The Challenge of Real-time Process Control for Production Systems

Franz Barachini, Norbert Theuretzbacher
ALCATEL Austria – ELIN Research Center
Floridusgasse 50 A–1210 Vienna, Austria

Abstract

Although the technology of expert systems has been developed substantially during the past decade, there still seems to be relatively little application to ***time–critical*** *problems because of their extensive computational requirements. One application area of particular interest is that of* ***process control****. Because this area requires* ***real–time*** *operation, the expert system must operate within the time scale of the process involved. Here we present techniques which have been used to implement PAMELA, a language suitable for building time–critical expert systems. We discuss substantial optimizations of the well known RETE algorithm and present run–time measurements based on these optimizations. Despite the critics on RETE's real–time behaviour we show that the presented optimizations and extensions will cover the demands of many real–time applications. In order to efficiently support process control applications, some useful language constructs concerning* ***interrupt handling*** *and* ***rule interruption*** *are discussed.*

1 Motivation

PAMELA (**PA**ttern **M**atching **E**xpert system **LA**nguage) was born out of frustration. After having examined many AI–languages and tools, it became apparent that most of them were unable to handle *time–critical* problems efficiently. Here we raise some of the problems which we tackled with PAMELA.

It came to our attention that although several other inference algorithms were developed [Mc Cracken, 1978; Mc Dermott et. al. 1978], most of the declarative languages with reasonable performance use the RETE algorithm [Forgy, 1979; Forgy, 1982] as an indexing scheme. Consequently, we concentrated our study on this algorithm and others which have been derived from it [Miranker, 1987; Scales, 1986].

It became clear that a pure production system is not fully applicable to process control applications, since asynchronous peripheral events may influence the recognize–act cycle during operation. Interrupt–handling facilities are fundamental for creating timely and elegant solutions to real–time problems. Being able to interrupt the recognize–act cycle and to modify an existing *working memory element* (WME) within a interrupt routine would be a nice feature. However, two problems arise when incorporating such a feature. Firstly, consistency of the *RETE network* cannot be guaranteed when the CHANGE[1] command is performed immediately during an interrupt. Secondly, in most of today's production systems it is impossible to access a WME outside the scope of a rule.

[1] This command is used to modify a WME.

2 Improvements on the RETE–Algorithm

Although several proposals for RETE run–time improvements have been made [Schor et. al., 1986; Scales, 1986], there remain some parts worthy of optimization. For the following discussion it is assumed that the reader is familiar with the RETE algorithm and the vocabulary normally used in reference to OPS [Brownston et. al., 1986; Forgy, 1979].

Every implementation of the RETE algorithm requires the sequentialization of the network except when employing parallel architectures [Gupta, 1986b; Gupta et. al., 1986c; Kelly and Seviora, 1987; Miranker, 1986; Tenorio, 1984]. A well known method of implementation [Forgy, 1982] is designed specifically for interpreters.

2.1 Eliminating the Explicit Token–Stack

In PAMELA [Barachini, 1987] nodes are represented as procedures which receive a token as a parameter, thus eliminating the need for a *token–stack*. Instead of the *token–stack*, the processor's stack is used. Procedures are called recursively when the network is traversed. The recursion depth depends on the complexity of the most specific *left hand side* of the rules (i.e. containing the greatest number of patterns).

PAMELA maintains node types (such as negative–join nodes, positive–compare nodes, negated nodes, etc.) for *two–input nodes*. Except for conditions and calls to successor nodes, each two–input node type uses the same code for determining the consistently bound tokens and for handling the token memories. For each node type there exists a specific *node handler* which receives the *node–number* of the node to be processed along with the token. For each individual node, the unique conditions and calls to successive nodes are executed within a particular part of the current node handler.

2.2 Optimization During CHANGE and REMOVE

Within two–input nodes PAMELA maintains two memories (the *left counter memory* and the *right counter memory*). For each incoming token, the counter memory indicates the number of consistently bound tokens in the opposite token–memory.

