CIS-TR 85-06
Design Issues in a Rule-Based System

Stephen Fickas, David Novick

Department of Computer and Information Science
University of Oregon

This paper appeared in Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments, Volume 20, Number 7, July 1985.

Design Issues in a Rule-Based System

Stephen Fickas
Computer Science Department
University of Oregon

1. Introduction

This paper discusses a language and associated environment for building rule-based programs. The language and environment are encapsulated in a system we call ORBS (Oregon Rule Based System). In tune with this conference, our focus will be on the interplay between language and environment design. However, we will broaden this somewhat to include design constraints placed by our program development model[1] as well. Instead of attempting a complete design rationalization of ORBS, we will concentrate on design decisions that highlight the coupling between language, environment, and development model.

ORBS falls in the class of rule-based systems that often is categorized as forward-chaining or data-driven. Other systems in this category include OPS5 [9], YAPS [2], and all of the Hearsay family [6, 11, 15]. What is often taken as the inverse of this set are the backward-chaining or goal-directed languages such as EMYCIN [17]. Arguments for and against the forward and backward approaches have been made many times elsewhere (see [12] for examples of each), and we will not address the issue further in this paper.[2] While we believe that many of the arguments we make for the design of ORBS are applicable to rule-based systems in general, any reference to the term *rule-based* should be taken in light of the classification above.

It is often difficult in a tightly coupled system such as ORBS to separate language from environment, or even the ORBS environment from the Lisp environment on which it rests. Further, while we use the term *rule-based* to describe ORBS, an ORBS program may make use of frame-based, procedural, and message-passing languages in carrying out a computation. Complicating matters further, an ORBS program may be embedded in a program written in any of these other languages. Some of the newer AI languages argue (see for instance, LOOPS [4]) that this is just right; complex programs often do not fit into a single language, but instead will need a variety of representations. We have two things to say about this view. First, we believe it. Our experience in building programs solely with a rule-based language such as OPS5 was one of attempting to by-pass its rather rigid and limited representation. In the same vein, we have felt the same restrictions when building purely logic-based, procedural or object-oriented programs. Second, not much thought has gone into the effects of throwing together a language salad. How will each language interact with the others? What is the development model with such a conglomeration? Because ORBS is part of such a salad, we have found ourselves constantly running up against these questions. We attempt to come up with a few answers in the following sections.

[1] We use the word *development model* here and throughout the paper to refer to the construction of application programs using ORBS, as opposed to the development of ORBS itself.

[2] More recent efforts have attempted to combine aspects of both (c.f., [14])

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the

We finally must note that our work builds on much good work before us. Part of our goal in the construction of ORBS is to better formalize and support the good ideas of past rule-based systems. By the same token, we have spent frustrating sessions dealing with their ill-designed features. We hope to have learned from these experiences as well.

2. Design Prejudices

Our experience with building rule-based systems has lead us to a collection of design prejudices. We state them here not because they are new — most have been stated as part of other language efforts — but because they show our biases.

The development of rule-based programs is best suited to an incremental, interactive approach. Stated in the negative, we do not believe a traditional edit/load/run cycle is conducive to the construction of rule-based programs. While this point may now seem obvious in AI systems, we note that many existing rule-based systems are based on some form of a fixed edit/load cycle.

Mapping domain knowledge to rule representation (i.e., knowledge acquisition) is difficult[3], *and should be supported.* Some researchers have focused on eliminating the mapper (A.K.A. knowledge engineer), and allowing the domain expert to directly add, modify and debug rules. Our approach is to look towards some of the newer work in knowledge-based software development for help. For instance, part of the ORBS environment is concerned with the *specification* of rule-based programs.

A problem-independent language (such as ORBS) should provide general language constructs. Languages designed around specific problem domains (e.g., cognitive modeling) do not often travel well outside of their restricted area. However, the use of general language constructs places a greater burden on the user to represent his or her problem-dependent knowledge. The more general the language, the less help it is in modeling specific domains.

Reuse can mitigate complexity. Given general language-constructs, the programmer is faced with the complex task of defining more specific domain concepts. We have attempted to lessen this problem by supplying the programmer with catalogs of program pieces that have been indexed around domains

[3] Known in some circles as the Feigenbaum bottleneck.

publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1985 ACM 0-89791-165-2/85/006/0208 $00.75

and problem types. Instead of starting from scratch on a new problem, the programmer may be able to load in a skeleton that contains objects, rules and control knowledge from a related problem. The programmer's task is then to tailor the skeleton to the problem at hand. Section 4 discusses this use of used components in more detail.

