# The Taming of R1

Arnold van de Brug, Judith Bachant, and John McDermott
Carnegie-Mellon University

For several years, Digital Equipment Corporation has used a system called R1 (or sometimes Xcon) to configure the computer systems it manufactures. The most recent account of R1's development[1] notes that as R1 has grown to several thousand rules, maintaining and developing it have become substantially more difficult. As we explored what kind of tool could facilitate knowledge acquisition for R1, we saw that the most valuable tool would (1) help determine the role any new piece of knowledge should play and (2) suggest how to represent the knowledge so it would be applied whenever relevant. Several researchers in recent years[2-4] have stressed that to maintain and to continue to develop a knowledge base it is critical to identify the various knowledge roles and to represent the knowledge in a way that does not conflate these roles. What is not yet clear is how many interestingly different roles exist and, if there are many, how one identifies the appropriate subset for a particular expert system.

We believe we will find the answers to these questions by studying knowledge roles in different problem-solving methods. Our approach is to develop knowledge acquisition tools that make explicit the knowledge representation implications of various methods. Until recently, most of the research in knowledge acquisition tools has concentrated on tools for classification problem-solvers.[5] Because knowledge acquisition tools such as Teiresias,[6] ETS,[7] and More[8] presuppose relatively similar problem-solving methods, the systems built with these tools have similar knowledge roles. However, knowledge acquisition tools for constructive problem-solvers are now being developed—e.g., Salt[9] and Sear, the knowledge acquisition tool we describe in this article—that are based on problem-solving methods significantly different from classification problem-solving methods. Because the problem-solving method that Sear presupposes is significantly different not only from classification problem-solving methods but also from the constructive method presupposed by Salt, the diversity of possible roles is becoming more apparent.

In this article, we discuss how a problem-solving method can influence the development of a knowledge acquisition tool. We first investigate why adding knowledge to R1 in its current form is difficult and find that the main reason is that R1's knowledge roles are ill-defined. We then examine another computer-system configurer, Rime, and the explicitly defined knowledge roles that are part of its problem-solving method. Finally, we indicate how Rime's problem-solving method served as the basis for the knowledge acquisition tool, Sear.

This article is a revised version of an earlier paper, "Doing R1 with Style," presented at the Second Conference on AI Applications, Miami, Fla., 1985.

## R1's approach to configuration

In performing the configuration task, R1 takes as input a list of components a customer has ordered and produces as output a set of diagrams of the interrelationships among those components. The initial list of components may be incomplete (i.e., it may not be possible to configure a functional system with that set of components) and, if so, R1 must add appropriate components.

**R1's problem-solving method.** Because of the number of possible combinations of components, the only reasonable approach to configuring the components is to construct (as opposed to select) an appropriate system. Generally, con-

structive tasks perform heuristic search, that is, a combinatorial search in which candidate partial solutions are constructed and their potentials evaluated. R1 can for the most part, however, avoid the combinatorial search (and thus avoid backtracking) by using small local searches for additional information at steps where there is ambiguity about what next action is most appropriate.[10] In other words, local cues are ordinarily sufficient to drive R1 along a path to a solution.

R1's problem-solving method selects the next piece of knowledge to apply from among those associated with the currently active subtask. Ordinarily, only a few are relevant at any given time. A piece of knowledge is considered relevant whenever the pattern defining its relevance can be instantiated by elements describing the current state of the world. When more than one piece of knowledge is relevant, the problem-solving method relies on very general heuristics, such as the recency of the elements and the specificity of each pattern, to determine which piece to apply.

Thus, R1's problem-solving can be characterized as follows: Given that it's involved in some task, it will take whatever next action (i.e., apply whatever knowledge) is relevant; if more than one piece of knowledge is potentially relevant, the choice will be made on the basis of very general considerations; if there is no more knowledge relevant to the current task, R1's attention returns to the parent task; whenever R1 does not have enough information to confidently prefer one possible action to all other candidate actions, it does some local problem-solving (e.g., by invoking some information-gathering subtask) until sufficient information has been collected.

**Why it's hard to add knowledge to R1.** R1's problem-solving method does not provide expert configurers with clear guidelines about what knowledge they are expected to share. In particular, a person adding knowledge to R1 could use substantially more help in (1) how to go about bounding the potential relevance of a piece of knowledge and (2) how to determine which piece to apply when more than one is relevant.

