Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule-Base

Elliot Soloway
Department of Computer Science
Yale University
New Haven, Connecticut 06520

Judy Bachant and Keith Jensen
Digital Equipment Corporation
Intelligent Systems Technology Group
Hudson, Mass. 01749

Abstract

XCON is a rule-based expert system that configures computer systems. Over 7 years, XCON has grown to 6,200 rules, of which approximately 50% change every year. While the performance of XCON is satisfactory, it is increasingly becoming more difficult to change. With the goal of facilitating maintenance, DEC has developed a new rule-based language, RIME, in which the successor to XCON, XCON-in-RIME, is being written. This paper evaluates the potential for enhanced maintainability of XCON-in-RIME over XCON.

I. Introduction: Motivation and Goals

The following properties of XCON, an expert system, make it a particularly interesting system to examine:

- *XCON performs a complex design task:* XCON configures computer systems for DEC; XCON is used in a production mode, day in, day out -- it has been used since January 1980.
- *XCON is a very large rule-based system:* currently there are approximately 6,200 rules in XCON, which draw on a database of approximately 20,000 parts.
- *XCON undergoes constant change:* 50% of the rules in XCON are changed each year.

While there is no problem with XCON's performance, DEC nonetheless decided to redesign XCON: as we will describe below, it has become increasingly more difficult to *change* XCON. Since XCON must continually be updated to reflect new products and new computing concepts coming out of DEC, it was deemed desirable to develop a rule-based architecture that would be more supportive of this type of activity. In this paper, then, we will present an assessment of the redesigned XCON, called XCON-in-RIME, from the perspective of maintainability; we will mount two types of arguments (an *in principle* argument and an *in practice* argument) to support the view that XCON-in-RIME will be more maintainable. While the discussion here necessarily will be focused on XCON and XCON-in-RIME, we feel that the issues we raise will become increasingly more relevant --- and familiar --- as expert systems grow in size and complexity.

[1] The following are trademarks of Digital Equipment Corporation: XCON, RIME, XCON-in-RIME, DEC.

II. Problems With XCON's Current Rule-Based Architecture

XCON started as a relatively small, rule-based system (about 700 rules) (McDermott, 1982). It has grown to over 6,200 rules to meet the needs of DEC. Frankly, there is no end insight: XCON will continue to expand and change. Unfortunately, the problems of continually updating such a large system *do not grow linearly;* moving from 700 rules to 6,200 rules, with 50% of the rules changing every year, makes for an exceedingly difficult software enhancement problem. In particular, two basic properties of production rules give rise to these difficulties:

Dynamic properties of rules: As the number of rules grows --- and as different programmers work on the same rule-base, with different levels of understanding of what is in the rule base *and why* --- inadvertently, rules that are not appropriate become triggered, resulting in unwanted and undesirable interactions among the rules. In OPS5, control of rule firings is either implicit, in the domain-independent, conflict resolution strategies (e.g., recency), or it is explicit, but buried in rules themselves (e.g., special tricks are used to cause one rule to fire over another.)

Static properties of rules: There are no language restrictions on the number of functions a particular rule can perform. For example, in Figure 1, we see an Englishified XCON rule that performs a number of functions (i.e., actions on the right-hand side of the rule). This open-endedness causes problems as the rule-base grows. In particular, a typical strategy for extending the rule base to handle a new device is to copy the rules that worked for a similar device and then edit them to handle the new device. Unfortunately, in the editing process, one isn't always sure what the rationale for all the functions are. The result is that one often inadvertently changes a function, and causes run-time problems; alternatively, one doesn't change the functions, but keeps them in the new rules --- not feeling all that confident about why they are there.

