CIS-TR 85-10

An Environment for Building Rule-Based Systems: An Overview

Stephen Fickas, David Novick, and Rob Reesor

Department of Computer and Information Science
University of Oregon

AN ENVIRONMENT FOR BUILDING RULE-BASED SYSTEMS: AN OVERVIEW

by Stephen Fickas, David Novick, and Rob Reesor

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

Summary

This paper presents an overview of the Oregon Rule-Based System (ORBS), an environment which supports the construction and development of domain knowledge and control knowledge for rule-based expert systems. We describe the operation of the system, with particular attention to issues of control knowledge. The paper then presents an interactive model for development of application systems. Finally, the paper discusses KATE, a knowledge-based, problem-solving specification language and its use in constructing expert systems.

1. Introduction

In this paper, we present an overview of an environment that supports the construction and development of domain knowledge and control knowledge for rule-based expert systems. The environment grew from our experience in constructing both AI and non-AI software. The environment, called ORBS (Oregon Rule-Based System), is founded on the following three propositions:

- Problem Solving systems are best suited to an interactive, incremental development approach (See discussions in [3]).
- At least some of the methods that have grown up around more traditional software development models can also be used effectively in the construction of problem-solving systems. In particular, some of the research results from the Software Engineering field are applicable to development of expert systems.
- The reuse of past efforts (for all kinds of software) is a powerful means of achieving complete, correct, and unambiguous systems.

In support of these ideas, ORBS provides the following components: 1) a specification language, KATE [10], for specifying rule-based programs; 2) a structure editor for modifying rules, facts, and control strategies; 3) a break package which supports incremental development and allows user-defined breaks to be built out of more primitive events; and 4) catalogs of useful domain and control pieces that can be combined to build new systems.

2. The ORBS System

The ORBS language builds on ideas found in several existing systems, including Hearsay III [6], LOOPS [4], RLL [14], and YAPS [1]. A detailed design rationale for of the system can be found in [9]. An expert system written in ORBS contains a set of relational facts in a data base, a set of forward-chaining rules that trigger on those facts, and a set of scheduling functions that determine which triggered rule to execute. A rule contains one or more left-hand-side (LHS) patterns, zero or more filters, one or more right-hand-side (RHS) actions, several system-defined fields, and zero or more user-defined rule-attribute fields.

In execution, ORBS proceeds through a sequence of match/choose/act cycles. The cycle, depicted in figure 1, is as follows: The matching process produces zero or more activations. This is called the conflict set.[1] Each cycle produces a new conflict set. An activation represents the match of a rule's LHS against facts in the database. We may want to choose none, one, some, or all activations for invocation. In ORBS, the choice is made by a control strategy defined by the system developer; ORBS has no built-in control strategy.[2] The developer builds a control strategy using software tools provided by 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. In particular, when building a new system the user may either a) select from a catalog of complete control strategies, b) build a new strategy using a catalog of pieces, or c) build a new strategy by using a combination of existing pieces and newly defined pieces. ORBS supplies a strategy-building language based on McDermott and Forgy's notation for combining scheduling functions [16]. Currently, the catalog contains complete control strategies which emulate OPS5, YAPS,

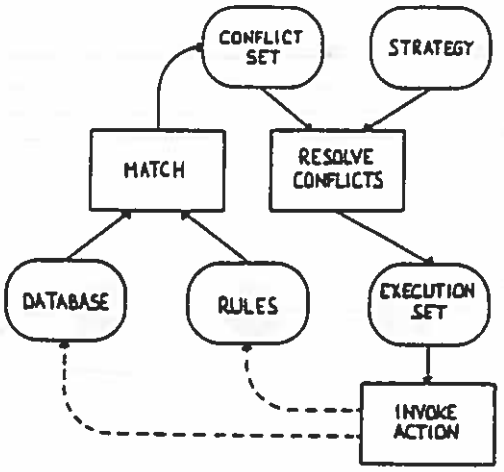

Figure 1. The ORBS match/choose/act cycle.