The relevance of each piece of R1's knowledge is defined by a pattern (i.e., a set of conditions); the pattern specifies, for some subtask, the circumstances under which the piece of knowledge can be applied. Because R1 has no defined knowledge roles, the way relevant pieces of knowledge are chosen cannot be explicitly expressed. The problem-solving method provides no vocabulary for an expert to describe the various roles knowledge will play in the performance of a task. Knowledge has been represented in R1 in various ways; regularities, to the extent they exist, have gone unnoticed. One has to know R1 well to modify its behavior in some desired fashion. Since R1 now has so much knowledge, gaining such familiarity has become more and more formidable. It is thus hard to communicate to the variety of people adding knowledge to R1 what is required of them. R1's problem-solving method is just a problem-solving inclination that has to be further specified by the knowledge it uses.

## Rime's approach to configuration

The configuration task Rime is being groomed for is identical to R1's task. But Rime currently has only about an eighth of the knowledge R1 has. Rime's competence is in the area of unibus configuration (which was R1's earliest area of expertise) where it can configure three of the twelve system types that R1 can configure.

**The problem-solving method.** The primary problem-solving method used by Rime has been derived from work done on R1-Soar[11]—an experiment in knowledge-intensive programming using a general problem-solving architecture called Soar.[12] The major difference between R1-Soar and Rime is that for Rime Soar's general problem-solving method has been tailored for tasks that can be solved by a strongly recognition-driven problem-solver. Also, Rime has several "auxiliary methods," not described in this article, that are useful in a variety of special circumstances.

Problem-solving in Rime (as in R1-Soar) is done in problem-spaces. A problem-space consists of a set of operators and pieces of knowledge that indicate the conditions under which these operators might appropriately be applied. A problem-space is the arena within which part of a complex operator from a parent problem-space is implemented. Rime's problem spaces serve much the same function that subtasks do for R1; each problem space corresponds to a part of the configuration task that expert configurers have named. The difference between R1 and Rime is that Rime's problem-solving method imposes an additional level of organization on its knowledge. Within each problem-space there are six roles for knowledge to play: propose-operator, reject-operator, evaluate-operator, apply-operator, recognize-success, and recognize-failure. Rime's primary problem-solving method is defined in terms of these knowledge roles. Before this method is described, some discussion of each of these roles is appropriate.

*Propose-operator.* Propose-operator knowledge suggests what operators might solve the problem at hand under the current set of circumstances; it is also knowledge of the relative static desirablity of those operators. For example, three operators that might be proposed in the configure-unibus problem space are the configure-module, the configure-backplane, and the configure-bus-repeater operators. Each is sometimes appropriately applied. The configure-module and configure-backplane operators, if applicable, are always to be preferred to the configure-bus-repeater operator. The choice between configure-module and configure-backplane is dictated by a variety of situational cues.

*Reject-operator.* Reject-operator knowledge is used to reject inapplicable operators proposed by propose-operator

knowledge. For example, the configure-module operator might be rejected because of insufficient power of a particular type. The principal reason for separating propose-type knowledge from reject-type knowledge is to allow additional information to become available (the reasons why certain operators are rejected) which in turn can make it possible to select a more appropriate operator.

*Evaluate-operator.* Here the task is to favor one of the proposed operators on the basis of whatever domain-specific considerations are relevant. A piece of knowledge favors one proposed operator over another under some specific set of circumstances. For example, the configure-module operator would be preferred to the configure-backplane operator if the pinning type of the next module to be configured is the same as the pinning type of the next available slot in the backplane being filled.

*Apply-operator.* Apply-operator knowledge defines the actions that are to be performed when some operator is applied. For example, when the configure-module operator is applied, the boards comprising the module must be associated with particular slots in the backplane.

*Recognize-success.* Recognize-success knowledge indicates how to determine when a subtask has been satisfactorily completed. For example, if all modules have been configured then there is nothing more to do in the configuration-module problem-space.

*Recognize-failure.* Recognize-failure knowledge indicates how to determine when the current approach to performing a subtask is not going to succeed. For example, no space remaining in the backplane currently being filled indicates that more space must be identified before the task can be finished.

