

# Frontiers in Run-Time Prediction for the Production-System Paradigm

*Franz Barachini*

■ Efficient indexing schemes have influenced the acceptance of production systems in the industrial world. However, in embedded-control systems, production systems have not been applied intensively because of their nondeterministic run-time behavior. Thus, nonpredictability of response times is a major obstacle to the widespread use of expert systems in the real-time domain. The RETE and TREAT algorithms and their offspring play a major role in the implementation of efficient pattern-matching systems. Therefore, it is worthwhile to investigate run-time predictability for these match algorithms. This article presents three different schemes for estimating the time needed for operations in the production-system execution model.

Engineers have recently been combining control theory, real-time systems, and AI. Such systems are considered intelligent when they are able to perform complex actions in response to the sensed environment. In intelligent real-time systems, there is a trade-off between acting and reasoning. Time is a valuable resource that is lost when the system must reason about actions before performing them.

Reasoning and, thus, search can be performed on two levels: the problem-space level and the knowledge base level (Tambe and Newell 1988). With the problem-space level, the question is how to find a sequence of operations that map an initial state to a goal state. Usually, a sequence of steps is required to find the goal state. If more than one operator is applicable to a state, search is required to select the best one. This search can be applied on the knowledge base level by, for example, production systems that browse through the available knowledge that is appli-

cable in a specific state. Because these production systems constitute an integral part of an intelligent real-time system, they must fulfill stringent timing requirements. They are not candidates for integration into real-time systems when run time cannot be predicted (Stankovic and Ramamithram 1990).

In the real-time domain, we distinguish between hard and soft real time. In a *hard real-time system*, schedulers rely on deterministic tasks to guarantee results before specific deadlines are encountered. Thus, run-time prediction of tasks is necessary to match these deadlines. In contrast, a *soft real-time system* is designed so that a missed deadline can be accepted.

The motivation for this research was driven by the safety bag expert system (Klein 1990), an embedded control system for railway switching that is used in the railway stations of three European countries. Parts of the control system are implemented in PAMELA (Barachini 1991), an expert system shell providing an optimized RETE (Forgy 1982) algorithm. Years ago, a request came from the railway authorities for a system that could guarantee a reasonable worst-case match time for a peripheral request. This requirement is understandable because in typical real-time systems, scheduling is mostly based on worst-case execution times of the tasks involved. Only the theoretical worst-case behavior of the match could be presented as an answer to the railway authorities. The fundamental problem with production systems is that the worst-case execution time is often orders of magnitude larger than the average-case execution time. A proof also showed that the match is NP-hard (Tambe, Kalp, and Rosenbloom 1991; Barachini 1990). Obviously, this

*Although it is well known that high performance does not mean real time, engineers tend to speed up applications to meet certain deadlines.*

answer was unsatisfactory to the railway authorities.

To have an indication of whether a deadline will be met, match-time predictability is a necessity for real-time applications such as the safety bag expert system. This necessity is justified because most of the overall run time of the safety bag system is used in the match phase. However, the safety bag system is designed so that it can be interrupted at any time by the process periphery. These interrupts are only scheduled at specific preemption points in the right-hand side of rules. Therefore, a preemption point is the only position where peripheral events are handled. In PAMELA, a preemption point is set before and after each basic action (make, change, remove). Between preemption points, interrupts are queued. Thus, for the electronic interlocking system, it is important to know how long the match will take and how much time until the next preemption point is encountered. Therefore, match-time prediction with fine granularity is an issue for the safety bag production system. The predictability of the points of preemption would allow us to estimate a priori the time required to react to peripheral events. When this time exceeds certain deadlines, the entire match can be postponed, or emergency strategies can be executed.

In this article, I investigate the predictability of the time needed to perform a basic action and the time needed to perform all basic actions of a rule. Although the right-hand side of a rule might include statements other than basic actions, the presented methods (microlevel reasoner, upper bound, and extended upper bound) deal with match-time prediction only.

The time for the execution of a basic action determines the granularity at which interrupts can actually be handled. This time is known in the literature as the *responsiveness* of a system. I show that realistic upper bounds for basic actions can be calculated a priori. This calculation is a prerequisite for hard real-time systems. Additionally, I show that reasonable approximations of run time for complete right-hand sides can be predicted by one of the methods. The prediction of approximations is a prerequisite for soft real-time systems, where no guaranteed answering times for particular events are required but where an estimation of the average time for a typical request is required. Not restricted to RETE, the presented methods can also be applied to TREAT (Miranker 1990), although in this article, I only discuss RETE networks.

## Related Work

Besides the methods proposed, I can identify three other possible methods of meeting certain deadlines.

First, the tuning of applications is a common technique used by designers to meet and verify timing constraints. Although it is well known that high performance does not mean real time, engineers tend to speed up applications to meet certain deadlines. However, this method is promising only when a complete simulation of the process periphery is available. Speedups can be achieved by optimizing the existing match algorithms (Miranker et al. 1990; Schor et al. 1986) according to the needs of the application; using higher-performance chips; or applying parallel, special-purpose hardware to increase the performance of the match (Bahr et al. 1991; Gupta and Tambe 1988; Kelly and Seviora 1987). However, speedup of the match is limited to a magnitude of 40 (Gupta 1986). The next step then would be to exploit parallelism on the rule-and-task level (Ishida 1991; Harvey et al. 1989). An alternative is to apply totally new match algorithms, such as *collection match* (Acharya and Tambe 1993), which shows large time improvements over classical RETE implementations. Although in practice, most of the performance problems can be solved by engineers using one of these alternatives, these alternatives do not solve the run-time prediction problem.

Second, the approach by Ishida, Gasser, and Makoto (1992) improves the ability to build production systems that can adapt to changing real-time constraints. The approach extends *parallel production systems*, where global control exists, into *distributed production systems*, where problems are solved by a society of agents using distributed control. Adaptive work allocation is provided by introducing special reorganization primitives that control the size of the agent population and the resource allocated to each agent.

The third possibility is the idea to limit match complexity as presented by Haley (1987); Wang, Mok, and Cheng (1990); Tambe and Newell (1988); Paul et al. (1991); and Tambe, Kalp, and Rosenbloom (1991). Haley and Wang try to put limits on the amounts of data approaching the expert system. Tambe's UNI-RETE shifts match combinatorics from knowledge search to problem-space search. Paul et al. designed a real-time architecture that is implemented on top of CPARAOPS5. This architecture uses data partitioning and redirection of token streams to achieve small execution-time variances.

## Basics

A production system consists of a rule set, called the *production memory*, and a database of assertions, called the *working memory*. Each rule consists of condition statements, the left-hand side, and a set of actions, the right-hand side.

The right-hand side specifies information that is to be added to, or removed from, the working memory. There are three possible basic actions in the left-hand side. *Make* adds a new working-memory element to the working memory. *Remove* deletes an existing working-memory element from the working memory. *Modify* modifies an already existing working-memory element. Each working-memory element corresponds to a certain type of data, called its *working-memory type*. Each right-hand side consists of one or several basic actions.

The production system repeatedly performs the so-called recognize-act cycle. During this cycle, a right-hand side is executed, leading to a match phase that determines a conflict set of satisfied rule instantiations. The conflict set resolution procedure selects one rule to be fired, and the act procedure executes the right-hand side.

