Historical Rete Networks for Debugging Rule-Based Systems

Sharon M. Tuttle
Department of Computer Science
University of Houston
Houston, TX 77204-3475

Christoph F. Eick
Department of Computer Science
University of Houston
Houston, TX 77204-3475

Abstract

To debug a forward-chaining rule-based program, certain 'historical' information is needed. System builders should be able to directly request such information, instead of having to rerun the program one step at a time or search a trace of run details. As a first step in designing an explanation system for answering such questions, this paper discusses a proposal for storing a forward-chaining program run's 'historical' details in its Rete inference network, used to match rule conditions to working memory, without seriously affecting the network's run-time performance. We call this proposed modified Rete network a historical Rete network. Various algorithms for maintaining this network are discussed, along with how it can be used to analyze what happened during a program run.

1 Introduction

A forward-chaining rule-based program's execution is data-driven: data changes determine what happens next. A rule with satisfied left-hand-side conditions, said to be eligible to fire, is chosen to fire: to have its right-hand-side actions executed. Those actions may change working memory, causing previously unsatisfied rules to be eligible, and vice versa. So, which rule fires determines which rules next have a chance to fire. To debug such a program, detailed information is needed, such as the order of rule firings, and when facts were in working memory. This historical information is needed to understand the program's behavior.

In debugging large forward-chaining programs, system builders with traditional debugging tools have a tedious job ahead. These languages often allow one to run a program one rule-firing at a time, and to see what is in memory or in the agenda, the ordered list of currently-eligible rules. A trace of rule-firings and working memory changes may also be kept. There has been research into easing the debugging of these programs; [5] presents a graphics-based debugger, while [1], [13], [7], and [6] suggest changes to rule-based languages, such as adding rule-sets, that might, among other goals, make debugging easier.

We are interested in using explanation as a tool in debugging these programs. Most explanation research has focused on end-users of backward-chaining systems (for example, see [16], [14]). (However, [5] considers its graphical forward-chaining debugger to be providing graphical explanation.) While discussing explanation's general utility, [3] mentions that it was helpful in debugging MYCIN. MYCIN, however, is a backward-chaining system, with goal-oriented execution, and its explanation was designed accordingly.

Many questions for debugging forward-chaining programs deal with a run's historical details. For example, system builders might ask which fired rule's right-hand-side actions added fact F to working memory, allowing it to trigger some other rule. They might ask which fired rules used F to satisfy their left-hand-side conditions, to determine F's impact, or they might ask why rule R did not fire at time T. These questions can be answered using current tools, but tedium would be reduced if they could be directly answered. To do so, we must somehow store and maintain a run's historical information.

Many forward-chaining systems use inference networks to efficiently match left-hand-side (or LHS) rule conditions to facts. A widely-used algorithm for this is Rete ([9], [15]), developed for OPS5 ([8]) and also used in NASA's CLIPS ([10], [4]). In Rete, a network is built, which includes tests for LHS conditions and for certain combinations of LHS conditions. When a fact matches a condition, an instantiation, showing that this condition is satisfied by this fact, is stored in the network; instantiations are also stored for combinations of conditions satisfied by sets of facts. A rule instantiation is a set of facts that satisfies all of a rule's LHS conditions. When a fact is added to working memory, it is sent through the network, causing rules to become eligible to fire, if its addition causes
instantiations of those rules to be created. Likewise, rule instantiations may be removed, if a rule requires that this fact not be true. When facts are removed from working memory, that is also propagated through the network, causing the removal of instantiations including those facts. It might be feasible to "tag" these instantiations with when they occurred, as they occur.

Research involving Rete has focused on improving its performance, as in [12]; however, using it to store temporal information as suggested above is unexplored. This paper proposes a generalization of the Rete network, called a historical Rete network, that can store and maintain a run's historical information. It should run almost as quickly as regular Rete, so that run-time operation during program development is not overly impeded. Adding a temporal dimension to Rete in this way will allow us to store and maintain historical information about a run, a necessary and important component for providing explanation to help with debugging forward-chaining programs.