In software engineering terms (Brooks, 1975), what happens to a large rule base as it changes over time is a "degradation in integrity:" what may once have been a coherent rule base, turns into a rat's nest of special rules, tightly coupled rules, etc. While software engineers have been able to label this problem, e.g., see (Soloway, 1987) they have not presented a general solution to the problem. Note that by "degradation" we do not mean that the performance of the system is necessarily impaired, e.g., XCON continues to function quite productively. However, from a rule-developer's perspective, the rule-base no longer has its initial unity of structure, of coherence, thus making additional changes increasingly more problematic.

```
Name: r1-unmounted-ubx-options
LHS: Describes certain types of cabinet mountable
     disk drives and information necessary to place
     one in a cabinet, cable it, and create output
RHS:  * Marks the drive "temporarily configured",
      * marks the placement in the cabinet "used",
      * identifies all of the information for
        connecting  the drive to it's controller,
      * identifies the containing information
        between the drive and cabinet,
      * and creates output labeling
```

Figure 1: An XCON Rule

III. A New Language for Rewriting XCON: RIME

XCON-in-RIME is the successor to XCON; it will perform the same function as XCON but it is intended to be more maintainable, i.e., its integrity should be easier to preserve over time. RIME is the language in which XCON-in-RIME is being written. In turn, RIME produces OPS5 code. The major advance of RIME over, say OPS5, is that one can more easily make explicit domain knowledge, both in structuring of the rules themselves and in controlling the firing of the rules. (See also (van de Brug, et al., 1985, Chandrasekaren, 1983, Neches, et al., 1984, Clancey, 1983, Clancey & Letsinger, 1981).) Below, we identify the more important language features of RIME:

Problem Space - provides a domain-specific "bucket" into which to throw rules that have a common purpose. For example, in XCON-in-RIME, there are 40 problem spaces, each dealing with one functional aspect of the configuration problem, e.g., CONFIGURE-MODULE, SELECT-MODULE, SELECT-CONTAINER. Some problem spaces are hierarchically organized, e.g., SELECT-MODULE and SELECT-CONTAINER are functions that must be done in order to effectively CONFIGURE-MODULE.

Problem Solving Method - a domain-independent sequence of steps to solve a type of problem; each problem space uses one problem solving method. Of the 6 current methods, the most frequently used one is PROPOSE/APPLY, which is, for example, the method used for achieving CONFIGURE-MODULE, SELECT-MODULE, and SELECT-CONTAINER. In effect, methods explicitly acknowledge that there are problem solving algorithms. For example, in the PROPOSE/APPLY method there are the following steps (note the following is a simplified description):

- *PROPOSE Step:* first an operator (or operators) is suggested that might be relevant to the achievement of the current goal, e.g., in Figure 2 we present two rules that suggest a slot that might be used in finding a place for a drive. (Operators typically either represent objects, as in the example above, or actions, as in the case of proposing to go off to another problem space.)

- *ELIMINATE Step:* then, there are *domain-specific* rules that evaluate the appropriateness of the candidate operators and prune the operators down to one, e.g., in Figure 2 we present a rule that decides among the slots being proposed.

- *APPLY Step:* the selected operator is activated and executed.

- *EVALUATE Step:* finally, the goal is reviewed, and if it has been achieved, the problem space can exited; if the goal was not achieved, then the difficulty is handled by going through the problem space once again, or by going to another problem space.

Note that within each step the actual order of "rule firing" or activation is irrelevant. Control is realized either by the domain-independent steps in a problem solving method or by the domain-specific task level control of entering another problem space.

Subgroup - In order to help insure "one function, one rule," there is an additional Dewey-Decimal-like, *domain-specific*, classification imposed on the rules: each class makes explicit the function that the rule is performing. For example, in Figure 3 we present three rules, each of which performs a single function, along with the subgroup classification scheme. Note that this classification scheme is not related to control and implies nothing about the order of rule activation.

Rule Type - To help insure the creation of rules consistent with the categories of permisable rules, there are *rule templates* that serve as guides for rule creation.

With this necessarily brief description of XCON and XCON-in-RIME, we can now proceed to assess the impact of XCON-in-RIME's new architecture on its maintenance.