Every two–input node produces a shared token–memory (*output–memory*) accessed from the successors of the two–input node. As an extension to Forgy's implementation

[1982], two *backlink–pointers* are stored for each token in the token–memory. The first *backlink–pointer* references the counter for that part of the token received from the left predecessor node. The second references the counter for that part of the token having arrived from the right predecessor node.

When a token with a negative tag enters a two–input node (indicating the deletion of a specific WME) within OPS83, it is subject to the same tests applied to the token, which had previously arrived with a positive tag. These tests are not repeated in PAMELA. Let us consider a token X entering a two–input node during a remove operation. The output memory below the two–input node contains one or more tokens that are the result of concatenating X with another token from the opposite token memory. Such tokens may be identified and removed by scanning the output memory for all tokens which contain X as a left or right subpart. When we find such a token containing X the backlink pointer represents the number of consistently bound opposite tokens. Thus we know how far to search in the output memory in order to remove all tokens containing X. We have to decrement the counters which implies an algorithmic overhead but we don't have to recheck the inter–element conditions.

Note that a counter with value zero avoids PAMELA having to search the output memory (Gupta's and Forgy's measurements [1983] show that this occurs frequently). Even for counters greater than zero, search time may be reduced significantly since the search for tokens can be terminated once the last element has been found.

2.3 Node Reduction

In addition to the algorithm described above, *node reduction* of the network may be an efficient run–time optimization in PAMELA. The following PAMELA rule–fragments serve as an example for the discussion. The *right hand sides* are not relevant and therefore omitted.

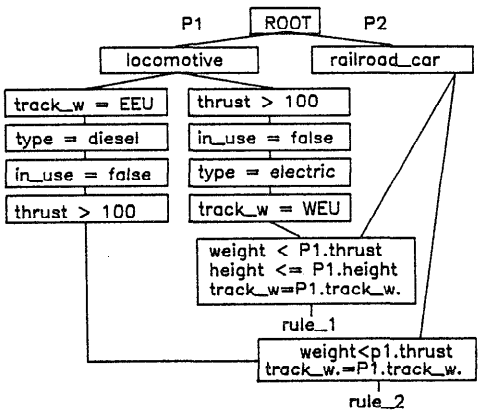
```
RULE_1 : RULE;
 P1 locomotive
      (thrust > 100; in_use = false;
       type = electric; track_w = WEU)
 P2 railroad_car
      (weight < P1.thrust; height < P1.height;
       track_w = P1.track_w)
==>
 /* attach action in RHS */
END RULE_1;

RULE_2 : RULE;
 P1 locomotive
      (track_w = EEU; type = diesel;
       in_use = false; thrust > 100)
 P2 railroad_car
      (weight < P1.thrust; track_w = P1.track_w)
==>
 /* attach action in RHS */
END RULE_2;
```

The conventional network established by the RETE algorithm for these rules is illustrated in Figure–1.

2.3.1 Sorting, Compressing and Sharing of Nodes

The RETE algorithm has the advantage that one–input



Figure–1 : Conventional RETE–Network

nodes are shared as far as possible. Two equal chains of one–input nodes having different successor nodes are replaced by only one chain of one–input nodes. The last node of this new chain now contains more than one successor node. To obtain optimal one–input node sharing, the sequence of the one–input nodes within the equal chains has to be the same. Therefore PAMELA sorts the intra–element conditions before the network is constructed. In addition, one–input nodes are compressed by reducing one–input node chains to a single one–input node if each of the nodes has only one successor.