The rest of the paper is organized as follows. Section 2 discusses using rule-firings as the basis for time. The modifications to Rete, using time-tags along with current and past partitions to store run history, are covered in section 3. Since agenda state may be needed in debugging, section 4 describes how an agenda copy may be reconstructed. Section 5 then briefly discusses how historical information may be gathered for answering two different debugging-related questions. Finally, section 6 concludes the paper.

2 Time-related considerations

To understand a forward-chaining program's behavior, we need to know not only what rules fired, but also when they fired, and when facts were in working memory. To store these temporal details, we need a time basis. We could tag occurrences based on the real time at which they occurred. However, this would make it hard to compare the relative times of events between two runs, such as seeing if a rule fired at the same time in both runs. We could instead base time on events occurring within a program, such as working memory changes. This unit allows easier inter-run comparison; it is easy to see if a rule fires after the fifth working memory change in two runs. However, it is too fine-grained; it can become quite large just from loading initial facts, before a single rule fires. A measure more central to forward-chaining execution would be better.

Conceptually, rule-firings are the major units of action in a forward-chaining program. A counter could start at zero at a run's beginning and increase with each rule-firing. Like a memory-changes-based counter, a rule-firings-based counter is comparable between runs: the fifth rule to fire does so at the same "time" in two runs, at time counter value five. Existing systems also support this choice. The Transparent Rule Interpreter (TRI), the graphical debugger described in [5], uses rule-firings as the 'time' scale in its 'musical score' framework for graphically representing forward-chaining execution. Also, as mentioned, most forward-chaining systems allow one to run a program a rule-firing at a time; one rule firing is considered one program 'step'. For debugging, then, since rule-firings are central to forward-chaining execution, we will use rule-firings as the time basis.

3 Modifications to the Rete network

3.1 Regular Rete networks

Before discussing historical Rete, we will briefly review regular Rete. (See also [9], [15], and [12].) A Rete network contains three kinds of memory nodes: alpha, beta, and production nodes. (For simplicity, 'node' will stand for test nodes along with their corresponding memory.) There is an alpha node for each LHS condition, which stores instantiations for facts matching it. A beta node contains instantiations representing two or more consistently-satisfied LHS conditions from a rule, and a production node holds instantiations satisfying all of a rule's LHS conditions.

In Rete, two alpha nodes representing rule conditions are joined into a beta node, containing instantiations whose facts consistently satisfy both conditions. That beta node is (typically) joined with another alpha node into another beta node, containing instantiations whose facts consistently satisfy these three conditions, and so on for all of a rule's LHS conditions, at which point, instead of leading to a beta node, they are joined into a production node, containing eligible-to-fire rule instantiations.

So that they will be less cluttered, this paper's figures use non-standard notation, depicting the instantiations in a network as variable bindings. Figure 1 shows a simplified Rete network for a single rule, rule-13:

(defrule rule-13
  (p ?X ?Y)
  (q ?Y ?Z)
  (r ?X ?Q)
=>

451)
3.2 Historical Rete networks

Historical Rete networks, proposed by this paper, differ from regular ones in two key ways: each instantiation in the network has a time-tag, giving when it was in effect, and each memory node is divided into a current partition, containing instantiations currently in effect, and a past partition, containing instantiations in effect earlier in this run.

A time-tag is a set of one or more intervals stored with a fact or instantiation, giving when it was true. (For brevity, we use 'true' to describe a fact in working memory, a (non-rule) instantiation of condition(s) satisfied by working memory, or a rule instantiation that is eligible to fire.) This time-tag differs from [2]'s, p. 43, because that time tag consists of only one integer, and is associated just with facts, giving when they joined working memory or were last modified; ours stores more information, about both facts and instantiations. An interval is a component \((z, y)\) in a time-tag, where \(z\) is when that item became true and \(y\) is when it became no longer true. An open interval, \((z, \ast)\), indicates that the item is still true.

Each wme hash table entry now also includes the fact's time-tag. When a fact is deleted, its table entry is not removed; instead, its open time-tag is closed with the current time. So, the wme hash table contains facts that are or have been true during a run.