IV. The Problems of Software Maintenance: In General and In XCON

In a software maintenance task there is an existing body of code that must be augmented in some manner. Typically, the augmentation is readily understood --- the programmer knows what needs to be done. However, the problem is in understanding the existing body of code, and then knowing where and how to add the augmentation *so as not to disturb the rest of the code.* Thus, on the one hand, the maintainer's job will be facilitated if the code is "readable," while on the other hand, the code will remain in a readable state if the programming language facilitates "good programming practice." In effect, reading and writing are duals of each other, with the goal being "maintaining readable code." The question, then is, what will enhance the

```
Rule Name:
  select-drive-space:propose:
    110f:lowest-drive-slot
LHS:
  Identifies the lowest numbered drive
  slot in the current cabinet
RHS:
  Proposes that slot

Rule Name:
  select-drive-space:propose:
    110j:exclusive-rackmount-drive-space
LHS:
  Identifies a drive slot in which only
  certain types of drives can be mounted
  the current drive is one of those types
RHS:
  Proposes that slot

Rule Name:
  select-drive-space:eliminate:
    340c:prefer-exclusive-space
LHS:
  Two proposed slots,
    one of which has restricted use
RHS:
  Eliminates the other slot
```

Figure 2: Sample XCON-in-RIME Rules

```
Rule Name:
  configure-device:apply:
    200a:mark-device-configured
LHS:
  Unconfigured device chosen for activity
RHS:
  Marks it's status "configured"

Rule Name:
  configure-device:apply:
    420a:update-contained-number
LHS:
  The current device has a "position-on-bus" identified
RHS:
  That is the number used to identify this device
  on the output by filling in "contained-number"

Rule Name:
  configure-device:apply:
    430a:update-containing-info
LHS:
  The fact that the device being configured belongs
  in a cabinet and the previously chosen cabinet
RHS:
  Identifies that the device is contained in this cabinet
```

#	LEVEL 1	LEVEL 2	LEVEL 3
200	update-status-or-phase	component	
420	update-containership	contained	number
430	update-containership	containing	

Figure 3: Sample Subgroup Schema -- Rule Type: Apply

readability/intelligibility of code? Two properties can be identified that directly influence this issue:

- *Homogeneity:* a small number of readily discernible *plans* are used over and over again to accomplish the various, desired goals. A plan is a sequence of language constructs used to accomplish some stereotypic (i.e., oft occurring) goal (Rich, 1981, Soloway & Ehrlich, 1984). In contrast, non-homogeneous code contains idiosyncratic, different solutions to similar goals. For example, in the configuration task, the code of laying out a cabinet for different cabinets and different computers, should still have some common appearance. Afterall, the goals that need to be achieved are similar. Moreover, a reader should not be able to tell who wrote a particular chunk of code to realize a particular cabinet layout; different programmers should be using the same set of programming plans to realize comparable goals.

- *Predictability:* (1) the reader knows where to look next for an answer to a question, (2) the reader is not surprised by what he comes upon in the code, and (3) the reader can trust that nothing untoward is being done behind the scenes. For example, if one rule (in the case of production rule programming) serves more than one function, then point (3) may be violated. Similarly, if rules that are intended to serve a related function are distributed over the rule base, the reader may not realize he needs to look in a non-local region for key rules, and hence (1) may be violated.

Clearly, homogeneity and predictability are related: by definition, predictable code will be homogeneous, and vice versa. Homogeneity focuses on a property of the *code itself,* while predictability focuses on a property of the *use of the code.*

Those who have had to maintain XCON have repeatedly observed: (1) that XCON grows continually more non-homogeneous, and (2) that predictability in the XCON rule-base is exceedingly difficult. Why? The basic problem seems to be the fact that what a code reader needs to know about a subset of rules, say, in XCON *is not explicit* in the rules; a code reader needs to talk to the person who created the rules and/or tap into the "institutional memory" of how the rules evolved to where they are. For example, XCON rule developers use various tricks to force rules to fire in a particular sequence. And still further, rule developers use certain rules for more than one purpose. Thus, rule developers are often uncertain as to what XCON rules are really doing, and therefore they are afraid to modify the rules, lest some unwanted behavior might result. The problem, in a nutshell, then, is that a rule developer needs to understand at least a major portion of the rules before he can effectively make some change to the rule base.

We hasten to point out that XCON rule developers are not malicious individuals, purposely trying to undermine the project with their non-homogeneous, idiosyncratic code! Rather, the problem is that there have been few external, explicit mechanisms to capture the otherwise implicit knowledge. For example, the OPS5 language encourages the style of programming that has evolved, e.g., there are no effective language constructs to aid the rule developer in creating rules that do not have some order dependence. Also, coding practices have evolved without clear guidelines as to how rules should be written. Again, this is not really a fault of the rule developers; the issues of homogeneity, predictability, and nature of the task (a very large rule-base that continually is modified) were not apparent when XCON started to evolve. In fact, the lessons learned in working on XCON directly lead to XCON-in-RIME --- where weaknesses of the sort identified here are meant to be addressed.

