SRC TR 87-172

A Modifiable Approach to Expert Systems Development

by

J. C. Sanborn

A Modifiable Approach to Expert Systems Development†

James C. Sanborn

Systems Research Center
University of Maryland
College Park, Maryland 20742

Arpanet: sanborn@maryland.umd.edu

Abstract

Rule based expert systems programmers experience similar difficulties in developing and maintaining large application programs: rules become instantiated when they shouldn't; the execution order of rules is undesirably nondeterministic, or worse, simply incorrect; and modifications to program behavior are difficult or unwieldy. All of these problems arise from the *control strategies* used by the development language, their implementation, and the programmers control over (and awareness of) them. This paper explores the impact of rule based program control on overall program modifiability. We present a language designed with efficiency, modifiability, and ease of use in mind. Throughout, we discuss traditional control strategies, improvements made through our research, and directions for further study.

1. Introduction

Maintaining correct behavior of a rule based program under modifications to the rule base is one of the biggest problems confronting expert system developers. While the apparent extensibility of rule based programming is initially attractive, users soon learn that adding a new rule to the rule base (or changing an existing rule) may have a drastic, rippling effect on the desired behavior of the program. In this paper, we explore some of the reasons behind this contradictory state of affairs. We present a language designed with efficiency, extensibility, modularity, and modifiability in mind. In section 5, we compare this language to other approaches, showing how the use of control information impacts the ultimate modifiability of rule based programs. Finally, in section 6, we discuss some new ideas in rule based program control. In the following two sections, we pin down the notion of modifiability, and briefly review some popular control strategies.

2. Modifiability Issues

What do we intend by *modifiability* in a rule base? Basically, this implies the ability to make changes to any aspect of a rule based program — rules, data, and control — in relative isolation. For one thing, we would like to avoid the ripple effect of rule modifications on program behavior mentioned above. One means of achieving this is to partition the rule base in such a way that modifying a rule in any partition will at most affect program behavior within that partition. We say that a rule based programming language that supports such a partitioning of the rule base has the **modularity** property. As we shall see, rule base modularity is crucial to maintaining proper behavior.

Modularity of the rule base limits the scope of behavior changes, but does not eliminate them. We say that a rule based language is **extensible** if rules may be added, removed, or modified without impact on the program's behavior with respect to existing rules. Thus, an extensible rule base is truly a collection of independent "programs." Adding a new rule *should* have an effect on program behavior; the extensibility property says that this effect is based *soley* on the addition of the new rule, and not on the relationship of this rule to the other rules in the system.

Another modifiability property is the ability to easily change parameters within rules. This applies to the types and representations of data manipulated by rules, as well as functions applied to data from

† The language discussed in this paper was developed at the Denver Aerospace and Baltimore Laboratories facilities of the Martin Marietta Corporation from 1983 to 1986.

within rules. As an example, we may only want a rule to be applicable if the value of some parameter is within a certain range, and it may be desirable to change this range dynamically during program execution.

Finally, control of program behavior should be dynamically modifiable. That is, we would like rules to influence control knowledge and strategies. An example of this is a program that uses a goal directed strategy to generate a hypothesis, and (subsequently) a heuristic search strategy to plan actions based on the hypothesis. We will see more examples of dynamic control in sections 4 and 5.

3. Control Issues

The basic operation of rule based program interpreters has come to be known as the **recognize/act cycle**. Given a set W of facts and a set R of rule definitions, an iteration of this cycle consists of the following sequence of operations:

1. From W and R, determine the set I of rule instantiations ($I \subseteq 2^W$ such that, for all $i \in I$, i unifies with the conditions of some $r \in R$).
2. Determine a set $F \subseteq I$ of rule instantiations to be executed, or "fire."
3. Execute the actions of each instantiation in F in some order.

The behavior of a rule based program is clearly governed by whatever happens in step 2. This operation is known as **conflict resolution** (CR). Typically F is a singleton set,[1] so that the CR process induces a "best first" partial ordering on I, generating F from the "best" instantiation. Next, we consider the two classes of control information which may be used by CR.

