THE EXECUTION KERNEL OF RC++: RETE*, A FASTER RETE WITH TREAT AS A SPECIAL CASE

Ian Wright[1]
IKuni Inc.
3400 Hillview Avenue, Building 5,
Palo Alto, CA 94304, USA
wright@ikuni.com

James Marshall
Department of Earth Science and Engineering
Imperial College London
Exhibition Road
London, SW7 2AZ, UK
James.A.Marshall@imperial.ac.uk

KEYWORDS
Rule-based languages, RETE, pattern matching.

ABSTRACT

Some behaviour of computer game agents can be naturally expressed as collections of rules and knowledge bases. General-purpose rule-based languages provide high-level constructs for expressing complex conditional behaviour. We examine the run-time kernel of RC++, a rule-based language developed for game AI, to explore the costs associated with adopting general-purpose, rule-based approaches for computer game production. The kernel of RC++ is the RETE* algorithm, an extension of the RETE algorithm with better time characteristics, but also able to exhibit the beneficial properties of TREAT (a low memory cost alternative to RETE) when required. RETE* achieves this functionality and performance by employing (i) asymmetric deletion, (ii) dual tokens, and (iii) a dynamic beta-memory cut mechanism. The dynamic beta cut allows the RETE/TREAT trade-off to be exploited by users. Theoretical and empirical performance comparisons for RETE, TREAT and RETE* are provided. The implications for the utility of rule-based programming for the computer games industry is discussed, and we conclude that there is still some way to go before rule-based programming can be employed in the game-making process.

INTRODUCTION

Rule-based programming is a natural approach to specifying agent behaviour: rules define conditions on internal state (interpreted as sensory information or memory stores) and associated actions that can alter memory or produce actions in a virtual world. General-purpose rule-based languages (in contrast to simple, relatively stateless, scripting languages) provide powerful pattern-matching constructs that automate some of the programming tasks associated with rule execution. Both the SOAR architecture, which has been used to develop 'bots' for Quake II (Laird and van Lent, 99), and our own work on RC++ (Wright and Marshall, 00), a rule-based language implemented for Sony's PlayStation2 architecture, are `general-purpose' rule-based languages of this type. Our aim in this paper is to explain what is occurring 'under the hood' in such systems, using the execution kernel of RC++ as representative example. We believe it is important to inform other game developers of our experience working with rule-based languages, and present some of the difficulties of the approach in a production setting.

The organisation of the paper is as follows. First, in the major section of this paper, we describe the advantageous properties of RETE*, the RETE variant at the heart of RC++, and present theoretical and empirical investigations of its performance. Second, we discuss the implications for deploying rule-based languages in game production.

1 RETE

The RETE algorithm is a network-based algorithm designed to speed the matching of patterns with data. The original RETE was developed by Forgy, and is the execution kernel of the rule-based language, OPS5 (Forgy, 81; Forgy, 82), which is the language basis of both the SOAR architecture and the RC++ language. In rule-based languages, patterns constitute rule conditions. Rules are executed, or "fired", if program data, or "working memory" (WM), can be consistently matched with rule conditions. The RETE algorithm decides rule firing. RETE employs a static discrimination network, generated by the language compiler, that represents data dependencies between rule conditions. RETE avoids unnecessary recalculation of condition matches during addition of data to WM by storing intermediate matches at network junctures called beta-memories. Addition and deletion of data to and from WM is symmetric in RETE: the sequence of operations to delete data is the same as those to add data. RETE trades space for time: results of matching are incrementally cached in memory for subsequent re-use.

The RETE algorithm has compile-time and run-time parts. At compile-time the left-hand sides (LHS) of rules are compiled to a discrimination network represented by an op-code language. The RETE network is a data-flow network, and represents data dependencies between rule conditions. At run-time data items representing changes to the contents of WM, called "change

[1] This work was undertaken at Sony Computer Entertainment Europe's Team Soho game development studio in London as part of an effort to develop new AI technologies for game production.

tokens" (ctokens), enter the network at the root and are processed through the network. Ctokens represent either additions to WM or deletions from WM. The network contains two types of node: test nodes and join nodes, and two types of associated run-time memories: alpha-memories that store atomic "data tokens" (dtokens) for input to top-level join nodes, and beta-memories that store complex dtokens (concatenations of atomic tokens) at join nodes. The dtokens stored in alpha-memories correspond to the contents of WM. Test nodes test values of ctokens, whereas join nodes compare values between an entrant ctoken and existing dtokens.

For example, consider the RC++ rule:

```
RULE PickupObject
       [Goal   ^Status   active   ^Type   holds
^Object ?w]
       [Object ^Name ?w]
       NOT [Monkey ^Holds ?w]
->
       ADD [Goal ^Status active ^Type pickup
^Object ?w]
ENDRULE
```

The rule specifies three conditions, two that check the existence of data in WM, and a NOT condition that checks for non-existence of data in WM. The conditions compile to the RETE network shown in figure 1.

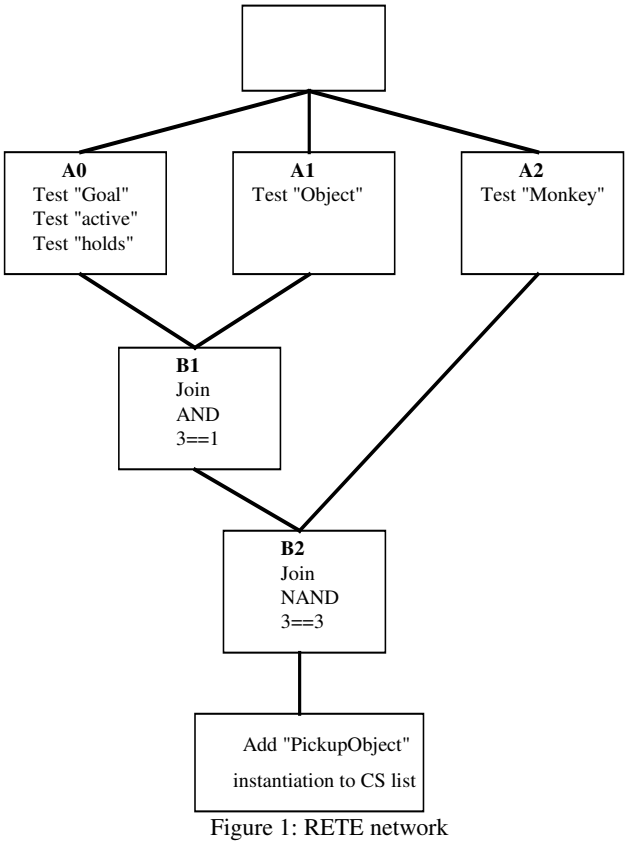

Figure 1: RETE network

The first layer of the network is the set of test nodes that determine whether a ctoken partially matches one of the rule conditions (nodes A_0, A_1, and A_2). For example, the ctoken:

```
+[Goal active holds ladder]
```

represents the addition of a data item to WM. It enters the network at the root and is processed in a depth-first manner through the network. The token passes the first test corresponding to the first rule condition at A_0, but fails the other tests. Hence, the token enters the next level of the network as the LHS of the AND join node B_1 only. Node B_1 checks whether there is a shared value between a ctoken entering from the LHS (resp. RHS) of the join and stored dtokens in the RHS (resp. LHS) memory (in general, a join node may check for an arbitrary number of shared values). For example, consider that the alpha-memory associated with node A_1 contains the dtokens:

```
[Object chair]
[Object ladder]
```

The join is specified as (AND 3==1) (see figure 1), which means that the third attribute of a LHS token must match the first attribute of a RHS token. Therefore, the ctoken

```
+[Goal active holds ladder]
```

from the LHS will join with the second item of the alpha-memory. The join corresponds to the shared rule variable "w" that appears in the first and second conditions of the example RC++ code. The matching LHS and RHS tokens are joined to form a complex token; in this case:

```
[[Goal active holds ladder]
[Object ladder]]
```

This token is stored as a dtoken in the join's beta-memory and a copy passed as a positive ctoken to the next level of the network.

