Proc 3 Int. Conf on Super computing, MAY 1988
vol 2, 179-156

The Modification of a Rule-based Diagnostic System for Routinized Parallelism on the Butterfly™ Parallel Processor

Albert Boulanger
BBN Laboratories Inc.
10 Moulton St.
Cambridge MA 02238

Abstract

This paper describes the process of converting a simple rule-based system from the KEE™ expert-system shell to the Flex rule system. Flex is part of BBN's AI-TOOLS environment developed for parallel processors like the Butterfly computer. The concept of routinized parallelism within rule-based systems will be discussed. Using the example rule-based system, an approach will be presented for organizing rules into a structure that is amenable to routinized parallelism.

1. Introduction & Background

We have been developing an environment for building expert systems which capture and use the so-called "shallow knowledge" of experts. We have decided to call this knowledge "routine". As the term suggests, routine knowledge is used in solving problems "by routine". The process of making problem solving routine can be seen as a mapping of situations to actions as directly as possible. Therefore rules in a routine-knowledge rule system are represented as situation-action pairs that are relatively independent of each other. Interactions among rules are allowed to occur only between structured collections of rules called rule packets. Interactions between rules within a rule packet are not allowed. Because of this restructuring, rules in a packet can be evaluated in parallel. Dependencies are resolved between packets. The degree of potential parallelism among packets depends on the nature of the domain.

Achieving the potential parallelism in a rule-based system is a strong function of how the rules are linked and run together. Structural information about collections of rules, which is often lost or implicit, can be made explicit to modularize the rule-base and hence help in building a data-flow model of the rules. This structural information which is used in BBN's FLEX [18] system to organize rules into packets and identify dependencies between packets is readily available from experts and can be obtained during the knowledge acquisition process.

The FALCON system shall be used as a case study of parallelizing a rule-based expert system. The FALCON system [17] diagnoses a batch chemical process which makes paracetamol. Paracetamol, also known as acetaminophen, is the active ingredient in Tylenol®. The batch chemical process has five main stages. See Figure 1 for a schematic layout of the process. The original FALCON system was developed in the KEE™ expert-system shell [13]. The part of the rule-base that involved diagnosing quality control problems, which includes approximately 30 rules, was ported to BBN's AI-TOOLS environment [4], a system designed for parallel processors like the Butterfly computer.[1] The rule-based part of the AI-TOOLS environment is called Flex.[2] FALCON was used as the basis for a demonstration of process simulation and process diagnostics on a Butterfly system. For the purposes of the demonstration, data was provided by an object-oriented simulation of the chemical process.

The problem-solving strategy of FALCON is to first determine whether there is a problem in product quality by inexpensive checks of such final product attributes as smell, color, and texture. Then more expensive data is used to isolate the stage(s) at fault. For each of the stages selected as possibly faulty[3], a rule packet was run to isolate the problem within the stage. A diagram of the problem-solving flow is given in Figure 2.

As in KEE, rules in Flex are organized into rule packets. Each packet is a description that consists of a set of rules, local variables, and a list of arguments. The rules within a rule packet have access to all of the packet's local variables and arguments. Instances of rule packets, once they are stable, can be expanded and compiled to run as functions.

---

[1]Since this system was small, the port was done manually. However, there should not be any obstacle in building a computer tool to automate this process.

[2]The environment also includes a version of Xerox's portable implementation of the emerging Common Lisp object-oriented programming standard (CLOS) that has extensions for parallelism. The environment also includes editing and browsing facilities. See [4] for a further description.

[3]The FALCON rule-base can only select problems in four of the five stages, and it can only isolate the problem *within* three of the four stages: PNP Preparation, PAP Preparation, and Acidulation.