The rapid construction of partial programs is particularly useful in the ill-defined domains that we find most appropriate to a rule-based approach. This rapid prototyping issue overlaps with the design of both the language (e.g., strong vs. weak typing) and the environment (e.g., component cataloging).

As can be seen above, our development model is based on 1) rapid prototyping, 2) reuse of components, and 3) a process from the traditional software lifecycle, specification. We will attempt to show how each has had an impact on our design decisions.

3. The ORBS Language

ORBS is both a language and environment for building rule-based programs. A program written in ORBS contains a set of tuples in a data base, a set of condition/action rules that trigger on those tuples, and a set of scheduling functions that determine which triggered rules to execute. Figure 1 conveys this graphically.

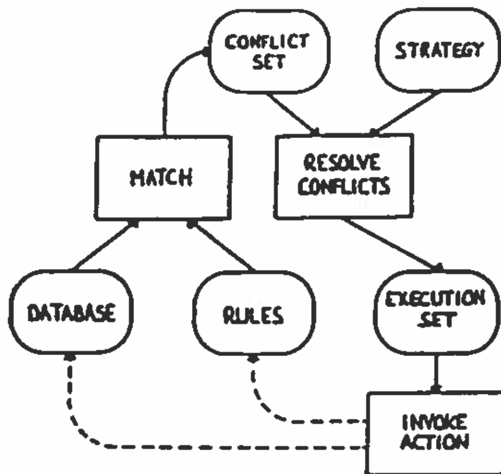

Figure 1

An example of a rule embodying a piece of circuit design knowledge is shown in figure 2. The rule is taken from a larger ORBS program, written by Vangelis Simoudis, that automates portions of a silicon compiler. We have chosen this rule because it is simple, and it demonstrates some of the interaction between rules, frames, and message-passing in ORBS. The danger of viewing such a rule in isolation is the pigeon-holing of the system as a particular kind of problem solver, e.g., classification, simple synthesis. As we have stated earlier, our goal in ORBS is to handle a wide-range of problem types. Simoudis' compiler, for instance, uses two separate ORBS systems, one doing constraint-based problem solving, and the other design-rule checking.

If you are building a ring oscillator that must be both small and fast, then use inverter-type-3 as the basic building block.

```
(rule small-fast
       (ring-osc -ro)
 test  (eq (send -ro 'speed) 'fast)
       (eq (send -ro 'area) 'small)
 -->
       (fact cell-choice -ro inverter-3)
 :::
       (task: choose-inverter)
       (status: active)
       (see-for-ref: simoudis))
```

Figure 2

The LHS (Left Hand Side) is made up of one or more clauses that are matched against data base tuples. Above, the LHS consists of the single clause (ring-osc -ro). Pattern variables are allowed in a clause (they are distinguished from literals by a prepended hyphen). The (optional) keyword "test" marks the beginning of the filtering predicates. In this case we assume that -ro has been bound to a frame object (discussed in more detail in the next section) with attributes speed and area. The function send passes a message to the object, asking in this case for the values of its speed and area attributes. The "-->" marks the beginning of the RHS (Right Hand Side) actions. Each action is a lisp expression; actions are evaluated sequentially. In the above case, the action adds a fact to the data base. As with the filtering predicates, an action can be any Lisp expression. The ":::" marks the beginning of rule attributes (task, status, and see-for-ref in this case). Attributes are gettable and settable from both the system and the application program.

In the next 3 sections, we will discuss each of the major language components -- database, rules, control -- in more detail. In each section, we will first give an overview of the component, and then the major design rationale.

3.1. The data base

An ORBS database consists of zero or more tuples, often called *facts* for historical reasons. Each fact can take as arguments either Lisp objects or frame objects. For example, four database facts are given below.

```
(location sue office)
(actions (go stop continue))
(hit {person} {person})
(ring-osc {ring-oscillator})
```

The notation {"name"} represents the instance of a frame object. Frame objects are defined in the Moss language, which is based on Strobe [16], and is another component of our integrated system for building knowledge-based programs. A frame object implemented in Moss (henceforth known simply as a frame) has one or more slots. For example, the ring-oscillator frame from our small-fast rule contains the slots x-dimension, y-dimension, area, and speed.

Each slot of a frame has one or more facets, either user defined or system defined. Facets can be used to assign default values, do procedural attachment, and of particular interest, define data type information. In the latter case, the user can define arbitrarily complex "type predicates" through the DATATYPE facet of a slot; any attempt to assign a value to a slot must first pass any predicates attached to the DATATYPE facet.