Join nodes incrementally compute matches between the memories on their input edges and store the results as complex tokens in beta-memories. The number of matches between a new entering token and the existing stored state on the opposite input edge determines the number of new complex tokens. In this example, the resulting complex token is stored in beta-memory, β_1. Here it will stay until either of the constituent atomic tokens are deleted from the network. If new tokens enter the RHS of node B_2 the stored state may be repeatedly reused, avoiding the need to recalculate the join. Figure 2 shows the position of alpha and beta-memories within the network and their contents.

The resulting ctoken copy must now be processed through the network. It enters the LHS of join node B_2 in figure 1. B_2 is a NAND node that implements a check for non-existence. If a match is not found between an entrant LHS (resp. RHS) token and existing RHS (resp. LHS) dtokens in the associated alpha or beta-memory then the entrant token is allowed to pass through the NAND join. Otherwise, token progress through the network is halted. In this example, there is no match between the new complex token entering on the LHS and:

```
[Monkey chair]
```

stored in alpha-memory, α_1 (see figure 2). The token therefore

passes all the discrimination tests in the network, which means that all rule conditions are satisfied by this particular combination of items in WM. The combination is represented by the complex token, now called a "rule instantiation", in which the rule variable "w" is bound with the value "ladder". The instantiation is added to the conflict set (CS) (the set of rules scheduled to fire) for execution of the action part of the rule. Execution of rule actions creates new tokens to be processed through the network that may add, delete or modify WM. The RETE processing cycle repeats until no more rules fire and the program terminates.

Deletion is cost symmetric to addition in RETE. For example, the negative ctoken:

```
-[Goal active holds ladder]
```

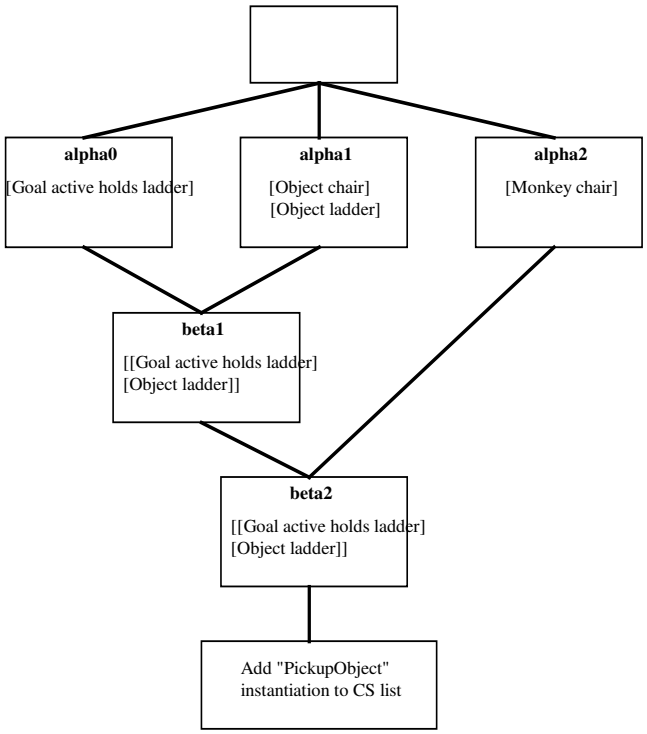

Figure 2: Alpha and beta-memories

represents a deletion of a data item from WM. Quoting Forgy (1982): "The tag in a token indicates how the state information is to be changed when the token is processed. The + and - tokens are processed identically except: (i) The terminal nodes use the tags to determine whether to add an instantiation to the conflict set or to remove an existing instantiation. When a + token is processed, information is added; when a - token is processed, an instantiation is removed. (ii) The two-input nodes [join nodes] use the tags to determine how to modify their internal memories. When a + token is processed, it is stored in the internal memory; when a - token is processed, a token with an identical data part is deleted. (iii) The two-input nodes use the tags to determine the appropriate tags for the tokens they build. When a new output is created, it is given the tag of the token that just arrived at the two-input node." Negative token processing removes all stored state that is dependent on the existence of the data part of the token. A complete description of the RETE algorithm is presented by Forgy (1982).

2 TREAT

The TREAT algorithm (Miranker, 89) also decides rule firing. However, TREAT does not use a discrimination network and does not cache intermediate results at network junctures; instead, pattern matches are recalculated as required. The only stored state is WM and the conflict set (CS). Deletion is inexpensive in TREAT, compared to RETE, because the same sequence of operations that occurred during an addition need not be performed during deletion. TREAT is addition/deletion asymmetric. The extra time cost incurred during addition, compared to RETE, due to the lack of precalculated intermediate matches, is offset by the time saved when data is deleted from WM. TREAT trades time for space: matching results are not cached for subsequent re-use, and hence the memory cost of TREAT is significantly less than RETE. The additional claim for TREAT is that for most rule programs faster execution can be achieved by not maintaining beta-memories and avoiding the associated expensive deletion (i.e., it is possible that TREAT may be superior in both time and space). A TREAT extension, called LEAPS, is the execution kernel of VENUS (Browne et al., 94). LEAPS has better space complexity characteristics compared to both RETE and TREAT (Miranker et al., 90) and is therefore suited to rule firing on very large databases. LEAPS does not fully enumerate the whole conflict set but instead processes a single rule instantiation per cycle.

The development of the TREAT algorithm was motivated by three observations on RETE: (i) beta-memories redundantly store the same state (e.g., see β_1 and β_2 in figure 2), (ii) the CS contains much of the information stored in beta-memories, albeit in an unstructured list, (iii) deletion is expensive due to the need to remove state stored in beta-memories. TREAT maintains alpha-memories but does not maintain beta-memories, and deletion, wherever possible, is processed by direct examination of the CS for removal of invalidated rule instantiations. In consequence, addition of a token requires full computation of all joins that would otherwise have been cached in RETE beta-memories. Deletion, however, requires search of the CS for complex tokens representing rule instantiations that contain the data item to be deleted. If a match is found the instantiation is directly removed from the CS. This is a much less costly process than deletion in RETE, under reasonable assumptions, such as a relatively small CS size.

