Performance Evaluation of Rule Grouping on a Real-Time Expert System Architecture

Ing-Ray Chen, *Member, IEEE*, and Bryant L. Poole

***Abstract*— This paper uses a Markov process to model a real-time expert system architecture characterized by message passing and event-driven scheduling. The model is applied to the performance evaluation of rule grouping for real-time expert systems running on this architecture. An optimizing algorithm based on Kernighan–Lin (KL) heuristic graph partitioning for the real-time architecture is developed and a demonstration system based on the model and algorithm has been developed and tested on a portion of Advanced GPS Receiver (AGR) and Manned Maneuvering Unit (MMU) knowledge bases.**

***Index Terms*—Expert systems, real-time architectures, Markov models, performance, rule-based systems, rule grouping, graph-partitioning algorithms.**

I. Introduction

THE INCORPORATION of rule-based systems in real-time control systems has emerged as a state-of-the-art demand in recent years [1], [2], [11]. One central issue is how to make expert systems real-time, that is, how to ensure that the timing and functional requirements of expert systems are satisfied. This issue is interesting mainly because the exponential search time behavior exhibited by rule-based expert systems makes them difficult to apply to real-time applications.

One strategy for using expert systems in real time involves the Maintenance System for AI Knowledge Bases (SKRAM) [7]. SKRAM uses a real-time expert system architecture, called the Activation Framework (AF) [6], to support a dynamically prioritized, message-based, distributed, real-time Ada run-time environment. Under SKRAM, the user can specify groups of expert system production rules using SKRAM's Activation Framework Language (AFL) syntax, or expert system shells supported by SKRAM, such as the C Language Integrated Production System (CLIPS) [5] and the Automated Reasoning Tool (ART) [17]. These user-specified rule groups are automatically translated by SKRAM's translator into efficient Ada run-time processes (with each process corresponding to a rule group) which communicate with each other via message-passing. To facilitate real-time computing, the priorities of these rule groups are dynamically recomputed as follows: whenever a new fact is generated and sent from one rule group to others during run time, the priority of each receiving rule group is calculated as a function of the global importance of the rule group and the sum of the important levels of all pending messages received by the rule group. Then the rule group having the highest priority will be scheduled to run next. The effect of this *event-driven scheduling* mechanism is that it guarantees that the most important process (or rule group) will always be executing at any time, thereby guaranteeing timely responses to the evolving real-time environment. Furthermore, since a message is only sent to relevant rule groups instead of all rule groups when a new fact is generated, this real-time architecture also possesses an intelligent matching capability similar to that of the Rete algorithm [4], except that a distributed strategy, rather than a centralized one, is adopted for managing the pattern matching process. In the extreme case when only one rule is allocated to a group, the communication network connecting all the groups is itself like a Rete network [9].

This real-time expert system architecture, however, has a discernible process-level overhead for carrying out the event-driven scheduling. First, the host processor is interrupted to determine the most important process every time a message is delivered from one process to another process, or whenever an I/O event occurs. (Note: a message is sent carrying the I/O information when an I/O event occurs). Second, if the currently executing process indeed becomes less important than some other process, then the operating system must perform a context switch to allow the most important process to run. This introduces substantial overhead because the processor is optimized to compute with register and cache data, which are lost in a context switch.

This issue is further complicated by a rule grouping which allocates rules to separate groups. If each rule is allocated to a separate group, then the process-level overhead within a processor will be significant because the real-time operating system has to deal with a large number of processes since each group is a separate process. On the other hand, if many rules are allocated to a group, then, although the process-level overhead is reduced (because there are fewer processes), the overhead required for scheduling rules within a single process may be significant because a similar mechanism (e.g., the Rete algorithm) has to be applied for scheduling the execution of rules within a group, thus introducing a lot of system maintenance overhead for keeping track of which rules are ready and which rule should be fired, etc. Both overheads, one at the process level and the other at the rule level, degrade the performance of the system since they increase the time required for firing productions rules. Therefore, the optimizing

Manuscript received September 23, 1991; revised July 9, 1992. This work was supported in part by the AFOSR under the 1991 Summer Research Program and the National Science Foundation under Grant CCR-9110816.

I.-R. Chen is with the Institute of Information Engineering; National Cheng Kung University; No. 1, University Road; Tainan, Taiwan ROC.

B. L. Poole is with the IBM Federal Sector, Boulder, CO 80301 USA.

IEEE Log Number 9413316.

rule grouping should balance these two overheads, thereby minimizing the total system overhead. It should be noted that while the AF architecture also allows rule groups to run on separate processors to increase production parallelism (similar to that in [10]), the paper will only focus on the process-level and rule-level overheads which exist in a uniprocessor real-time system.

Current research investigations of rule grouping for expert systems [12], [16] are not tied in with real-time architectures and thus do not consider this design tradeoff. The basic approach of these rule grouping algorithms is to select a distance metric between each pair of rules and, after allocating each rule to a separate group, iteratively merge groups with the minimum inter-group distance until a stopping condition is met. These algorithms stop either after a predetermined number of groups is obtained [16] or the inter-group distance between the next two groups to be combined is no longer positive [12]. These rule-grouping algorithms are not applicable to expert systems running on real-time architectures because there is no provision for balancing the process-level and rule-level overheads.