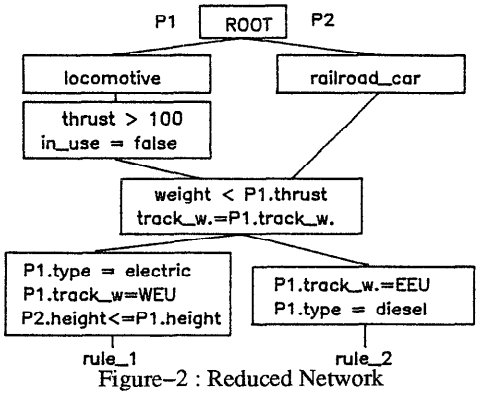
Since two–input node processing requires an excessive amount of run–time, it is desirable to reduce the number of these nodes as much as possible. Two–input nodes with the same left predecessors, right predecessors and conditions can be combined into one node with two successors[2]. The requirement that both nodes have the same predecessor can be artificially satisfied by placing all the different intra–element conditions between the nodes and their identical predecessors behind the shared two–input node. We call these nodes placed behind shared two–input nodes ***special one–input nodes***. They are treated like one–input nodes, except that they maintain one associated token–memory.

This "aggressive" algorithm is performed recursively over the whole network. The newly generated *special one–input nodes* are sorted in order to allow optimal special one–input node sharing. Compared to the original network there is a node reduction of more than 40% in our example (see Figure–2).

Although the described method yields a minimal network[3], it is not clear in advance whether this method always improves run–time performance. It can be shown that the improvements due to the proposed aggressive two–input node sharing depends heavily on world data (working memory content). Thus the run–time behaviour turns out to be problem dependent. If we assume that on the average 10% will pass successfully from the cross–product of the tokens at two–input nodes in our example, then run–time will decrease.

2 Sharing is also done if only sub–parts of two–input nodes are equal.

3 The discriminating network is minimal in that it contains the minimum number of nodes.



Figure–2 : Reduced Network

But if we assume 50% passing tokens[4] the run–time will increase.

There are several possibilities for two–input node sharing available within PAMELA. Actually PAMELA offers 10 sharing levels. Some Sharing levels represent sure cases e.g. when they are applied run–time will always decrease. The other levels don't always yield a decrease in run–time.

3 Uniprocessor based Measurements

Currently, PAMELA code can only be executed on INTEL–286 processors. ALCATEL–Austria develops its own specially tailored hardware and software – it would be unfair to perform run–time measurements on this architecture. The absolute performance measurements are therefore made on an IBM–PC/AT.

In order to cover a broad range of different expert system bench–marks we selected the widely referenced applications MAB, EMAB, COMBI, RUBIK and TOURNEY as test cases. They already have been used as bench–marks by Gupta [1986c] and Miranker [1987]. We didn't check larger expert systems because of limited PC resources[5].

We concentrate on the most run–time efficient sharing level denoted by *SHARING* and on the worst level denoted by *REDUCING*. Table–1 represents the standard PAMELA implementation already including the optimizations discussed in chapters 2.1 and 2.2 (elimination of the token–stack, optimization during CHANGE and REMOVE). The measurements are given in seconds. The conflict set resolution strategy is identical to OPS83.

The last column in Table–1 contains the number of rule–firings. Positions indicated with *)[6] couldn't be tested because of memory restrictions of the PC. It would be unfair to include run–time of input/output routines when comparing two inference engines. Thus we didn't add the run–time of the output routines to the overall run–time of the inference engines. Hence OPS83's run–time is not exactly the same as indicated by Forgy.

[4] Gupta's and Forgy's measurements [1983] show that, in practice, this very high inlet/outlet ratio occurs very rarely.

[5] PAMELA is certainly applicable for large Expert Systems. ALCATEL–Austria is currently implementing an Expert system with PAMELA for monitoring railway stations, which consists of hundreds of rules.

[6] Dynamic memory overflow occurs for these examples.

	no sharing	sharing	reducing	OPS83	rfire
MAB–NASA	0.49	0.38	2.63	0.72	81
EMAB	30.21	28.67	*)	869.5	903
COMBI	41.80	41.50	42.20	1346.0	1387
RUBIK	115.33	113.83	*)	129.5	342
TOURNEY	85.60	80.90	*)	*)	543

Table–1 : Run–time Measurements

Analyzing the results, we recognize that careful two–input node sharing is about 22% better than the unshared version of MAB–NASA. Although this is not of an order of magnitude we proved that two–input node sharing is an advantage for certain examples.