The RETE algorithm maps the left-hand sides of rules to a discriminating data flow network. The data elements flowing through this network are called *tokens*. When a working-memory element is added to, or removed from, the working memory, a positive-tagged token (plus token) or a negative-tagged token (minus token) representing this action is passed to this data flow network. There are different node types available in this network. The constant condition tests are performed at so-called one-input nodes, and matched tokens are stored in alpha memories. Copies of matched tokens are sent to successor nodes. Typically, the most run-time-intensive node is a two-input node. Tokens arriving at two-input nodes are compared with the tokens stored in the opposite-token memory. Successfully joined tokens are then resent to a successor node. In this manner, tokens flow through the network until they arrive at the end node. If one token arrives at the end node, an instantiation enters the conflict set. For a more detailed description of the RETE algorithm and pattern matching, see the example in the sidebar.

## Predictability Issues

In most production-system languages, basic actions are indivisible subtasks. Whenever an

interrupt that is modifying working-memory contents rises during the match phase, it cannot be handled immediately (Barachini and Theuretzbacher 1988). For the sake of consistency, it must be postponed and can only be executed after the completion of the current action. In PAMELA, at the end of each basic action, the system reaches a so-called *preemption point*. At this preemption point, any interrupt is handled that might have arisen since the previous preemption point. As explained for the safety bag production system, the time required for an action is particularly crucial because it represents the time that the system needs to react to an external interrupt.

The run time for a rule's right-hand side represents the time for a complete rule firing. Predictability at this level enables the system to determine whether the rule can be executed within an allocated time frame.

The most general level is predictability at the expert system level. This level is impossible for interrupt-driven expert systems that are part of a process control system because facts are created and modified dynamically by peripheral events. The behavior of the periphery can hardly be forecasted in any industrial environment. Therefore, only limited assertions can be made about rule-firing sequences. I show that run-time prediction of embedded expert systems can be performed with finer granularity, but I believe that it will never be achieved on the expert system level in a tractable manner.

An obvious approach to run-time predictability is profiling the quantities that one wants to predict. However, several tries with different match algorithms showed (Barachini, Mistelberger, and Gupta 1992) that execution times of basic actions or complete right-hand sides cannot be used as a basis for run-time prediction. The first kind of measurement my colleagues and I performed was the investigation of run-time behavior used by plus or minus tokens for specific working-memory types over all fired rules. A huge variance over these activations can be found. This variance is also true with other expert systems. Thus, we cannot rely on these meaningless data for run-time prediction. To find meaningful data, we restricted ourselves to the study of run time required for a working-memory-element action of a particular working-memory type but, in this case, only in the scope of a specific rule's right-hand side. Unfortunately, the measurements showed that this time was also rather irregular. It highly depends on the number of actions

*... run-time prediction of embedded expert systems can be performed with finer granularity, but I believe that it will never be achieved on the expert system level in a tractable manner.*

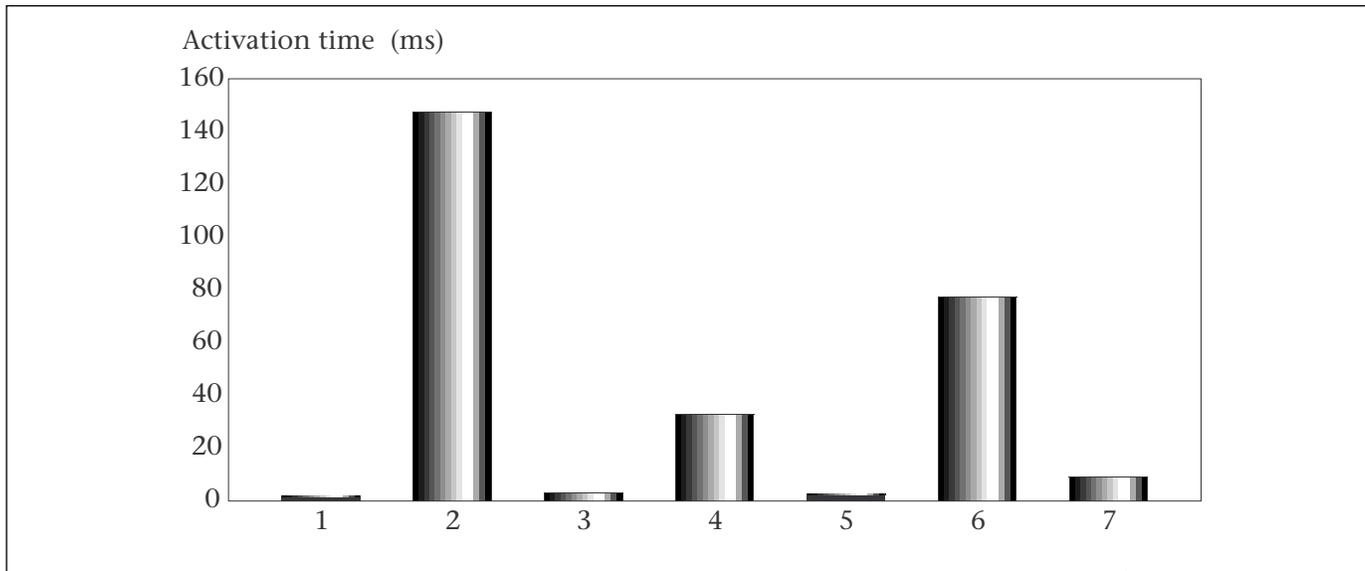


Figure 1. Make Goal Action-Time in Rule Sequence\_5 in RUBIK.

performed in the network, that is, on the number of matches and the number of insertions and deletions. For example, in the RUBIK benchmark, the rule Sequence\_5 has a simple right-hand side, consisting only of one *make* statement for the working-memory type *goal*. Figure 1 shows the evolution of the time required for the right-hand-side firing of this rule. It was fired seven times during the execution of a specific cube configuration. In all cases, the execution-time variance was too high to be useful. These discouraging results are general in nature. They hold for OPS-based (Cooper and Wogrin 1988), as well as PAMELA-based, expert systems, irrespective of their implementation method. Therefore, in contrast to classical approaches where predictability is verified with pre-run-time profiling methods, such methods are not promising for production systems reacting to continuously changing environments.

## Benchmarks

The three presented run-time prediction methods were applied to different production systems: a pool of toy problems and 7 industrial systems, ranging from 29 to 250 rules. All the methods showed similar behavior. I present representative results of two toy and two applied systems: (1) EMAB, the monkey and bananas production system written by the National Aeronautics and Space Administration in an extended version with 29 rules; (2) RUBIK, a production system with 71 rules that solves the Rubik's cube by J. Allen; (3)

VAHINE, a production system with 55 rules (by Alcatel RC) that advises road maintenance in winter for given weather conditions; and (4) CARS, a production system with 62 rules (by Alcatel RC) that dynamically maximizes traffic throughput on a network of roads and crossings.

CARS is a typical real-time production system in the sense that it continuously gathers data from the periphery through interrupts and polling mechanisms. It is especially time critical because it has to react immediately to traffic congestion. Although VAHINE and CARS are only medium sized with respect to the number of rules, they were selected because they use vast amounts of data produced by the process periphery. Thus, the size of their working memory represents a fruitful playground for the investigation.