Another class of algorithms [3], [13], [15] deals with a more general graph-partitioning problem as follows: given a graph G with costs on its nodes and edges, partition the nodes of G into k subsets of specified sizes, $s_1, s_2, \cdots, s_k$, so as to minimize the total cost of the edges cut. The problem can be related to rule grouping as follows: a) each node corresponds to a rule with its cost proportional to the size of that rule; b) each edge going from rule i to rule j is assigned a cost of one if rule j uses a fact generated by rule i, or zero otherwise; and c) minimizing the total cost of the edges cut when the graph is partitioned into k subsets of specified sizes corresponds to minimizing the message flow between these k subsets, thereby reducing the process-level overhead. Therefore, for a selected $(k, s_1, s_2, \cdots, s_k)$, these graph-partitioning algorithms may be utilized as subroutines to optimize the performance of the system since they produce partitions that will minimize the process-level overhead. The question that remains is how to determine the best $(k, s_1, s_2, \cdots, s_k)$. Our approach is first to use modeling techniques, rather than experimental evaluation (which is more laborious and expensive) to capture the essential characteristics of the real-time architecture, taking into account the tradeoff between the rule-level and process-level overheads. Under this methodology, a quantitative model is constructed, it is *parameterized* to reflect the selection of $(k, s_1, s_2, \cdots, s_k)$ under study utilizing a graph-partitioning algorithm as a routine, and then the model is *evaluated* to determine the performance of the system under the selection. Finally, the model is validated and verified by comparing the model outputs with performance measures obtained from empirical data.

The purpose of this paper is to estimate the performance of various rule groupings of a real-time, rule-based system. Thus a system can be optimized using the appropriate partitioning. While current research is helpful in developing this rule-grouping algorithm, the research does not address key characteristics specific to the AF architecture. In particular, no existing rule-grouping algorithm can balance the tradeoff between rule-level and AFO-level overheads. Section II presents a Markov model for estimating the performance of expert systems running on this real-time architecture with a single processor. The model accounts for the process-level and rule-level overheads. A performance equation is derived from the Markov model and is used to estimate the performance of an expert system which has been partitioned and optimized by a rule grouping algorithm. Section III discusses techniques for obtaining the model parameters. In Section IV, an optimizing rule-grouping algorithm is developed using the Kernighan–Lin (KL) algorithm as its core. Section V demonstrates the feasibility of the model and algorithm by applying them to the Advanced GPS (Global Positioning Systems) Receiver (AGR) [7] and the Manned Maneuvering Unit (MMU) [14] knowledge bases. Finally, Section VI summarizes the paper and outlines some future research areas.

II. Model

In the construction of the model for describing the run-time behavior of expert systems running on the real-time architecture, we first distinguish the following two classes of processes: a) Application Code Objects (ACO's), each of which corresponds to a group of production rules, and b) System Code Objects (SCO) which perform I/O functions such as sensing, actuating and displaying. We assume that there exist I/O-processing devices for interfacing with sensors, actuators, and display devices (as real-time systems should have). The host processor runs ACO's most of the time, but can be temporarily interrupted by an I/O-processing device when an I/O event occurs. In the latter case, the SCO corresponding to the particular I/O event will be scheduled to run.

We use a continuous-time Markov model[1] to describe the real-time architecture. With the Markov model, the system execution is considered as a progression through the following states:

- *ACO*: in this state an ACO is scheduled to run. The ACO may perform useful work, e.g., firing a rule, generating some new fact, etc., or it may use the CPU time for the rule-level overhead which includes determining a) which rules are ready by applying a match algorithm such as the Rete algorithm for saving the states of the rules within a single ACO; and b) which rules should be fired among all ready rules.
- *OS*: in this state, the operating system takes control. This state models part of the process-level overhead due to event-driven scheduling. The operating system gets control because the currently executing ACO sent a message (i.e., new fact) to other processes. The CPU power is used to deliver the message, calculate the priorities of the receiving ACO's, and compare the priorities of the sending and receiving ACO's to determine which ACO should be scheduled to run next.

[1] A continuous-time Markov chain is a stochastic process $\{X(t), t \geq 0\}$ taking on values in the set of nonnegative integers such that the conditional distribution of the future state $X(t+s)$ given the present $X(s)$ and the past $X(u), 0 \leq u < s$, depends only on the present and is independent of the past states [18].

- *SW*: in this state, the operating system performs a context switch by saving the states of the sending ACO (or the ACO which just returned to AF) and allocating the CPU to the ACO having the highest priority. This state models part of the process-level overhead.
- *SCO*: in this state, an input or output SCO takes over due to an I/O event (viz, an alarm event). The CPU power is used for performing a) a context switch to transfer the control to the SCO corresponding to the I/O event; b) an I/O write or read operation; and c) a context switch to transfer the control to the ACO having the highest priority after the I/O operation is done. This state also models part of the process-level overhead.
- *T*: the termination state.