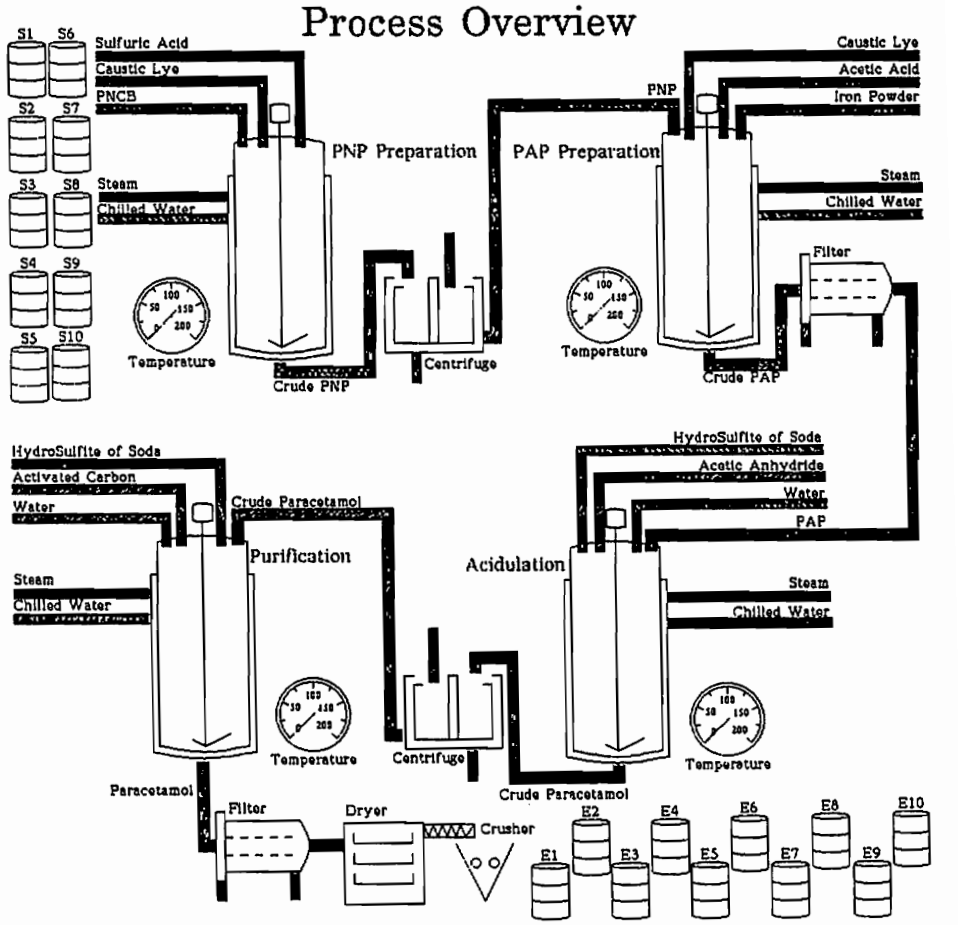

Figure 1: Schematic layout of the FALCON's associated chemical process. The five stages of the process are PNP Preparation, PAP Preparation, Purification, Acidulation, and Drying/Crushing.

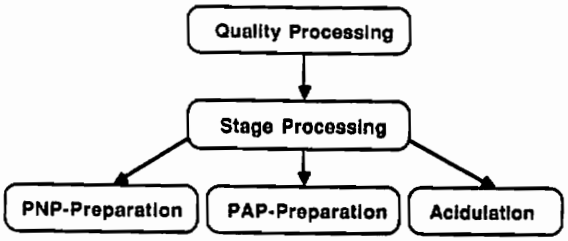

Figure 2: The problem-solving flow of the FALCON diagnostic rules.

Figure 3 is an example of the rule-development interface for Flex within the AI-TOOLS environment. The two major panes in the middle of the screen are a structure editor for Flex rules and a window for browsing through the history of rule execution. Some of Flex's syntax is illustrated in this figure. For instance, examine the last of the three rules in the structure editor and its corresponding paraphrasing (the PNP-RULE) in the history pane. Clauses in the rules are the #[] structures visible in the structure editor pane. These are called selectors.[4] The rules access an object-oriented database, and there is a nested-object-referencing syntax (for example [ANALYTICAL-TEST PH] gets the PH slot of the ANALYTICAL-TEST object). There is also an implied disjunction construction {} that is useful in checking for multiple values of a reference.

2. Related Work

The parallelization of OPS rule-bases has probably been the most explored of all rule systems. The CMU group has looked at both idealized parallel processors, the ILLIAC-IV, and the DADO processor [16], [7], [10], [11] for OPS. OPS on the NON-VON has also been investigated [19]. The parallelization of EMYCIN backward-chaining rules has been studied using Multilisp on a simulated parallel processor [14].[5] Our current paper emphasizes the approach one needs to take to parallelize a routine-knowledge rule-system.

For a discussion of the *performance* of such a parallelized system, see [3]. The work described in that paper demonstrated substantial performance improvements with a parallelized rule system running on a 16-node Butterfly. The system, which involved approximately 250 rules divided into 51 rule packets, achieved a performance improvement over the serial version by a factor of 7.[6]

---

[4]The semantics and part of the syntax of selectors comes from the work of Michalski [15].

[5]This work shares the *future* construct with the FALCON parallel implementation. See next section for more details on the future construct.

[6]See footnote [13] for current limitations to speedup.