If rule conditions were only positive, the TREAT algorithm would be relatively simple; however, complications arise due to the presence of negative conditions that test for non-existence. A new positive token may match a negative condition and potentially result in the withdrawal of rule instantiations from the CS that depend on the non-existence of the new token. However, in this case, the CS cannot be directly searched to remove instantiations because complex tokens only represent the presence of data items, not the absence of data items. Therefore, TREAT temporarily considers negative conditions to be positive, and uses the token to build new instantiations, which then may be matched against instantiations in the CS. If a match occurs the instantiation is removed.

A new negative token may match a negative condition and potentially result in the addition of new rule instantiations to the CS (i.e., a non-existence condition becomes satisfied through the removal of a data item). In this case, TREAT cannot search the CS directly, and must also recompute intermediate beta-memories to

add new instantiations.

This summary has ignored implementation details; however, the main characteristics of TREAT are the lack of beta-memories, and use of the CS to directly remove instantiations. A complete description of the TREAT algorithm is presented by Miranker (1989).

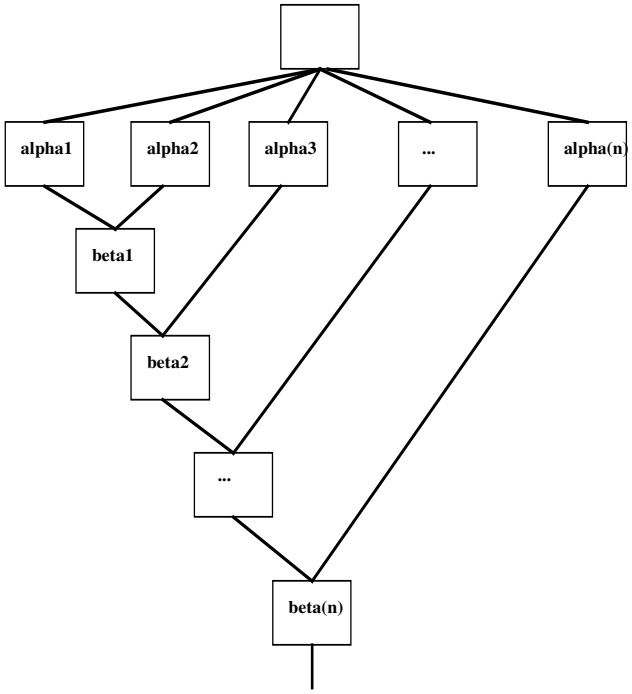

Figure 3: General and simplified RETE network

3 A Model of TREAT and RETE

Miranker (1989) provides a simple mathematical model of join processing in order to compare RETE and TREAT processing costs. That analysis is now extended and modified to more clearly understand the differences between RETE and TREAT, and prepare for the analysis of RETE*. The following simplifying assumptions are made: (i) only positive conditions are considered, (ii) only a single rule is considered (i.e., various optimisations due to test and join sharing between rules are ignored), (iii) constant matching probabilities approximate the highly variable and dynamic result of joins, (iv) an atomic ctoken enters the network at a single point, and hence the corresponding dtoken is stored in one and only one alpha-memory, (v) searching a memory to delete a token costs the same as searching a memory in order to compare and generate a new complex token, (vi) CS search costs are ignored, and (vii) each discrimination network is of the form shown in figure 3. Many of these assumptions are invalidated in practice; despite this, the resulting mathematical model remains useful for approximate reasoning about the relative performance of the algorithms.

3.1 TREAT Model

TREAT token addition requires full computation of all relevant beta-memories (which, of course, are temporarily stored in a scratch pad, not stored between processing cycles and not associated with nodes in a network). The order of join consideration in TREAT may be dynamic, as there is no compile-time discrimination network to impose a static ordering on joins; in consequence, the alpha-memory a token first enters is irrelevant to the analysis. Assume a ctoken enters at α. It must be compared against the contents of α_1 (see figure 3). K_1, the number of comparisons required in a network with two alpha-memories is:

$$K_1 = a_1$$

where a1is the size of α_1. If a network has three alpha-memories, TREAT must form β_1 and compare its contents against α_2. The size of β_1 depends on a_0 and a_1, and the probability that a token from the LHS memory will match a token in the RHS memory, which depends on the type of join and the contents of WM. Assume constant matching probabilities and denote the probability that a token from β_{n-1} will match a token from α_n as p_n (where n>0 and $\beta_0=\alpha_0$). Therefore, the size of β_1 is:

$$B_1 = 1 p_1 a_1$$

The number of comparisons in a network of three alpha-memories is:

$$K_2 = K_1 + B_1 a_2$$

that is:

$$K_2 = K_1 + a_1 p_1 a_2$$

Hence, by induction, the recurrence relation, n = 2, 3, ...

$$K_1 = a_1$$
$$K_n = K_{n-1} + a_n \prod_{i=1}^{n-1} a_i p_i$$

which can be rewritten as the function:

$$K(n) = \begin{cases} a_1 & n = 1 \\ \sum_{i=1}^{n-1} a_{j+1} \prod_{j=0}^{i-1} a_{j+1} p_{j+1} + a_1 & otherwise \end{cases}$$

where n>0 and p_0=0. Deletion of a token in TREAT requires removal of the dtoken from the correct alpha-memory. Using assumption (iv), average cost of deletion in TREAT is:

$$D(n) = \frac{1}{n+1} \sum_{i=0}^{n} a_i$$

The memory cost of TREAT is the total size of the alpha-memories:

$$M(n) = \sum_{i=0}^{n} a_i$$

However, this is an underestimate, as it ignores the CS, and other temporary state.

3.2 RETE Model

RETE has beta-memories and therefore token entry point is important for determining addition and deletion costs. The analysis begins by determining the size of β_1.

$$B_1 = a_0 a_1 p_1$$

The size of β_2 depends on the size of β_1 and the probability of matching with tokens in α_2 (see figure 3).

$$B_2 = B_1 a_2 p_2$$

and in general:

$$B(n) = a_0 \prod_{j=1}^{n} a_j p_j$$

If a ctoken enters at α_0 it must be compared against the contents of α_1. Therefore, $R_{m,n}$, the number of comparisons required for a ctoken entering the mth alpha-memory in a network with n+1 alpha-memories, is:

$$R_{0,n} = K(n)$$
$$R_{1,n} = K(n)$$

because RETE is identical to TREAT if a token enters the network at the first or second alpha-memory. The number of comparisons required when adding a token at α_2 is:

$$R_{2,n} = B(1) + B(1) p_2 a_3 + B(1) p_2 a_3 p_3 a_4 + \ldots$$
$$+ B(1) p_2 a_n \prod_{i=3}^{n=1} a_i p_i$$

This equation requires some explanation. First, the ctoken enters α_2. It must be compared against the contents of β_1 in order to generate new complex dtoken additions to β_2: this cost accounts for the first term. Second, new additions to β_2 must each be compared to the contents of α_3. The new additions to β_2 are $B(1)p_2$ because one new token entered α_2, and the probability of a match between the contents of β_1 and α_2 is p_2. This explains the second term of the equation. Similarly, any new additions to β_3 must be compared against α_4, and so forth. The last term expresses the general pattern. Substituting for B(1):

$$R_{2,n} = a_0 a_1 p_1 + a_0 a_1 p_1 p_2 a_3 +$$
$$+ a_0 a_1 p_1 p_2 a_3 p_3 a_4 + \cdots + a_0 a_1 p_1 p_2 a_n \prod_{i=3}^{n-1} a_i p_i$$

By induction and rewriting as a function:

$$R(m,n) = \begin{cases} K(n) & m \le 1 \\ \sum_{i=m}^{n} \frac{a_0 a_i}{a_m} \prod_{j=1}^{i-1} a_j p_j & otherwise \end{cases}$$

Unlike Miranker (1989), however, the analysis prohibits a token from entering more than one alpha-memory (this would violate assumption (iv), also used in the TREAT model).