3.1. Domain Independent Control Information

For most general-purpose interpreters (e.g., OPS [FORGY77], YAPS [ALLEN83], etc), control information is obtained from rule *syntax*. Such information is **domain independent**, as it may be extracted from any rule defined to the system. Some of the more popular domain independent CR strategies are:[2]

1. **Refraction**: instantiations that are fired are removed from I, so that physically identical instantiations (i.e., those matching the exact same data items) are fired at most once.
2. **Specificity**: the intent of this strategy is to prefer rules that are more specific instances of other rules. For example, rule r1 below is more specific than rule r2.

```
(rule r1                    (rule r2
  if (a =x =x)                if (a =x =y)
     (b =y)                      (b =z)
  then ...)                   then ...)
```

   Specificity is usually determined when a rule is initially defined to the system. Two approaches to specificity are

   (a) **Complexity**: a numeric value is derived based on the composition of a rule's conditions (number of patterns, different variable names, etc) and is used as a weight in deriving specificity relations during execution.

   (b) **Unification**: we attempt to unify the conditions of the defined rule with those of each previously defined rule. A new rule is more specific than an existing rule if the conditions of the old rule unify with a proper subset of the conditions of the new rule.

---

[1] This need not be the case. Haps, for instance, supports firing more than one instantiation per cycle. In the general case, this is undesirable, since two instantiations asserting contradictory information may be fired on the same cycle, however there are situations where firing several instantiations "in parallel" is advantageous (see, for example, [BALZER80]).

[2] While we will not have much to say in this paper regarding the usefulness and tradeoffs of the various CR strategies, an excellent discussion of these topics may be found in [SAUERS85].

3. **Recency**: prefer instantiations containing "newer" data to those matching older data. The intent here is to make behavior of the program sensitive to changing data. Usually, recency is implemented by recording the "time" at which a data item is asserted; in CR, instantiations may then be sorted according to the most recent piece of data.
4. **Priority**: many languages provide the user with a explicit means recording the relative "importance" of defined rules. If it has been asserted that rule r1 takes priority over rule r2, *all* instantiations of r1 will be preferred over *any* instantiations of r2.
5. **Random**: when all else fails, any of the remaining instances may be selected.

The CR process applies one or more of theses strategies in some order. OPS, for example, uses the order

refraction → priority → recency → specificity → random.

This gives an interpreter flexibility over many application domains. Unfortunately, the exclusive use of domain independent information is inadequate for the control of most non-trivial applications. For example, one of the most common control patches during the testing phase of expert systems development is to change rule priorities. As more and more of these patches appear over time, it becomes difficult to manage or analyze the behavior of the system. In these situations, both modularity and extensibility are compromised.

3.2. Domain Dependent Control Information

Most applications need to make use of problem-specific control information. For example, goal directed reasoning requires the explicit representation of domain dependent goal information as data. If CR is to make use of domain dependent information, the language must incorporate a means of defining control information to the system explicitly, as well as for distilling this information out of rule instantiations.

"Meta-level" reasoning is another example of the use of domain dependent information in program control. The use of "meta-rules" allows the programmer to manipulate control at the program behavior "object-level" (i.e., reasoning in the application domain). In this way, information specific to a given application situation may be used to help guide control. Figure 3.1 shows a sample meta-rule (called "cant-fix-with") in a "device Maintenance" domain. This rule states that repair methods using unavailable tools should not be explored. Clearly, there is substantially more work involved in representing control information in this manner, but the resulting program has excellent modularity and the potential for extensibility.

4. Basic Haps

Here we present an overview of the Haps rule based programming language. We restrict our discussion of the language to issues of modifiability; for a more complete treatment of the language, see [SANBORN85] and [MARSHALL85].

```
MRule: cant-fix-with
  if <goal> = (repair <device>)
     and <means> = (to-repair <device> use <method>)
     and (uses <method> <tool>)
     and (unavailable <tool>)
  then (eliminate-method <method>)
```

Figure 3.1 — *A Meta-Rule*

4.1. Design Goals

Haps was initially conceived of and designed by Ron Sauers [SAUERS83]. At the time, there were no commercially available rule based languages, and languages such as OPS were deemed insufficient for application domains of interest. Having experienced the shortcomings of existing systems, the design goals for Haps became:

1. Efficiency of both the runtime and development environments: rules should compile quickly, be easy to change during development, and run fully compiled in the runtime environment for increased execution speed.
2. Multiple data representations: the language should be capable of matching patterns, arrays, strings, records, and user defined types.
3. Control of program behavior: modularity should be encouraged, CR should be dynamically modifiable, both by the user and the program itself, and special purpose control strategies should be easy to integrate with defaults.
4. Access to the target language (Lisp): it should be easy to call Lisp from anywhere within a rule for both value and effect.
5. Learning the language should be an easy, incremental process, even for users unfamiliar with Lisp.