Frames themselves are related by inheritance: a frame may have zero or more parents and zero or more children. Slots and facets are inherited down ancestral chains. For example, a portion of the ring oscillator frame has the following structure:

```
Frame ring-oscillator
    (super-frames circuit-component graph-object)
    (sub-frames nil)
    (author simoudis)
    (created July 2, 1984, 2120)
  (slot speed
    (facet value)
    (facet DATATYPE speed-class)
  (slot area
    (facet IF-NEEDED calculate-area)
    (facet DATATYPE area-class)
  (slot x-dimension
    (facet value)
    (facet DATATYPE real)
  (slot y-dimension
    ...)
  ...
```

The above description of Moss is both simplified and incomplete. Our objective is to introduce enough of the language so that the following discussion can be understood. In particular, we find the interesting topic not the language itself, which adds little to frame-based language research, but the way it interacts when tied in to a rule-based system.

Finally, it is important to note that ORBS is implemented in Moss, and in fact, is a Moss frame object itself. In this way, ORBS can be viewed as one resource of a knowledge-based program development system. In particular, multiple instantiations of the ORBS system are possible, allowing separate ORBS systems to be inserted into a larger application program. For instance, one ORBS system can reside in the database of another ORBS system (as an argument of a fact) or be attached directly to a frame slot. We will see examples and discuss the ramifications of this later.

Having finished our overview of the ORBS database, we will now look at the major design decisions that lead to its current form.

The ORBS database is weakly typed.

We can contrast this to database languages that provide fully typed objects and relations. The Expert System language Hearsay III provides such typing[4]. In the following discussion, we will use Hearsay III as the embodiment of the fully typed approach. We view the design issues as follows:

- By supporting a full type lattice[5] extended to include user-defined type-checking code, Hearsay III gains all of the attendant error detection capabilities, e.g., an attempt to add an undefined relation or one with a mis-typed argument is flagged immediately.
- The Hearsay III approach allows more succinct database queries. By using typed pattern variables, we can ask for all tuples that contain person objects, or just tuples that contain a subtype of person, e.g., students.
- The Hearsay III type structure allows domain concepts to be organized around class hierarchies.
- Our experience with Hearsay III, and in a more limited sense with OPS5 keyword declarations, shows that considerable time must be spent in getting the type lattice right. It is seldom the case that the right set of types and subtypes can be determined without prototyping and experimentation.
- Hearsay III provides no clean way to continue processing once a change to the type lattice is made. Instead, the entire system must be recompiled, loaded and run (Goodwin discusses related problems in [10].).
- Our experience with weakly typed languages, such as YAPS and ORBS, is that initial prototype programs can be built quickly. These early programs are of enormous help in defining the right frame classes and subclasses.

Our choice of an untyped database was based finally on three concerns. First, we are advocating an interactive, incremental model of development much like that proposed by Interlisp and its successors. If we chose to use the Hearsay III model, then we would have to find a way to make changes to the type lattice in mid-computation so that processing could continue (a capability strongly advocated by Goodwin [10]). We note that at least one promising approach to incremental type definition is DUCK's "walk" mode [14], which allows a type hierarchy to be defined and modified dynamically.

Our second concern is with rapid prototyping. Experience with ORBS has shown that initial (albeit incomplete) versions of a system can be brought up quickly. In Hearsay III, our experience was the opposite: much of development time was spent in defining and debugging the type lattice. Even initial prototypes took substantial effort to bring up.[6]

Finally, we find that the incorporation of frames within the database partly compensates for weak typing. A major contribution of the type lattice in Hearsay III was the organization of domain objects into a class hierarchy. Frames, lacking from Hearsay III, provide this capability as well. Further, the DATATYPE facet in Moss allows us to type any slot we wish. Thus, we can choose certain slots as critical, leave others untyped, and change our mind as development progresses.

Given our choice of weak typing, we do lose some functionality. For one, we cannot as elegantly query the database using the type lattice as a filter. Instead, we have to rely on filtering predicates to narrow down the objects and relations we are interested in. Secondly, type errors caught immediately in Hearsay III may not manifest themselves until late in a computation in ORBS. If we misspell a database relation (e.g., ring-ocs), its effect might be that a rule that should have matched will not match. Unraveling this kind of error is clearly difficult. This is one point where our choice of a language feature has affected our corresponding environment tools. Specifically, we have spent considerable effort building tools that will allow these errors to be pinpointed and fixed. As an example, the user may query why a particular rule did not trigger. The system will answer with one or more pattern clauses that failed to match, and/or predicates that returned nil (the system keeps a history of such events). The user can further inspect a rule by asking the system to match a specific database fact against a specific LHS clause. The matcher will report any argument mismatches. Once a bug has been found, the user can correct it and ask the system to revert to a previous cycle and try again. .