Just as R1 always selects the next piece of knowledge to apply from among those pieces of knowledge associated with the currently active subtask, so Rime selects the next piece of knowledge to apply from among the knowledge associated with the currently active problem space. But whereas there was little more to say about how R1 selects knowledge, Rime's problem-solving method is substantially better specified. R1 and Rime use essentially the same knowledge to perform their tasks, but because Rime's method makes the roles that knowledge can play explicit, it is easier to talk (and think) about how Rime uses its knowledge and about what knowledge it needs to perform its tasks. Whenever it performs a subtask, Rime always sequences one or more times through a series of steps. Within each step, there is never any issue of what action to perform next because the method was designed to eliminate all control issues inside steps. Thus, although Rime ordinarily has many rules that are satisfied at any given time, it never matters in what order the satisfied rules are executed. Within any step, any of the rules that are satisfied can be executed in any order. When no rules are satisfied, control moves to the next step.

*Step 1: Propose candidate operators (propose-operator and reject-operator).* Step 5 of Rime's method (below) applies the operator that has been selected; if this operator is complex, its application becomes a problem to be solved in another problem-space. Rime's first step in the new problem-space is to propose operators that are relevant to the current situation and to reject those whose pre-conditions are not satisfied. Rime can be sure at the end of this step that all of the operators that could plausibly be applied are available for consideration.

*Step 2: Eliminate obviously inferior candidates. Rime's second step is to try to prune some of the candidate operators.* If there are candidate operators whose preconditions are all satisfied, any operators whose preconditions are not all satisfied are pruned. If there are candidate operators whose preference class is lower than the preference class of other candidate operators, those operators in the lower preference class are pruned.

*Step 3: Evaluate the remaining candidates (evaluate-operator).* During the third step, Rime compares the candidate operators that remain after the second step with one another. Each circumstance that suggests that one of the operators is less appropriate than another results in the less appropriate operator being pruned. At the end of this step, all of the evidence that Rime has that allows it to discriminate among the candidate operators has been taken into account.

*Step 4: Select one operator.* In Step 4, if more than one candidate remains, Rime selects one of the candidates at random.

*Step 5: Perform the actions associated with that operator (apply-operator).* Rime's fifth step is to apply the operator selected in the previous step. If the operator can be realized within the current problem-space, whatever actions are performed to realize this operator are performed during this step. If the operator is complex and can only be realized by invoking another problem-space, the problem-space in which the selected operator can be realized is invoked.

*Step 6: Quit if there is nothing more to do (recognize-success and recognize-failure).* In the sixth step, Rime looks for evidence that it has done all that can be done for now in the current problem-space. If it recognizes that it has done everything it can, control returns to Step 5 in the parent problem-space.

*Step 7: Iterate.* If Rime finds itself at the seventh step, it knows that it is appropriate to iterate through the steps again, and so it goes to Step I.

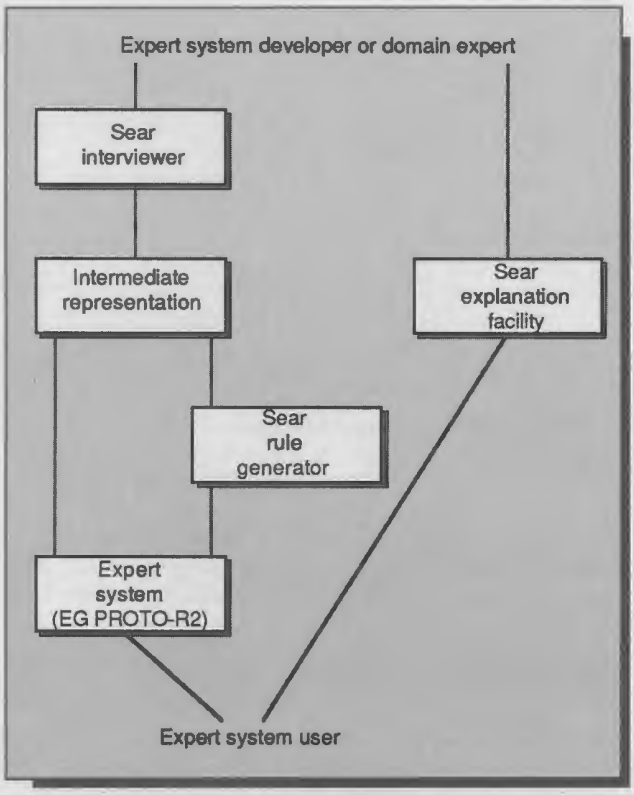


**Figure 1.** An overview of Sear.

*Why it's easier to add knowledge to Rime.* The problem-solving method used by Rime provides more direction to someone adding knowledge than does R1's method. This is in part because Rime's method integrates explicitly defined knowledge roles and in part because the person adding the knowledge can specify the conditions under which one piece of knowledge should be applied in preference to another.

