining domain knowledge. Once this is accomplished, the problem is one of translating user simulation into machine strategy. The current model relies on the users to supply the insight necessary to replace their strategies with existing scheduling functions (or to generate them from scratch if missing). This is a problem for two reasons: 1) users may not be able to see any coherent strategy in the scheduling choices they made, and 2) even if a strategy is discerned, users may not be able to map it onto the system's scheduling functions.

We are attacking the problem along two fronts. First, we are attempting to categorize the user's scheduling actions in a *scheduling vocabulary*. For instance, if the user chooses all activations of rule R for k cycles, but none after that, we might characterize the control as a while-do loop with to-be-determined exit condition C, or alternatively as a repeat-do loop with count k. This necessarily requires the system to

keep a history of not only the user's actions, but pertinent information on each scheduling cycle.

Second, we are attempting to map the scheduling vocabulary onto existing functions. In our revised model, then, users initially simulate control while a domain prototype is constructed, and then ask the system for help in taking them out of the scheduling loop. Our final goal is a system which can analyze users' scheduling actions and recommend a set of pre-defined scheduling functions for replacement. In its most general form, this is a problem of learning by example with all of the attendant difficulties.

8. Status and Future Work

A prototype ORBS system has been implemented on a VAX using the Maryland extension to Franzlisp [2].[8] This system successfully achieves our first goal, that of separating domain and control knowledge so that control strategies can be tailored to the problem at hand, and partially meets our second goal, that of supplying environmental support. Our current efforts include

- Broadening the system to allow more straightforward construction of backtracking control strategies. In practice this means a closer modeling of the Hearsay III scheduling blackboard and context mechanisms.
- Continuing work on automating the strategy construction process, such as the learning by example problem discussed in the last section.
- Improving organization of the scheduling component catalog. This must include a vocabulary for talking about control problems.

To summarize, we believe that control in rule-based systems must be treated as a difficult problem. The solution requires a separate representation, reusable components, and tools to support testing and debugging.

Acknowledgments

Other past and current members of the ORBS project include Michael Hennessy 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.

[8] Current work on the system is being carried out on a Symbolics 3600.

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] Bobrow,D., Stefik, M.
The LOOPS Manual,
Xerox PARC, Palo Alto, 12/83

[4] Clancey, W.
The Advantages of Abstract Control Knowledge in Expert System Design,
Tech Report HPP-83-17, Computer Science Dept., Stanford, 11/83

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

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

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

[10] Nilsson, N.
Principles of Artificial Intelligence,
Tioga Publishing Co., 1980

[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.