One final note relating to database typing: in practice, we find that we use the ORBS database as a means of organizing frames (The KEE system [13] has taken this to the extreme by eliminating the database altogether, instead relying on the system to manage and index frames.). Tuples often represent indexes as opposed to

[4] Hearsay III is based on the relational database language AP3

[5] Hearsay III allows a subtype to have more than one supertype with the type ENTITY acting as the universal element

[6] We must be fair to Hearsay III here. Rapid prototyping was never a goal of the language. Quite the opposite, the development model assumed that Hearsay III would be used for large problems where much time would be spent on defining relations and types offline before being translated to code

domain concepts. The latter are now represented as frames and their associated slots. A good example is taken from the small-fast rule, where the complex ring-oscillator frame is indexed in the database by a simple relation

```
(ring-osc (ring-oscillator))
```

Given this simple use of the database, we feel that typing is where it belongs: in the language that models domain concepts, i.e., frames.

The database is unstructured.

Languages like Hearsay III and BB1 [11] provide the notion of problem solving levels for both domain and control knowledge. As pointed out in [11], this provides a powerful organizational abstraction in certain planning domains. In Hearsay III, database levels are actually classes defined as part of the typing mechanism. That is, level X is defined by defining a class X; newly created objects can be placed in this class, i.e., placed at this level. Rules (Knowledge Sources) could also be placed in classes in the same way. In BB1, levels have a second, temporal dimension. This can be used to track alternative solution paths.

We have decided against an explicit representation of database structure in ORBS. Our reasons are twofold. First, the introduction of levels forces a particular abstract planning view of the world. Objects in the database must have an associated abstract level; relations are used to mark inter-level connections. While this is natural for some problem domains, it is artificial and counter-productive in others.

Secondly, we find that we can simulate the level notion, when needed, by more primitive ORBS mechanisms, e.g., typing of frame objects, explicit level arguments in database relations.

Does this mean that an unstructured database is best? We would answer no, but any structure added appears to be problem dependent. This is somewhat of a dilemma in ORBS, which aspires to problem independence. Our solution is *not* to add new language features, but instead to rely on cataloging of skeleton applications in representative domains. In this way, we can configure the database structure to match the problem type. For instance, if we wish to build a planning system, we may load in a skeleton that contains frames, rules, and scheduling functions that handle multilevel planning. An example of this type of dynamic system configuration is given in section 4.

3.2. Rules

An ORBS rule contains one or more LHS patterns, zero or more filters, one or more RHS actions, several system-defined fields, and zero or more user-defined fields. The ORBS match/schedule/execute cycle is as follows:

(1) Match each rule against the data base. Separate *activations* are created for each different match.

(2) Make sure each activation passes any filtering predicates. Ones that do are placed in a conflict set.

(3) The conflict set is passed through a set of scheduling functions (previously defined by the user).

(4) The outcome is an execution set containing zero or more activations, which are chosen for execution.

(5) The cycle repeats until explicitly halted by the application program.

As an example of rule syntax, we repeat the rule from figure 2.

```
(rule small-fast
      (ring-osc -ro)
 test (eq (send -ro 'speed) 'fast)
      (eq (send -ro 'area) 'small)
 ->
      (fact cell-choice -ro inverter-3)
 :::
      (task: choose-inverter)
      (status: active)
      (see-for-ref: simoudis))
```

Our model for rule structure comes from a hybrid of YAPS and Hearsay III. We have striven for a readable syntax that allows extension. We have made the following design choices in defining our rule representation.

Pattern clauses are separated from filtering predicates.

Only literals and wildcards may appear in a pattern. More complicated predicates may be applied in the optional test section. This is in contrast to systems such as OPS5 which allow predicates to be embedded within a LHS clause. We argue for separation of clauses and predicates on two grounds: 1) additional syntax must be introduced within a clause to describe embedded predicates, syntax which potentially introduces new bugs, and often makes clauses difficult to understand, and 2) by separating matching and predicate application, we have separated two debugging concerns. In particular, the ORBS environment contains tools that allow a user to trace/break the matching of the pattern clauses of a rule *and/or* the satisfaction of its predicates. We have found this capability quite valuable in rule debugging. In contrast, systems like Hearsay III and OPS5, which intermix matching and predicate testing, are found to be more difficult to debug; more time is spent on narrowing the error to one or the other. Thus, to provide the type of debugging tools needed to overcome the weakly typed database (see the previous section), the language must separate matching from filtering.

Rules should be packageable.