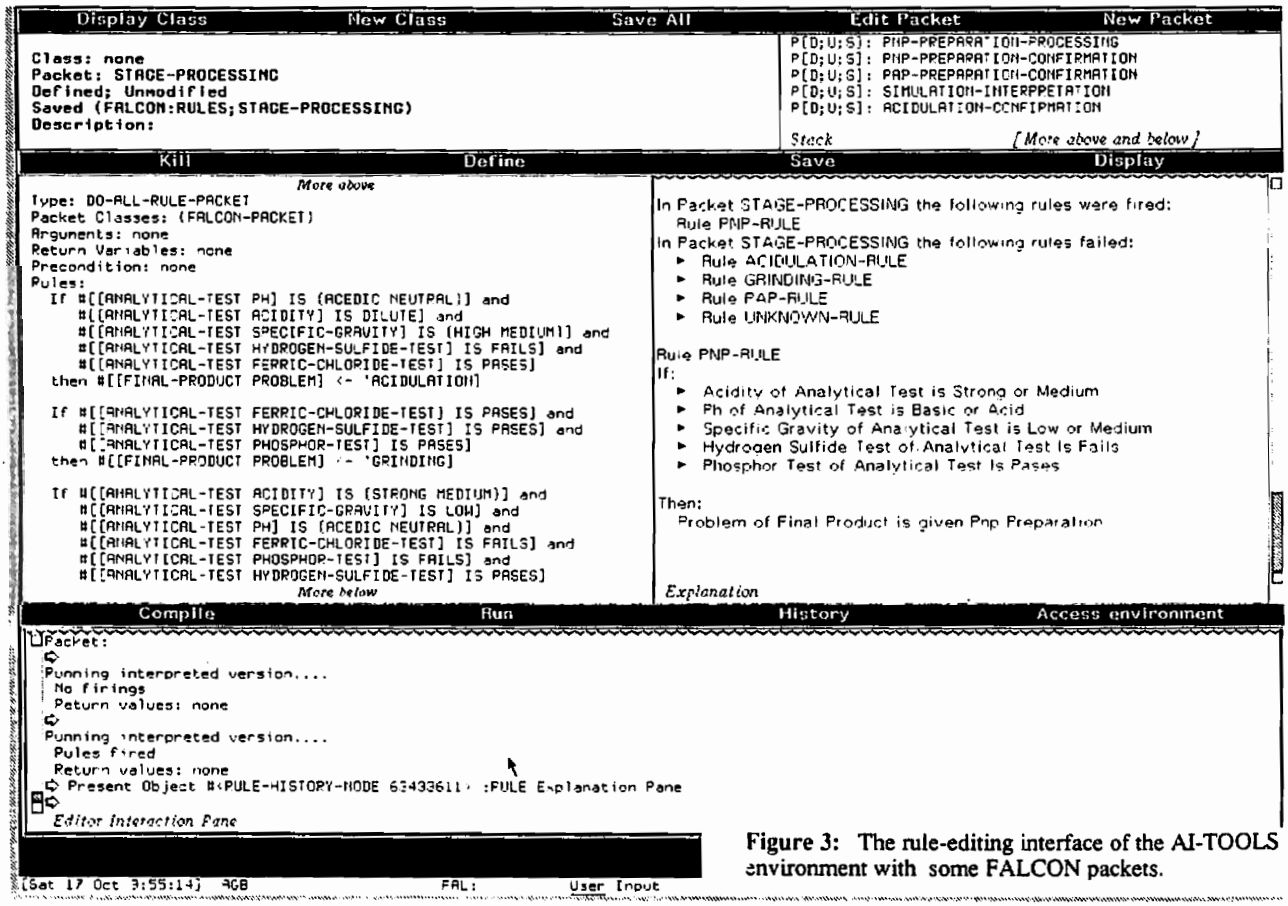

Figure 3: The rule-editing interface of the AI-TOOLS environment with some FALCON packets.

3. The Butterfly Parallel Processor

The BBN Butterfly computer is a shared-memory multiprocessor that contains up to 256 processor nodes interconnected by a high-performance logarithmic switching network. The memory-management hardware and the Butterfly switch permit the memories on each of the processor nodes to be treated as a pool of shared memory that is directly accessible by all processors. The processor configuration on which FALCON was developed had eight processors.

This pool of shared memory is crucial to the design of Lisp on the Butterfly computer. It enables Butterly Lisp to preserve the shared-memory quality that has always been characteristic of the Lisp language. This means that data structures of arbitrary complexity can be easily passed from one context to another by simply transmitting a pointer, rather than by copying. This approach has significant ease-of-programming and efficiency advantages.

Currently, Scheme is offered for the Butterfly system, although the AI-TOOLS environment will be for the Butterfly Common Lisp product. Butterfly Scheme is a shared-memory, multiple-interpreter system. In the Scheme implementation, the shared-memory capability of the Butterfly computer is utilized to provide a single heap, mapped identically by all interpreters.

Butterfly Scheme uses the *future* mechanism.[7] The following form:

```
(future <lisp-expression>)
```

causes the system to record that a request has been made for the evaluation of `<lisp-expression>`, and to commit resources to that evaluation when they become available. Control returns immediately to the caller, returning a new type of Lisp object called an *undetermined future*. This new object acts like a placeholder for the value that the evaluation of `<lisp-expression>` will produce. The undetermined future object can be stored into Lisp data structures, and passed as an argument, but it will be coerced to the value of `<lisp-expression>` if an operation requires the value of `<lisp-expression>`. Butterfly Scheme accomplishes this by suspending the task that caused the coercion until the task that is evaluating `<lisp-expression>` completes. See [1] and [2] for more details on Butterfly Scheme.