Fig. 1. shows the model which is constructed as follows:

1) When the system is in the ACO state, the CPU is used by an ACO for processing an arriving fact, i.e., determining which rules are ready to fire due to the arrival of the new fact, and selecting one to fire among the set of ready rules, if any. Firing a rule may generate new facts which may a) instantiate other rules within the same ACO, thus requiring the ACO to process the new fact again, or b) instantiate rules in other ACO's, thus requiring the ACO to send a message carrying the fact to other ACO's. The time required for the ACO to process an arriving fact is assumed to be exponentially distributed with a constant rate λ. The probability of termination, executing the same ACO again, and returning the control to the operating system when a new fact is generated are q, r, and $1-r-q$ (or $\bar{r}-q$), respectively. In Fig. 1 these correspond to the horizontal transition at rate $q\lambda$, self-looping transition at rate $r\lambda$ (not shown in the figure), and diagonal transition at rate $(\bar{r}-q)\lambda$, respectively.
2) When the operating system takes over (i.e., in state OS), the time that is required for the operating system to deliver a message from the sending ACO to another ACO and to re-compute the importance levels of these two ACO's to determine which one has the higher priority is assumed to be exponentially and randomly distributed with a constant service rate μ. Assume that the average number of ACO connections for each rule having at least one ACO connection is m. Then, the overall rate for the operating system to deliver a message and re-compute the importance levels of all involved ACO's is μ/m. Further, assume that with probability p, the control will be transferred to a new ACO. Hence, with probability $\bar{p}$ the sending ACO will retain the CPU. These events correspond to the diagonal transitions at rate $(p\mu)/m$ if the CPU is allocated to a new ACO, or at rate $(\bar{p}\mu)/m$ if the CPU is retained by the sending ACO, respectively.
3) When the operating system allocates the CPU to a new ACO, the system is in the SW state. The CPU time is used for performing a context switch. The time for performing a context switch is assumed to be exponentially and randomly distributed with a constant context-switching rate θ. After the context switch is performed, the CPU is allocated to a new ACO and the system is again in state ACO. This event corresponds to the horizontal transition at rate θ.
4) If the list of SCO's to be checked by the operating system at regular intervals exists, then state SCO exists. In this case, whenever an I/O event occurs, the control is passed (by a context switch) to the SCO corresponding to the I/O event. The interarrival time for I/O events is assumed to be exponentially distributed with a constant alarm rate σ. The event corresponds to the vertical transition at rate σ from state ACO to state SCO . When the system is in state SCO, the time required for the system to return to state ACO is assumed to be exponentially distributed with a constant rate α. This quantity includes the time required for SCO's to read from or write to some buffer in I/O-processing devices as well as that required for performing a context switch twice: one to switch from state ACO to SCO and one to switch back from SCO to ACO . This event corresponds the vertical transition at rate α from state SCO to state ACO.
5) The system continues to perform state transitions until it reaches state T.

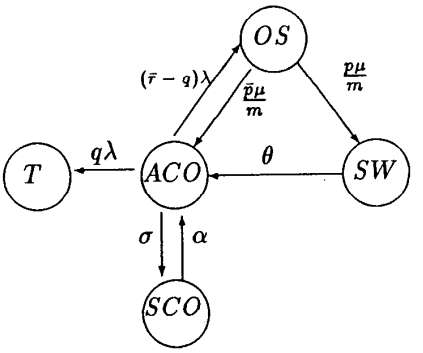


Fig. 1. The Markov model for the real time architecture with a single processor.

The performance measure of interest in this model (Fig. 1) is the mean time to termination ($MTTT$), i.e., the mean time required for the system to transit to state T from state ACO. This measure reflects the average time required for a rule system to terminate. Thus, the design goal in this case is to minimize the $MTTT$.

A simplifying model is the case in which a rule system never terminates (e.g., it runs an infinite loop in an embedded process-control system). This is shown in Fig. 2. In this case, the performance metric of interest is the effective production rate, namely, $\lambda P_{ACO}(\infty)$. This performance metric measures the number of facts generated (and thus correspondingly the number of rules fired) per time unit. This performance metric accounts for the tradeoff between the process-level and rule-level overheads by considering the proportion of time the system stays at state ACO (which accounts for the process-level overhead) and the rate at which the system fires a rule given that the system is in state ACO (which accounts for

the rule-level overhead). An important observation is that λ decreases as more rules are grouped within one ACO because more time is spent within an ACO for handling a new arriving fact over a larger pattern matching network.

This performance metric can be computed by first solving for $P_{ACO}(\infty)$ from the following set of linear equations describing the Markov model in Fig. 2 [18]:

$$P_{\mathrm{ACO}}(\infty) + P_{\mathrm{SCO}}(\infty) + P_{\mathrm{SW}}(\infty) + P_{\mathrm{OS}}(\infty) = 1$$
$$\frac{\mu}{m} P_{\mathrm{OS}}(\infty) = \bar{r}\lambda P_{\mathrm{ACO}}(\infty)$$
$$\theta P_{\mathrm{SW}}(\infty) = \frac{p\mu}{m} P_{\mathrm{OS}}(\infty)$$
$$\alpha P_{\mathrm{SCO}}(\infty) = \sigma P_{\mathrm{ACO}}(\infty).$$

This yields

$$P_{\mathrm{ACO}}(\infty) = \frac{1}{1 + \frac{\sigma}{\alpha} + \frac{m\lambda\bar{r}}{\mu} + \frac{\lambda\bar{r}p}{\theta}}$$

and therefore the equation for the effective production rate is given by

$$\mathcal{X}_S = \frac{\lambda}{1 + \frac{\sigma}{\alpha} + \frac{m\lambda\bar{r}}{\mu} + \frac{\lambda\bar{r}p}{\theta}}. \tag{1}$$

This performance equation has two important implications.