The bottom line is this: it is *not* surprising that XCON is very hard to maintain (e.g, change, add, delete rules): the language in which it is written, the architecture of the system itself, and the coding guidelines do not facilitate rule change. In what follows, we present a rationale for why XCON-in-RIME does address the specific weaknesses of XCON and thus why XCON-in-RIME should be more maintainable than XCON.

V. In Principle: Why XCON-in-RIME Should Be More Maintainable Than XCON

Over the years, the configuration group at DEC has had the need to "push around" a rule-base architecture, e.g., (Bachant & McDermott, 1984). This extensive experience has led directly to the design of RIME and to XCON-in-RIME. In what follows, we identify two major factors in which RIME/XCON-in-RIME differs from OPS5/XCON.

A. RIME as a Higher-Order Language

In order to appreciate the evolution of RIME/XCON-in-RIME from OPS5/XCON, one needs to look to the history of the development of programming languages. That is, programming languages have continued to evolve towards more problem-specific applications: e.g., FORTRAN (FORmula TRANslation) was considered a major improvement over assembly language, because it allowed *scientists to write in their own, natural language: mathematical equations*. Similarly, APL, the new crop of spreadsheet languages (e.g., LOTUS, MULTIPLAN), etc. have all been specifically crafted to allow domain specialists to talk to the computer in a language natural to the domain.

It would not be a distortion to view OPS5 as at the "assembly language level:" afterall, OPS5 is an almost totally domain independent programming language, which allows the programmer considerable control, and hence leeway. In contrast, RIME has been designed specifically to reflect what has been learned about configuration, and about writing *and changing* large rule bases. For example, as mentioned before, XCON rule developers forced rules to fire in specific orders and still attempted to reuse subsets of rules for multiple goals. In contrast, RIME attempts to understand this need explicitly, and has created explicit language constructs to deal with this type of situation. For instance, notions such as *problem space, problem solving method, method step, rule type*, have been created to help the rule developer in making explicit the heretofore implicit procedural relationships between rules. Thus, RIME can be viewed as more towards the "spreadsheet end of the problem independent/dependent language continuum." As such, then, RIME could be considered a "higher-order language" in comparison to OPS5, much as FORTRAN is considered to be a "higher-order language" relative to assembly language.

The next question is this: what predictions can be made about maintaining XCON-in-RIME, written in RIME, on the basis of experience gained in maintaining systems written in other higher-order languages? In particular, how do higher-order languages help with respect to homogeneity and predictability of code?

- *Homogeneity:* The constructs of a higher-order language can be viewed as techniques for realizing oft occurring goals in the problems towards which the language is directed. Thus, similar problems in a domain will have similar solutions, which in turn makes for more homogeneous and less idiosyncratic code.

- *Predictability:* Given that the language constructs are more directed towards problems in the domain, the decomposition in the code tends to reflect the decomposition in the problems more explicitly. Thus, it should be easier to identify where subgoals are achieved, and hence where code can be changed.

Given the positive effects promised by the use of higher-order languages, it would be remiss on our part not to point out that horrendous looking code has been written in higher-order languages. Nonetheless, while hard numbers are few and far between, the *overwhelming sense* of the software engineering community is that *the use of higher-order languages has had a positive impact on maintenance*, e.g., (McGarry, 1982). Thus, on these grounds alone, it is quite reasonable to predict that XCON-in-RIME, written in RIME, a higher-order language, should be significantly easier to maintain than XCON, written in a arguably lower-level language.

B. The Programming Environment: SEAR and Coding Guidelines

Language constructs are not enough to ensure that rule developers use the constructs in the desired fashion. SEAR is a tool being developed that will directly interpret RIME code. Currently, SEAR provides on-line enforcement of coding guidelines, e.g., there are templates for each rule type which guide the creation of rules. The coding guidelines, and their enforcement via SEAR, correspond to "structured programming" practices advocated by the software engineering community as leading to more readable code. However, unlike these vaguely worded practices, SEAR's can be tuned to the specifics of the problem.