4. The AI-TOOLS Environment and the Butterfly Processor

In the FALCON system, the rules and the simulation are executed on the Butterfly computer, and a Symbolics™ Lisp Machine serves as a user interface to the system. The simulation was written in an object-oriented simulation package for Scheme called POSSUM.[8] The AI-TOOLS environment with the Flex rule system on the Symbolics served as the place of development for the FALCON rules. The rules were ported to Butterfly Scheme by the extensible rule expansion/compilation feature of Flex. This will be illustrated below:

Original packet:

```
Rule Packet PNP-PREPARATION-PROCESSING

If
  #[[FINAL-PRODUCT CRYSTALITY] IS {ROUGH SMALL-CRYSTALS}] and
  #[[PNP-PREPARATION TEMPERATURE-TEST] IS FAILS]
Then
  #[[PNP-PREPARATION HEATING-TIME] <- POSSIBLY-MORE]

If
  #[[PNP-PREPARATION TEMPERATURE-TEST] IS FAILS] and
  #[[PNP-PREPARATION CAUSTIC-LYE-TEST] IS FAILS]
Then
  #[[PNP-PREPARATION CAUSTIC-LYE ADDED-AMOUNT] <-
     POSSIBLY-MORE]
```

Expansion:

```
(DEFINE (PNP-PREPARATION-PROCESSING)
  (COND-EVERY-R
   ((AND (OR (IS (GET-ACCESS '(FINAL-PRODUCT CRYSTALITY))
                 'ROUGH)
             (IS (GET-ACCESS '(FINAL-PRODUCT CRYSTALITY))
                 'SMALL-CRYSTALS))
         (IS (GET-ACCESS '(PNP-PREPARATION TEMPERATURE-TEST))
             'FAILS))
    (SET-ACCESS  '(PNP-PREPARATION HEATING-TIME)
                 'POSSIBLY-MORE) T)
   ((AND (IS (GET-ACCESS '(PNP-PREPARATION CAUSTIC-LYE-TEST))
             'FAILS)
         (IS (GET-ACCESS '(PNP-PREPARATION TEMPERATURE-TEST))
             'FAILS))
    (SET-ACCESS '(PNP-PREPARATION CAUSTIC-LYE ADDED-AMOUNT)
                'POSSIBLY-MORE) T)))
```

Normally, a Flex rule packet expands into a Common Lisp function, but for interfacing to the Butterfly computer, the rule expansion/compilation facility was configured to expand packets using a Scheme syntax.[9] `COND-EVERY-R` comes from the Zeta Lisp `COND-EVERY` construct, which is like `COND` except that every true condition is executed instead of only the first one. `COND-EVERY-R` in Butterfly Scheme is a macro that expands into a function that shrink-wraps every N clauses in a `FUTURE`. N is a chunking factor that is useful in trading-off `FUTURE` overhead costs for the grain-size of the work. See [3] for an empirical analysis of the optimum chunking factor for a 250-rule application.

[7] First implemented at MIT by Robert Halstead for the Concert processor.

[8] POSSUM was developed by the MITRE Corp. for the Butterfly computer.

[9] Another feature to note: The rule expansion/compilation facility allows the general features of Flex with which the normal rule interpreter must deal, to be dropped in the expansion if the feature is not used in an application. In Falcon, uncertainty is not used, so the expansion of the rule packet does not involve any part of the uncertainty mechanism.

5. Parallelizing FALCON in Flex

The procedure used is to analyze the data flow of the original KEE rule base and to reorganize the rules into packets, so that none of the rules in a packet change the value of a variable that is referenced by another rule in that same packet. If the rule base consisted of made-routine knowledge[10], then this data flow analysis is straightforward. This process will be illustrated below.

The first rule packet to be executed is the one that checks to see if there is some problem with product quality. Here are some example rules from that packet:

```
If
  #[[FINAL-PRODUCT SMELL] IS SWEET-SMELL] and
  #[[FINAL-PRODUCT COLOR] IS WHITE]
Then
  #[[FINAL-PRODUCT QUALITY] <- EXCELLENT]

If
  #[[FINAL-PRODUCT COLOR] IS WHITE-WITH-PINK-DOTS] and
  #[[FINAL-PRODUCT SMELL] IS SWEET-SMELL]
Then
  #[[FINAL-PRODUCT QUALITY] <- BAD]

If
  #[[FINAL-PRODUCT COLOR] IS WHITE-WITH-PINK-DOTS] and
  #[[FINAL-PRODUCT SMELL] IS PNCB-SMELL]
  #[[FINAL-PRODUCT CRYSTALITY] IS POWDER]
Then
  #[[FINAL-PRODUCT QUALITY] <- POOR]

If
  #[[FINAL-PRODUCT COLOR] IS YELLOWISH] and
  #[[FINAL-PRODUCT SMELL] IS SWEET-SMELL] and
  #[[FINAL-PRODUCT CRYSTALITY] IS (ROUGH POWDER FINE-POWDER)]
Then
  #[FINAL-PRODUCT QUALITY] <- BAD]

.
.
.
```