## The Microlevel Reasoner Method

The *microlevel reasoner* is a method that estimates match time for individual basic actions on the token level. The method relies on statistical data dynamically gathered at the node level of the RETE network.

At the beginning of its flow through the RETE network, the token filters through one-input nodes. Only a small fraction of run time is spent during one-input-node treatment (Gupta 1986), which is even true for systems dealing with a large working memory because optimized search techniques, such as hashing, can be applied. Therefore, worst-case

run time, which is linear in the number of incoming tokens for the one-input-node network, is small compared to overall match time. This upper bound can easily be calculated with the assumption that no filtering takes place in one-input nodes. I assume an implementation in which tokens of one working-memory element of a right-hand side are gathered at the end of one-input-node chains. The one-input nodes are treated first, and then the estimation task begins. Thus, the exact number of tokens entering the two-input-node network is known.

The token flow in the network is characterized by several elementary operations. In a two-input node, each incoming token is compared with every token from the opposite memory. For each successful match, one new token is created and stored in the input memory of the successor. To estimate the time spent in a particular node during a token treatment, the following amounts are important: (1) the number of tokens entering the node, (2) the time required for one token insertion into a token memory, and (3) the time required for a match.

All relevant token numbers for a specific token (for example, the number of new tokens produced by a two-input node) have to be estimated before actual matching starts. Because of the worst-case combinatorial behavior of RETE and TREAT, it is not feasible to estimate these numbers only by evaluating the largest theoretical number of tokens that could flow through the network. Most of the time, this upper bound would be too large to be of any significance.

For the two-input nodes, the estimation is based on the implementation of a token flow ratio  $p_k$  in each two-input node (see figure 2). The factor  $p_k$  represents the probability for a successful match at node  $k$  when a token is compared with another. This probability factor  $p_k$  is updated after each performed token treatment.

For the estimation, the following numbers are used: (1)  $V_k$ , the number of matches at node  $k$ ; (2)  $m_k$ , the number of tokens in the right memory of node  $k$ ; (3)  $n_k$ , the number of tokens in the left memory of node  $k$ ; (4)  $\delta m_k$ , the number of plus tokens entering the right memory of node  $k$ ; (5)  $\delta n_k$ , the number of plus tokens entering the left memory of node  $k$ ; and (6)  $p_k$ , the probability for a successful match at node  $k$ .

For tokens entering the right memory of node  $k$ , we get

$$V_k = n_k * \delta m_k \quad (1)$$

and

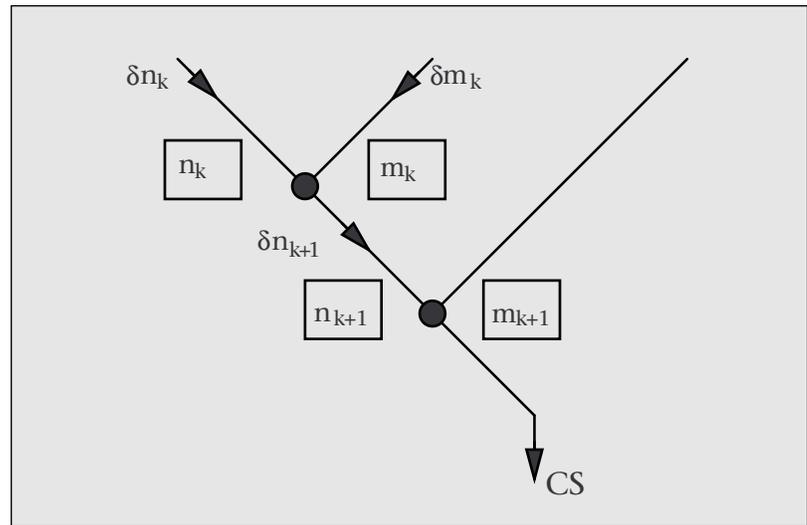


Figure 2. RETE Network ( $CS = context\ set$ ).

$$\delta n_{k+1} = p_k * V_k \quad (2)$$

for a two-input node and

$$V_k = n_k * \delta m_k \quad (3)$$

and

$$\delta n_{k+1} = 0 \quad (4)$$

for a not node. For tokens entering the left memory of node  $k$ , we get

$$V_k = m_k * \delta n_k \quad (5)$$

and

$$\delta n_{k+1} = p_k * V_k \quad (6)$$

for a two-input node and

$$V_k = m_k * \delta n_k \quad (7)$$

and

$$\delta n_{k+1} = (1 - p_k)^{m_k} * \delta n_k \quad (8)$$

for a not node.

Note that equation 4 does not mean that no tokens are actually created in the RETE network. It only means that those possibly created tokens are negative tokens; therefore, I do not take them into account because of PAMELA's special minus-token treatment (Barachini 1991). Compared to the positive-token treatment, the negative-token treatment is cheap because no rematch is performed. Also, note that for hash table-based memory implementation, formulas 1 to 8 must be implemented for each hash table bucket. Theoretically, the estimation results would be more precise. In contrast, the hash function must be evaluated for each token to find the correct buckets. The simulation of the execution time of the hash function puts too much overhead into the estimation algorithm. Thus, there is great evidence that the present methods are useful only for list-based implementations.

Using formulas 1 to 8, one can estimate the number of tokens that will be created in the

network. To estimate the time required for the plus-token treatment, the time for an insertion in a token memory and the time for a match on a T800 transputer (Bahr et al. 1991), with a precision of 1 ms, were measured. The following measurements were performed:

First was 5  $\mu$ s for a match invoking a simple equality test, which is the most common situation. However, because PAMELA offers the ability to call a C function within the test, the time for a single match can be significantly longer. Thus, we also have to make our predictions with larger match times.

Second was 50  $\mu$ s for an insertion in a right token memory. This time does not depend on the node or the token, whose length is always one in a right memory.

Third was  $(50 + 6*L)$   $\mu$ s for an insertion in a left token memory, where  $L$  is the left token's length. ( $L$  actually means the number of working-memory elements that  $L$  consists of.)

The microlevel reasoner method estimates the number of actions to be performed in the network (matches and insertions). The estimated number of actions is calculated by assuming a hypothetical match. During this action, the two-input-node checks are not performed because the calculation is based on equations 1 to 8, considering only statistical token flow ratios. As a result, we get the number of estimated actions in the network before a token starts traversing the network. By multiplying the number of estimated actions with the average run-time measurements presented earlier, we get the estimated match time for the whole next basic working-memory-element action. Working-memory element-type specific features are not taken into consideration.

## Results of the Microlevel Reasoner Method

The run-time prediction adds some burden to the whole execution time of the expert system. However, compared to match time, this additional time is small. On the transputer-based system, 50- $\mu$ s run-time overhead must be taken into consideration for each affected two-input node. This number represents the overhead for calculating the time for the next action, including the overhead for updating the internal counters. The complexity of the method is linearly dependent on the number of affected nodes for each basic action.

The measurements presented in this section were performed in the following way: By using the time required for each elementary

action, a simulator was implemented, which computes the time that it would take on a T800 transputer. This method has the advantage of being able to vary the values of the match time and study their influence on the quality of the prediction.