1) For a balanced k-way partition (e.g., $k = 1$ means that there is only one group), the system performance increases as λ, θ, μ, or α increases, and as $\bar{r}$, p, m, or σ decreases. Since θ, μ, α, and σ do not change from one partition to another partition, a graph-partitioning algorithm for generating the balanced k-way partition should minimize $\bar{r}$, p, and m and maximize λ in order to optimize the system performance.
2) To improve the performance of the system, it is desirable to decrease $\bar{r}$, p, and m by selecting a lower k value (thus minimizing the process-level overhead); however, this would decrease λ (i.e., increasing the rule-level overhead as there are more rules in a group) which will adversely degrade the system performance. Therefore, the goal is to select a best balanced k-way partition which can balance these two opposite effects.

In light of these implications, the optimizing rule grouping algorithm should a) utilize a graph-partitioning algorithm to minimize $\bar{r}$, p, and m and maximize λ for a selected k value; and b) vary k from 1 to N so as to select the best balanced k-way partition (from implication 2). Section IV will present an optimizing rule grouping algorithm based on this approach.

III. Parameterization of Model

Equation (1) can serve as a basis for predicting the performance of the system when all parameters are quantified after a particular graph-partitioning algorithm has been applied to partition the production rules into groups. Some of these parameters are machine-dependent but insensitive to the use of graph-partitioning algorithms. These are called *statistically measurable* parameters. Others are sensitive to the utilization of graph-partitioning algorithms and the way a group processes a new fact. These are called *computable* parameters. This section discusses techniques for quantifying these two types of parameters using an 80386-based machine as an example.

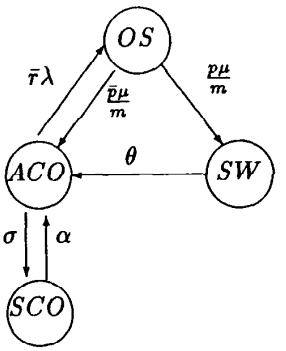


Fig. 2. The Markov model with no termination state.

A. Statistically Measurable Parameters

Statistically Measurable Parameters include θ, μ, σ, and α. These are discussed next.

- θ denotes the average number of times per second the CPU is capable of performing a context switch (dedicated for that purpose). For an 80386 machine loaded with AF, for example, the average time required for the AF operating system to perform a context switch is about 2 ms [19]. Therefore, $\theta \approx 500\ \mathrm{s}^{-1}$.
- μ denotes the average number of times per second the CPU is capable of a) delivering a message from an ACO to another ACO and (b) re-computing the priorities of these two ACO's to determine which one has the higher priority. For an 80386 machine loaded with AF, the average time required for the AF to perform this service is about 1.5 ms [19]. Therefore, $\mu \approx 700\ \mathrm{s}^{-1}$.
- σ denotes the average number of times per second the execution of the rule system is interrupted by I/O events. For example, if it is measured that there are about five data-sensing and five display operations per second, then $\sigma = 10\ \mathrm{s}^{-1}$.
- α denotes the average number of times per second the CPU is capable of performing I/O activities, given that every time an I/O activity occurs, the execution of the rule system is interrupted. Since the time required to service an I/O event includes the time for the system to switch back and forth between states ACO and SCO as well as that required to service an I/O operation,

$$\alpha \approx \frac{1}{\frac{2}{\mu} + \frac{2}{\theta} + D}$$

with D representing the average time required for an SCO to input/output the data to/from an I/O-processing device. D is a statistically measurable quantity, e.g., $D \approx 20$ ms for an 80386-based disk read/write operation.

B. Computable Parameters

Computable parameters include r, m, p, and λ. These parameters are sensitive to the use of graph-partitioning algo-

rithms. In addition, λ is sensitive to the Rete-like mechanism with which each rule group (or ACO) processes a new arriving fact.

We first define some data structures used by graph-partitioning algorithms. Then, we explain how to compute these parameters using these data structures. Let $A[1 \cdots N, 1 \cdots N]$ be the adjacency matrix that shows the connectivity of production rules, i.e., $a_{ii} = 0$ for all i and $a_{ij} = 1$ if rule j uses a fact generated by rule i; 0 otherwise. Let $B[1 \cdots N]$ be the output partition vector of a particular graph partitioning algorithm which has been applied to partition the rule production rules into k balanced groups, such that if $b[i] = j$ then rule i is allocated to the jth group. In the following we discuss techniques for computing r, p, m, and λ.

- r denotes the probability that when an ACO generates a fact, the fact will instantiate rules within the same ACO, rather than instantiating rules in other ACO's. We can approximate r by

$$r = \frac{C_i}{C} = \frac{\sum_{i=1}^{N} \sum_{\substack{j=1 \\ b[j]=b[i]}}^{N} a_{ij}}{\sum_{i=1}^{N} \sum_{j=1}^{N} a_{ij}} \tag{2}$$

  where C stands for the total number of connections in the connection matrix and C_i stands for the total number of internal connections for all k groups after the partition. When there are fewer groups, $r \to 1$ and, conversely, when there are many groups, $r \to 0$. For a given k, a graph partitioning algorithm should maximize r (thereby minimizing $\bar{r}$) as much as possible to reduce the process-level overhead.
- m denotes the average number of external groups that a rule connects to. This can be estimated in three steps: a) computing, for each rule having at least one external connection, the number of groups it connects to, b) accumulating the total count for all such rules, and c) dividing the result in b) by the number of all such rules. Hence, the average number of external ACO connections per rule is an output of a graph partitioning algorithm. Obviously, m increases as the number of groups increases and, as a result, the process-level overhead involved for sending a message from an ACO to all its connecting ACO's increases, a behavior which has been accounted for by the Markov model.
- p represents the probability that the operating system will perform a context switch after servicing a message delivery. If we assume that, when servicing a message delivery for an ACO, the operating system inspects each connecting ACO in succession to determine whether or not the CPU should be switched to that connecting ACO with a success probability p_o, then the probability that the operating system will perform a context switch is proportional to m as follows:

$$p = \sum_{i=1}^{m} p_o (1 - p_o)^{i-1}. \tag{3}$$

  This approach gives a reasonable of p since it can well explain why $p \approx 1$ when there are many groups.
- λ denotes the rate at which a new fact is generated by a rule group. The magnitude of this parameter largely depends on how many rules there are in each group. For that purpose, let λ_j denote this rate when there are j rules in each group. Furthermore, let λ_j relate to λ_1 (the new fact generation rate when each ACO contains exactly one rule) by the following equation:

$$\frac{\lambda_1}{\lambda_j} = \eta_j$$

  where η_j stands for the number of rules that have to be processed by a group containing j rules when a new fact arrives. The advantage of relating λ_j with λ_1 is that λ_1 is a measurable quantity and therefore λ_j can be computed from λ_1 as long as η_j can be estimated properly. For example, for 80386 machines loaded with AF with 1-Mflops processing capability, if each rule contains approximately 500 machine language (floating-point) instructions, then

$$\lambda_1 \approx 2000 \text{ s}^{-1}.$$

To properly estimate η_j, we distinguish the following two mechanisms with which each group handles a new fact.

— *Non-Data-Driven Mechanism:* In this scheme, all j rules within a group are encoded as a succession of `if-then` code blocks. Therefore, the processing time required to generate a new fact is about j times as large as when there is only one rule per group because all j code blocks must be examined before one is selected to fire. Under this *non-data-driven* scheme

$$\eta_j = j \text{ ; and } \lambda_j = \frac{\lambda_1}{j}.$$

Consequently, for a balanced k-way partition, since $j = N/k$

$$\lambda = \frac{\lambda_1 k}{N} \qquad \text{for non-data-driven mechanism}$$

where N is the number of rules in the rule system.

— *Data-Driven Mechanism:* In this scheme, each group maintains a pattern matching network (e.g., consisting of RTGO, BFO, and RO data structure objects in [8]) with a size proportional to the number of rules in each group. When a new fact is processed, instead of inspecting all j rules, only a portion of the rules in the pattern matching network is inspected before the most important rule is selected to run. The number of rules that are inspected when a new fact is processed, i.e., η_j, may be estimated by analyzing the following two cases: a) if the fact was generated by a rule within the same group (which occurs with probability r), then η_j is the average number of internal connections per rule; b) if the fact was generated by a rule in another group (which occurs with probability $\bar{r}$), then η_j is the average number of incoming connections per external rule, a quantity that is equal to the average number of external,

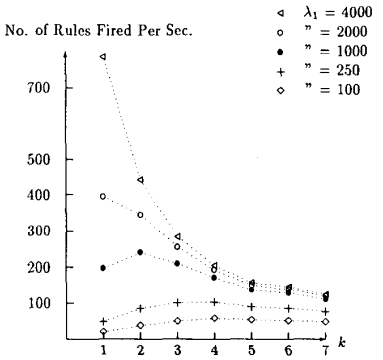


Fig. 3. Model outputs for the AGR system under the data-driven scheme.

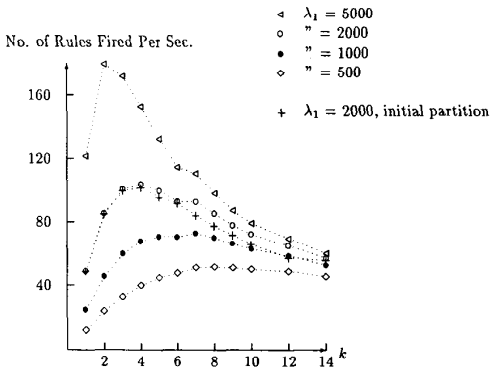


Fig. 4. Model outputs for the MMU system under the data-driven scheme.

outgoing connections per rule in the rule set divided by $k-1$. (Note that when there is only one group, i.e., $k=1$, case b) does not contribute to η_j since $\bar{r}=0$.) Thus under the *data-driven* scheme, when $k>1$

$$\eta_j = \max\left(1, r \times \frac{C_i}{N} + \bar{r} \times \frac{C_e}{N(k-1)}\right)$$

where C_i stands for the total number of internal connections for all k groups as defined in (2) and $C_e = C - C_i$, standing for the total number of external connections. (C is the total number of connections of the input connectivity matrix.) The max operator is used to make sure that at least one rule will be affected when a new fact is processed. In summary, for a balanced k-way partition

$$\lambda = \begin{cases} \frac{N\lambda_1}{C_i} & \text{if } k=1 \\ \frac{\lambda_1}{\max\left(1, \frac{rC_i}{N} + \frac{\bar{r}(C-C_i)}{N(k-1)}\right)} & \text{otherwise} \end{cases}$$

for data-driven mechanism.

IV. The Optimizing Rule Grouping Algorithm

In this section we present an optimizing rule grouping algorithm with a goal of determining the best balanced k-way partition for optimizing the system performance.

Input: adjacency matrix $A[1\cdots N,\ 1\cdots N], \theta, \mu, \sigma, \lambda_1, D,$ *and* p_o.