This addresses both language and development model issues. Systems such as LOOPS allow rules to be grouped into related collections. There is generally an activity pointer that designates what collection is current. Only rules from the current collection are allowed to match. AGE [15] uses pattern-directed invocation of a rule package. There are at least two attractive language properties of rule packaging: it allows a complex space to be broken into more understandable pieces; it tends to simplify rule construction and debugging by alleviating context issues (c.f., [1]).

ORBS currently contains only a weak and implicit form of rule packaging. Specifically, rules can be collected together by placing appropriate values in their attribute fields. For example, we could define a rule attribute *group*, and set it to *interfacing* for each of the circuit design rules we wished to package together. This is more or less the approach taken by Hearsay III, grouping rules by attributes. While this provides a method of defining the elements of the set, it does not provide a means of defining attributes of the set itself, e.g., when should it become active, what problem does it solve, what signals the end of activation.

There are two other aspects of rule packaging in ORBS that are worth mentioning. The first is tied to our desire to reuse components. One of our design goals is to make program construction

easier by allowing the programmer to use components found to be useful in the past. As a pertinent example, Simoudis' collection of rules dealing with circuit layout problems may be useful in a new problem dealing with space planning within the CS department. We would like to reuse the circuit layout rules in the new space planning system. As we will see in section 4, one of our environment tools allows rules to be packaged and retrieved for use in new programs.

Another type of rule packaging in ORBS is related to the ability to instantiate multiple copies of the ORBS system. In this way, rule-sets can be defined separately by placing them in separate ORBS systems. For instance, Simoudis' program uses one ORBS system to edit circuit layouts, and a separate ORBS system to do design rule checking. Each system has its own database and control structure. Communication is through message passing.

In summary, rule packaging comes at various levels. ORBS provides the ability to package rules as part of the development environment so that they can be used across programs, and the ability to package rules into separate programs, but provides only limited packaging of rules within a single program. One of the problems with extending the latter is its interaction with the ORBS control model discussed in the next section.

3.2. Problem Solving Control

The ORBS scheduler is called when all rule activations have been gathered for the current cycle, forming what is known as a conflict set.[7] The scheduler uses a *control strategy* to choose among competing solutions, and to order sub-tasks within some larger task. A control strategy consists of a directed acyclic graph with *scheduling functions* as nodes. The scheduling process is carried out by passing the conflict set along arcs and through nodes. Each scheduling function takes as input a set S of rule activations, and returns none, some or all of S. Generally, a function either removes or weights one or more activations. The new set is in turn passed along arcs to other scheduling function. The process completes when the last scheduling function (as designated by the user) returns zero or more activations for execution. In figure 3, we see a simple strategy that 1) passes the initial conflict set to function R2, 2) splits the output of R2 to COST and ENDURANCE, 3) takes the union of the two outputs, 4) passes the resulting set to AD1, and 5) passes the output of AD1 to the system rule-executer.

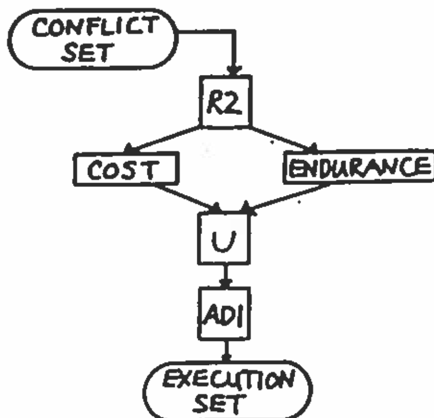

Figure 3.

[7]This is a historical name that is no longer entirely accurate. In particular, there need not exist any conflict within the set: all activations may be executable on the current cycle.

Several major design decisions went into the ORBS control model. They are listed below.

ORBS uses a state-based as opposed to an event-driven matching algorithm.

In an event-driven system, rules are matched by the occurrence of some database event, typically adding a new fact. For example, in Hearsay III, a knowledge source is matched on the insertion of a new relation on the blackboard. Because an event is seen only once, such systems must keep a record of rule matchings to avoid "losing" a rule that was not selected for execution on the same cycle that it matched. In Hearsay III, rule activations are placed in a permanent data structure called a scheduling blackboard. On each cycle, the Hearsay scheduler must weed through both rules that matched this cycle, and all waiting rule matches from previous cycles. Other AI systems use a similar structure called an agenda to hold pending tasks.

ORBS uses a state-based matcher, which matches strictly on the state of the database, and not on any addition or deletion actions. Conceptually, the entire conflict set is generated anew on each cycle. Hence, almost no history is needed between cycles. The system does record how many times a particular rule has been chosen for execution.