Knowledge must be added in such a way that the new knowledge interacts well with existing knowledge. A limited number of clearly defined knowledge roles make adding knowledge easier because there is less uncertainty in the mind of the person adding the knowledge about how and when knowledge gets used. Each rule is responsible for some aspect of the system's behavior. Rules that have the same knowledge role interact minimally; rules with different roles have well-defined interactions. Thus there is less danger that adding a piece of knowledge will result in some unwanted behavior because of some unexpected interaction with existing pieces of knowledge.

## Configuration knowledge acquisition

**Sear—a knowledge collector and organizer.** For R1 and perhaps many other large expert systems, the problem of knowledge acquisition can be more appropriately viewed as the problem of knowledge maintenance. Sear is a knowledge maintainer that presupposes a problem-solving method—the method used by Rime. Figure 1 shows an overview of Sear. Its major components are an interviewer and a rule generator. The interviewer elicits domain knowledge from a knowledge engineer or a domain expert and puts it into an intermediate representation. The purpose of the intermediate representation is to provide a storehouse of domain knowledge in a declarative form. The Sear rule generator converts the intermediate representation into OPS5 rules that "proceduralize" the knowledge; that is, the knowledge is represented in a way that tailors its usefulness to a problem-solving method so that there is no need to search the knowledge-base when solving a problem.* The explanation facility is independent of the expert system since access to knowledge in a declarative form is required for adequate explanation.

To give a sense of how Sear can assist the knowledge-base maintainer, we will focus on two of the roles that knowledge can play in Rime: rejecting an operator and applying an operator. We first identify the knowledge the rule generator puts in the rules that play each of these roles. We then indicate how much of that knowledge Sear needs to elicit from the user and how much is given by the knowledge role.

Each rule that Sear generates consists of condition elements and action elements. Condition elements are patterns or templates that match objects defined by attribute-value pairs; action elements can create or modify objects. The left hand side of a rule that rejects an operator consists of a condition element that identifies the role of the rule, a description of the operator to be rejected, and typically three or four more condition elements that define a class of situations in which it would be inappropriate to apply the operator. The right hand side of the rule is a single action element that tags the operator with the reason it is being rejected. An apply-operator rule has a condition element that identifies the role of the rule, and it typically has at least three or four condition elements that are instantiated by the objects to be operated on by the rule. There are several action elements, each of which somehow modifies the current state.

Although there is a significant amount of variation in the form of rules that play the same role, the fact that the knowledge in these rules will be put to the same use allows Sear, given a small part of some piece of knowledge, to form strong expectations about what the rest of the knowledge will be. In the case of reject-operator rules, Sear bases its expectations primarily on two pieces of information: the relevant problem-space and the knowledge role. In the example in Figure 2, the five prompts indicate what information

*It is perhaps worth noting that for people who find the concept of knowledge tailored to a single problem-solving method too limiting (and surely single-method problem-solvers are shallow—no matter what the method), Sear's intermediate representation could be the source of many different sets of rules, each set tailored to a different method and the sets collectively providing substantial robustness. (But see Lenat.[13])

```
PROBLEM-SPACE: CONFIGURE-MODULE
ROLE: REJECT-OPERATOR
MODULE: HEX-SLOTS-REQUIRED
RESTRICTION: SCOPE SLOTS
OPERATOR: REASON SLOT-SPACE HEX

(P CONFIGURE-MODULE:REJECT:300A:REJECT-FOR-SPACE
          (GOAL ↑ACTIVITY-PHASE CURRENT ↑STEP PROPOSE-OPERATOR ↑PROBLEM-SPACE CONFIGURE-MODULE)
     {    (OPERATOR ↑ACTIVITY-PHASE PENDING ↑STATUS PROPOSED ↑TOKEN <TOKEN>
               ↑PROBLEM-SPACE CONFIGURE-MODULE) <OPERATOR>}
          (MODULE ↑ACTIVITY-PHASE CURRENT ↑TOKEN <TOKEN>
               ↑HEX-SLOTS-REQUIRED {<HEX-SLOTS-REQUIRED> <> NIL})
          (CONTAINER ↑CAPACITY-PHASE CURRENT ↑CLASS BACKPLANE ↑TOKEN <BACKPLANE-TOKEN>)
          (RESTRICTION ↑TOKEN <BACKPLANE-TOKEN> ↑SCOPE SLOTS
               ↑HEX-SLOTS-REMAINING {>=<HEX-SLOTS-REQUIRED> <> NIL})
     →
          (MODIFY <OPERATOR> ↑STATUS REJECTED ↑REASON <SLOT-SPACE> ↑REASON-QUALIFIER HEX))


IF   THE ACTIVE PROBLEM-SPACE IS THE ONE IN WHICH MODULES ARE CONFIGURED
     AND A MODULE HAS BEEN PROPOSED
     AND IT REQUIRES MORE HEX SLOTS THAN ARE AVAILABLE IN THE BACKPLANE BEING FILLED
THEN REJECT THE IDEA OF CONFIGURING THAT MODULE
```