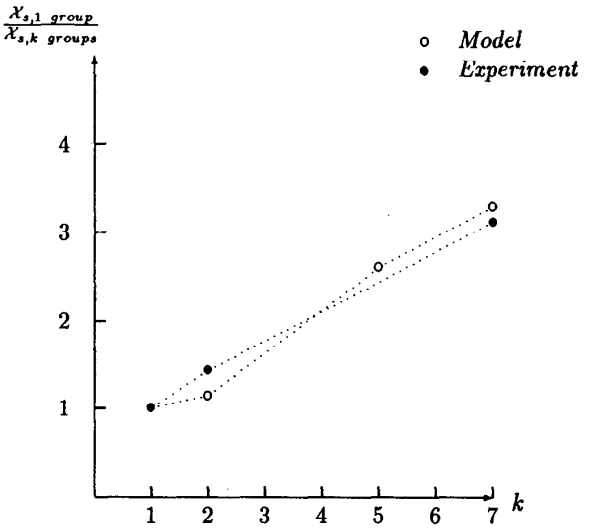


Fig. 5. Model outputs versus experimental results for the AGR system.

Output: the best balanced k-way partition vector $(B[1\cdots N.])$
Steps:

1) *determine N from the input adjacency matrix,*
2) *set* $k=1$,
3) *while* $k \le N$ *do*
   a) *parameterize r, p, m, and* λ *based on the balanced k-way partition generated by the KL algorithm,*
   b) *compute the performance metric of the system using (1),*
   c) *if the system performance is the greatest so far, record this value; also record the best k and the resulting k-way partition in the variable* $B[1\cdots N]$,
   d) $k = k+1$.
4) *output the best k and* $B[1\cdots N]$.

This optimizing rule grouping algorithm utilizes the KL algorithm [13] as a subroutine for k-way partitioning rules so as to minimize the total external cost on edge cuts, thereby minimizing $\bar{r}$, p, and m and, consequently, optimizing the system performance. With k varying linearly from 1 to N, this optimizing algorithm runs about 10 s for a rule system of size 7 (e.g., AGR) and about 5 min for a rule size of 92 (e.g., MMU) on an 80386-based machine loaded with AF.

V. Examples

In the following, the optimizing rule grouping algorithm is applied to the AGR and MMU knowledge bases. The objective is to identify the best k-way partition which can maximize the effective production rate (i.e., system performance) as a function of λ_1. Both systems run on an 80386-based machine loaded with AF with which the following performance measurement data are observed:

- $\mu \approx 700\ s^{-1}$;
- $\theta \approx 500\ s^{-1}$;

```
(defrule jamming_levels_increasing      "jamming increasing -> state 3"
  (rx_state 5)
  (current_cno ?cno)                    ;; was (?asv current_cno ?cno)
  (cno_trend ?trend)                    ;; was (?asv cno_trend ?trend)
  (test(< ?cno 25))                     ;; cno < 25 db-hz
  (test(< ?trend -1))                   ;; cno trend < -1 db-hz/10 secs
  (jamming_threat 1)
  (jamming_threat_trend 1)
=>
  (assert(rx_state 3))
)

(defrule signal_decreasing              "cno decreasing -> state 3"
  (rx_state 5)
  (current_cno ?cno)                    ;; was (?asv current_cno ?cno)
  (cno_trend ?trend)                    ;; was (?asv cno_trend ?trend)
  (test(< ?cno 20))                     ;; cno < 20 db-hz
  (test(< ?trend 0))                    ;; cno trend < 0 db-hz/10 secs
=>
  (assert(rx_state 3))
)

(defrule dynamics_levels_increasing     "dynamics increasing -> state 3"
  (rx_state_recovery 5) ;; was (rx_state 5 ?)
  (acceleration ?acc)
  (jerk ?j)
  (test(> ?acc 3))                      ;; acceleration > 3 g
  (test(> ?j 10))                       ;; jerk > 10g/sec
=>
  (assert(rx_state 3))
)

(defrule cno_meas_improving             "cno meas available => tracking svs"
  (rx_state 3)
  (current_cno ?cno)                    ;; was (?asv current_cno ?cno)
  (cno_trend ?trend)                    ;; was (?asv cno_trend ?trend)
  (test(> ?cno 25))                     ;; cno > 25 db-hz
  (test(> ?trend 0))                    ;; cno trend > 0 db-hz/10 sec
=>
  (assert(rx_state 5))
)
```

Fig. 6. An example AGR rule group in CLIPS format.

TABLE I
r, m, p AND λ AS A FUNCTION OF K FOR THE AGR SYSTEM

k	r	m	p	λ (data-driven)	λ (non data-driven)
1	1.00	0	0.00	500	286
2	0.46	1	0.80	995	571
3	0.32	2	0.96	1499	857
4	0.21	3	0.99	1986	1143
5	0.14	3	0.99	2000	1429
6	0.07	3	0.99	2000	1714
7	0.00	4	1.00	2000	2000

- $\sigma \approx 10\ \text{s}^{-1}$;
- $\lambda_1 \approx 2000\ \text{s}^{-1}$;
- $D \approx 0.02$ s;
- $\alpha \approx \frac{1}{\frac{2}{\mu}+\frac{2}{\theta}+D} = 35\ \text{s}^{-1}$.

A. Model Outputs

By applying the KL algorithm on the AGR rule system, it is observed that r, m, p, and λ (under data-driven as well as non-data-driven scheme) are related to k ($1 \leq k \leq 7$) as shown in Table I when $\lambda_1 = 2000$.