The function R(m,n) demonstrates that the presence of beta-memories does not reduce processing costs when a token enters a discrimination network at $\alpha 0$ or $\alpha 1$. In this case, cached state is unused. However, if a token enters the network at a "later" entry point the presence of pre-calculated matches in beta-memories helps to avoid an exponential increase in the number of comparisons required. However, ignoring worst-case CS size, RETE has greater memory cost, which is the sum of the size of alpha and beta-memories.

$$N(n) = \sum_{i=1}^{n} B(i) + \sum_{i=0}^{n} a_i$$

where N(n) is the memory cost of a RETE network with n+1 alpha-memories. Conditional on matching probabilities, beta-memories may, in general, increase exponentially with network size. Note that N(n) is an underestimate for true RETE memory costs because it assumes that complex dtokens consume the same space as atomic dtokens.

To compare RETE with TREAT assume that the cost of RETE token addition is the average cost of adding the token to each alpha-memory. In practice tokens exhibit a skewed entry point distribution.

$$R(n) = \frac{1}{n+1} \sum_{i=0}^{n} R(i,n)$$

Miranker assumes RETE deletion costs the same as RETE addition. In fact, as implied in (Forgy, 82), RETE deletion costs more. An entrant negative token at a join node causes: (i) matching of the negative token against the opposite memory at the join node to generate new, negative complex tokens (a process identical to that during addition), and (ii) a search of the current memory to remove dtokens that match the negative token (extra operations that do not occur during addition). In consequence, the comparison between TREAT and RETE can be made more advantageous to TREAT if the additional RETE deletion costs are included. The cost of the additional search of beta-memories is:

$$S(m,n) = \sum_{i=m}^{n} a_0 \prod_{j=1}^{i} a_i p_j$$

And, again, averaging over entry points:

$$S(n) = \frac{1}{n+1} \sum_{i=0}^{n} S(i,n)$$

hence, cost of RETE deletion is:

$$R(n) + S(n)$$

Table 1 summarises the equations that model TREAT and RETE.

	TREAT	RETE
Cost of adding a token	K(n)	R(n)
Cost of deleting a token	D(n)	R(n)+S(n)
Memory cost	M(n)	N(n)

Table 1: Equations that model TREAT and RETE time and space costs in a network of n+1 alpha-memories

3.3 A Crossover Point

Miranker (1989) quotes values of a=25.6 and p=0.039 for the average size of alpha-memories and the average matching probability at joins respectively for typical rule programs. Obviously, rule programs exhibit a wide range of alpha-memory sizes and matching probabilities; therefore, the following analysis can only be indicative.

Figure 4 shows RETE and TREAT time costs on arbitrary data set conforming to an 'average'rule-based program. The data set was randomly initialised to conform to the quoted averages. The cost of adding and deleting is summed and shown on the y-axis. In consequence, a further assumption is introduced: the number of additions and deletions exactly balance during a program run. The number of alpha-memories, shown on the x-axis, represents a measure of rule complexity. A greater number of alpha-memories correspond to a greater number of rule conditions. As expected, TREAT requires less comparisons when the presence of RETE beta-memories makes little difference during addition (i.e., rules with very few conditions, n=1, n=2). However, RETE "catches up" with TREAT: a crossover point is reached as network complexity increases. After the crossover point the deletion savings avoided by TREAT are more than offset by the additional costs of not maintaining beta-memories. The results suggest that TREAT has better average run-time performance than RETE when rules require less than 6-7 alpha-memories on average. The claim and justification for TREAT is that most rule programs conform to this constraint. A superior analysis would investigate a large sample of randomly generated data sets to investigate the distribution of crossover points.

Nayak et. al. (1988) present empirical results comparing RETE and TREAT on SOAR (Laird et. al., 87) production rules (SOAR is based on the OPS5 production system), with an average of 9 conditions per rule. Their results show that, "while RETE seems to be better than TREAT in most cases, there are some situations under which TREAT is comparable to RETE and may even be better." These empirical results conform to the derivation of a RETE/TREAT crossover point. It is generally thought that (i) rules with few conditions, combined with (ii) a volatile WM, are favourable conditions for TREAT style processing. The model explains (i) but fails to explain (ii) due to the restrictive assumption of constant matching probabilities.

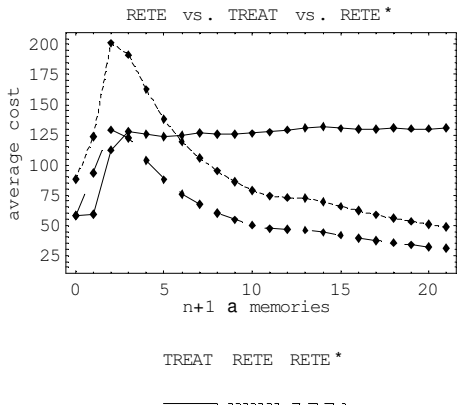

Figure 4: Indicative comparison of RETE (2R(n)+S(n)), TREAT (K(n)+D(n)), and RETE* (R(n)+S(n)), and the RETE/TREAT crossover point

4 RETE*

The RETE* algorithm is the execution kernel of RC++ (Wright and Marshall, 00), a rule-based extension to C++. RETE* maintains beta-memories; however, the RETE* algorithm is addition/deletion asymmetric: deletion costs less than addition, although deletion, in general, is more expensive than TREAT deletion. RETE* is also slightly faster than RETE on addition due to the presence of "dual tokens". RETE* employs a dynamic beta-memory cut that maintains a fixed upper-bound on total run-time beta-memory size. If the upper-bound is specified as zero, RETE* functions as a flavour of the TREAT algorithm. Hence, TREAT is a special case of RETE*. Users may explicitly control the time/space trade-off as exhibited by the two extremes of RETE (maintain all beta-memories) and TREAT (maintain no beta-memories). The '*' in RETE* denotes that the memory allocated to beta-memories ranges from 0 to n bytes.

RETE* aims to speedup standard RETE and allow users to exploit the RETE/TREAT crossover point. RETE* employs three main mechanisms to lower the time and space costs of RETE: (i) asymmetric deletion, (ii) dual tokens, and (iii) dynamic beta cut. Each is described in turn.

4.1 Asymmetric deletion

RETE is inefficient on deletion. Consider that a negative ctoken enters the LHS of an AND join node B_n. RETE performs two computations: (i) the ctoken is compared against the contents of α_n, and a complex negative ctoken is formed for each possible join; and (ii) the beta-memory, β_n, is searched for any matches between the new complex negative ctokens and existing stored complex dtokens; if a match is found, the stored state is deleted. Each newly generated ctoken is passed to the next level of the network, node B_{n+1}, and the deletion process continues. For example, if n new ctokens are generated there are n depth-first traversals of the sub-network below the join.