The most efficient MAB–NASA version of PAMELA performs only twice as good as OPS83. The reason for that behaviour is that the token–memories are rather small–sized and there is only a limited number of rule–firings. To overcome this disadvantage we insignificantly modified the world data in order to have more than one monkey searching for the bananas. This modified version yields 903 rule–firings instead of 81.

Table–1 presents the largest COMBINATION example not exceeding the memory limits of the PC. This example evaluates all possible combinations of terms of the sum yielding the number 14. The terms of the sum may be numbers between 1 and 5. Experiments showed, that PAMELA's high performance could also be demonstrated with lower numbers yielding less rule–firings.

The number of rule–firings in the RUBIK's example depends on the initialized scrambled position of the cube. Because of memory limits we chose a very simple initialization. The run–time difference between OPS83 and PAMELA is not an order of magnitude for this example.

Neither PAMELA nor OPS83 are able to solve the TOURNEY problem on the PC. There are too many MAKE and CHANGE statements within this bench–mark causing a memory overflow. The PAMELA bench–mark was performed on our own hardware. Since we reject comparing bench–marks running on different architectures the numbers for the TOURNEY bench–mark shouldn't be taken to literally.

In general we observed a better run–time behaviour of PAMELA compared to OPS83 when token–memory size was increased.

Obviously, comparing our run–time measurements with XC [Nuutila et. al., 1987] we are able to reject most criticisms of the RETE algorithm. We showed that substantial run–time optimizations of this algorithm enhance its real–time capabilities.

4 Stimulating the Recognize–Act Cycle

Current inference engines have not been designed with interrupt–handling facilities in mind. Such facilities are however essential for solving real–time problems. During the "normal" recognize–act cycle (*match–select–act*), no modifications or deletions of a WME can be performed within interrupts. Variable–binding outside the scope of a

rule is essential for a completely interrupt–driven expert systems.

PAMELA offers the possibility to stimulate the recognize–act cycle within interrupt handlers with the aid of the following functions:

```
SCAN(wme-type;      property-1,      ...
     property-N)
```

This function searches for a specific WME. The type and its properties are given as parameters. The function substitutes the variable binding mechanism of patterns and provides another view of the working memory.

```
Q_MAKE(wme-type;    assignment-1,    ...
       assignment-N, priority)
```

This function allows the creation of a WME within an interrupt routine. The priority parameter reflects the priority of the action.

```
Q_CHANGE(SCAN(...);        assignment-1,
         ... assignment-N, priority)
```

This function allows the modification of a WME outside the scope of a rule (e.g. within an interrupt routine).

```
Q_REMOVE(SCAN(...); priority)
```

This function allows the removal of a WME outside the scope of a rule.

Consistency problems may occur when applying these functions. Suppose that an interrupt occurs during the match phase of the RETE algorithm. The task stops, although the network is not fully updated (e.g. a MAKE action triggers only 6 instantiations instead of 12). When exactly the inverse action (Q_REMOVE) is performed within the interrupt routine, it removes the 6 instantiations but 6 other instantiations would enter the conflict set after returning from the interrupt routine. This would probably not be the user intended behaviour.

We therefore define the MAKE, CHANGE and REMOVE actions as *atomic indivisible actions*. During the execution of these actions no other action can be performed. Thus, actions performed within the interrupt routine (i.e. Q_MAKE, Q_CHANGE and Q_REMOVE) are queued. A priority level is attached to every queue (the last parameter of the Q_MAKE, Q_CHANGE and Q_REMOVE action), which is associated with the interrupt level of the interrupt routine currently initiating the action.

When the end of a *right hand side* is encountered, PAMELA schedules the FIFO–queues according to their priority level. The normal recognize–act cycle resumes after all queues are empty – otherwise the actions in the queues are performed.