[1] The name, used for historical reasons, is no longer entirely accurate. In particular, there need not exist any conflict within the set, as we will discuss later in this section, all activations may be executable on the current cycle.

[2] The lack of a built-in strategy does not condemn users to creating special strategies for each program; users may choose a pre-defined strategy from the ORBS catalog or catalogs developed by users of the system. The point of ORBS's policy about strategies, though, is that developers should not be restricted to limited, static domain-independent strategies [11].

Function ==> (pronounced "join") is a generalization of McDermott and Forgy's -> operator; that is, ==> selects the first (as ordered by arc number) non-nil input. Thus if split sent conflict sets of size 13 on each of its out-arcs, and if RFKS GOAL sent a conflict set of size 0 on its out-arc, then ==> would send the conflict-set of size 13 on its own out-arc, because the size 13 conflict set would be the highest-order non-nil conflict set it received.

The graphical representation forms part of the run-time development system as well. The graphic strategy representation dynamically displays the control process. Figure 3 shows ORBS midway through conflict resolution using the YAPS-emulation strategy. This graphic representation has proven useful for program analysis and debugging. For example, Figure 4 shows the conflict resolution process in one cycle of a rule-based test-case generator for software debugging; note that the display reveals that the R5KF scheduling function is not contributing to the decision process on this cycle. Note too that by including AD1Y, a scheduling function from YAPS that chooses a single arbitrary activation from the conflict set, our control strategy allows only one activation per cycle to be invoked. In many applications, this behavior is overly restrictive. For example, the test-case generator could fire multiple activations per cycle without conflicts. However, many rule-based systems have such a decision on the number of activations that can be invoked on any cycle "hard-wired" into the system itself. Thus in ORBS, the freedom to define strategies extends to the freedom to determine the number of activations fired on a given cycle—or even to the freedom to omit specifying any limit at all. That is, all rule ordering could be handled through the other scheduling functions. If we can guarantee that invoking two rules of the same type during the same cycle will not interfere with each other, we simply remove the AD1Y function to allow multiple activations.

2.1 Halting the System

Another decision, initially "hard-wired" into ORBS, was that the system would always halt when the control strategy produced an empty set of activations. We have now placed this decision in the user's hands. Specifically, ORBS halts only when given a halt command. The halt-on-empty-conflict-set behavior can be achieved by using scheduling function HECS, which issues the halt command when the conflict set is nil.

We find no need for HECS in many programs. In particular, an empty conflict set often signals that the program is stuck. In this case, we may want 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 a domain-dependent scheduling function that interacts with the user to add and remove facts from the database when the conflict set is nil.

When, then, does the system halt? Strictly speaking, whenever the function orbs-halt 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. It halts the system after all activations chosen in the current cycle have been invoked. In one application system developed in ORBS for VLSI silicon compilation, for 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 developer-written scheduling function checks to see if the user is willing to loosen the current set of constraints. If not, orbs-halt is called and the system gives up.