RETE* deletion is based on the observation that computation (i) is

almost always redundant. To understand why, consider that the negative ctoken entering a network fragment is:

```
-[Goal active holds ladder]
```

And the dtokens in β_n are:

```
[[Goal active holds ladder]
[Object red ladder]]

[[Goal active holds ladder]
[Object yellow ladder]]
```

That is, β_n contains two complex dtokens, each of which contains the atomic token to be deleted. Consider also that α_n contains:

```
[Object red ladder]
[Object yellow ladder]
```

and join at node B_n is between the third attribute of the LHS input and the second attribute of the RHS (AND 3==2). Standard RETE deletion proceeds by computing the join and generating two negative complex ctokens:

```
-[[Goal active holds ladder]
[Object red ladder]]

-[[Goal active holds ladder]
[Object yellow ladder]]
```

Each ctoken is then compared against the contents of β_n and if a match is found the stored state is deleted.

In contrast, RETE* does not form the complex negative tokens, does not compute the join, and therefore avoids multiple traversals of the subnetwork. Instead, memory β_n is directly searched. If the negative token partially matches stored state, the stored state is deleted. For example the negative ctoken:

```
-[Goal active holds ladder]
```

partially matches both complex dtokens in β_n and therefore both dtokens are deleted. The result is identical to RETE deletion, except the costly join computation is avoided. In consequence, negative complex tokens are not passed to the next level of the network; instead, the original negative ctoken is passed, and the same method may then be used to delete state from subsequent beta-memories. If no match is found between the negative token and contents of a beta-memory the deletion process backtracks to the last unprocessed edge in the network as normal.

NAND nodes introduce exceptions that slightly complicate the RETE* deletion scheme. NAND nodes check for non-existence. The case of a negative token entering a NAND node introduces the complication. When a negative token enters from the LHS of a NAND node the RETE* deletion scheme may operate as described. However, if a negative token enters from the RHS then the removal of a data item could satisfy a non-existence condition, generate new complex tokens, and potentially result in new instantiations in the CS. Hence, RETE* deletion operates as RETE deletion for this case and the join is computed. Table 2 summarises the situation.

Negative token entry point	New instantiations in CS possible?	Deletion scheme
LHS AND	No	Direct deletion (RETE*)
RHS AND	No	Direct deletion (RETE*)
LHS NAND	No	Direct deletion (RETE*)
RHS NAND	Yes	Join computed and complex negative tokens generated (RETE)

Table 2: RETE* deletion cases

Therefore, in three out of the four possible deletion cases RETE* avoids join computation during deletion. Under the assumptions of the model, deletion in RETE* does not cost:

$$R(n) + S(n)$$

but

$$S(n)$$

In consequence, RETE* has better overall time performance than RETE and hence the crossover point between RETE* and TREAT is reached in smaller size networks RETE* performance is always better than RETE performance due to cheaper deletion costs.

Scales (1986) first introduced the idea of asymmetric deletion to optimise RETE (normally called 'deletion optimisation'). However, there are two differences between Scales` deletion scheme and the RETE* deletion scheme. First, Scales' scheme does not extend to NAND nodes (i.e., the LHS NAND case is processed as a normal RETE deletion). Second, although join computation is avoided, copies of deleted complex dtokens are passed as ctokens to the next level of the network. Therefore, if a negative ctoken causes direct deletion of n dtokens then n new ctokens are generated requiring n traversals of the sub-network. But the generation of multiple complex ctokens is redundant. In contrast, RETE* deletion passes only the original ctoken to the sub-network and thereby avoids multiple traversals. The original ctoken contains sufficient information to match against parts of invalidated dtokens in later beta-memories.

Scales reports a run-time speedup of RETE processing of 28% on the SOAR 'eight puzzle' program using the inefficient version of asymmetric deletion. Asymmetric deletion represents a significant time saving over the original RETE algorithm.

4.2 Dual Tokens to Avoid Join Computation During Addition

A negative token entering a NAND node from the RHS is a special case for deletion. A further observation is that a positive token entering a NAND node from the RHS is also a special case for addition. In this case the only possible effect is to remove dtokens from beta-memories and potentially withdraw instantiations from the CS; in all other cases of positive token processing the result is

to add new state to the network. Consider that the RHS input to a NAND node is the positive token:

```
+[Object yellow ladder]
```

The LHS input memory of the NAND node is:

```
[Monkey chair]
[Monkey ladder]
```

and the NAND tests whether the first attribute of LHS token is the same as the second attribute of a RHS token (NAND 1==2). If a token entering from the LHS matches with RHS state then the token will not drop through to the next level of the network.

If a token enters from the RHS and matches with LHS state then RETE forms the complex negative ctoken:

```
-[[Object yellow ladder]
[Monkey ladder]]
```

(More precisely RETE generates a negative complex token if the 'match count' of the matching LHS token increases from 0 to 1. Otherwise, the state in subsequent beta-memories has already been deleted by a previous token that matched the same LHS token. For more details see (Forgy, 82).) The current beta-memory is searched and matching dtokens removed. The ctoken drops through to the next level of the network and deletes any subsequent stored state. In other words, a positive token entering the RHS of a NAND node can act like a deletion.

Positive token entry point	New instantiations in CS possible?	Addition scheme
LHS AND	Yes	Join computed and complex positive tokens generated (RETE)
RHS AND	Yes	(as above)
LHS NAND	Yes	(as above)
RHS NAND	No	Direct deletion via dual tokens (RETE*)

Table 3: RETE* addition cases

RETE* deletion is faster than RETE deletion because, in most cases, join computation can be avoided. Similarly, it is possible to avoid join computation for the special case when addition acts like deletion. To do so requires the generation of dual tokens. Dual tokens are now described followed by an explanation of how they are used to avoid join computation during token addition.

Dual tokens represent the non-existence of data items in WM, unlike normal tokens that represent existence. NAND joins generate dual tokens when a LHS token drops through to the next level of the network. For example, consider that the LHS input to a NAND is:

```
+[Monkey ladder]
```

and the NAND join is as before (NAND 1==2). Assume also that the RHS memory is empty; therefore, no matches can occur, the non-existence condition is satisfied, and the token may pass. RETE* appends a dual token to the positive token, creating a complex token with a dual part. The dual token represents the fact that there was not a data item in the RHS that had the value "ladder" for its second attribute:

```
+[[Monkey ladder]
-[Object 2 ladder]]
```

The dual part of the complex token is highlighted in bold. Note that dual tokens are partially specified: they do not completely specify attribute values (i.e., the only values specified are those that did not exist in order for the non-dual part of the token to pass the NAND test). As a further example, consider that the LHS input to a NAND is:

+[Monkey ladder red heavy]

and the NAND join is:

NAND
1,1==1,2
1,2==1,1
1,3==2,3

Where "(a,b) == (c,d)" is a join between the bth attribute of the ath atomic token of a LHS complex token with the dth attribute of the cth atomic token of a RHS complex token. Generated dual tokens always contain the same number of atomic tokens as those in the RHS input memory. In this example, the resulting ctoken with dual part is:

```
+[[Monkey ladder red heavy]
-[[Object 2 ladder 1 red]
[Object 3 heavy]]]
```

Hence, rules with NOT conditions will generate instantiations in the CS that have trailing dual tokens that represent the non-existence of data items in WM.