For reasons of statistical significance, I only present the results for those of the working-memory types appearing sufficiently often during the expert system execution. Tables 1 through 4 display the averages and the variances of the ratio of real time to estimated time for each of the applications. The results in tables 1 through 4 show a fairly small variance, allowing us to expect in most cases an error lower than a factor of 2.

Figure 3 provides more relevant information. It shows the distribution of the ratio of real time to estimated time for a so-called context switch. A *context switch* usually invalidates many already-performed matches and initiates abnormal match activities in the network. In this case, match-time prediction is especially hard to perform. However, even in such cases (see figure 3), the microlevel reasoner method evaluates meaningful results.

Figure 3 shows that about 50 percent of the time, the ratio (real time to estimated time) is between 0.9 and 1.1. It also shows that even in the worst case, for 80 percent of the time, the ratio is between 0.1 and 10.0, and 60 percent of the time, it is between 0.5 and 2.0. Even in the worst case, the result is still acceptable for soft real-time systems.

An interesting context switch occurs with the *goal* working-memory element in RUBIK. Although in this case, the variance was not small (table 2), a match time with a precision better than 10 percent is predicted by the microlevel reasoner method more than 90 percent of the time.

In all other cases where a basic action does not correspond to a context switch, the run-time prediction is even more accurate. This accuracy is expressed by the small variances of the corresponding working-memory types in tables 1 through 4.

The microlevel reasoner method works for complete rules in the same manner as for single basic actions, except that the tokens of all working-memory-element actions belonging to one right-hand side are gathered at the end of the one-input-node chains before the estimation task starts. As a typical example, I chose the right-hand side of RUBIK's rule *minus\_90*. The right-hand side consists of 21 *change* statements.

Figure 4 shows the actual (simulated) run time versus the estimated run time. Although

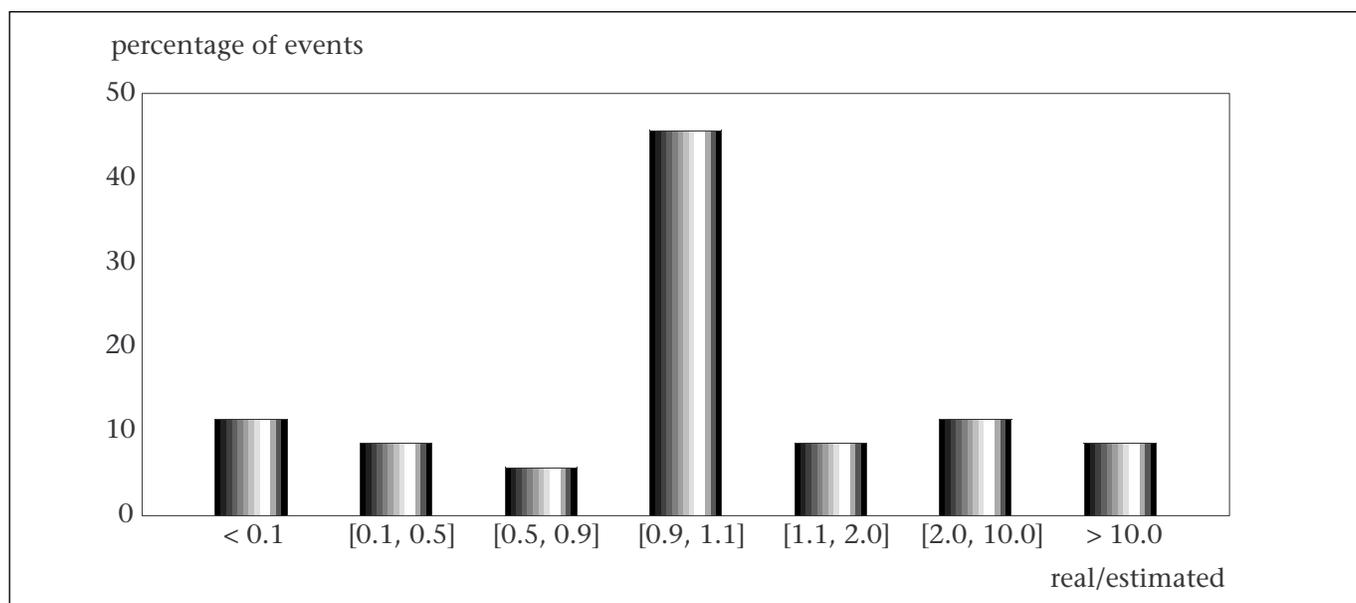


Figure 3. Distribution of Real Time versus Estimated Time for a Basic Action Initiating a Context Switch.

the estimated run time deviates slightly from the actual one, the estimation models the run-time behavior well. For all rules of the production systems tested, a notable correspondence between real and estimated rule-execution times was observed. In those cases where the right-hand side contains a context switch working-memory element, the accuracy of the prediction is severely reduced. This accuracy is also reduced for systems using deep networks because errors in the estimation of successful matches are amplified as one moves down the network. This reduction in accuracy is another reason that the method does not work well with RUBIK. The microlevel reasoner method does not provide either an upper or a lower bound on the amount of time required because the input-output ratios of tokens for two-input nodes vary and represent estimations only. Therefore, the method is not applicable to hard real-time systems.

### The Upper-Bound Method

For hard real-time systems, the *upper-bound method* was designed. The idea behind this method is to classify the tokens stored in beta memories in a way that facilitates the prediction of the number of matches. In contrast to the microlevel reasoner method, the upper-bound method always calculates an upper bound of the number of matches.

The upper-bound method classifies interesting attributes, that is, those that are tested

when a token arrives at the opposite memory of a node. As shown in figure 5, we assume that a two-input node receives tokens of working-memory type <A> on the left and tokens of working-memory type <B> on the right. It is further assumed that the node compares the attribute field\_a1 of the left incoming tokens with the attribute field\_b1 of the right incoming tokens. The left memory contains  $n$  tokens of type <A>. When a token of type <B> arrives, the value of its attribute field\_b1 is known. In the worst case, the two-input-node condition would be tested  $n$  times, comparing the complete left memory with token <B>. This comparison could theoretically yield  $n$  new tokens. However, if knowledge of the exact values of the attribute field\_a1 of the tokens stored in the left memory would be available, a maximum number of generated tokens much smaller than  $n$  could be deduced. The upper-bound method gains this information and partitions node memories into intervals. In my example, a token referencing a field\_a1 value would be assigned to a specific interval. Only tokens belonging to this interval are subsequently compared with the corresponding approaching token <B> from the opposite memory.