4.2. Implementation

Haps is a forward-chaining production system with several available data representations. The rule compiler generates pattern-matching networks similar to *Rete* networks described in [FORGY82]. One difference is that Haps networks are composed of Lisp function definitions; this makes rule modification easy during development, and increases runtime speed when these networks are themselves compiled. There is a window-oriented user interface capable of monitoring various changes during program execution, as well as providing ready access to most user-level functions. While this interface does not encompass such things as graphic or icon editors, Haps' integration with Lisp from within rules gives the programmer the freedom to use Haps in conjunction with any other Lisp program. This includes external programs that may asynchronously update Haps' working memory.

4.2.1. Data Representation

The two basic data types in Haps are **clauses** and **frames** are composed of data types available in Lisp.[3] The Haps Frame System is similar to frame systems such as NETL [FAHLMAN79] and will not be discussed here. Clauses in Haps are arbitrary list structures. Any valid Lisp object may appear in a clause, including other lists, so

```
(this is a clause)
(this (clause (contains) nested) lists)
(object square-37 (type square) (color blue))
(pretty-name square-37 "the red square numbered 37")
```

are all valid Haps clauses.

4.2.2. Rules

Haps rules have a familiar left-hand-side (LHS) right-hand-side (RHS) representation, with patterns on the LHS matched against working memory (WM) and actions written in both Haps and Lisp on the RHS. The LHS contains an implied conjunction of clause patterns (CPs) testing for the presence or absence of clauses in WM. The LHS may also include calls to Lisp testing values of variables bound during the matching phase. **Variables** are preceded by an equal sign (=) and must be consistently bound for a rule

[3] Haps is implemented on Symbolics Lisp Machines, so these data types include arrays, strings, structures, and flavors. In particular, flavors may be used directly as data structures in Haps' frame memory.

to instantiate. Question marks (?s) appearing in patterns are **who cares** variables; something is expected to appear in a ?'s location, but its value is ignored. Patterns preceded by a minus sign (-) are **negated**, they test for the absence of a clause in WM. A **label** is any atom surrounded by angle brackets (e.g., "<label>" is a label); these are bound to the matching clause to be used in the RHS. Finally, **predicates** may be applied to variables during the match phase; these are of the form "{=var &rest *tests*}" and are directly embedded within patterns. If the binding for =var is does not pass the *tests*, the rule will not be instantiated. A test can be anything from simple equality testing to an arbitrary Lisp call (see below).

The exclamation point (!) signals a break to Lisp and may appear anywhere in a Haps rule. For instance, if the clause !(< =x =y) appears as a test on the LHS of a rule, the rule will only instantiate if the value bound to =x is less than that for =y. The call to Lisp may be for value as well as effect; it may appear within a clause (in which case the matching clause must match the value returned by the call), or as a test argument in a predicate construct. On the RHS, a clause such as !(draw-line =x1 =y1 =x2 =y2) could be used for effect.[4]

Finally, the RHS of a Haps rule may contain Haps or Lisp commands (with the latter preceded by "!"). The relevant Haps RHS command are

(hassert ...clause$_i$...) - add the specified clauses to WM
(hremove ...<label$_i$>...) - remove the clauses bound to the specified labels from WM
(hmodify <label> ...=var$_i$ change$_i$...) - modify the clause bound to the specified label
(halt) - halts the Haps recognize/act cycle.

Any RHS clause not beginning with one of the above keywords or preceded by "!" is assumed to be an assertion. As an example, the Haps rule of figure 4.1 might be found in a "blocks world" planner.

```
(rule put-block-on-block                          ; name of the rule
  if (task (put =block1 =place)) <task>                  ; labelled "task" pattern
     (at {=block2 <> =block1} =place)                    ; block2 not equal block1
    -(on ? =block2)                               ; block2 is "clear" (nothing is on it)
     (holding robot =block1) <holds>              ; robot has block1
test (= (robot-location) =place)                  ; compare place to robot's location (in Lisp)
then !(show-robot-put =block1 =block2 =place)            ; do some graphics (in Lisp)
     (hmodify <holds> =block1 !(list 'none))      ; update what robot holds (using Lisp)
     (hremove <task>)                                    ; remove the task clause
     (at =block1 =place)                                 ; assert new info about block1
     (on =block1 =block2))                        ; (recall default RHS action is "hassert")
```

Figure 4.1 — *A Sample Haps Rule*

This rule instantiates whenever the planner tries to put a block somewhere where there is already a (different) block. It uses Lisp on the RHS to actually show the robot doing its thing, as well as in modifying the "holding" clause.