The class name of dual tokens cannot be obtained from dtokens in the RHS input memory because there are circumstances when that memory will be empty. Therefore, the RC++ compiler assigns node indices to each node, unique for every NAND join, zero otherwise, and constructs a class name table, indexed by the node index, which contains a list of the class names that may reside in the RHS input memory of the particular NAND node. Dual tokens may therefore be constructed even when a RHS input memory is empty.

Dual tokens are used to implement direct deletion in the case that a positive ctoken enters the RHS of a NAND node. In this case, instead of computing the NAND join the beta-memory is directly searched for any matches between the positive ctoken and dual tokens within dtokens; if a match is found the dtoken is deleted. The positive ctoken is then passed to the sub-network and processing continues. Any subsequent dtokens with matching duals are also deleted. Table 3 summarises the RETE* addition cases. In only one out of the four possible addition cases can RETE* avoid join computation during addition (the mirror image of RETE* deletion). In consequence, RETE* addition will always be slightly

faster than RETE addition, assuming that the cost of dual token creation is offset by the avoidance of some join computations. However, the speedup is achieved at the expense of extra storage of dual tokens. The theoretical model of RETE processing does not model NAND nodes; however, it is reasonable to assume that RETE* addition savings will be significant in networks with a high number of NAND joins with frequent RHS positive token entry. Section 5.1 presents empirical results for a program called "manners" that confirms this conjecture.

4.3 Dynamic Beta Cut

Ignoring CS storage RETE has worst-case space complexity of $O(W^{n-1})$, where W is the number of data items in working memory, and n is the number of rule conditions, whereas TREAT has space complexity of O(W). RETE has exponential space complexity, whereas TREAT has linear space complexity. The space requirements of RETE are a hindrance to its use in large databases (Miranker et al., 90) and embedded systems with low memory availability, such as home entertainment systems. To alleviate this problem, RETE* allows an upper bound on beta-memory consumption to be specified by language users. Beta-memories are discarded and retained at run-time depending on the current memory consumption. If a beta-memory is absent during token processing but is required to compute a join, RETE* recalculates the beta-memory. If the recalculation itself depends on prior joins with absent beta-memories, RETE* also recalculates, popping back up the network until either a stored beta-memory is found, or the alpha-memory layer is reached. Hence, users have control over the trade-off between speed of execution and memory consumption. Further, if the upper bound is set to zero, no beta-memories are stored, and RETE* functions as a flavour of the TREAT algorithm: every new token entering the network results in full join computation (with static join ordering). In addition, when tokens are deleted from the network RETE* need not search absent beta-memories, or compute joins; instead, the negative token is matched against the contents of the CS and instantiations directly removed. Therefore, in some cases, RETE*, with a low upper bound on beta-memory size, will benefit from the inexpensive deletion associated with TREAT. However, a guaranteed upper memory limit cannot be specified for total run-time memory consumption because alpha-memories are necessary to represent the contents of WM. Therefore as more tokens are added to a network, the size of RETE* alpha-memories continues to increase. In addition, pathological scenarios may occur when the CS becomes very large, which itself has worst-case space complexity $O(W^{n-1})$. Therefore, in order to fully tackle the combinatorial space problems associated with rule languages it will be necessary to investigate further space saving techniques, such as avoidance of full enumeration of the CS (Miranker et al., 90).

Pseudo-code for the beta-memory deletion algorithm is as follows:

1. while memory consumption exceeds memory upper limit and undeleted beta-memories exist
    1.1 find beta-memory with lowest 'order'
    1.2 delete contents of beta-memory
    1.3 set beta-memory valid flag to false
2. end while

The algorithm requires an ordering function that assigns a numerical value to each beta-memory. The memory with the lowest order is deleted first. The abstract order may be implemented in different ways to support different deletion schemes. For example, our first trial implementation of the beta cut uses a "most recently modified" heuristic that orders according to how recently memories have contributed to rule instantiation computation. This ordering requires a timestamp mechanism to be implemented in the RETE network, such that modifying a beta-memory (adding or removing a token) or performing a read on a beta-memory updates its 'last modified' timestamp.

Deleted beta-memories are recalculated on demand. That is, when RETE* requires the contents of an absent beta-memory, the join between the two associated input nodes is computed and the required beta-memory formed. If one or more of the input memories are also absent, the current recalculation task is pushed, and further recalculations are performed higher in the network. The pseudo-code below describes the process:

1. if beta-memory for comparison is not valid
  1.1 push beta-memory onto recalculation stack
  1.2 while more beta-memories on recalculation stack
    1.2.1 pop beta-memory from recalculation stack
    1.2.2 determine preceding network node through beta-memory's backpointer
    1.2.3 perform full join on left and right input memories and put results in beta-memory
  1.3 end while
2. end if

Entry point, join type, ctoken type	**Recalculate absent LHS beta-memory?**	**Recalculate absent RHS beta-memory?**
LHS, AND, +	N	Y
LHS, NAND, +	N	Y
LHS, AND, -	N (token drops through)	N
LHS, NAND,-	N (token drops through)	N
RHS, AND, +	Y	N
RHS, NAND, +	N	N (token drops through)
RHS, AND, -	N	N (token drops through)
RHS, NAND, -	Y	Y

Table 4: Recalculation cases for beta-memories

The stack-based approach allows backward chaining through invalidated beta-memories to the original alpha-memories. Recovered beta-memories are only deleted if the upper memory limit has been exceeded and once the originally absent beta-

memory is no longer required. Therefore, temporary "scratch pad" memory requirements may exceed the specified upper bound. If temporary memory requirements are still too high for the application the upper bound may be further decreased. Demand-driven recalculation of beta-memories requires the introduction to the RETE network of "validity" flags for beta-memories and backpointers from join nodes to input nodes.

Table 4 describes all cases for ctoken processing when an absent beta-memory is encountered. Special action must be taken in some cases. Cases where the ctoken can immediately drop through are indicated (inexpensive TREAT style deletion). Entry of a negative ctoken at the RHS of a NAND node requires recalculation of both the right beta-memory and the left beta-memory (if both absent), in order to calculate match counts (see footnote 1).

Implementation of the beta cut with a recency heuristic requires very little run-time overhead (validity flag checking, and timestamp updates), and some extra information from the compiler (node depths and node backpointer offsets). The result is a rule-based execution kernel that allows users to trade time for space according to the application constraints. However, like any memory paging mechanism (the beta-memories are "paged" in and out of memory) there is the possibility of pathological scenarios where the same sets of beta-memories are repeatedly re-deleted and re-stored over the run of a program.

5 EMPIRICAL RESULTS

The preceding analysis predicted that the performance of RETE and TREAT is problem dependent: some rule programs will run faster under RETE than TREAT and vice-versa. In addition, we expect that RETE* should perform better than RETE under all circumstances, and that RETE*(0) (RETE* with 0 bytes allocated to beta-memories), which is a flavour of TREAT, will perform better than RETE and RETE* on some rule programs. The results presented below confirm these expectations.