VI. In Practice: Providing Empirical Support For The Enhanced Maintainability of XCON-in-RIME

While an *in principle* argument needs to made, one would like to see *at least some glimmers* of evidence for the veracity of those *in principle* claims. In this section, then, we present empirical evidence that *does* bolster the *in principle* claims.

A. Data Collection Methodology

Our goal was to get a sense of strengths and weaknesses of XCON-in-RIME *from a user's perspective.* We interviewed, on a daily basis, 8 rule developers who rotated into the XCON-in-RIME project for a short period of time (1-2 weeks). These sessions were recorded on audio-tape. Interview data of this sort does not provide "statistical evidence" pro or con. However, anecdotal evidence of this sort has been found to be quite insightful and reliable, e.g., (Lewis, 1982, Littman, et al., 1986). Frankly, it does not seem appropriate at this stage to go to all the trouble of carrying out a methodologically rigorous, controlled-study --- the costs would be to high, and the benefits are not clear. Note that the observations described below were made on the basis of interviewing only 4 rotaters. However, the observations made by the additional 4 rotaters were in almost unanimous accord with those by the initial group of rotaters.

B. Observations On and Interpretations Of Rotaters' Experiences with XCON-in-RIME

The following is a distillation of comments made at the various debriefing sessions with the rotaters. In carrying out such a distillation, there is always the danger of oversimplifying or misrepresenting someone's comments. We have, of course, attempted to be as "fair" as we could in our interpretations. In what follows, we break the rotaters' comments down with respect to the issues of homogeneity and predictability of XCON-in-RIME code.

Comments on Homogeneity:

Observation of the Rotaters: "I can't tell who wrote the rules."

Interpretation: The rotaters all agreed that the rules they read in XCON-in-RIME had a certain homogeneity. In contrast, the rotaters all agreed that, by and large, they could tell who wrote a rule in XCON, i.e., that rules could differ substantially as a function of who wrote them. We feel that the homogeneity in the rules in XCON-in-RIME, in contrast to XCON, is quite telling: a reasonable interpretation of this difference is that XCON-in-RIME provides constraints and guidelines on the rule developers so that they tend to write similar looking rules. The similarity of the rules across different rule developers leads directly to the enhanced readability: when rule developer X sits down to read the current rule base, he will feel more confident that he has accurately assessed the content of the rules if the rules have a homogeneous nature. One of the major readability problems with the current rules in XCON is that rule developers have significant difficulty in figuring out what is being implied by the rules --- since different rule developers have different styles of writing rules.

Observation of the Rotaters: Each rotater had developed special rule writing "tricks" for creating rules in XCON.

Interpretation: We asked the various rotaters if they had developed any special techniques for coding rules in XCON. Each said they had. For example, one rotater introduced a mini-context mechanism by including a very general rule at the end of a set of rules; this general rule, then set a marker, which, in turn would allow another set of rules to fire. Another rotater included extra conditionals in his rule in order to insure that that rule would fire at a special time. Thus, this point is similar to the last point: the rules in XCON were often coded by rule developers using idiosyncratic styles ---- thus, making the XCON rule base less homogeneous and less readable by other rule developers.

Observation of the Rotaters: The tricks the rule developers were using typically permitted them to control the order in which rules were firing.

Interpretation: While in their "pure" state, production rules are not meant to have this almost algorithmic character, the reality is that problems may require this type of procedurality. In XCON-in-RIME this procedurality is *explicitly* acknowledged, and the problem spaces, steps, etc. allow a rule developer to explicitly encode the sequentiality that they wanted --- that they were using implicitly in XCON, and doing so with various coding tricks. Again, readability can only be enhanced if rule developers are given tools --- the explicit vocabulary of problems spaces, steps, etc. --- to help them in writing rules. The use of this explicit vocabulary facilitates the development of a homogeneous rule set.

Comments on Predictability:

Observation of the Rotaters: "I know where to go in the rule base to add some new rule."

Interpretation: A comment made almost universally by the rotaters was they felt that they could pinpoint where they needed to make a change in XCON-in-RIME's rule base. In contrast, a major problem with XCON's rule base was the difficulty in locating the place where the change needed to be made.