4.3. Control Strategies

The integration of Haps programs with Lisp provides a simple but powerful approach to control in the language. While the default control strategies are sufficient in many applications, users, as well as their programs, may cause program behavior to change dynamically during execution. There are two basic control strategies in Haps: a modifiable, domain independent conflict resolution mechanism; and a domain

[4] Unlike some other rule based languages, the RHS of a Haps rule is not evaluated conditionally; every clause in an instantiated RHS is evaluated when the instantiation is fired.

dependent "control-level" approach based on the use of explicit control data within Haps rules.

4.3.1. Conflict Resolution

Haps provides a sequence of conflict resolution strategies similar to those discussed in section 3.1, but with several notable differences. The default application sequence is refraction → importance → recency → complexity → arbitrary. The definitions of refraction and recency are as in section 3.1. A complexity value is derived through examination of a rule's LHS; the number of clauses and variables, as well as the number of predicate tests applied to variable bindings contribute to this value. Haps uses complexity rather than unification in testing for rule specificity because computation of the latter is time-consuming even for reasonably small rule bases. Since conflicts are usually resolved via recency, the cost of computing the unification — which must be done at rule definition time — was deemed unreasonable due to its effect on rule compilation time.[5]

The **importance** strategy used in Haps differs from the traditional priority approach in several useful ways. Rule importance may be defined at rule definition time, or any time thereafter. Additionally, it may be changed dynamically. Another difference is that rule importances may be defined relative to other *sets* of rules. For example, executing

(raise-importance (rule3 rule4) (rule7 rule1) (rule2))
(lower-importance (rule0 rule5) (rule6))

in Haps imposes the following hierarchical ordering on eight rules:

{rule6} → {rule0,rule5} → {rule3,rule4} → {rule7,rule1} → {rule2}

Instantiations of rule6 are "least" important, while those of rule2 are "most" important. That is, whenever an instantiation of rule2 enters the set I, it will be preferred over all other instantiations; an instantiation of rule6 will only be selected when I does not contain instantiations of any of the other rules. The advantages of this approach over priorities are discusses in section 5.

It is important to note that the user or program may change importance relations at any point after a rule has been defined (during execution, for example). In addition, there is a simple procedure for the system to add, remove, or modify control strategies; or change the means of selecting F from I. In this way, the Haps environment has generally modifiable domain independent control.

4.3.2. Control-Level Rules

Most applications written in languages like Haps store domain dependent control information in WM and match it directly in rules. An example of this is an application controlling a system which must be initialized before running. Rules mentioning subsystem **x** might include a condition such as "(initialized **x**)" and thus not be instantiable until the initializations have been noted by the application. We call this these data items **embedded control data** since their use embeds control information within rules.

In Haps, a **control-level rule** is a domain independent Haps rule that matches and manipulates embedded control data. Such data has been used in most production systems in the past; the Haps approach to its use is somewhat unique in that control-level rules have been provided as a sort of Haps "system code," providing a general format for user programs. In this way, many of the problems encountered in other languages do not arise in Haps programs. At present, two of these rule based control-level strategies have been implemented: **goal-directed search** with automatic AND/OR subgoaling, and **heuristic search** with local state memory and automatic backtracking.