One can imagine that this partitioning is performed recursively down the network for each two-input node. When the prediction starts, the upper-bound method knows which tokens from which interval to compare with. Theoretically, in RETE's worst case, one would

WMT	monkey	goal	object
$\mu$	0.99	1.02	1.03
$\sigma$	0.14	0.69	0.19

Table 1. The Averages and the Variances of the Ratio of Real Time to Estimated Time for the EMAB Application

WMT	bot-cor	cub-ord	face	fr-face	face-rel	goal	sou	task
$\mu$	1.00	1.03	1.05	1.30	1.00	1.90	0.98	1.00
$\sigma$	0.98	0.89	0.88	1.78	0.00	8.82	0.37	0.00

Table 2. The Averages and the Variances of the Ratio of Real Time to Estimated Time for the RUBIK Application

WMT	action	alarm	clock	message	sensor	weather
$\mu$	1.03	1.00	0.99	0.99	1.06	0.92
$\sigma$	0.04	0.00	0.00	0.00	0.03	0.03

Table 3. The Averages and the Variances of the Ratio of Real Time to Estimated Time for the VAHINE Application

WMT	time	car	cross
$\mu$	1.05	0.93	0.97
$\sigma$	0.03	0.01	0.01

Table 4. The Averages and the Variances of the Ratio of Real Time to Estimated Time for the CARS Application

(WMT = working-memory type,  $\mu$  = average,  $\sigma$  = variance)

assume that every token stored in the opposite memory is a candidate for the match. The upper-bound method reduces the candidate set by taking only the tokens of the corresponding interval from the opposite memory into consideration. It is important to note that the upper-bound method does not only work for integer attributes but also for float and character string attributes.

### Results of the Upper-Bound Method

The following formula constitutes a metric for the run-time complexity of the estimation task itself:

$$\begin{aligned} &\text{run-time upper bound} \\ &= (K_1 * Nb\text{-intervals}^{Nb\text{-nodes}} \\ &+ Nb\text{-nodes} \log Nb\text{-intervals}) . \quad (9) \end{aligned}$$

The first term represents the complexity of the algorithm performing the actual match prediction. The second term represents the complexity of the algorithm classifying each token in its proper place. *Nb-intervals* is the number of intervals used in the upper-bound method to classify the tokens in one memory according to one compared attribute.  $K_1$  is a

constant value. *Nb-nodes* represents the number of affected nodes for each recognize-act cycle. This number depends on the nature of the production system but usually is small. Therefore, to perform a rapid prediction, the number of intervals should also be small. Having a lot of intervals allows the computing of a good upper bound but at the cost of increased time. A compromise must be found between the quality of the prediction and its run-time complexity.

Table 5 represents the averages and the variances of the ratio (numbers of comparisons performed to numbers of comparisons predicted) for all basic actions of one application. Note that the upper-bound method calculates upper bounds; therefore, the averages are smaller than 1. Compared to the microlevel reasoner method, the quality of the prediction is worse, but it is still better than the theoretical worst case would lead us to expect.

### The Extended-Upper-Bound Method

With the upper-bound method, the classifying intervals are defined at compile time. This solution has some drawbacks. First, it is not

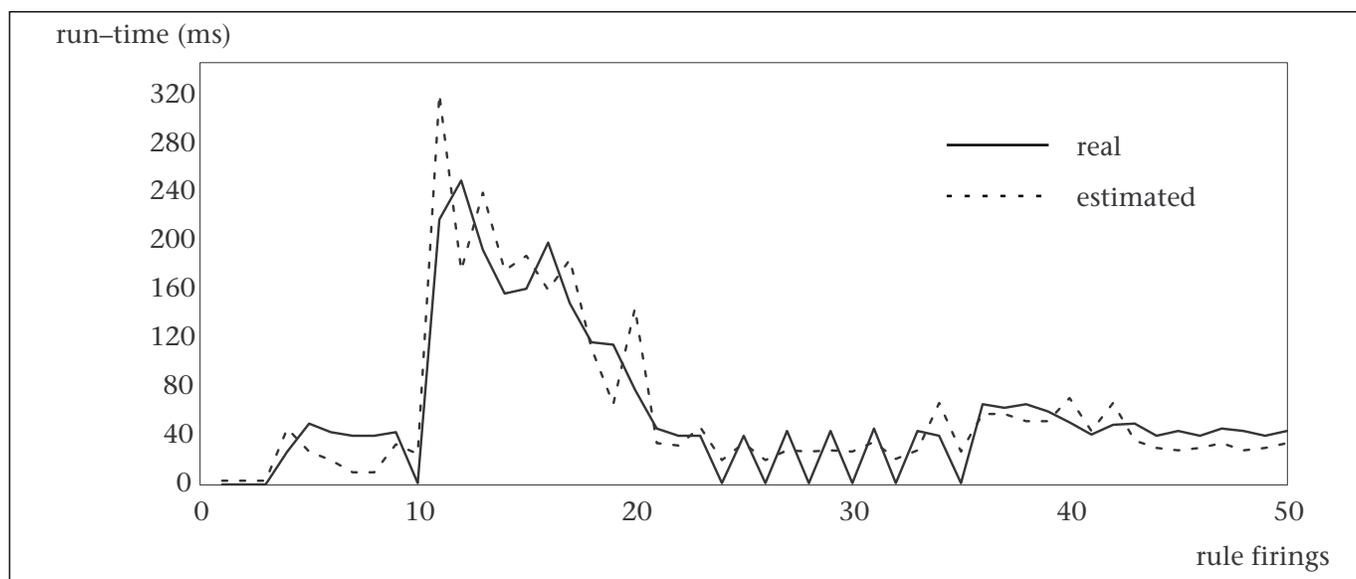


Figure 4. Rule-Execution Times versus Estimated Run Times of Rule *minus\_90* in RUBIK.

easy, even for the application programmer, to find the right number of intervals and their correct boundaries. Second, a static interval definition is not sufficient to guarantee an equal distribution of the tokens.

In contrast to the upper-bound method, the extended-upper-bound method takes into consideration that intervals can be rearranged dynamically during the run time of the expert system. In the best case, intervals can be rearranged in such a way that there is exactly one token for each interval. This arrangement corresponds to the unique-attribute representation used for the UNI-RETE (Tambe 1991) algorithm. In this sense, the extended-upper-bound method is a generalization of the unique-attribute representation. However, in practice, more than one token will correspond to one interval because it is run time intensive to rearrange the intervals after each basic action. Also, when working-memory elements hold exactly the same value for join attributes, the tokens are kept together. In general, there is a trade-off between the numbers of intervals and the quality of the prediction.

In theory, we expect better results by classifying intervals dynamically during run time. The nonempty token memories have to be reclassified after having computed the new interval definitions. In practice, it is unwise to recalculate new interval boundaries after each basic action: This process is time consuming because tokens have to be reassigned to intervals. The extended-upper-bound method was tested after each basic action was

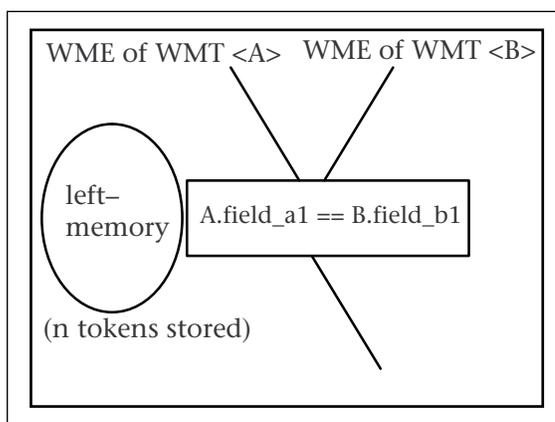


Figure 5. A Simple Two-Input Node (WME = working-memory element; WMT = working-memory type).

reclassified, after 10 basic actions, and after 50 basic actions.