and so-on for 11 more rules

Each of these rules updates the quality slot of the final product. Furthermore, the quality of the final product is not used in any of the conditions of these rules, so each of these rules can be evaluated in parallel if we use appropriate data-base locking on the quality slot. One may think that this data-base locking could be a potential bottleneck, but one should observe that on average only one (sometimes two or three) rules will fire. This is true for this application, and it is also true for rule-based systems in general [9], [11]. Parallelizing this packet was easy.

The second rule packet isolates the stage(s) at fault. Here are the rules from that packet:

```
If
  #[[ANALYTICAL-TEST FERRIC-CHLORIDE-TEST] IS FAILS] and
  #[[ANALYTICAL-TEST HYDROGEN-SULFIDE-TEST] IS FAILS] and
  #[[ANALYTICAL-TEST PHOSPHOR-TEST] IS FAILS]
Then
  #[[FINAL-PRODUCT PROBLEM] <- 'UNDETERMINED]

If
  #[[ANALYTICAL-TEST PHOSPHOR-TEST] IS PASSES] and
  #[[ANALYTICAL-TEST HYDROGEN-SULFIDE-TEST] IS PASSES] and
  #[[ANALYTICAL-TEST FERRIC-CHLORIDE-TEST] IS PASSES]
Then
  #[[FINAL-PRODUCT PROBLEM] <- 'GRINDING]

If
  #[[ANALYTICAL-TEST PHOSPHOR-TEST] IS PASSES] and
  #[[ANALYTICAL-TEST HYDROGEN-SULFIDE-TEST] IS FAILS] and
  #[[ANALYTICAL-TEST SPECIFIC-GRAVITY] IS (LOW MEDIUM)] and
  #[[ANALYTICAL-TEST PH] IS (BASIC ACID)] and
  #[[ANALYTICAL-TEST ACIDITY] IS (STRONG MEDIUM)]
Then
  #[[FINAL-PRODUCT PROBLEM] <- 'PNP-PREPARATION]

If
  #[[ANALYTICAL-TEST HYDROGEN-SULFIDE-TEST] IS PASSES] and
  #[[ANALYTICAL-TEST PHOSPHOR-TEST] IS FAILS] and
  #[[ANALYTICAL-TEST FERRIC-CHLORIDE-TEST] IS FAILS] and
  #[[ANALYTICAL-TEST PH] IS (ACIDIC NEUTRAL)] and
  #[[ANALYTICAL-TEST SPECIFIC-GRAVITY] IS LOW] and
  #[[ANALYTICAL-TEST ACIDITY] IS (STRONG MEDIUM)]
Then
  #[[FINAL-PRODUCT PROBLEM] <- 'PAP-PREPARATION]

If
  #[[ANALYTICAL-TEST FERRIC-CHLORIDE-TEST] IS PASSES] and
  #[[ANALYTICAL-TEST HYDROGEN-SULFIDE-TEST] IS FAILS] and
  #[[ANALYTICAL-TEST SPECIFIC-GRAVITY] IS (HIGH MEDIUM)] and
  #[[ANALYTICAL-TEST ACIDITY] IS DILUTE] and
  #[[ANALYTICAL-TEST PH] IS (ACIDIC NEUTRAL)]
Then
  #[[FINAL-PRODUCT PROBLEM] <- 'ACIDULATION]
```

One should note that these rules are also independent of each other. This rule set is run after the first because the analytical tests are more expensive to perform. This is an example of a domain-specific serialization of the rules.