To achieve goal-directed (e.g., backward-chaining) behavior in a forward-chaining interpreter, the conditions (LHS) of rules must match goal expressions, and the actions (RHS) must insert subgoal expressions into WM. Haps provides a set of Haps rules and Lisp functions for defining and manipulating arbitrary

[5] In some cases, especially where most of the rules are very similar, the unification approach to specificity may helpful. Since Haps provides for user-defined strategies, this straightforward algorithm is easily implemented and incorporated with the other domain-independent strategies (see [SANBORN85]).

```
(rule succeed-conjunctive-goal
  if  (goal =goal ? ? (status {=status == ACTIVE}) ?) <goal>
      (goal ? ? ? (status SUCCEED) (parent =goal))
     -(goal ? ? ? (status ACTIVE) (parent =goal))
      (conjunct =goal =subgoals) <conjunct>
  then  (hremove <conjunct>)
        !(remove-subgoals =subgoals)
         (hmodify <goal> =status SUCCEED))

(rule fail-conjunctive-goal
  if  (goal =goal ? ? (status {=status == ACTIVE}) ?) <goal>
      (goal ? ? ? (status FAIL) (parent =goal))
      (conjunct =goal =subgoals) <conjunct>
  then  (hremove <conjunct>)
        !(remove-subgoals =subgoals)
         (hmodify <goal> =status FAIL))
```

Figure 4.2 — *Control-level Rules for Automatic AND Subgoaling*

AND/OR goal hierarchies. The general form of a goal clause in WM is

(goal *name* (type *type*) (info *info*) (status *status*) (parent *pname*)).

Lisp functions are defined for generating and asserting goal clauses, as well as defining arbitrary conjunctions and disjunctions of subgoals. Figure 4.2 shows the two control-level rules implementing automatic conjunctive subgoaling; their two disjunctive counterparts are similar (see [SANBORN85]).

The second set of control-level rules, implementing a heuristic search (HS) strategy, is intended to simulate the A^* state space search algorithm [HART68]. This presents a problem in a forward-chaining environment with global memory: namely, how does the interpreter deal with state dependent information? Since most forward-chaining production systems do not provide a backtracking facility, it is not generally recognized that a language like Haps may be used to implement state space search. Indeed, the notion of "state" in a production system refers to the current set of instantiations, the rules, and the data defined to the system; once fired, the actions of a rule are not easily retracted.

In Haps, control-level rules have been used to implement **state selection** in HS. The approach represents state information as clauses in WM. There are two means of encoding state dependent information: if possible, all such information is represented as part of the state itself; otherwise, changing external data items must include an explicit "state" parameter. For example, the set of six Haps clauses

(state STATE3) (status STATE3 current) (parent STATE3 STATE1) (children nil)
(cost-info STATE3 (cost 3) (estimate 27)) (info ((at *at-loc*) (goal *goal-loc*)))

might represent a state in a "path-planning" application (where the *locs* are points in space, the estimate is Euclidian distance, and the cost is the actual distance travelled in reaching *at-loc*). Since the only changing data in this case are the *locs*, all state dependent information is represented in the "info" parameter of the state representation. As with other domain dependent strategies, Haps provides Lisp functions for generating and manipulating state representations.

The approach to implementing HS assumes that the user develops rules to generate new states (from a current state), as well as rules testing goal criteria. The "state-select" control-level rule, shown in figure 4.3, must be of lower importance than state generating rules to achieve the desired HS behavior. A Lisp function "heuristic-value" is used to order states for selection; by changing its definition, the search path can be modified (simply adding the cost and the estimate gives the A^* heuristic — so long as the estimate

is a lower bound on actual cost). For more details on this approach, see [SANBORN86].

5. Does Haps Help?

In this section, we compare Haps' control strategies to those found in other languages. We believe our approach provides useful "default control" and that the extensibility and modifiability of the implementation makes it well suited to large applications where dynamic behavior change may be essential.

Domain independent control in Haps is augmentable. Additionally, the use of importance over simple priorities provides a means of partitioning rule bases via **class hierarchy**. For example, when using the goal directed search control-level rules, an application's behavior will be very different if these rules are set to be more (or less) important, as a class, than user-defined rules that assert goals. This definition also provides an easy means of distinguishing between the object- and meta-levels in such reasoning systems: meta-level rules are, as a class, made more important than rules at the object-level.

But how *extensible* is this? The notion of "class" above comes about only through explicit mention of a rule in a certain set of rules, relative to other sets of rules. Future modifications to the rule base may not take relative importance into account, resulting in improper behavior, or ad hoc patches to control (as with priorities). So importance is better than priority, but has the same negative impact on modifiability.