Our rationale for selecting a state-based approach is based on two concerns. The first involves the interaction between rules and objects that can cause side-effects. Specifically, the addition of frame objects and procedures into a rule-based system allows certain events to go unnoticed by the rules. For instance, a rule has no means of detecting a frame slot-changing event. In the state-based approach, we can build rules that constantly poll relevant frame objects to check for certain key values being present.[8] For instance, suppose that we had a system that monitored a person's age:

```
(rule birthday
      (person -p)
      (date -month -day -year)
 test (eq (send -p 'birth-month) -month)
      (eq (send -p 'birth-day) -day)
 -->
      (send -p 'set-age (+ (send -p 'age) 1))
 :::
      (task: monitor)
      (status: active))


(rule legal-voter
      (person -p)
 test (geq (send -p 'age) 18)
 -->
      (fact task mail-voter-material -p)
 :::
      (task: monitor)
      (status: active)
      (see-for-ref: federal-voting-law))
```

In the above example, we have used a frame to record relevant information about a person. If we used an event-driven matcher, legal-voter would not see a change to the database, and hence would not trigger. Of course, we could modify the birthday rule so that it left a marker in the database that signalled a birthday

[8]This is not quite as expensive as it sounds; ORBS uses an incremental matcher that avoids rematching the pattern of each rules every cycle.

event. The legal-voter rule could likewise be modified to trigger on this. Below, we will propose a more general approach to this type of event signalling as one way of more usefully coupling rules and frames.

In a state-based matcher like the one ORBS employs, the single clause of legal-voter would match every cycle; it is up to the test predicate to check for the relevant change in the person's age. We can use a scheduling function to guarantee that legal-voter will execute only once (on a person's 18th birthday), and not continue to send voter pamphlets for the rest of a person's adult life.

The second problem deals with monotonicity. In an event-driven system, a rule R may trigger on facts known on cycle i, but R may not be chosen for execution until cycle i+k. In the k intervening cycles, new facts may be added and old ones deleted or modified. Hence, a triggered rule may become out-of-date while it is waiting. The Hearsay III approach is to attempt to "rematch" an activation right before it is executed; if it fails to rematch, it is ignored. A state-based matcher does not have to deal with obsolete activations. The execution of an activation uses the latest facts known. However, neither matching scheme solves the problem of non-monotonic reasoning in general, e.g., what to do with a fact/hypothesis resting on now erroneous data.

There are other problems with the state-based approach. First, the polling strategy above only can detect the after-effects of an event, not the event itself. Thus, if we want to know any time a slot value changes, regardless of its new value, we have no clean way of encoding a corresponding rule. Second, it is difficult to build agenda-based problem solvers, where a record of pending and competing tasks is kept from cycle to cycle. We see several possible solutions.

Solution 1: Unfold the rules

We might "unfold" rules like legal-voter into the slots themselves. Thus, we could attach a rule to the age slot of a person frame that checks everytime it is changed. This moves us back to an event-driven model. In fact, the Mss language allows such procedural attachment. However, we try to avoid it for several reasons. First, the dispersement of rules throughout a set of frames makes it hard to generalize, catalog, and reuse interesting pieces of procedural knowledge. In our experience, it leads to slot-dependent and duplicate procedures. While some types of procedures do belong with a slot, and not as a part of the rule base (e.g., graphics routines), many others are ill-suited as attached procedures.

Second, ORBS provides a sophisticated and extensible control model to schedule competing tasks and subtasks. It is unclear how this model will be extended to handle the many mini rule-bases spread throughout the frame space. For instance, the change of the age slot to 18 may trigger two competing attached procedures, one advocating a medical career, one advocating a law career. The only way we now have of handling this type of problem is to attach an entire ORBS system to an individual slot. Such an attached rule system can be executed on specified slot-changing events. Thus, the person frame could be modified so that an ORBS system is attached to the age slot. When the person became 18, the ORBS system could be called on to do the necessary problem solving to determine a career path.

Solution 2: make frame modification an observable event

An event-driven matcher becomes feasible in ORBS if rules can detect not only the addition and deletion of relations, but the change of slots as well. Thus, we can retain our explicit rule base, but at the same time allow rules to be associated with one or more slots. Hence, we might define the following security rule:

```
(rule secure-slot
      (*changed* -slot -frame)
 test (eq (send -frame 'type) 'secure)
 ->
      ((fact security-alert -slot -frame))
```

Here, the system must monitor all slot modification, and relay the information to the rule base, hypothesized above as a special relation *changed*. As another example, we can add a clause (*changed* age person) to legal-voter to monitor a person's age slot. One part of our current work on ORBS is to integrate both state-based and event-driven rules.[9] Major issues involve rule representation, event monitoring (including side-effects wrought by procedures) and notification, and extension of the control model to handle event-driven activations.