Finally, if the problem is either with PNP-preparation, PAP-preparation, or Acidulation, then in the original system, a rule packet was run to isolate the problem with the corresponding stage of the chemical process. Let us now examine the original packet for PNP-preparation:

```
If
  #[[PNP-PREPARATION TEMPERATURE-TEST] IS FAILS] and
  #[[PNP-PREPARATION CAUSTIC-LYE-TEST] IS FAILS]
Then
  #[[PNP-PREPARATION CAUSTIC-LYE ADDED-AMOUNT] <-
     POSSIBLY-MORE]

If
  #[[PNP-PREPARATION CAUSTIC-LYE ADDED-AMOUNT] IS
     POSSIBLY-MORE] and
  #[[PNP-PREPARATION PNCB-POWDER COLOR] IS REDDISH-YELLOW]
Then
  #[[PNP-PREPARATION CAUSTIC-LYE ADDED-AMOUNT] <- MORE]

If
  #[[PNP-PREPARATION CAUSTIC-LYE ADDED-AMOUNT] IS
     POSSIBLY-MORE] and
  #[[FINAL-PRODUCT SMELL] IS ACID-SMELL]
Then
  #[[PNP-PREPARATION SULFURIC-ACID ADDED-AMOUNT] <- MORE]

If
  #[[FINAL-PRODUCT CRYSTALITY] IS (ROUGH SMALL-CRYSTALS)] and
  #[[PNP-PREPARATION TEMPERATURE-TEST] IS FAILS]
Then
  #[[PNP-PREPARATION HEATING-TIME] <- POSSIBLY-MORE]

If
  #[[PNP-PREPARATION HEATING-TIME] IS POSSIBLY-MORE] and
  #[[PNP-REACTION-TIME] >= 6]
Then
  #[[PNP-PREPARATION HEATING-TIME] <- MORE]
```

The rules in this packet are not independent and they have an implied ordering. They cannot work correctly if each rule is executed in parallel. Note that there are two types of rules in this packet: one that creates an initial hypothesis and another

[10] This is opposed to some applications in systems such as OPS where the Turing-equivalence of the system is used to do "rule-based" programming. To put this in another way, one can make use of the fact that a system is Turing-equivalent to do anything in it that one can do in any other computer language -- including general applications programming.

that confirms or refines the hypothesis. The author of the initial set of rules, Fenil Shah, in fact described this dual purpose of the rules verbally, but the distinction is not explicit in the original rule packet. One could rewrite this rule packet as two -- one for each intention of the original packet:

Initial Set:

```
If
  #[[FINAL-PRODUCT CRYSTALITY] IS (ROUGH SMALL-CRYSTALS)] and
  #[[PNP-PREPARATION TEMPERATURE-TEST] IS FAILS]
Then
  #[[PNP-PREPARATION HEATING-TIME] <- POSSIBLY-MORE]

If
  #[[PNP-PREPARATION TEMPERATURE-TEST] IS FAILS] and
  #[[PNP-PREPARATION CAUSTIC-LYE-TEST] IS FAILS]
Then
  #[[PNP-PREPARATION CAUSTIC-LYE ADDED-AMOUNT] <-
     POSSIBLY-MORE]
```

Confirming Set:

```
If
  #[[PNP-PREPARATION CAUSTIC-LYE ADDED-AMOUNT] IS
     POSSIBLY-MORE] and
  #[[PNP-PREPARATION PNCB-POWDER COLOR] IS REDDISH-YELLOW]
Then
  #[[PNP-PREPARATION CAUSTIC-LYE ADDED-AMOUNT] <- MORE]

If
  #[[PNP-PREPARATION CAUSTIC-LYE ADDED-AMOUNT] IS
     POSSIBLY-MORE] and
  #[[FINAL-PRODUCT SMELL] IS ACID-SMELL]
Then
  #[[PNP-PREPARATION SULFURIC-ACID ADDED-AMOUNT] <- MORE]

If
  #[[PNP-PREPARATION HEATING-TIME] IS POSSIBLY-MORE] and
  #[[PNP-REACTION-TIME] >= 6]
Then
  #[[PNP-PREPARATION HEATING-TIME] <- MORE]
```

The rules now within each packet can run in parallel; intentions that can be valuable in the life cycle of the rule system as it grows have been recovered and preserved. The rule packets of the other two stages were split up this way as well.

Another approach to parallelizing these rules would be to build a rule execution/compilation environment that could discover the data dependencies among rules and create an appropriate partitioning of the rules (as is done in OPS systems [8] and in EMYCIN [20]). From the examples above, we see that such an approach is inappropriate for a routinized-knowledge rule-base since a well-constructed rule base of this type will have a partitioning that reflects the data dependencies. This explicit partitioning is used to direct the parallelization. In fact, one could claim that for routinized knowledge, dependencies among rules within a rule packet indicate the failure of the rules to capture the natural partitioning of the domain -- it is the *goto* for rules.

Finally, observe that it is possible to increase the parallelism beyond what can be obtained by the process described above by taking a "cross-product" of rule packets that is akin to the data-base *join* operation. This process is complicated[11], generates a multiplicative increase in rules, and will not be elaborated here. Suffice to say, one can push the speed-memory trade-off even beyond what has been presented here for processors that offer massive parallelism.

6. Implications for Knowledge Acquisition

Knowledge acquisition is concerned both with the way in which knowledge is to be organized in the target rule system, and in the way humans organize knowledge. Knowledge in a routine knowledge system consists of packets of rules; these packets have external data dependencies but no internal data dependencies. The constraint that there be no data dependencies within packets substantially reduces the potential complexity of the rule system. If we visualize this as a data flow graph, we can see that organizing rules into such packets substantially reduces the number of entitities that have to be considered and the number of arrows (representing dependencies) linking these entities. Instead of having to create an entity for each rule and arrows denoting the data dependencies between all the rules in the system, we only have to create entities for each rule packet and arrows representing dependencies between these rule-packet entities.