Observation of the Rotaters: "The rules are more organized."

Interpretation: By and large the rotaters all said something like the above statement. In unpacking what it means to be "organized," it appeared that the rotaters felt that rules had *specific* places to be, i.e., those familiar with the configuration task found the problem spaces, subgroup classification scheme, etc. to be natural, organizational units. Again, this observation reflects both a broader understanding of the task of configuration as well as the encoding strategy dictated by the design of XCON-in-RIME, i.e., the fact that there are multiple classification levels using explicit criteria.

VII. Concluding Remarks

Based on the two types of arguments just presented, there is clearly a prima facie case that: *XCON-in-RIME should be easier to maintain than XCON.* While that difference should be readily observable, it would nonetheless be more than academic to gather data on two types of measures:

- *Human performance:* How long does it take to change/add a rule(s)? How many bugs are made? How long does it take to identify and fix bugs?

- *Assessing readability status of rule base:* Does the rule base degrade as new rules are added/changed? How homogeneous are the rules after 6 months, 12 months, etc.?

However, in order to capture such data, we would first need to define some metrics (e.g., how does one quantify homogeneity?). Moreover, we should not expect that *all* the maintenance problems will be alleviated by XCON-in-RIME; afterall, there are many problems yet to be discovered (e.g., what happens when the rule base hits 18,000 rules? 27,000 rules?).

Finally, economic reasons dictate that an evaluation, of the sort described here, be carried out before one undertakes a redesign/reimplementation of a system of the magnitude of XCON. Moreover, as expert systems continue to become more of an engineering enterprise, we will need to develop a range of evaluation tools: *evaluation is an integral part of an engineering effort.* Thus, besides evaluating XCON-in-RIME's design, we have attempted to articulate one strategy for carrying out a design evaluation: *in principle* and *in practice* type arguments.

Acknowledgements

We would like to thank Diane Muise and Michael Grimes for their continuing contributions to the RIME Project, and Virginia Barker and Dennis O'Connor for their unflagging support, encouragement, and above all, patience.

References

Bachant, J., McDermott, J. R1 Revisited: Four Years in the Trenches. *AI Magazine*, 1984, *5(3)*, .

Brooks, F. *The Mythical Man-Month.* Addison-Wesley Publishing Co., 1975.

Chandrasekaren, B. Towards a Taxonomy of Problem Solving Types. *AI Magazine*, 1983, *4(1)*, .

Clancey, W. *The Advantages of Abstract Control Knowledge in Expert System Design.* Proceedings of the AAAI National Conference on AI, Washington, DC, 1983.

Clancey, W., Letsinger, R. *NEOMYCIN: Reconfiguring a Rule-based Expert System for Application to Teaching.* Proceedings of the Seventh IJCAI Conference, 1981.

Lewis, C. *Using the "Thinking-aloud" Method in Cognitive Interface Design.* Technical Report RC 9265, IBM Watson Research Center, Yorktown Heights, NY., 1982.

Littman, D., Pinto, J., Letovsky, S., Soloway, E. Software Maintenance and Mental Models. In Soloway, E., Iyengar, S. (Eds.), *Empirical Studies of Programmers,* Ablex, Inc., 1986.

McDermott, J. R1: A Rule-based Configurer of Computer Systems. *Artificial Intelligence*, 1982, *19*, .

McGarry, F. *What We Have Learned In The Past 6 Years: Measuring Software Development Technology.* Proceedings of the Seventh NASA/Goddard Workshop on Software Engineering, Md., 1982.

Neches, R, Swartout, W., Moore, J. *Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development.* Proceedings of the IEEE Workshop on Principles of Knowledge-based Systems, Denver, CO, 1984.

Rich, C. *Inspection Methods in Programming.* Technical Report AI-TR-604, MIT AI Lab, 1981.

Soloway, E. *"I Can't Tell What In The Code Implements What In The Specs".* Proceedings of the Second International Human-Computer Interaction Conference, Honolulu, Hawaii, 1987.

Soloway, E., Ehrlich, K. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, 1984, *SE-10(5)*, 595-609.

van de Brug, A., Bachant, J., McDermott, J. *Doing R1 With Style.* Proceedings of the Second IEEE Conference on AI Applications, Miami, FL, 1985.