Time-tags tell when things were true, and so are part of a run's historical information. The partitions allow the historical Rete network operation to be surprisingly unchanged by the addition of time-tags, and the associated effect that instantiations that no longer hold have their time-tags' intervals closed, but are not removed from the network. If we keep these no-longer-true instantiations in past partitions, then the instantiations in each memory node's current partition are exactly those in the corresponding normal Rete network memory node. The actions involving all instantiations in past partitions in normal Rete involve all instantiations in current partitions in historical Rete.

So, a memory node's current partition contains currently-true instantiations, and its past partition contains instantiations that were true, but no longer are. What if a no-longer-true instantiation becomes true again? Then, we will have a current instance in the current partition, whose time-tag is a single, open interval, and a past instance in the past partition, whose time-tag interval(s) are all closed. This approach lets us avoid accessing the past partition when adding an instantiation. However, the past partition will be searched when an instantiation becomes no longer true, to append the newly-closed interval to

![Figure 1: A regular Rete network](image-url)
Here is a simple partition example. Consider an instantiation \(i\). If it becomes true at time \(a\), its time-tag is \((a\ *\)\), and it goes in a current partition. If it becomes no longer true at time \(b\), its interval becomes \((a\ \ b)\), and \(i\) is moved from a current partition to a past partition. If \(i\) is a memory node's only instantiation, then that node's partitions contain:

- current: —
- past: \(i\), \((a \ b)\)

If \(i\) becomes true again at time \(c\), \(i\)'s past instance will be in the past partition, and its current instance will be in the current partition:

- current: \(i\), \((c \ *\)\)
- past: \(i\), \((a \ b)\)

Finally, if \(i\) becomes false again at time \(d\), then its current instance is removed from the current partition, and the now-closed time interval is added to the existing past partition entry's time-tag:

- current: —
- past: \(i\), \((a \ b)(c \ d)\)

Figure 2 shows a historical Rete network as it would be right before time 4 for the initial facts given and for these rules:

1. \(\text{defrule rule-1}
   \begin{align*}
   (p \ ?X \ ?Y) \quad (q \ ?Y \ ?Z) \\
   (r \ ?X \ ?W)
   \Rightarrow \\
   (\text{add} \ (r \ ?X \ ?Z)) \\
   (\text{delete} \ (p \ ?X \ ?Y))
   \end{align*}
\)
2. \(\text{defrule rule-2}
   \begin{align*}
   (r \ ?X \ ?W) \\
   (s \ ?Z \ ?X)
   \Rightarrow \\
   (\text{add} \ (q \ (?W*s?X \ ?Z)))
   \end{align*}
\)

Following the chronology shown, one can see how facts propagate through the network, how time-tags are set, and how instantiations move from current to past partitions. The instantiation of rule-2 matching facts \((r \ 4 \ 6)\) and \((s \ 2 \ 4)\) will fire next, at time 4. The action is basically the same as in classical Rete, but now one can see such historical details as, for example, why rule-1 could not fire at time 3: because it had no true instantiations then.

Conceptually, a historical Rete network looks like figure 2; however, we will implement it slightly differently, incorporating hashing. Hashing has been proposed for Rete, to improve performance (as in [11]). For historical Rete, past partitions should be hashed, so that a past partition entry can be found quickly.

To add a fact in historical Rete at time \(t\), we follow the same basic actions as in regular Rete, except that each new fact and instantiation has the time-tag \((t \ *\)\) appended to it. In more detail, the fact is hashed, a wme hash table entry is created if needed, and \((t \ *\)\) is added to the table entry's time-tag. Then, it is tested against every alpha node, and pointers are added from the wme hash table entry to every alpha node matching the fact. For each match, the resulting instantiation is propagated through the historical network, to see if any instantiations need to be added or removed. Each new instantiation is propagated as in a normal Rete network, except that (1) new resulting instantiations are placed in the corresponding memory node's current partition (instead of in its 'only' partition), (2)
beta tests are performed for instantiations in current partitions (but these contain the same instantiations as the normal case's 'only' partitions), and (3) new resulting instantiations have the time-tag interval \((t \ast)\) appended to them.