The most useful improvement found in Haps' domain independent strategies is the dynamic modifiability of the control strategies.[6] During execution, the application order of existing strategies may be changed (selecting via complexity or unification before recency, for example), and strategies may be added or removed. There is an ability for these strategies to access domain dependent, instantiation-level information. However, control reasoning at this level is complicated and time consuming; it is desirable for the CR phase be executed as quickly as possible to maintain the sensitivity of the application to changes in WM. The use of control-level rules allows domain dependent, rule-level reasoning and

```
(rule select-state
  if (status =cs {=cs-status == CURRENT}) <cs>
     (status {=s1 <> =cs} {=s1-status == UNEXPANDED}) <s1>
     (status {=s2 <> =cs =s1} UNEXPANDED)
     (cost-info =s1 (cost =c1) (estimate =e1))
     (cost-info =s2 (cost =c2) (estimate =e2))
test (<= (heuristic-value =c1 =e1) (heuristic-value =c2 =e2))
then (hmodify <cs> =cs-status EXPANDED)
     (hmodify <s1> =s1-status CURRENT))

(rule goal-test   ; for the "path planning" example
  if (status =state CURRENT)
     (goal-location =goal-loc)
     (info =state ((at =goal-loc) (goal ?)))
then !(show-path)
     (halt))
```

Figure 4.3 — *Control-level Rules for Heuristic Search*

[6] Many new rule based languages are beginning to incorporate similar features. The ORBS project [FICKAS87] for example, forces users to define control strategies by presenting them with sets of instantiated rules and allowing them to select a set to be fired. The system uses this selection information during future executions to make control decisions on its own.

maintains higher execution speed. Thus, we prefer the use of control-level rules to reasoning about domain dependent information within the CR phase.

The compilation of Haps rules into networks of (compiled) Lisp functions encourages modular rule base design, so many of the common problems inherent in large applications may be avoided by encoding rules which load and discard other rule bases as they are needed. However, this is another implicit means of guiding control; proper behavior remains tied to the programmer's ability to maintain this implicit modularity. While this approach results in acceptable execution speed, reasonable modularity, and dynamic control by the application; extensibility of the rule base in isolation of control is limited.

Finally, the use of control-level rule in Haps differs from most other implementations. In OPS, for example, users may select a "means-ends analysis" strategy implementing goal-directed search. However, this is achieved by embedding control data of the form "(goal ...)" in OPS rules: these tokens *must* appear as the first pattern in the rule, and CR applies strategies only to clauses matching these patterns. Thus, rules written using this strategy will not behave correctly under other strategies, so there is no control modifiability. The control-level goal rules in Haps however, are just Haps rules; they are subject to whatever domain independent strategies are currently in use. This is a fundamental difference between control in Haps and in other languages: in Haps, domain dependent strategies are not embedded within domain independent strategies. Control in Haps *is* modular with respect to domain dependent or independent strategies.

6. Remedies

We have seen that while control in Haps is an improvement over previous efforts, many of the limitations on modifiability persist. Are there some approaches that might be employed to improve modifiability? In this section, we consider several directions for further research in modifiable control.

6.1. Dynamic Control

While Haps does provide a capability to modify control dynamically, it is limited by the accessibility of relevant control information to rules. That is, changes in control are determined by the programmer writing the rules; applications depend on these rules being executed at the proper time to effect changes in behavior. Ideally, this function should be performed asynchronously in a control monitoring environment. Success and failure criteria for various operations would be represented explicitly in this environment and used to modify control based on current data and goals. This approach is not unrelated to the use of meta-level rules in some systems (see, for example, the "blackboard" approach of [BALZER80]), but would represent an improvement in endowing a control mechanism with the power to independently examine WM regardless of current instantiations.

6.2. Class Relations

In section 5, we showed that Haps' importance strategy is more powerful than simple inheritance when viewed as imposing a class hierarchy on a rule base. However, since the hierarchy is implicit as regards control, modifications to the rule base may lead to improper behavior. Imposing an explicit class membership on each rule would lead to greater extensibility, since new rules could be easily placed in an existing class. For example, the implementation of HS outlined in section 4.3.2 depends on the use of importance levels to achieve the proper behavior. By representing the three levels as classes the "goal", "select", and "expand", no importance information needs to be changed when adding new goal-test or state-expansion rules.