In summary, we find rules, frames, attached procedures and object-oriented programming interact in complex ways. In ORBS, we view the rule base as the primary repository for procedural knowledge. By having access to the rule base as a whole, we can reason about this knowledge, e.g., generalize it, detect bugs, catalog it.

Control strategies are tailorable to the problem.

Different problems require different control strategies. Many existing rule-based systems provide only a fixed control strategy. While it is sometimes possible to implement different control strategies using a fixed strategy 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 to explicate the implicit control decision points in a rule-based system, and to make all control points accessible to and modifiable by the user or application program [8]. Specifically, our experience with building rule-based systems in languages such as OPS5, YAPS, PROLOG, and Hearsay III has 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.

In ORBS, a control strategy is user-defined. For a user to build a scheduler in ORBS, he or she must 1) define a set of scheduling functions, and 2) define how those functions are to be combined to form a control strategy, i.e., define the arcs of the graph. In ORBS, the environment supports this construction process. ORBS maintains scheduling components found useful in previous systems. These components can range from individual scheduling functions to larger pieces of control strategies. ORBS also supplies a "manual" scheduler, which allows the user to incrementally build

[9] The current system only allows state-based rules.

up a control strategy.[10] Initially, the manual scheduler forces the user to make all scheduling decisions (e.g., select the activations from the conflict set to be invoked). As time passes, the user may add in scheduling functions, but still remain in the scheduling loop. Eventually, the complete graph will be built, forming a coherent strategy. At this point, the user has been completely removed from the loop. One of the members of the ORBS group, Keith Downing, is looking at ways that this process might be automated by examining the choices made by the user, and comparing them to the known scheduling functions. The goal is for the system to recognize the strategy being carried out manually by the user, and replace the manual scheduler with the appropriate set of scheduling functions.

This ends our discussion of the ORBS language components. We next take up the environment that surrounds them.

4. The ORBS Environment

The ORBS environment includes the following components: 1) a specification language, KATE, for specifying rule-based systems [6,7], 2) graphical editors for modifying rules, flavors, facts, control strategies, 3) a break package that supports incremental development, and 4) a catalog of useful domain and control pieces that can be melded together to build new programs.

We believe the best way to discuss the ORBS environment is by following a small example of a programmer building an ORBS program. Suppose that we wished to build an ORBS program that laid out lab, office, and storage space in a Computer Science department. Our first step would be to use Kate to specify the components of the problem. Our overall goal for Kate is to allow rule-based systems to be specified by a combination of component reuse and tailoring. Kate maintains a catalog of skeleton components from past development efforts. These skeletons are currently handcrafted to represent the "essential" pieces of an ORBS program for some particular problem. Each skeleton contains a set of frames, a set of rules, and a control strategy. These pieces generally do not constitute a complete program. Instead, they are crafted as components to be mixed in to a larger system. Two skeletons of interest in our layout problem are CS-Dept-World and Circuit-Design-World. The first contains frames (Faculty, Staff, Student) and rules ("schedule department meetings at 1600 on Tuesdays") dealing with department life. The second contains a control strategy that is suited to a constraint-based planning problem.

Once these are loaded, the Kate editor allows unwanted pieces to be trimmed (e.g., the Circuit-Design-World frames), relevant pieces to be further refined (e.g., Circuit-Design-World scheduling functions), and missing pieces to be added (e.g., additional space planning rules). While our current research effort is aimed at supplying these editing operations at the level at which they are stated above, currently only primitive addition, deletion, and modification actions are available. Kate does present each of the components graphically: frames are shown as an inheritance graph; rules are shown in component form; the control strategy is represented as a graph of scheduling function nodes and connecting arcs. Each of these can be edited graphically as well. For instance, the user can add and delete nodes and arcs from the control strategy by manipulating representative graph icons. The graph is also used as a tracing tool by marking and highlighting icons as the scheduling process executes.

[10] The control strategy can be modified at the beginning of any cycle. This may be done by the user for debugging purposes, or by the running program as part of the application.

Once we have tailored our frames, rules, and control strategy to match the layout problem, we are ready to test the program. We will use the ORBS break package to do tracing and as a platform from which to patch bugs. The break package allows us to set breaks as follows:

- Break on <pattern> being added to or deleted from the data base.
- Break when <rule pattern-clause(s)> matched.
- Break when <rule test-clause> fails.
- Break when <rule> matched.
- Break when <rule> selected for execution.
- Break after <rule> executed.
- Break every cycle.
- Break before scheduling functions run, after scheduling functions run, after a particular function run, on empty execution set.