2.2 Higher-Order Control

ORBS also provides facilities for what has been termed "higher-order" control [5]. That is, a system may have knowledge of and make decisions about its own decision-making process. One way higher-order control may be used in ORBS is through dynamic (i.e., run-time) changes to the strategy. Different circumstances may require a single system to make decisions in different ways. 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 implemented this control knowledge in ORBS as a domain rule:[4]

```
(defrule nearly-out-of-time
        (time-remaining -tr)
 test   (< -tr critical-time)
 -->
        (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 domain-knowledge rules (first-order control). When and if rule nearly-out-of-time is invoked, it causes the standard scheduling strategy to be changed to a keyword-based beam-search. The beam-search on the keyword represents knowledge that a non-optimal solution is always possible. While it might not always be the best, we will use it when time runs out.

However, second-order control knowledge presents problems for ruled-based systems even such as ORBS. 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. One of our current research efforts is attempting to deal with this problem by analyzing the semantics of control of a rule-based system, including dynamic strategy change and multiple contexts. We hope to develop 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, similar to Hearsay III's scheduling blackboard [7]. Thus the focus of our attention on this problem continues to be the dilemma that system flexibility and extensibility are often achieved only at the cost of increased complexity and difficulty of use.

In the remainder of the paper, we look at 1) the interactive, incremental model used by ORBS; and 2) our attempts to use formal specification techniques in building rule-based systems.

---

[3] Figures 2 through 8 are screen dumps to an Imagen laser printer from a Symbolics 3600 running ORBS, shown in whole or in part as appropriate. We have enhanced parts of the figures to compensate for poor reproduction quality.

[4] In this rule, as in other ORBS rules, identifiers beginning with a hyphen are pattern variables, thus -tr will be bound to the number which is in the second position in the fact matched by the pattern. In this example, the fact (time-remaining -tr) is maintained by the system and critical-time is a global variable.

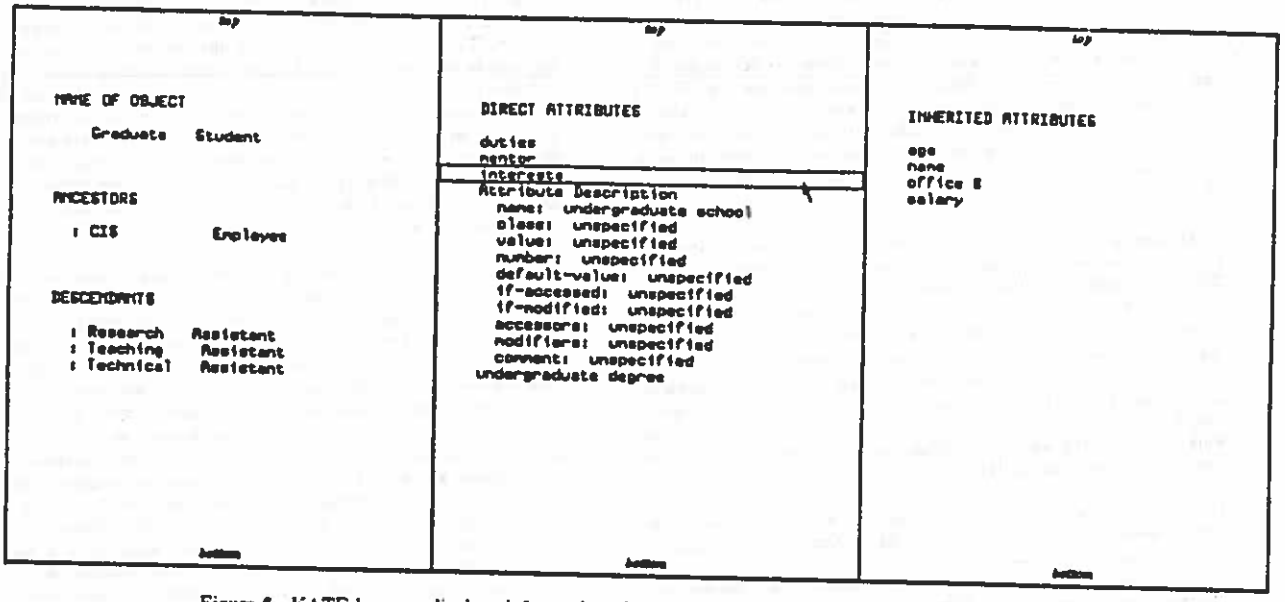