Fig. 3 shows that for the AGR knowledge base when the data-driven scheme is used to process arriving facts, the best value of k depends on the magnitude of λ_1 (which reflects the degree of the rule-level overhead). The figure shows that, if λ_1 is in the same order of magnitude as θ or μ (which occurs when each rule contains many instructions), then the best partition favors a high k. Conversely, if λ_1 is an order of magnitude higher than θ or μ (which occurs when each rule contains only a few instructions), then the best partition favors a low k. The physical interpretation is as follows: in the latter case the time required for the operating system to schedule ACO's is an order of magnitude longer than that required for an ACO to process an arriving fact, select a rule, and fire the rule. Consequently, the system performance is improved by shifting the time-consuming scheduling work from the operating system to within an ACO. An important observation from this figure is that, for each λ_1 value, there exists a k value under which the system performance is maximized. Since it is estimated $\lambda_1 \approx 2000$ for the AGR knowledge-base system, our model suggests that the best partition should group all rules together when the data-driven scheme is used.

Remark: We also observed a similar trend for the case when the *non-data-driven* scheme is used to process arriving facts, except that the latter case requires a higher value of λ_1 for the best partition to favor a low k value because more rules

```
RCO example-rule-group IS
DECLARE
  acceleration IS INTEGER;
  current_cno IS INTEGER;
  cno_trend IS INTEGER;
  jamming_threat IS INTEGER;
  jamming_threat_trend IS INTEGER;
  jerk IS INTEGER;
  rx_state IS INTEGER;
  rx_state_recovery IS INTEGER;

  RECEIVE  acceleration,current_cno,cno_trend,jamming_threat,
           jamming_threat_trend,jerk,rx_state,rx_state_recovery;
  SEND rx_state;

BEGIN
-- jamming levels increasing
  IF rx_state = 5 AND current_cno < 25 AND cno_trend < 88
     AND jamming_threat = 1 AND jamming_threat_trend = 1
  THEN  rx_state := 3;
-- signal decreasing
  IF rx_state = 5 AND current_cno < 20 AND cno_trend < 0
  THEN  rx_state := 3;
-- dynamic levels increasing
  IF rx_state_recovery = 5 AND acceleration > 3 AND jerk > 10
  THEN  rx_state := 3;
-- cno meas improving
  IF rx_state = 3 AND current_cno > 25 AND cno_trend > 0
  THEN  rx_state := 5;
END;
```

Fig. 7. An example AGR rule group in AFL format.

have to be inspected when a new fact arrives. As a result, $k = 3$ is the best number of groups under the non-data-driven scheme for the AGR rule system as opposed to $k = 1$ under the data-driven scheme.

A similar procedure is applied to the MMU system. Fig. 4 shows the model outputs for the MMU system under the data-driven scheme. The result shows that $k = 4$ is the best number of groups for $\lambda_1 = 2000$. All curves in Fig. 4 are due to partitions generated by the KL graph-partitioning algorithm except for the one labeled with *initial partition* which is formed by arbitrarily assigning rule 1 to group 1, rule 2 to group 2, and so on, in a round-robin fashion until all rules have been assigned. This initial partition has been used by the KL graph-partitioning algorithm as the starting partition. It is easily seen that the KL graph-partitioning algorithm can generate a more optimized k-way partition than the initial partition for $\lambda_1 = 2000$, although they both predict $k = 4$ is the best number of groups for the MMU system.

B. Experimental Results

Fig. 5 compares the model outputs with the experimental results for the AGR system with the y-coordinate representing the performance ratio between the one-group case and the k-group case with the x-coordinate varying k from 1 to 7. This experimental result is obtained by encoding the AGR rules in the form of Activation Framework Language (AFL) rule groups based on the output generated by the optimizing rule-grouping algorithm and having the AFL translator [9] automatically translate them into groups of C program modules incorporating the data-driven mechanism as described in [8]. Figs. 6 and 7 show an instance of an AGR rule group consisting of 4 rules encoded in CLIPS and AFL formats, respectively. These translated C program modules (with each module corresponding to a rule group) are subsequently linked with test modules and the Activation Framework For C Language (AFC) run-time libraries into one executable program. To ensure that no injected data are lost, the times at which data are injected into the system are manually adjusted (separately for each k value) in the input file such that the execution order of rules is repeated, e.g., rule 2 first, followed by rule 6 and then by rule 3, etc. The times required for the AGR system to output an expected sequence of logging records in the output file are measured for all k values and are used as the performance standard with which the model outputs are compared. It was observed that, including the disk operation overhead required to inject and log the data, 7.19 s are needed for seven groups, 3.3 s are needed for two groups, and 2.3 s are needed for one group for the expected execution order to be observed.

Fig. 5 shows that, for the data-driven scheme, the model outputs correlate well with the experimental results. The result that one group is the best selection for the AGR system is not surprising because the AGR system consists of only seven rules and therefore the rule-level overhead is not significant when compared with the process-level overhead, especially when the data-driven mechanism is utilized. However, for a system of moderate size, e.g., the MMU system with 92 rules, one group is not necessarily the best selection. As the the number of rules increases, the rule-level overhead also increases significantly, especially when all 92 rules are put into one group. As a result, for the MMU system, $k = 4$ becomes the best selection.