Our hypothesis is that the human mind does, in fact, organize routine knowledge into the equivalent of rule packets and that the impressive human performance in the handling of routine tasks is partially attributable to the parallel execution of such *rule packets*. We further hypothesize that it is only when routine knowledge *fails* that humans resort to more complex and more abstract levels of knowledge. Such failure implies either that the human is confronting a novel situation or that feedback from a situation that was assumed to be routine has shown that it is not properly covered by the applied routine knowledge.

These hypotheses strongly influence our view of the knowledge acquisition process. Since our goal is to construct rule packets of the type we have already described, we seek to identify groups of rules which are relevant to a particular aspect of the domain but which have no data dependencies with each other. Since our hypothesis is that human routine knowledge actually consists of such rules we assume that rules of that sort should be easily elicited from human experts and our experience has confirmed that assumption.

Even without the motivation of our hypotheses, there is an advantage to acquiring knowledge and structuring rules using this approach. That advantage is the reduction in system complexity mentioned above and the resulting ease of maintenance. Inevitably, rule systems must be extended and modified after they have been developed. Such maintenance is much easier when the rules are organized into packets relevant to particular aspects of the domain and there are no data dependencies within packets. The maintenance problems that

[11] Especially when one wants to preserve the ordering of evaluation of conditions because they involve more expensive-to-obtain data.

result from not organizing rules into such packets are analogous to the problems caused by *goto* in procedural languages.

7. Summary

Our routine-knowledge rule system is less powerful in a computational sense than OPS, which is a Turing-machine-equivalent programming language. Rather than increasing concurrency by increasing the number of rules affected by an action, our system increases concurrency by requiring that rules in a packet be independent. Thus, we trade computational power, which is not needed for this kind of rule system, for execution performance.

Our position is that routine knowledge can be implemented in a rule system given the following condition: That the variables in a rule packet that are accessed in the left-hand side of the rules in that packet are disjoint from the variables modified in the right-hand side of the rules in that packet.

We view this characterization of routine-knowledge rule systems not as an implementation-imposed limitation, but as a reflection of the nature of the routine-knowledge that humans use in most of their decision making. It is fortuitous that this independence of rules from each other makes routine-knowledge rule-systems candidates for large-scale parallelism. All that is needed are routine-knowledge rule systems sufficiently large to take advantage of large-scale parallelism.[12] In our view, with the exception of database-related bottlenecks,[13] there are no theoretical limitations imposed by a rule interpreter to the amount of parallelism that can be achieved in a completely routinized-knowledge rule system.

We can imagine rule systems that are partitioned into routine-knowledge sections and what we might call nonroutine-knowledge sections. They could equal the computational power of an OPS-like production system in their nonroutine-knowledge sections while maximizing parallelism in their routine-knowledge sections. Currently, the best choice for implementing the nonroutine sections would be any Turing-equivalent computer language, although work has been done to explore languages for expressing generic problem-solving strategies encountered in expert system applications [5].

In closing, we would like to speculate that the theoretical limitations to parallelism in nonroutine-knowledge rule-systems, limitations that were well documented by Gupta, Forgy, [9], [10], [11], and Oflazer [16], are as valid for human beings as they are for computer systems. Knowledge that has not been made routine consists of chunks of knowledge that have sequential data dependencies. We conjecture that there is, in fact, a cognitive process of routinization of problem-solving skills in human experts, and that humans can execute this knowledge "in parallel". The process of acquiring expert knowledge in the form of rules is often an eliciting of this type of made-routine knowledge.

Human beings, like computers, can handle routine-knowledge rules in parallel because these rules are independent of each other. Nonroutine-knowledge involves some sequential processing -- even in humans.[14]

8. Acknowledgment

The author wishes to thank the other people involved in building the FALCON demonstration: Fenil Shah, Ken Anderson, Barry Coflan, Mike Thome, and Jeff Mattson. The author also wishes to thank Lisa Sokol of the MITRE Corporation for permitting the use of the POSSUM object-oriented package for the FALCON demonstration. This work was paid for, in part, by DARPA contract number MDA903-84-C-0033.[15]

[12] One could use the cross-product method mentioned above to enlarge the rule set size.