When adding a fact to working memory, the computational cost of keeping historical information is quite low. The computations for deciding if an instantiation has to be propagated are the same as in regular Rete. The additional overhead comes from appending the time-tag intervals.

Deleting a fact in historical Rete at time \(t\) has a few additional differences. In normal Rete, to delete a fact, its wme hash table entry is found, and that entry's pointers are followed to reach the alpha nodes for LHS conditions matching this fact (and which lead to instantiations using it). We traverse the parts of the network reachable from each such alpha node, and remove the instantiations using this fact. Finally, its wme hash table entry is deleted. In historical Rete, neither the wme hash table entry nor the instantiations using this fact are removed; instead, their open time-tag intervals are closed, and those instantiations are moved to past partitions. As mentioned earlier, this will involve a past partition search, to look for already-existing past partition entries for those instantiations.

In more detail, the fact's wme hash table entry is found, and its time-tag's open interval is closed with \(t\). If it was \((a \ast)\), now it is \((a \; t)\). Then, as in regular Rete, the entry's pointers are followed to the alpha nodes matching this fact. We traverse the parts of the network reachable from each of these alpha nodes, and each encountered node's current partition instantiations are searched to see if they include this fact, which is the same as searching the instantiations in the regular network node's 'only' partition. However, besides removing each instantiation using this fact from the current partition, we will also close its time-tag with \(t\), and see if it already has a past partition entry. If so, then we append the newly-closed interval to the already-existing entry's time-tag; otherwise, the instantiation and its newly-closed time-tag are copied into the past partition. With hashed past partition nodes, this will normally take constant time; so, the additional work in the historical case, closing the time-tags, searching the past partitions, and copying items into past partitions, should not take unreasonably longer.

These preliminary intuitions suggest that the addition of time-tags and current and past partitions does not fatally increase the overhead of adding and deleting facts in historical Rete. They also suggest that we have met our goal of keeping the run-time operation of historical Rete reasonably close to that of original Rete, while still allowing historical information to be maintained within. The major cost will be the storage of the historical information. During program development, these costs should be reasonable, since they will facilitate debugging-related question answering by the system.

4 Agenda reconstruction

Since the agenda determines which rule fires next, its changing contents and their order may be useful in debugging. For example, the number of instantiations above one of interest at a certain time may explain why it was not chosen to fire. Such details may make it easier to determine what would be needed for that rule to fire then.

Since past agenda states may be useful, we should store agenda history. We could store an agenda copy for every time value; then, to see time \(T\)'s agenda, we would retrieve that copy. However, the instantiations in common between 'consecutive' agenda copies makes this a poor use of space. It seems preferable to store enough information to reconstruct an agenda copy when desired. This copy could be used by an explanation system to answer questions, printed for system builder use, or modified to answer follow-up questions. Furthermore, the information used to reconstruct the agenda may also be useful for other purposes, perhaps more conveniently than if it were in the form of literal agenda copies.

Information stored with moderate redundancy can be used to reconstruct the agenda from a desired time. This information is stored in an agenda-changes list, a chronological list of changes made to the agenda during a run. Three kinds of changes can occur: a rule instantiation can be added (an ADD), removed to be fired (a DEL/FIRE), or removed because its LHS conditions are no longer satisfied (a DEL/REMOVE). Each change also includes when it occurred, even though the list is ordered, for easy location of changes from a particular time period, and it contains some representation of the instantiation being added, deleted/fired, or deleted/removed.