Figure 6. KATE browser displays information about the CIS-People component. The attribute "undergraduate school" has just been expanded; the cursor is positioned to expand "interests" next.

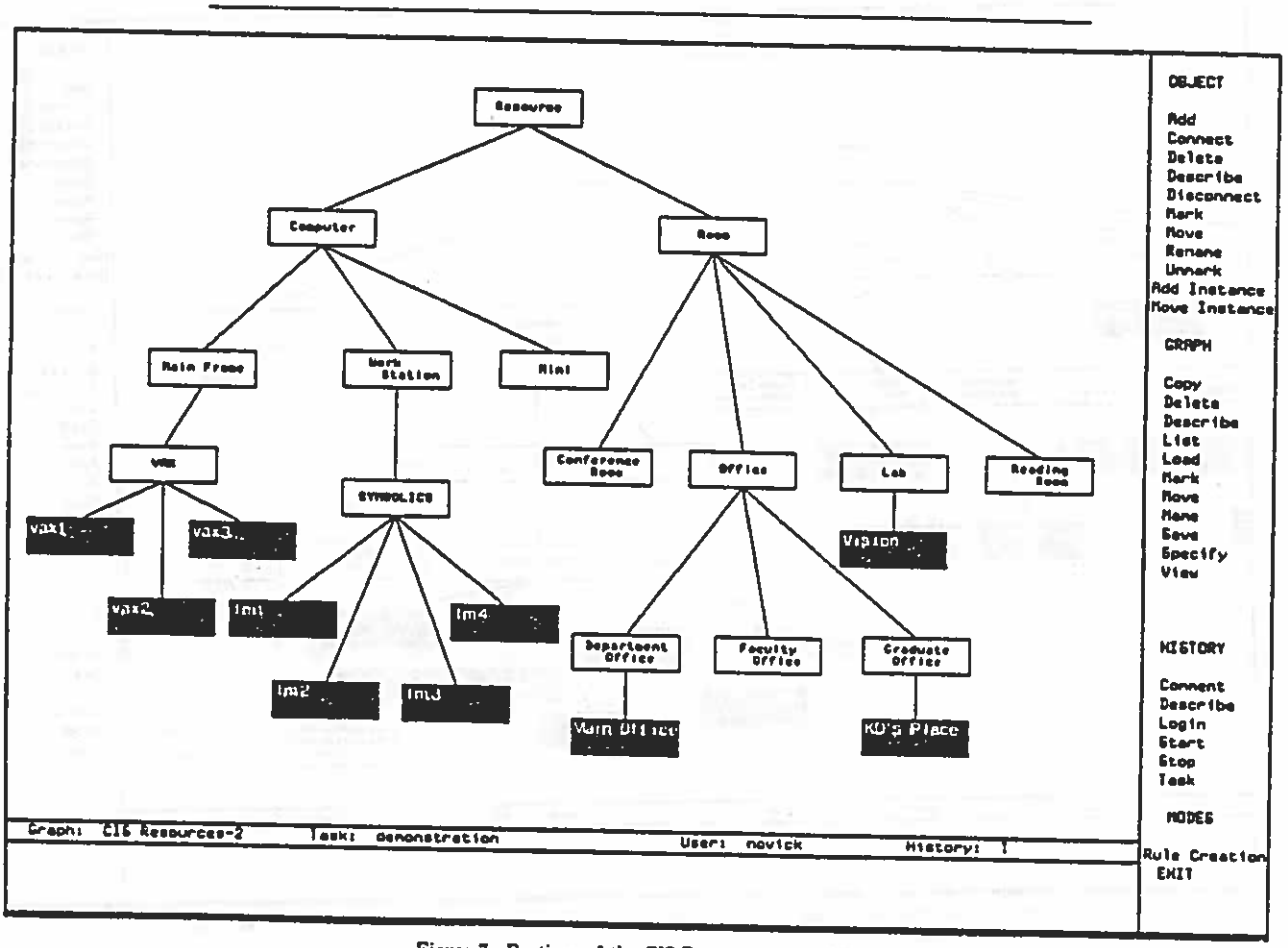

Figure 7. Portions of the CIS-Resources component.

another catalogued component from VLSI chip layout. Our long-range goal is to allow these multi-domain components to be loaded and melded together with assistance from KATE. However, with the existing primitive editing functions, we find that the translation process from VLSI to office planning is more painful than simply manually adding the components from scratch. As we have previously indicated, much of our current effort is directed towards adding the type of sophisticated reasoning necessary to make KATE a better specification assistant. This includes automating portions of the loading, melding, and tailoring process, as well as checking completeness and consistency as the specification evolves.

When the user finishes the specification, KATE will automatically map it to an ORBS program. From here the user may use the tools discussed in section 2 to test the program further and to debug it. This brings up a very difficult problem: because the user is free to modify the program itself, the KATE specification can easily become out of date. As suggested by Wile [18], one approach is to modify the specification, and then replay the mapping process to generate a new program. We are not sure this is a viable approach when constructing AI systems. Our experience suggests that the specification undergoes constant change; remapping after each new increment seems prohibitively inefficient.

Our current model uses KATE to build a prototype ORBS program, and then ORBS tools to further refine the program. While this seems to be adequate as a process model, it has inadequacies in terms of documentation. In particular, we find it useful to be able to trace a particular ORBS object, rule, or scheduling function back to the KATE specification. In this way we can rationalize an ORBS program. Once the user modifies the ORBS program directly, we lose these specification links.

5. Status

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 [2] under Vax Unix 4.2. The system is also being ported to a Tektronix 4404 AI Workstation (Pegasus).

6. Summary

In constructing ORBS, we have attempted to pay attention to the non-language aspects of the system. This includes both the model of program development, and the tools that support it. We have also tried to use some of the modern ideas from the field of Software Engineering, a field concerned with building software systems in general. While we do not believe there is a precise analogy between building non-AI software and AI software, we do believe there are some similarities. KATE is a first attempt at exploiting these.