It's obvious that the scheduling method described above may cause difficulties in the case of alarm–handling problems. In process control applications, as those designed at ALCATEL–Austria[7], *right hand sides* may contain a reasonable number of statements. If queue scheduling was only to be allowed at the completion of the *right hand side*, significant delays would result, degrading real–time response. Therefore, the user may define *synchronization–slots* within *right hand sides*. When a *synchronization–slot* is encountered, the queues are scheduled immediately. Furthermore every MAKE, CHANGE or REMOVE action serves as an implicit *synchronization–slot*.

At this point, one may ask if there is any difference between scheduling the queues at the end or somewhere in the middle of a *right hand side*. For OPS–like production systems there is no difference, because the select algorithm of the conflict set is performed after all the MAKEs, CHANGEs or REMOVEs are accomplished – regardless if they were queued or not. For PAMELA programs there is a great advantage in using *synchronization–slots*, since PAMELA offers a real *DEMON–Concept*.

The *DEMON–Concept* is completely different to the concept defined by Lee Brownston et al. [1986]. They define the demon as an instance of a rule, which **enters** the conflict set as soon as it has matched the data that it requires. We define the demon as a rule which is **fired immediately** after it has matched the data that it requires. Hence, PAMELA maintains a separate demon conflict set. A select algorithm is applied on this special demon conflict set **after every** atomic action. Consequently, *right hand sides* of rules may be preempted when they include MAKE, CHANGE or REMOVE statements which trigger demons themselves. Since demons are fired immediately, alarm–handling is managed efficiently. Figure–3 shows the extended recognize–act cycle.

An additional difficulty arises when an interrupt occurs within a *right hand side*. Suppose that the demon deletes or modifies a WME currently used in the *right hand side* of the interrupted rule. When returning to the interrupted rule, the WME previously bound may have vanished. To solve this problem PAMELA provides the functions EXISTS `(Pattern)` and MODIFIED `(Pattern)` in order to allow the user to determine whether a WME was deleted or modified by a demon. It is up to the programmer to use these functions when sensitive data is processed within a *right hand side*.

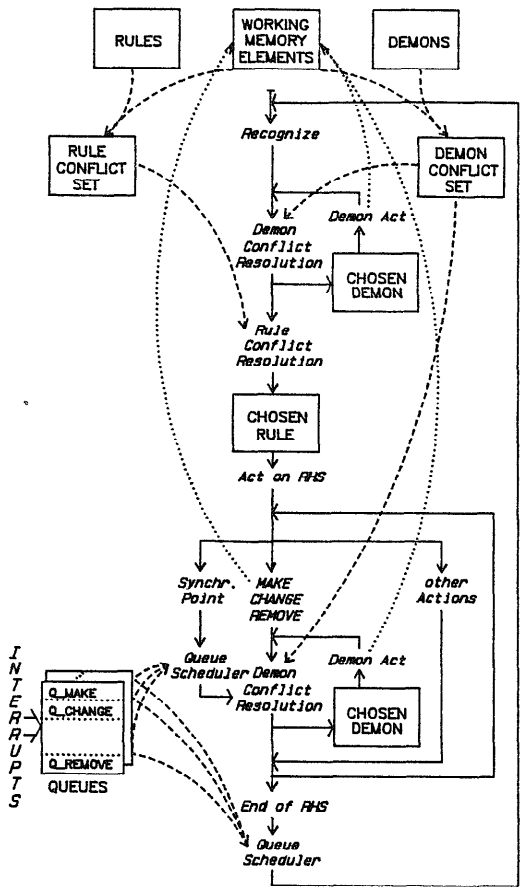
5 Future Directions

Many optimizations on the RETE algorithm have already been discussed and implemented by Gupta et al. [1986a], Shore et al. [1986], Miranker [1987] and Scales [1986]. Some optimizations proposed by these authors and the optimizations presented in this paper are certainly approaching the limit of the performance on uniprocessor–based systems. Concentrating all these optimizations within one product would certainly decrease run–time further. Yet we believe that additional run–time optimizations can only be achieved by switching to parallel architectures.