We can construct an agenda copy from time \(T\) by going through the agenda-changes list in order, and adding to our copy only those ADD'ed instantiations still on the agenda at that time. Because a low-priority rule could be instantiated from a run's beginning, but not fired for a long time because of higher-
priority rules, we must start at the beginning of the agenda-changes list. However, we can stop at the first agenda-change after the desired time. For each ADD we encounter, we search for that instantiation in its rule's production node, and find its time-tag interval opening with the time of this ADD. If this interval closed before time $T$, then this instantiation left the agenda before then, and should not be added to our copy; otherwise, it does belong on our agenda copy. Since past partitions are hashed, finding the instantiation in its production node should take less time than other alternatives involving searches of unhashed data structures. Note that, since we process the ADD's in chronological order, we can add instantiations to the copy as the system does during run time: search down the copy until we reach an instantiation with equal or less priority, and insert the new one above that one.

As a simple example, consider the agenda-changes list in Figure 3. To make it easier to read, current and past instantiations are not indicated, and letters are used for rule instantiations. Assume that the priorities of the instantiated rules labelled by $A$, $B$, $C$, $F$, and $G$ are all zero, and that those of $D$ and $E$ are 1.

To reconstruct the agenda right before time 2, before $E$ fired, we start at the top of the agenda-changes list: $A$'s addition. According to $A$'s production node entry, it left the agenda at time 1, and so does not belong in the copy. The next entry is $B$'s addition at time 0. $B$ is still on the agenda, and so was also there at time 2; $B$ begins the copy.

$C$ is also added at time 0, but is removed, by being fired, at time 1, so it is not put on the copy. Since the next entry is a DEL/FIRE, we go on to the addition of $D$. $D$ is not removed until time 3, so it belongs on the copy. $D$'s rule's priority of 1 is greater than $B$'s, which is 0, so $D$ is placed on top of $B$:

$D$ — priority: 1
$B$ — priority: 0

The DEL/REMOVE is ignored, and the next entry, $E$, goes in the copy, since it is still on the agenda at time 2, since its priority is equal to $D$'s, but $E$ is more recent, $E$ goes on top of $D$:

$E$ — priority: 1
$D$ — priority: 1
$B$ — priority: 0

The next entry occurred at time 2; since we want the copy from right before time 2, we stop now. The desired agenda copy is the one above.

Although DEL/FIRE's and DEL/REMOVE's are ignored in agenda reconstruction, they may be useful when agenda states over a period of time are desired. To observe the agenda between times $T$ and $U$, DEL/FIRE's and DEL/REMOVE's allow us to construct a copy for time $T$, then apply each of time $T$'s agenda-changes to the copy to get time $(T + 1)$'s agenda, and so on to time $U$, updating the previous time's copy instead of completely rebuilding it for each time. Similarly, they can let us see what happened to the agenda, and in what order, during a single time counter value.

5 Retrieving historical information

Now that we know how to save historical raw data, we will discuss briefly how it might be used in answering two debugging-related questions. Consider the question: What fired rule instantiations' LHS's included fact $(p \times y)$? We will not discuss all the aspects involved in answering this question, but will instead discuss which historical data should be collected for use in eventually answering it, and how to obtain it.

Although not directly requested, the answer should include the time periods that this fact was true. This might help the system builders to notice, for example, that during one period it contributed to several rules' firing, and in another it did not. We should also include the time of firing of the rule instantiations; this reveals when this fact played a role, and gives time information for follow-up questions.
One approach to collecting the needed information is to search every production node likely to contain an instantiation using \((p \ X \ Y)\), and for each such instantiation found, to see if it actually fired. If so, that instantiation and when it fired belong in the answer. More specifically, we go to \((p \ X \ Y)\)'s own hash table entry, and the time tag there gives us the time periods that it was true. Then, we follow this entry's pointers to the alpha nodes matching this fact. From each of these alpha nodes, we traverse the historical network to production nodes reachable from that node. At each of these production nodes, we search for instantiations containing \((p \ X \ Y)\). Each such instantiation is an eligible-to-fire instantiation that had this fact matching a LHS condition; to see if it fired, for every time value that it left the agenda, we see what rule instantiation did fire at that time. For example, if its time tag is \((s \ t)(w \ z)\), then we see what rule instantiations fired at times \(t\) and \(z\). If it did fire, then this instantiation belongs in the answer.