5.1 Time Results

	RETE	RETE*	RETE*(0)
Manners16	0.4072	**0.1008**	0.1028
Manners32	7.1504	1.7148	**1.6808**
Manners64	480.38	68.77	**65.03**
SimMatches16	0.0776	**0.07**	0.2556
SimMatches32	4.7084	**3.058**	10.5228
DCGS	0.1049	**0.0699**	0.1202

Table 5. Time results in seconds averaged over 25 runs

The ‘Manners’ rule program is a standard benchmark program that plans acceptable seating arrangements at a dinner party. This is a combinatorial problem: the more guests the more expensive the computation. RETE* is fastest on Manners16 (average 0.1008 seconds), followed closely by RETE*(0) (average 0.1028 seconds). RETE is much slower (average 0.4072 seconds). For Manners32 and Manners64, however, RETE*(0) is slightly faster than RETE*. ‘Manners’ contains 8 rules with a high proportion of RHS deletion commands. Therefore, RETE* with asymmetric deletion performs considerably better than standard RETE (on average about 4 times faster). In addition, RETE*(0) performs slightly better than RETE*, suggesting that ‘Manners’ is best suited to TREAT-style execution.

The ‘SimulateMatches’ rule program simulates soccer teams playing against each other in competitive leagues. Again, this is a combinatorial problem: the more teams in a league, the more matches to play. RETE* is fastest on SimulateMatches32 (average 3.058 seconds), followed by RETE (average 4.7084 seconds). RETE*(0) is the slowest at 10.5228 seconds (about 3.5 times slower than RETE*). Similar results hold for SimulateMatches16. ‘SimulateMatches’ contains a number of ‘setup’ rules that create records to store the results of games, and then ‘processing’ rules that perform the actual match result simulation. RETE* and RETE benefit from the priming of beta-memories with partial joins. RETE*(0) must recompute such partial joins, which explains the poor performance. The different results for ‘Manners’ and ‘SimulateMatches’ highlight the problem-dependent performance of rule-based execution kernels.

The DCGS program (Directed Cyclic Graph Search) consists of three rules that perform a depth-first search of a directed cyclic graph in order to find a route from a start location to a goal location. Again, RETE* is the fastest execution method (average 0.0699 seconds), followed by RETE (average 0.1049 seconds). RETE*(0) is again slowest at 0.1202 seconds (1.72 times slower than RETE*). The presence of RETE* and RETE beta-memories outweighs the associated deletion costs for the DCGS program. The results are summarised in table 6.

	Manners (16,32,64)	**SimMatches (16,32)**	**DCGS**
RETE	4.03 to 7.38 times slower	1.11 to 1.54 times slower	1.50 times slower
RETE*	0.98 to 1.05 times slower	**Fastest**	**Fastest**
RETE*(0)	**Fastest**	3.44 to 3.65 times slower	1.72 times slower

Table 6: Summary of time results

The empirical results suggest that RETE* is generally faster than standard RETE and RETE*(0). The empirical results demonstrate speed-ups of approximately 9.8% (SimMatches16), 35.0% (SimMatches32), and 33.4% (DCGS), values within the range of theoretical prediction. However, RETE* is between 75.2% and 76.0% faster than RETE on ‘Manners‘, a significant performance improvement not explicable in terms of asymmetric deletion alone. In this case, the presence of RETE* dual tokens speed up the processing of positive tokens that invalidate instantiations, which explains the large performance improvement over RETE processing. However, RETE*(0) is the faster execution method on ‘Manners‘, confirming that TREAT style processing can be faster than RETE on certain rule programs. However, RETE* remains competitive with RETE*(0) on ‘Manners’ due to the presence of dual tokens. RETE* is the faster execution method overall, particularly when it is considered that it may be instantiated as

RETE*(0) when required.

5.2 Space Results

	RETE	RETE*	RETE*(0)
Manners16	114	227	122
Manners32	334	671	254
Manners64	1443	3041	1051
SimMatches16	63	92	55
SimMatches32	133	212	175
DCGS	64	84	51

Table 7: Maximum memory consumption in kilobytes

The space results of running RETE, RETE* and RETE*(0) on the test programs are presented in table 7. The maximum allocated memory space during the run is measured, including temporary 'scratch pad' memory allocated during a RETE*(0) recalculation process. As expected, on 'Manners' RETE*(0) generally has the lower memory costs, whereas RETE* has significantly worse memory costs compared to RETE and RETE*(0). This is due to the extra storage required for dual tokens (which also accounts for the good time performance of RETE*, compared to RETE, on 'Manners'). RETE*(0) is much the better method for 'Manners', as it is the fastest, and also has the lowest memory costs. If unbounded RETE* is too memory expensive for a particular application it is possible to try RETE*(n) or RETE*(0) by simply altering the value of the memory bound parameter.

On DCGS RETE* again has the worse space characteristics compared to RETE. RETE*(0) has the lowest memory costs. The pattern is repeated for 'SimulateMatches', except in one case RETE*(0) has worse memory costs than RETE. How can this be? The reason is that during program execution tokens entered the RHS of the last join node before a TERM node (rule is satisfied) resulting in the full recomputation of all prior missing beta-memories. The implementation of dynamic beta-cut only deallocates beta-memory after the final join is calculated. Hence, for a time, all beta-memories are resident in memory (including dual tokens). Obviously this is a major drawback to the current implementation and needs to be altered. A better implementation of dynamic beta-cut would deallocate during the recalculation process such that the maximum memory cost would equal to the largest beta-memory in the recalculation chain, rather than the size of the all the recalculated beta-memories. In addition, dual tokens would not be generated for RETE*(0), as they play no useful role in this extreme case. An improved implementation of RETE*(0) would substantially change the memory costs of RETE*(0) on all test programs, resulting in RETE*(0) having the lower memory costs overall.

5.3 Summary of Results

Experimental results have confirmed theoretical expectations: TREAT style processing is sometimes superior to RETE style processing and vice versa. RETE* is always faster than RETE. As RETE* can be instantiated as RETE*(0), the RETE* algorithm is a better and more flexible method than either RETE or TREAT alone. Dual tokens have associated memory costs but appear to significantly speedup processing in certain cases. Asymmetric deletion results in about a quarter speedup over standard RETE. The current implementation of beta-cut is limited because deallocation of temporary beta-memories does not occur during a recalculation chain. For rule programs that favour TREAT style processing it is possible to use RETE*(0) in order to gain better time and space characteristics.

6 UTILITY-BASED DYNAMIC BETA CUT

Nayak et. al. (1988) discuss the desirability of a RETE/TREAT hybrid that decides whether to use RETE or TREAT style processing according to the nature of the rule program. Fabret at. al. (1993) present an algorithm that decides what beta-memory state is profitable to maintain in a RETE network based on a static, compile-time analysis of the rule program. The Gator discrimination network (Hanson, 93; Hanson et al., 95; Hanson et al., 97) is a generalised network that includes RETE and TREAT-style processing as a special case. A Gator network contains only those beta-memories that contribute to a reduction in processing time. The decision to maintain a beta-memory is based on compile-time, heuristic cost predictions parameterised by statistics collected from run-time database queries. Hanson et al. (1995) report significant speedups over RETE and TREAT with this method, although inaccuracies in cost formulae can result in Gator networks that perform worse than TREAT. Hybrid approaches are the way forward for improving the performance of rule-based languages, but compile-time prediction of run-time beta-memory utility is necessarily approximate and limited because the same rule program can exhibit very different run-time behaviour depending upon the contents of working memory. There simply isn't sufficient information at compile-time to fully determine the best discrimination network.