Finally, we note that the ORBS language itself is under constant evolution as we discover new problems and missing functionality. As an example, several graduate students are currently working on extending the langauge to include backward chaining and hypothetical reasoning. In summary, we believe we are far from a steady-state system; indeed, the development of ORBS constitutes an experimental method for exploring issues in artificial intelligence environments.

7. Acknowledgments

Other past and current members of the ORBS project include Allen Brookes, Kim Dannewitz, Keith Downing, Michael Hennessy, and Bill Robinson from the University of Oregon, and Bill Bregar from Oregon State University. Evangelos Simoudis has been a valuable first user of the system.

8. 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] Barstow, D., Shrobe, H., and Sandewall, E., (eds.), Interactive Programming Environments, McGraw-Hill, 1984.

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

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

[8] Fickas, S., Automating software development: a small example, in *Symposium on Application and Assessment of Automated Software Development Tools*, San Francisco, 1983.

[9] Fickas, S., Design Issues in a Rule-Based System, in *ACM SIGPLAN 85 Symposium on Programming Languages and Programming Environments*, Seattle, 1985.

[10] Fickas, S., Laursen, D., Laursen J., A Knowledge-Based Software Specification Environment, presented at *Rutgers Workshop on Knowledge-based Design*, 7/84.

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

[12] Goldman, N., Three Dimensions of Design, Proceedings of the Third Annual National Conference on Artificial Intelligence, AAAI, Washington, D.C., 1983.

[13] Greenspan, S., Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition, PhD Thesis, Computer Science Dept, Toronto, 1984.

[14] Griener, R., Lenant, D., A representation language language, in *1st National Conference on AI*, Stanford, 1980.

[15] London, P., Feather, M., Implementing Specification Freedoms, in *Science of Computer Programming*, Number 2, 1982

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

[17] Swartout, W., Balzer, R., On the inevitable intertwining of specification and implementation, *CACM* 25(7) (1982).

[18] Wile, D., Program Developments: Formal Explanations of Implementations *CACM* 26(11), 1983.