Because this is the first test of our program, we will cautiously place breaks at the beginning of every cycle, and at every scheduling function. Suppose that our first break occurs at the first scheduling function. We can interrogate the function for its input, in this case the initial conflict set. Suppose that it is empty. Its likely we have a bug since no rules matched on the first cycle. From this break we can do the following:

- Pretty Print the conflict set, data base (on any cycle), rule, process history.
- Add a fact to, delete a fact from, or destructively edit a fact in the data base.
- Add, delete, or edit a rule.
- Remove activations from the conflict set (not applicable in this case).
- Determine why <rule> did *not* fire in cycle k. Answer is either "did not match" or name of scheduling function that eliminated <rule> from the conflict set.
- Match <rule> against {<fact>}. User chooses subset of facts to match against <rule>. Answer is either "match" or LHS pattern(s) that failed to match.
- Single step the scheduling functions. The break package is called after each scheduling function returns.
- Print process history. This includes, on a cycle by cycle basis, what facts were added, what facts were deleted, what activations were executed.
- Revert to beginning of previous cycle k (the system will restore the database and rules to their original form, but not associated frames!).
- Continue processing.

We have several choices here. First, we can attempt to determine which rules should have matched and ask the system what caused them to mismatch. From this information, we may wish to edit a rule or the database, revert to the beginning of the first cycle, and continue processing from there. Debugging continues in this break/fix/continue cycle.

5. Summary

In this paper, we have attempted to show the interaction between the design of the ORBS language, and 1) the development model used to build ORBS programs, and 2) the environment that surrounds the language. We particularly note the difficult problems brought about by combining various language forms — frame-based, procedural, message-passing — with a rule-based system. We have rationalized ORBS language design in terms of the above issues.

In regards to ORBS as the pinnacle of rule-based language design, we note that all of our system components — environment, development model, Mom, Kate — are under constant revision as new design problems are uncovered. This leads us to believe that the current incarnation of the language is far from mature. We expect much hard analysis and experimentation lie ahead of us.

The ORBS system is implemented on the Symbolics 3600. A non-graphics version of the system is implemented in Franzlisp using the Maryland Flavors package [3] under Vax Unix 4.2. The system is also being ported to a Tektronix 4404 AI Workstation (Pegasus).

Acknowledgments

Past and current members of the ORBS project include Allen Brookes, Kim Dannewitz, Keith Downing, Michael Hennessy, David Novick, Rob Reesor, and Bill Robinson from the University of Oregon, and Bill Bregar from Oregon State University. Vangelis Simoudis has been a valuable first user of the system.

We would also like to thank two former members of the Hearsay III design group, Lee Erman and Phil London, for their comments on earlier drafts of this paper.

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

References

[1] Aikins, J.
Prototypes and production rules: a knowledge representation for computer consultations,
PhD thesis, Computer Science Dept, Stanford University, 8/80

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

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

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

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

[6] Fickas, S.
Specification Automation
International Workshop on Models and Languages for Software Specification and Design, Orlando, 4/83
Available from Departement d'Informatique, Universite Laval

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

[8] Fickas, S., Novick, D.
Control in Rule Based Systems: Relaxing Restrictive Assumptions,
In *5th International Conference on Expert Systems and Their Applications*, Avignon, 1985

[9] Forgy, C.
OPS5 Reference Manual,
Computer Science Dept., Carnegie-Mellon University

[10] Goodwin, J.
Why programming environments need dynamic data types,
Interactive Programming Environments,
Eds. Barstow, Shrobe, Sandewall, McGraw Hill 1984

[11] Hayes-Roth, B.
BB1: an architecture for blackboard systems that control, explain, and learn about their own behavior,
HPP 84-16, Heuristic Programming Project,
Stanford University, 1984

[12] Hayes-Roth, F., Waterman, D., Lenat, D.
Building Expert Systems,
Addison-Wesley, 1983

[13] The Knowledge Engineering Environment,
IntelliCorp, 707 Laurel Street, Menlo Park, Ca. 94025

[14] McDermott, D.
DUCK: A Lisp-based Deductive System,
Department of Computer Science, Yale University, 1983

[15] Nii, H., and Aiello, N.
AGE: A Knowledge-based Program for Building Knowledge-based Programs,
In *Proceedings of 6th International Conference on Artificial Intelligence*, 1979

[16] Smith, R.
Structured Object Programming in STROBE,
Schlumberger-Doll Research, Ridgefield, Ct., 1984

[17] van Melle, W.
A domain-independent production-rule system for consultation programs,
In *Proceedgins of 6th International Conference on Artificial Intelligence*, 1979