6.3. Success Statistics

Another means of improving control strategies is by adding new types of control information. Success statistics refer to performance information for various rules and rule bases in solving the problems for which they were designed. For programs executed over long periods of time, such statistics have an obvious use in directing control. A control rule using this kind is shown in figure 6.1.

CRule: time-critical-task
if G is a time-critical goal with priority = NOW!
and task T is a means of achieving G
and there are no other tasks T' that have achieved G in less time
then prefer achieving G via T

Figure 6.1 — *Control-level Rule using Success Statistics*

6.4. Problem Solving Contexts

One of the biggest problems with production system languages is a lack of explicit **focus** by the interpreter on rules which are appropriate to solving the current problem. Indeed, in any given situation, only a small number of the total rules in the rule base are applicable; others only slow the system down trying unsuccessfully to become instantiated.

Representing a **problem solving context** (PSC) explicitly for a rule has many advantages. Modularity of the rule base is imposed automatically, increasing the system's ability to swap compiled rule bases in and out of focus according to the program's current goals. In this way, CR is applied only to that segment of the total rule base relevant to the current problem. Such a rule base is also extensible through the addition of rules in an existing PSC which generating new PSCs. Best of all, by changing PSCs often, the usual problems associated with attaining desired system behavior are minimized: the rule based program is now goal-directed in terms of generating and reasoning about the success and failure of PSCs, and data-driven in its reasoning within a given PSC. Local control strategies may be defined for a PSC, so that, for example, one PSC may generate a hypothesis in a goal-directed manner, and based on the outcome, some other PSC may use heuristic search to generate a plan to test the hypothesis.

Haps' original goals included a simpler version of the PSC facility known as a "goal-context." This capability has not been implemented in the current version of the language.

7. Conclusions

We have seen that, by placing most of the burden of control maintenance on ad hoc user-controlled methods, most existing control strategies inhibit modifiability. Haps is an improvement, particularly in its use of control-level rules and its ability to change the control strategies dynamically. However, without explicit representation and reasoning about control-level information, as well as the partitioning of large rule bases through the use of local problem solving contexts, the modifiability problem in rule based systems development will persist. The Haps project has enlightened us to the speed at which rule may be defined and manipulated; to make truly large application systems viable, we need to carry this knowledge into the domain of reasoning about control.

Acknowledgements

Since 1983, many people have been involved with the design, implementation, testing, and application of Haps. I would like to thank Ron Sauers and Jonathan Bein for designing, building, and putting up with Haps version 0. David Marshall and I designed and built the current version of Haps, which is in use today at several Martin Marietta sites. Finally, I'd like to thank Jeff Van Baalen for initially getting me involved with the Denver AI Unit.

References

[ALLEN83] Allen, E. "YAPS: Yet Another Production System," TR# 1146, Computer Science Dept., U. Maryland, 1983.

[BALZER80] Balzer, R. et. al. "Hearsay III: A Domain-Independent Framework for Expert Systems," *Proc. AAAI-80*, 1980.

[FAHLMAN79] Fahlman, S. "NETL: A System for Representing and Using Real-World Knowledge," MIT Press, 1979.

[FICKAS87] Fickas, S. "Development Tools for Rule Based Systems," in *Expert Systems: The User Interface*, Hendler (ed). Ablex Press, (in press).

[FORGY77] Forgy, C. & McDermott, D. "OPS, A Domain-Independent Production System Language," *Proc. IJCAI* **5**, 1977.

[FORGY82] Forgy, C. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence* **19**, 1982.

[HART68] Hart, P., Nilsson, N., and Raphail, B. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. SS&C*, SSC-4(2).

[MARSHALL85] Marshall D. and Sanborn, J. "The Haps 1.1 Maintenance Report," Martin Marietta Corp. Report DA-TR-85-140, 1985.

[SANBORN85] Sanborn, J. and Marshall, D. "The Haps Users Manual", Martin Marietta Corp. Report MML-TR-86-20, 1985.

[SANBORN86] Sanborn, J. "Heuristic Search Methods for Production Systems," Martin Marietta Corp. MML-TR-86-36, 1986.

[SAUERS83] Sauers, R. and Walsh, R. "On the Requirements of Future Expert Systems," *Proc. IJCAI* **8**, 1983.

[SAUERS85] Sauers, R "Controlling Expert Systems," in *Computer Expert Systems*, Bolc & Coombs (eds), Springer-Verlag, 1985.