[13] Although it is true that there can be, and often is, a bottleneck due to the *variance* in task size. We believe however, that this variance can be made insignificant by careful partitioning of the work. Executing a whole rule with many conditions within a future can sometimes be too large a task size relative to the other rules, so we may need to split the condition evaluation of a rule into different tasks. This in not done currently. We have found that there is, in fact, a variance that is significant in the size of rules (measured by the number of conditions) in a 250-rule application. The statistics in [9] and [11] also support this observation. Another problem in achieving theoretical speedup is the overhead cost of the future mechanism and current limitations with future scheduling. These should become insignificant as Lisp on the Butterfly processor improves. See [6] for more details on the statistics on parallelizing the 250-rule knowledge base.

[14] With the search for emergent properties in the sense of Hopfield [12] in parallel neural-networks, one will find emergent serialization!

[15] Butterfly is a trademark of Bolt Beranek and Newman Inc.
KEE is a trademark of Intellicorp.
Symbolics is a trademark of Symbolics, Inc.
Tylenol is a registered trademark of McNEILAB, Inc.

References

[1] Allen, Donald C., Seth A. Steinberg, & Lawrence A. Stabile.
Recent Developments in Butterfly™ Lisp.
In *Proceedings AAAI-87: Sixth National Conference on Artificial Intelligence*, pages 2-6. Morgan Kaufmann, Los Altos, CA, July 13-17, 1987.

[2] *Butterfly Scheme Reference Manual*
Beta Test Version (11/5/87) edition, BBN Advanced Computers, 10 Fawcett St, Cambridge MA, 02238, 1987.

[3] Boulanger, Albert.
Parallelism in the Execution of a Routine Knowlkedge Rule System on the Butterfly™ Computer.
Report 6436, BBN Laboratories, 10 Moulton St, Cambridge MA, 02238, December, 1986.

[4] Boulanger, Albert, Dawn MacLaughlin, Marc Villain, & Ken Anderson.
Parallel Expert Systems Execution Environment: A Functional Specification. Version 2.1.
Report 6225, BBN Laboratories, 10 Moulton St, Cambridge MA, 02238, April, 1987.

[5] Chandrasekaran, B.
Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design.
IEEE Expert 1(3):23-30, Fall, 1986.

[6] Falk, Gil, Ken Anderson, Albert Boulanger, Dawn MacLaughlin, Marc Vilain, & Rich Shapiro.
Parallel AI Tools Project.
In *Butterfly Users Group Meeting*. BBN Advanced Computers, 10 Fawcett St, Cambridge MA, 02238, Spring, 1987.

[7] Forgy, Charles L.
Note on Production Systems and ILLIAC-IV.
Report CMU-CS-80-130, Carnegie-Mellon University, July, 1980.

[8] Forgy, Charles L.
Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.
Artificial Intelligence 19:17-37, 1982.

[9] Gupta Anoop, & Charles L. Forgy.
Measurements on Production Systems.
Report CMU-CS-83-167, Carnegie-Mellon University, December, 1983.

[10] Gupta, Anoop.
Implementing OPS5 Production Systems on DADO.
Report CMU-CS-84-115, Carnegie-Mellon University, March, 1984.

[11] Gupta, Anoop.
Parallelism in Production Systems.
Report CMU-CS-86-122, Carnegie-Mellon University, March, 1986.
This is Anoop's PhD thesis and is forthcomming as a book in Pittman's artificial intelligence series.

[12] Hopfield, J.J.
Neural Networks and Physical Systems with Emergent Collective Computational Abilities.
Proc. Nat. Acad. Sci. USA 79:2554-2558, April, 1982.

[13] *IntelliCorp™ KEE™ Software Development System User's Manual*
KEE Version 3.0, July 25, 1986 edition, IntelliCorp, 1986.
Document number: 3.0-U-1.

[14] Krall, Edward J., & Patrick F. McGehearty.
A Case Study of Parallel Execution of a Rule-Based Expert System.
International Journal of Parallel Programming 15(1):5-32, 1986.

[15] Michalski, R.S., & R.L. Chilausky.
Learning by Being Told and Learning from Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis.
International Journal of Policy Analysis and Information Systems 4(2):125-161, 1980.

[16] Oflazer, Kemal.
Partitioning in Parallel Processing of Production Systems.
In *Proc. 1984 International Conference on Parallel Processing*, pages 92-100. IEEE Computer Society Press, August, 1984.

[17] Shah, Fenil.
FALCON: Fault Location and Control Optimization Planner.
In *Conference of Simulation and Design*. Instrumentation Society of America, April, 1987.

[18] Shapiro, R.
FLEX - A Tool for Rule-Based Programming.
Report 5643, BBN Laboratories, 10 Moulton St, Cambridge MA, 02238, May, 1984.

[19] Shaw, David Elliot.
NON-VON's Applicability to Three AI Task Areas.
In *IJCAI-85: Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 61-72. Morgan Kaufmann, Los Altos, CA, August 18-23, 1985.

[20] van Melle, W.
A Domain Independent System that Aids in Constructing Knowldge Based Consultation Programs.
Report STAN-CS-80-820, Stanford University Computer Science Department, 1980.
PhD Dissertation.