An alternative is to perform beta-memory maintenance at run-time. The RETE* dynamic cut mechanism can be extended to implement a RETE/TREAT hybrid by ordering beta-memories according to a utility measure that represents their contribution to reducing computation costs. This would allow the RETE/TREAT crossover point to be exploited dynamically at run-time. This more sophisticated RETE* cut mechanism is not required for our current applications but it would be of interest to pursue such an implementation; therefore a possible sketch mechanism is presented.

An individual beta-memory is subject to three events: (i) a positive ctoken is added, (ii) a negative ctoken is added, resulting in a search of the memory to delete matching dtokens, and (iii) the memory is searched to compute a join. The time cost of event (i) is negligible, whereas events (ii) and (iii) cost search time. However, if the beta-memory is absent, the time cost of event (ii) is negligible (there is no state to be removed from the beta-memory), but event (iii) will cost more due to the need to recalculate the state that would have been stored. There are both advantages and disadvantages to storing state in a beta-memory, as the preceding analysis has shown. The utility of retaining a beta-memory depends on the distribution and frequency of these events during the program run. A utility measure dynamically computed based on past events can be heuristically used as a predictor of future utility. The RETE* beta cut mechanism can be modified to cut based not only memory requirements but also on a watershed utility value: beta-memories with sufficient utility are maintained, others are

deleted. Utility can be measured by approximating true costs via a model of beta-memory computation time, or measuring computation time directly by counting clock cycles expended during the different beta-memory events. RETE* with utility-based beta cut would automatically approximate the optimal balance between RETE and TREAT style processing by deleting and undeleting beta-memories during the program run. However, this has yet to be tested and verified. It may be the case that implementation details offset any speedup. In addition, any heuristic predicting future utility on past utility will be incomplete; hence, some rule programs will represent pathological cases that will defeat the speedup mechanism.

The RETE* dynamic beta-cut provides a mechanism for a new exploration of hybrid approaches, in particular flushing and maintaining beta-memories at run-time based on the dynamic properties of the rule program, a process analogous to page caching in virtual memory systems. However, the memory allocation schemes used in RETE* need further refinement before such an investigation can proceed.

7 GENERAL-PURPOSE RULE-BASED LANGUAGES INSUFFICIENT FOR GAME AI

High-level pattern matching helps the AI programmer by removing the need to write code to execute rules. Instead, the programmer only specifies the rules, allowing full concentration on the AI problem. However, the cost of this level of abstraction is that all rule programs are executed by the same kernel, which is general-purpose. Hence, the ability to exploit unique properties of an AI problem for efficiency gains is lost, which is a significant drawback for applications such as real-time computer games that need to maintain frame rates. Further, the semantics of general-purpose pattern-matching easily leads to combinatorial explosions of time and space requirements, even with the use of complex RETE-style speedup algorithms such as RETE*. In essence, rule-based languages give a little with one hand -- the increased level of abstraction -- but take a lot with the other -- the increased computational cost. The trade-off is further biased against rule-based languages because current requirements for game agents usually translate into relatively stateless, reactive programs. In these cases, simpler, dedicated approaches will result in faster code. In the short to medium-term the power of general-purpose pattern-matching is not required and cannot be afforded. If console processing power increases and there is greater need for more sophisticated AI characters then general-purpose rule-based languages may become a more attractive proposition for studio level production.

A further difficulty, however, which is more fundamental, is that of rule construction: the rule-based kernels discussed here enforce an explicit, ‘symbolic’ representation of knowledge. This is appropriate when the AI programmer has a good understanding of the problem domain and is able to explicate the relationship between sensory input and action output. Then the rules will flow. But the rule-based programming approach is of no help when the major difficulty is discovering the very rules themselves. The major problem facing the game AI programmer is not the programming language employed, but the task of constructing control programs that behave appropriately in virtual worlds. Rule-based languages allow models of behaviour to be conveniently expressed as rules, but the problem of rule construction remains. From this perspective, general-purpose rule-based languages are of uncertain help to a minor problem.

REFERENCES

Fabret, F.; M.Regnier; and E.Simon. 1993. "An adaptive algorithm for incremental evaluation of production rules in databases." In *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, 1993.

Forgy, C.L. 1981. "OPS5 user manual." Technical Report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Forgy, C.L. 1982. "Rete: a fast algorithm for the many pattern/many object pattern match problem." *Artificial Intelligence,* vol. 19(1): 17-37.

Hanson, E.N. 1993. "Gator: a discrimination network structure for active database rule condition matching." Technical Report UF-CIS-TR-93-009, CIS Department, University of Florida.

Hanson, E.N.; S.Bodagala; M.Hasan; G.Kulkarni; and J.Rangarajan. 1995. "Optimized rule condition testing in Ariel using Gator networks." Technical Report TR-95-027, CISE Department, University of Florida.

Hanson, E.N.; S.Bodagala; U.Chadaga; M.Hasan; G.Kulkarni; and J.Rangarajan. 1997. "Optimized trigger condition testing in Ariel using Gator networks." Technical Report TR 97-002, CISE Department, University of Florida.

Laird, J.E. and M. van Lent. 1999. "Developing an Artificial Intelligence Engine." In *Proceedings of the Game Developers Conference*, March 16-18, San Jose, CA, 577-588.

Laird, J. E.; N.Allen; and P.S.Rosenbloom. 1987. "SOAR: an architecture for general intelligence." *Artificial Intelligence,* vol. 33(1): 1-64.

Miranker, D. 1989. *TREAT: A new and efficient match algorithm for AI production systems.* Pittman/Morgan Kaufman, 1989.

Miranker, D.P.; D.A.Brant; B.Lofaso; and D.Gadbois. 1990. "On the performance of lazy matching in production systems." In *Proceedings of the Eigth National Conference on Artificial Intelligence*, 685-692.

Nayak, P.; A.Gupta; and P.Rosenbloom. 1988. "Comparison of the Rete and Treat production matches for Soar (a summary)." In *Proceedings of the Seventh National Conference on Artificial Intelligence*, 693-698.

Scales, D.J. 1986. "Efficient matching algorithms for the SOAR/OPS5 production system." Technical report STAN-CS-86-1124, also numbered KSL-86-47, Department of Computer Science, Stanford University, CA 94305.

Browne, J.C.; A.Emerson; G.G.Mohamed; D.P.Miranker; A.Mok; L.Obermeyer; F.Haddix; R.Wang; and S.Chodrow. 1995 "Modularity and rule based programming." *International Journal on Artificial Intelligence Tools*, vol. 4, no. 1&2 (June). A prior version was published in *Proceedings of the Sixth International Conference on Tools with Artificial Intelligence*, November 1994

Wright, I. P. and J.Marshall. 2000. "RC++: a rule-based language for game AI." In *Proceedings of the First International Conference on Intelligent Games and Simulation* (GAME-ON 2000), Nov. 11-12, Imperial College, London.