The number of tokens in an interval could either be constant or variable. Different experiments were performed, such as storing in one interval, at most, the square root of the total number of tokens contained in the token memory. Obviously, the best results can be obtained by limiting the number of tokens to one, as shown in table 8. This limitation corresponds exactly to the unique-attribute representation used in some SOAR (Laird, Newell, and Rosenbloom 1987) applications.

Table 6 shows the quality of the prediction when the intervals are updated after each

Application	EMAB	CARS	RUBIK	VAHINE
$\mu$	0.89	0.77	0.18	0.62
$\sigma$	0.30	0.30	0.20	0.50

Table 5. Quality of Run-Time Estimations for the Upper-Bound Method.

Application	EMAB	CARS	RUBIK	VAHINE
$\mu$	0.72	0.79	0.01	0.69
$\sigma$	0.41	0.28	0.26	0.50

Table 6. Results of the Extended-Upper-Bound Method When Updating the Intervals after Each Basic Action.

Application	EMAB	CARS	RUBIK	VAHINE
$\mu$	0.70	0.78	0.005	0.61
$\sigma$	0.41	0.28	0.38	0.50

Table 7. Results of the Extended-Upper-Bound Method When Updating the Intervals after 10 Basic Actions.

Application	EMAB	CARS	RUBIK	VAHINE
$\mu$	0.86	0.82	0.01	0.71
$\sigma$	0.08	0.27	0.22	0.47

Table 8. Results of the Extended-Upper-Bound Method When Updating the Intervals after Each Basic Action, Limiting the Content of Each Interval to One Token.

$\mu$  = average,  $\sigma$  = variance

basic action and when, at most, the square root of the total amount of tokens in the corresponding token memory is considered. The same holds for table 7, but this table reveals worse behavior because the reclassification is performed regularly after 10 basic actions.

For RUBIK, the prediction is not as good as it is with the other expert systems. The average is poor compared to the microlevel reasoner method. Also, the extended-upper-bound method performs worse than the upper-bound method in this case. There is one common reason for this behavior. The previously described strategy to split intervals into two equally sized intervals is especially disadvantageous for RUBIK because working-memory data are clustered in certain intervals. Thus, it costs many recognize-act cycles until the method generates sufficiently small intervals. In contrast, the user could determine optimal static intervals when the semantics of the application and the distribution of the working-memory elements are exactly known.

The tables presented so far don't show the

dynamic behavior of the presented algorithms. Figure 6 shows the quality of the prediction in a certain time frame for the VAHINE expert system. The dashed line represents the actual performed comparisons in the network. The solid line shows the number of comparisons performed by the upper-bound method. The dotted line represents the number of comparisons predicted by the extended-upper-bound method. For this example, it can be observed that the extended-upper-bound method calculates upper bounds that are significantly better than those calculated by the upper-bound method. However, in general, the extended-upper-bound method is not much better than the upper-bound method and never justifies the run-time overhead caused by dynamic boundary search and the reclassification of intervals. The following formula constitutes a metric for the complexity of the run-time estimation task itself:

$$\begin{aligned} &\text{run-time extended upper bound} \\ &= (K_1 * Nb\text{-intervals})^{Nb\text{-nodes}} \end{aligned}$$

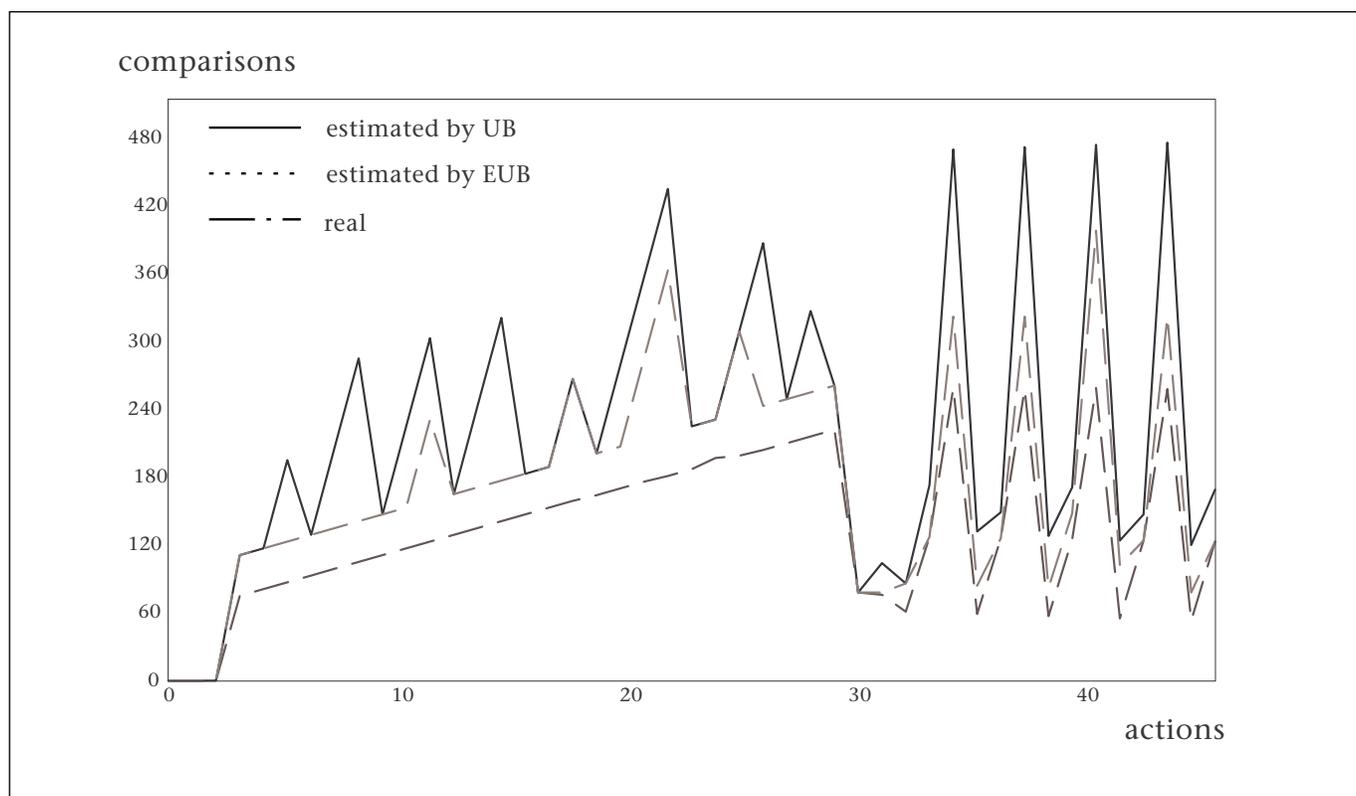


Figure 6. The Quality of the Extended-Upper-Bound (EUB) Method and the Upper-Bound (UB) Method for VAHINE.

$$+ Nb\text{-node} \log Nb\text{-intervals} \\ + T_{calc} \text{ .} \quad (10)$$

Note that the complexity is similar to the upper-bound method but enhanced with  $T_{calc}$ . This factor represents the time that is needed to find new meaningful boundaries and reorganize all memories according to these boundaries. It is exactly this action that makes the extended-upper-bound method run time intensive.