VI. Summary

We have developed a model for evaluating rule grouping on a real-time expert system architecture characterized by event-driven scheduling and message-passing. Based on the model, we derived the system performance equation and developed an optimizing rule-grouping algorithm utilizing the KL graph-partitioning heuristic for k-way partitioning the rule system and for optimizing the system performance. Our results show that the best k-way partition is a tradeoff between the rule-level and process-level overheads: if the rule-level overhead dominates the process-level overhead (e.g., when there are a lot of rules, or a lot of instructions per rule, and/or the non-data-driven mechanism is used), a high number of groups is favored; conversely, if the process-level overhead dominates the rule-level overhead (e.g., each rule is simple) then a low number of groups is favored. Using The Activation Framework as an example architecture, we demonstrated the applicability of the model by comparing the model outputs with empirical results for the AGR and MMU knowledge bases.

Some possible future research directions include a) investigating whether an unbalanced k-way partition can perform better than a balanced k-way partition by refining the model to accommodate groups of different size; b) refining the modeling of the data-driven mechanism by considering the overhead required for executing different conflict resolution algorithms;

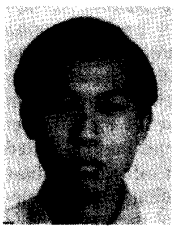
and c) developing and validating a model for real-time expert system architectures with multiple processors.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their many useful criticisms and suggestions. Their detailed comments have greatly improved the quality of the paper. The authors also thank Dr. P. E. Green and his supporting staff at the Real-Time Intelligent Systems Corp., Worcester, for providing needed help on the Activation Framework software.

REFERENCES

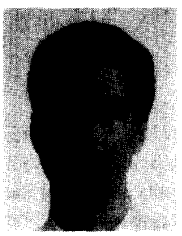
[1] *6th Int. Conf. on Applications of AI in Eng.*, Oxford, UK, July 1991.
[2] *3rd IFAC Int. Workshop on AI in Real Time Control*, Sonoma Valley, CA, Sept. 1991.
[3] E. R. Barnes, "An algorithm for partitioning the nodes of a graph" *SIAM J. Algebraic and Discrete Methods*, vol. 3, no. 4, pp. 541–550, Dec. 1982.
[4] C. L. Forgy, "Rete: A fast algorithm for the many pattern/many object pattern match problem," *Artificial Intell.*, pp. 17–37, 1982.
[5] J. L. Giarratano, *CLIPS User's Guide*, Artificial Intelligence Section, Lyndon B. Johnson Space Center, Houston, TX, June 1988.
[6] P. E. Green, "AF: A framework for real-time distributed cooperative problem solving," *Distributed Artificial Intelligence*,Michael N. Huhns, Ed. New York: Morgan Kaufmann, 1987, pp. 153–176.
[7] P. E. Green, J. Duckworth, L. Becker, and S. Cotterill, "Maintenance system for AI knowledge bases, Phase I—Final report: Design of the SKRAM system," Contract F33615-90-C-1470, Feb. 1991.
[8] P. E. Green, "A data-driven mechanism for the execution of production rules in real-time computer based systems," *The Real-Time Intelligent Systems Cooperation*, June 1991.
[9] ——, *AFL Users Manual*, Version 1.5, *The Real-Time Intelligent Systems Cooperation*, Aug. 1991.
[10] A. Gupta, *Parallelism in Production Systems*. New York: Morgan Kaufmann, 1987.
[11] *IFIP Working Conf. on Dependability of AI Systems*, Vienna, Austria, May 1991.
[12] R. J. K. Jacob and J. N. Froscher, "A software engineering methodology for rule-based systems," *IEEE Trans. Knowledge and Data Eng.*, vol. 2, no. 2, June 1990.
[13] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, pp. 291–307, 1970.
[14] D. G. Lawler and L. J. F. Williams, "MMU FDIR automation task," Tech. Rep. NAS9-17650, McDonell Douglas Astronautics—Engineering Services, Houston, TX, Feb. 1988.
[15] C. H. Lee, C. I. Park, and M. Kim, "Efficient algorithm for graph-partitioning problem using a problem transformation method," *Computer-Aided Des.*, vol. 21, no. 10, pp. 611–618, Dec. 1989.
[16] M. Mehrotra, "Rule groupings: a software engineering approach towards verification of expert systems," NASA Contract NAS1-18585, Final Rep., Feb. 1991.
[17] W. Mettrey, "A comparative evaluation of expert system tools," *IEEE Computer*, Feb 1991.
[18] S. M. Ross, *Introduction to Probability Models*, 4th ed. New York: Academic Press, 1989.
[19] "The real-time intelligent systems," in *Activation Framework Operating System Environment*, Users Manual, AFC Version 2.6, Aug. 1991.
[20] Worcester Polytechnic Institute, *Knowledge Representation into Ada Methodology*, Project Review Meeting, May 1991.



Ing-Ray Chen (S'86–M'90) received the B.S. degree from the National Taiwan University in 1978, and the M.S. and Ph.D. degrees in computer science from the University of Houston, University Park, in 1985 and 1988, respectively.

He was an Associate Professor of Computer and Information Sciences at the University of Mississippi. He is now an Associate Professor at the Institute of Information Engineering at the National Cheng Kung University, Tainan, Taiwan. His research interests are in reliability and performance analysis and real-time intelligent systems.

Dr. Chen is a member of ACM.



Bryant L. Poole received the B.M. degree in music and the M.S. degree in computer science from the University of Mississippi, University, MS, in 1985 and 1992.

He is an Associate Programmer at the IBM Federal Sector in Boulder, CO. His research interests include artificial intelligence and computer music

Mr. Poole is a member of the Association of Computing Machinery.