**Figure 2.** A sample reject-operator rule.

Sear asked the user for in this particular situation; the rest of the figure shows the OPS5 rule Sear created on the basis of that information and an English translation of the rule. The rule illustrates how much can be inferred by Sear given just a rudimentary understanding of the computer system configuration domain and an indication of the role the knowledge will play. Sear's knowledge of the configuration domain is currently limited to a small collection of quite general heuristics; for example, Sear knows that particular kinds of containers are ordinarily associated with particular kinds of objects. When Sear is told that the problem-space is the one in which modules are configured and that the knowledge role is reject-operator, it can create a goal element. It knows, since the rule concerns configuring modules, that the operator to be rejected is likely to have configure-module as its problem-space. It also knows that the rule will need an element that matches some module, so it prompts for the attributes of module that are relevant to the decision to reject. When Sear is told that hex-slots-required is a relevant consideration, it assumes that there will be a corresponding restriction element that requires hex-slots-remaining to be less than hex-slots-required and prompts for the scope of that element. It also assumes that the restriction is associated with the backplane currently being filled. Sear asks for the reason the operator is to be rejected and then creates an action element to modify the rule.

In the case of apply-operator rules, Sear bases its expectations primarily on the same two pieces of information: the relevant problem-space and the knowledge role. In the example in Figure 3, the five prompts indicate what information Sear asked for. Here, all that Sear needs to determine to generate a rule is how the backplane that is being configured and the box that it will occupy will be changed when the operator is applied. When Sear is told that the problem-space is the one in which backplanes are configured and that the knowledge role is apply-operator, it can create a goal element and also can create the appropriate operator element. Sear knows, since the problem-space has to do with configuring backplanes and the knowledge role is apply-operator, that the rule is likely to extend its understanding of the partial configuration involving the box and the backplane. Given the actions specified by the user, it infers that the rule will need to match the element describing the box's role in the resulting partial configuration, the element describing the box, the element containing information about the backplane's position in the box, and the element describing the backplane.

**Rime's future.** R1 in its current form is an extremely successful expert system. It can configure almost all of Digital's PDP-11 and VAX-11 computer systems, and knowledge engineers continue to extend the knowledge base to handle new products as well as add new functionality. But both of these tasks are becoming increasingly difficult. R1 consists of about 4000 production rules and a database of 10,000 component descriptions. As R1 continues to grow, the maintenance task could well become impossible. Thus there are strong reasons to consider rebuilding R1 using Sear.

However, the work we have done so far on Rime does not provide a clear picture of how much effort would be required to rebuild R1, nor does it tell us how much easier the new system would be to maintain. Rime currently consists of only about 250 rules and can configure only three system types. There is some reason to think that Rime's knowledge is more densely represented than R1's knowledge; part of not having a clear understanding of the roles knowledge plays in R1 is that redundancy creeps into the system. However, on the basis of the work that has been done on Rime so far, it seems unlikely that Rime's knowledge could be more than twice as dense as R1's. Of more concern than the relative size of R1 and Rime is how to extract knowledge from R1 and convert that knowledge into Rime rules. R1 rules represent configuration knowledge accumulated over

DOCKET ALARM

# Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

## Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

## Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

## Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

## API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

**LAW FIRMS**
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

**FINANCIAL INSTITUTIONS**
Litigation and bankruptcy checks for companies and debtors.

**E-DISCOVERY AND LEGAL VENDORS**
Sync your system to PACER to automate legal marketing.