As another example, consider the question: Why did rule \(X\) not fire at time \(T\)? Again focusing on what historical data should be collected and how it can be obtained, we first note the two basic reasons for a rule not firing: either its LHS was not satisfied, or it was eligible to fire, but was not on top of the agenda. To find out which is the case, we search rule \(X\)'s production node, and, examining the time tags, we see if any of its instantiations were true right before time \(T\). For example, if its time tag is \((s \ t)(w \ z)\), then we see what rule instantiations fired at times \(t\) and \(z\). If it did fire, then this instantiation belongs in the answer.

If \(X\) was not eligible to fire at time \(T\), then, to find out which of its LHS conditions were not satisfied then, we backtrack up from rule \(X\)'s production node, checking the time tags of all instantiations in the alpha and beta nodes encountered. If a beta node has no instantiations that were true at that time, then the corresponding inter-condition test was not satisfied then. Likewise, if an alpha node has no instantiations that were true then, then no fact matched the corresponding LHS condition then. Eventually, in this way, we collect the unsatisfied conditions and inter-condition tests for this rule at this time.

In the case that \(X\) was eligible to fire at time \(T\), we could simply see what did fire then; however, that would not give such potentially useful information as, for example, how far down the agenda rule \(X\)'s instantiations were at time \(T\). So, we reconstruct an agenda copy from then using the agenda reconstruction algorithm. Then, we search the copy from the top to see how far down the instantiation(s) of rule \(X\) were. We can also see which instantiations above rule \(X\)'s highest had higher priorities, and which were just more recent.

In these examples, the historical information in the historical Rete network makes obtaining specific details about a run straightforward and reasonable. This will help a great deal in developing a practical system for answering questions such as the above.

6 Summary and conclusions

We have proposed a new use for the Rete network: to store and maintain historical run information. The main contribution of this paper is that it explains how Rete can be modified for this purpose. The historical information thus stored can be accessed by a question answering system designed to help with debugging forward-chaining programs.

It is feasible to store and maintain this information without degrading run-time performance during program development too badly. In fact, it is noteworthy that it can be integrated so easily into Rete. The basic Rete propagation is almost unchanged; the additions involve peripheral actions such as appending time tags, and moving no-longer-true instantiations from current to past partitions. Using hashing in various places, the time for these additional duties should be reasonable. Keeping an agenda changes list as well will also allow reasonable agenda reconstruction when a past agenda state is desired.

Using time tags incorporated into Rete to store history has several benefits: it makes retrieval of specific historical details easier, thus making it easier to answer questions for debugging, and at the same time it makes maintaining and storing the historical information easier, since we simply tag events as they occur. In addition, improvements to classical Rete may also be applicable to historical Rete. For example, \([12]\)'s parallel Rete could be made historical; since a rule firing is a synchronization point (p. 43), instantiations being added and removed in parallel could be tagged with a rule firing based time tag.

Storing a run's historical details is an early necessary step for designing explanation to help with debugging forward-chaining programs. Such explanation will reduce the tedium for system builders, reducing the time they must spend examining system traces or single stepping through a program to find out if
or when something occurred. We plan to implement these ideas, to see if integrating this information into Rete works as well in practice as it appears that it should in principle. We will then design and build a question-answering system, determining what questions are useful for debugging forward-chaining programs, and designing algorithms for answering them, using the information in the historical Rete network. Future work will also include more detailed analyses of the complexities of maintaining and accessing this historical information.

Historical Rete stores information for a single program run, since one usually debugs a specific run. In principle, it is not limited to that; it could be used over longer periods of time, perhaps for several runs of the same program. It would be interesting to see if information from multiple program runs could facilitate the debugging of rule-based programs in general. Furthermore, such information might be useful input for improving the efficiency of a rule-based program, such as in, for example, the restructuring rules in ch. 6 of [2].

Both debugging and testing require knowledge of what has occurred during a program run — therefore, additional future work may include using this historical information in the development of further testing and debugging tools for forward-chaining programs. As the size of these programs increases, such software engineering tools will become more necessary. These methods for maintaining and storing run history should help in developing such tools.

References