## Discussion and Practical Impacts

I think that for researchers and practitioners, it was important to discover the limits of what is feasible in the area of run-time prediction for production systems. I showed that in principal, production systems can meet deadlines with fine granularity by using the upper-bound or the extended-upper-bound method. The time for the execution of a basic action can be predicted with these two methods. Hence, the granularity at which interrupts can be handled can be determined, and thus, predictability can be guaranteed. As a practitioner, I advocate cautious use of the two methods because it might happen that more central processing unit time is used for the

estimation algorithms than for the match itself. Indeed, both methods are exponential in the number of affected nodes. Recent experiments have shown that this danger can be reduced for the upper-bound method by selecting only a few intervals or knowing the size and the distribution of working-memory elements in the working memory. However, to use these methods requires good knowledge of the production-system application. Although the upper-bound application is useful for many applications, the extended-upper-bound method cannot be applied frequently because its run-time estimation algorithm consumes more central processing unit time than the match phase for systems where more than 10 nodes are affected in a match. This consumption is because the boundary search and the dynamic reclassification of intervals are too costly. Nevertheless, for learning systems such as SOAR, the extended-upper-bound method could be used to search for boundaries on newly added rules, provided that the creation of new identifier symbols could be handled. Once these boundaries are established, the system could continue with the upper-bound method. The extended-upper-bound method would then only be applied once after



```

Rule_1 : RULE;
P1 (locomotive   thrust > 100; in_use = false;
    type = electric; track_w = WEU)
P2 (railroad_car weight < P1.thrust;
    height < P1.height;
    track_w = P1.track_w)
==> /* attach action in right hand side */
END RULE_1;

```

Figure 2. Simple Left-Hand Side of a Rule.

nodes or *beta memory* when they appear after two-input nodes.

Whenever a fact (working-memory element) is added or deleted from working memory, a token containing the working-memory element is sent to the root node. The root node sends the token to all the successor nodes. It is then tested if it fulfills the property checked in the one-input nodes. In case of a positive test, it is sent to the successor nodes; otherwise, it is discarded. When a token enters a two-input node, it is first stored in its corresponding left or right memory. If a token from the left memory and a token from the right memory fulfill all two-input-node conditions, the two tokens are joined to form a new one. This new token is sent to the next node. Any token that filters down to an end node (rule\_1) indicates an *instantiation* of the production, which means the rule will be taken into consideration during the conflict-resolution phase. In the presented example, rule instantiations of type rule\_1 are stored in the conflict set. In case of a third pattern, the conflict set would form a new left memory.

Negated conditions, which are not presented in the example, are handled by not nodes. The not node maintains a counter for each token that has entered the left memory. This counter represents the number of consistent tokens in the right memory. If the token that has entered the left memory ever becomes consistent with one of the tokens in the right memory, then this token can be sent to the successor node. If the counter of a token changes from zero to nonzero, then this token no longer represents a match, and all previously joined tokens must be removed.

Let us assume that somewhere outside in the production-system program, a working-memory element of type *locomotive* with the following properties is created: thrust = 120, in\_use = false, type

= electric, height = 4, and track\_width = WEU. As previously explained, this action is performed with a *make* statement. A token of type <+, locomotive> is sent to the root node. The + tag indicates that working-memory element *locomotive* has been added to working memory. The token is sent from the root node to the two successor nodes. One successor node (the railroad car node) discards the token because the test in the node does not correspond to the content of the token. The other successor node accepts the token and sends it on to the next one-input node. All the subsequent one-input nodes accept the token; therefore, it is stored at the end of the one-input-node chain in the left memory. After arriving at the left memory, the two-input node compares the token with all tokens in the opposite right memory and checks whether the intraelement conditions are fulfilled. In the example, there is no token in the opposite right memory; therefore, there is nothing to check. Thus, the algorithm stops.

Let us assume that during the execution of the production system, another working-memory element of type *railroad\_car* with the following properties is created: weight = 20, height = 3, track\_width = WEU. A token of type <+, railroad\_car> is sent to the root node. This time, the token runs down the railroad car branch and is stored in the

right memory. After arriving at the right memory, the two-input node compares the token with all tokens in the opposite left memory and checks whether the intraelement conditions are fulfilled. One token (the locomotive token) is stored in the left memory. Because the intraelement conditions are fulfilled, both tokens are joined and sent to the next node. In the example, the new combined token is sent to an end node. Thus, the new token is stored in the conflict set, and the rule instantiation rule\_1 is ready to fire. Following the described procedure, partial instantiations of rule\_1 have been remembered in the left memory and the right memory. When a working-memory element is deleted, a token with a minus tag is sent to the root node and is subject to the same tests as the token with the corresponding plus tag. In case that a token with a minus tag approaches the left memory or the right memory, the token is deleted. A modification of a working-memory element is treated by sending first a minus-tagged token followed by a plus-tagged token.

The RETE algorithm stores results from previous recognize-act cycles in the left memory and the right memory and uses them in subsequent cycles. It also exploits similarities between conditions of rules, which is not shown in the example. For more details, please read Forgy (1982).

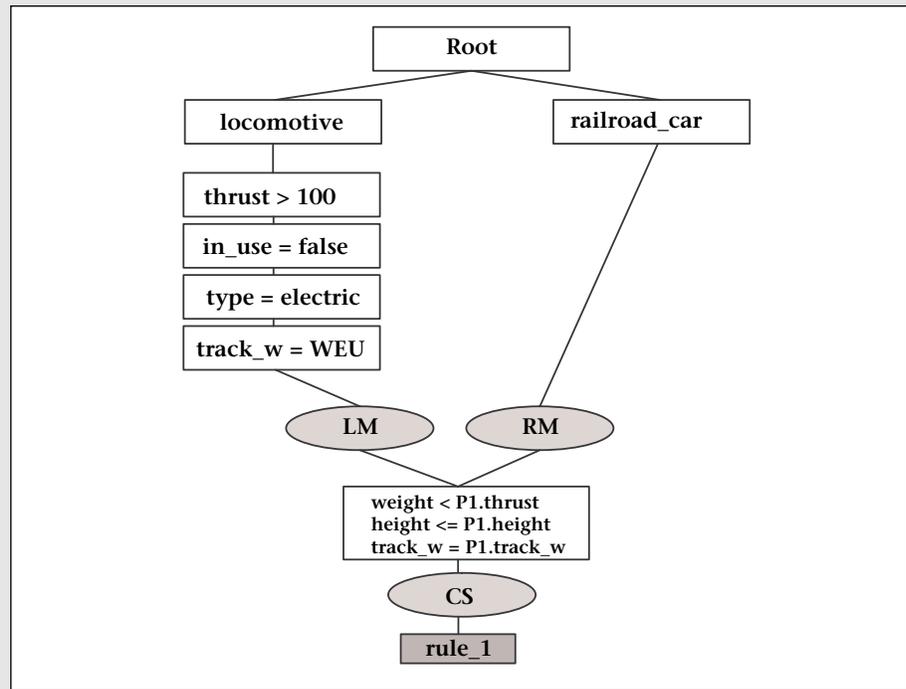


Figure 3. RETE Network (LM = left memory; RM = right memory; CS = context set).

many thousand recognize-act cycles to establish the interval boundaries. The activation of the extended- upper-bound method could be performed when the expert system task is idle. However, if the number of affected nodes increases without bound, then the upper-bound method would also not be applicable.