6 Summary

Implementation techniques especially suited for process–control applications applied in the AI–language PAMELA have been introduced. We discussed interrupt handling features reaching beyond the full set of constructs of current production system languages.

[7] Currently an expert system monitoring a railway station is being implemented.



Figure–3 : PAMELA's Recognize–Act Cycle

Based on the presented run–time measurements, we have every reason to believe that PAMELA is currently world–wide among the fastest uniprocessor based production system implementations.

Acknowledgements

We are especially grateful to Bill Barabash for providing us with all the Production System bench–marks. Without these bench–marks a serious comparison with OPS83 could not be performed. We owe thanks to the members of the PAMELA group – Ernst Bahr, Uwe Egly, Reinhard Granec, Konrad Mayer, Eduard Mehofer, Brigitte Ruzicka, Norbert Schindler, Gabriele Schmidberger, Manfred Twrznik, Dietmar Weickert and Istvan Zsolnay – who contributed to the ideas that have evolved into PAMELA.

References

Barachini F., 1987 : "PAMELA – Eine Deklarative Programmiersprache für Echtzeitanwendungen", Austrian Conference on Artificial Intelligence.

Brownston L., Farrell R.G., Kant E., 1986 : "Programming Expert Systems in OPS5", Addison Wesley.

Forgy C.L., 1979 : "On the Efficient Implementation of Production Systems", Ph.D. Thesis, Carnegie–Mellon University.

Forgy C.L., 1982 : "RETE : A Fast Algorithm for the Many Pattern/Many Object Pattern Matching Problem", Artificial Intelligence, Vol. 21, pp. 21–37.

Gupta A., Forgy C.L., 1983 : "Measurements on Production Systems", Technical Report, Carnegie–Mellon University.

Gupta A., Forgy C.L., Kalp D., Newell A., Tambe M., 1986a : "Results of Parallel Implementation of OPS5 on the Encore Multiprocessor", Draft Report, Department of Computer Science, Carnegie–Mellon University.

Gupta A., 1986b : "Parallelism in Production Systems", Ph.D. Thesis, Carnegie–Mellon University.

Gupta A., Forgy C., Newell A., Wedig R., 1986c : "Parallel Algorithms and Architectures for Rule–Based Systems", in the 13th Annual International Symposium on Computer Architectures, IEEE & ACM.

Kelly M.A., Seviora R.E., 1987 : "A Multiprocessor Architecture for Production System Machine", Proceedings of the AAAI–87, Vol1, pp. 36–41.

McCracken D., 1978 : "A Production System Version of the Hearsay–2 Speech Understanding System", Ph.D. Thesis, Carnegie–Mellon University.

McDermott J., Newell A., Moore J., 1978 : "The Efficiency of Certain Production System Implementations", Waterman D.A. and Hayes–Roth F., Ed., *Pattern–Directed Inference Systems*, Academic Press, New York, pp.165–176.

Miranker D. P., 1986 : "The performance Analysis of TREAT : A DADO Production System Algorithm", International Conference on Fifth Generation Computing, Tokyo 1984, revised article.

Miranker D. P., 1987 : "TREAT : A New and Efficient Match Algorithm for AI Production Systems", Ph.D. Thesis, Columbia University.

Nuutila E. et al, 1987 : "XC – A Language for Embedded Rule Based Systems", Sigplan Notices V22 #10.

Schor I. M., Daly P. T., Lee H. S., Tibbits B. R., 1986 : "Advances in RETE Pattern Matching", Proceedings of the AAAI–86, Philadelphia, 226–232.

Tenorio M. F. M., 1984 : "Parallelism in Production Systems", Ph.D. Thesis, University of California.

Scales D. J., 1986 : "Efficient Matching Algorithms for the SOAR/OPS5 Production System", Report No. KSL 86–47, Stanford University.