In contrast to the upper-bound and the extended-upper-bound methods, the microlevel reasoner method is linear in nature and predicts match time of basic actions, as well as complete rule-execution times, with an error of less than a factor of two for non-context switch actions. The microlevel reasoner method cannot give a rigorous upper bound for run time. It is therefore well suited to systems that have soft time constraints only. Such systems do not require guaranteed answering times for particular events but, rather, an estimation of the average time for a typical request. Based on these estimations, exception handlers can be raised that are able to deliver incomplete or suboptimal results to a high-level reasoning system before a user-defined deadline is exceeded. The microlevel reasoner unhesitatingly can be applied to real-world production systems.

Unfortunately, I must relegate to future research the challenging question of whether complete production systems can meet deadlines. It is well known that the fundamental question in its most general form cannot be decided, but progress on the run-time prediction of specific production-system classes can still be made.

### Acknowledgments

I am grateful to Anoop Gupta, who helped me to gain deep insight into this subject during my visiting period at Stanford University in 1989. For the discussions about this article and all the comments received, I owe thanks to Milind Tambe and Anurag Acharya from Carnegie Mellon University and Toru Ishida from Kyoto University. Finally, I am grateful to the anonymous referees and the colleagues in my department.

### References

- Acharya, A., and Tambe, M. 1993. Collection-Oriented Match. In *Proceedings of the Second International Conference on Information and Knowledge Management*, 516–526. Washington D.C.: ACM Press.
- Bahr, E.; Barachini, F.; Doppelbauer, J.; Gräbner, H.; Kaspavec, F.; Mandl, T.; and Mistelberger, H. 1991. A Parallel Production System Architecture. *Journal of Parallel and Distributed Computing* 13:456–462.
- Barachini, F. 1991. The Evolution of PAMELA. *Expert Systems, The International Journal of Knowledge Engineering* 8:87–98.
- Barachini, F. 1990. Match-Time Predictability in Real-Time Production Systems. In *Parallel Processing and Engineering Applications*, ed. R. A. Adey, 209–219. New York: Computational Mechanics Publications.
- Barachini, F., and Theuretzbacher, N. 1988. The Challenge of Real-Time Process Control for Production Systems. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 705–709. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Barachini, F.; Mistelberger, H.; and Gupta, A. 1992. Run-Time Prediction for Production Systems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 478–485. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Cooper, T. A., and Wogrin, N. 1988. *Rule-Based Programming with OPS5*. San Mateo Calif.: Morgan Kaufmann.
- Forgy, C. L. 1982. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem. *Artificial Intelligence* 19:17–37.
- Gupta, A. 1986. *Parallelism in Production Systems*. San Mateo, Calif.: Morgan Kaufmann.
- Gupta, A., and Tambe, M. 1988. Suitability of Message Passing Computers for Implementing Production Systems. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 705–709. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Gupta, A.; Forgy, C. L.; Newell, A.; and Wedig, R. 1986. Parallel Algorithms and Architectures for Rule-Based Systems. In *Proceedings of the Thirteenth Annual International Symposium on Computer Architectures*, 28–37. Washington, D.C., and New York: IEEE Computer Society and the Association of Computing Machinery.
- Haley, P. V. 1987. Real-Time for RETE. In *Proceedings of ROBEX-87*, 36–41. Triangle Park, N.C.: Instrument Society of America.
- Harvey, W.; Kalp, D.; Tambe, M.; McKeown, D.; and Newell, A. 1989. Measuring the Effectiveness of Task-Level Parallelism for High-Level Vision, Technical Report, CMU-conflict set-89-125, Computer Science Dept., Carnegie Mellon Univ.
- Ishida, T. 1991. Parallel Rule Firing in Production Systems. *IEEE Transactions on Knowledge and Data Engineering* 3(1): 11–17.
- Ishida, T.; Gasser, L.; and Makoto, Y. 1992. Organization Self-Design of Distributed Production Systems. *IEEE Transactions on Knowledge and Data Engineering* 4(2): 123–134.
- Kelly, M. A., and Seviara, R. E. 1987. A Multiprocessor Architecture for Production System Machines. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 36–41. Menlo Park, Calif.: American Association for Artificial Intelligence.
- Klein, P. 1990. The Safety-Bag Expert System in the Electronic Railway Interlocking System ELEKTRA. In *Proceedings of Expersys-90*, 177–182. Los Angeles: IITT-International.
- Laird, J. E.; Newell, A.; and Rosenbloom, P. S. 1987. SOAR: An Architecture for General Intelligence. *Arti-*

*ficial Intelligence* 33(1): 1–64.

Miranker, D. 1990. *TREAT: A New and Efficient Match Algorithm for AI Production Systems*. San Mateo, Calif.: Morgan Kaufmann.

Miranker, D. P.; Brant, D. A.; Lofaso, B.; and Gadbois, D. 1990. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 685–692. Menlo Park, Calif.: American Association for Artificial Intelligence.

Paul, C. J.; Acharya, A.; Black, B.; and Strosnider, J. K. 1991. Reducing Problem-Solving Variance to Improve Predictability. *Communications of the ACM* 34:81–93.

Schor, I. M.; Daly, P. T.; Lee, H. S.; and Tibbits, B. R. 1986. Advances in RETE Pattern Matching. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, 226–232. Menlo Park, Calif.: American Association for Artificial Intelligence.

Stankovic, J. A., and Ramamithram, K. 1990. What Is Predictability for Real-Time Systems? *Real-Time Systems* 2(4): 247–254.

Stankovic, J. A., and Ramamithram, K. 1988. Real-Time Computing Systems: The Next Generation. In *Tutorial on Hard Real-Time Systems*, ed. J. Stankovic, 14–37. Los Alamitos, Calif.: IEEE Press.

Tambe, M. 1991. Eliminating Combinatorics from Production Match, Ph.D. diss., CMU-conflict set-91-150, Computer Science Dept., Carnegie-Mellon Univ.

Tambe, M., and Newell, A. 1988. Some Chunks Are Expensive. In *Proceedings of the Fifth International Conference on Machine Learning*, 451–458. San Mateo, Calif.: Morgan Kaufmann.

Tambe, M.; Kalp, D.; and Rosenbloom, P. 1991. UNI-RETE: Specializing the RETE Match Algorithm for the Unique-Attribute Representation, Technical Report, CMU-conflict set-91-180, Computer Science Dept., Carnegie-Mellon Univ.

Wang, C.-K.; Mok, A. K.; and Cheng A. M. K. 1990. MRL: A Real-Time Rule-Based Production System. In *Proceedings of the Real-Time Systems Symposium*, 267–276. Washington, D.C.: IEEE Computer Society.



**Franz Barachini** received his M.Sc. in 1980 and his Ph.D. in 1983, both from the Technical University in Vienna. In 1989, he was a visiting scientist at Stanford University, working in real-time knowledge-based systems. From 1986 to 1993, he headed the Knowledge-Based Systems

Department at the Alcatel-Austria Research Laboratory in Vienna. In 1994, he became head of the Voice Processing Department. Since 1992, he has also been a lecturer and, since 1994, an assistant professor at the Technical University in Vienna. He is a member of the Austrian and the American Association for Artificial Intelligence.