N92-16591

ADDING RUN HISTORY TO CLIPS

Sharon M. Tuttle and Christoph F. Eick

Department of Computer Science
University of Houston, Houston, Texas

Abstract. To debug a CLIPS program, certain 'historical' information about a run is needed. It would be convenient for system builders to be able to ask questions requesting such information. For example, system builders might want to ask why a particular rule did not fire at a certain time, especially if they think that it should have fired then, or they might want to know at what periods during a run a particular fact was in working memory. It would be less tedious to have such questions directly answered, instead of having to rerun the program one step at a time or having to examine a long trace file.

This paper advocates extending the Rete network used in implementing CLIPS by a temporal dimension, allowing it to store 'historical' information about a run of a CLIPS program. We call this extended network a ***historical Rete network***. To each fact and instantiation are appended ***time-tags***, which encode the period(s) of time that the fact or instantiation was in effect. In addition, each Rete network memory node is partitioned into two sets: a *current* partition, containing the instantiations currently in effect, and a *past* partition, containing the instantiations which are not in effect now, but which were earlier in the current run. These partitions allow the basic Rete network operation to be surprisingly unchanged by the addition of time-tags and the resulting effect that no-longer-true instantiations now do not leave the Rete network.

We will discuss how historical Rete networks can be used for answering questions that can help a system builder detect the cause of an error in a CLIPS program. Moreover, the cost of maintaining a historical Rete network is compared with that for a classical Rete network. We will demonstrate that the cost for assertions is only slightly higher for a historical Rete network. The cost for handling retractions could be significantly higher; however, we will show that by using special data structures that rely on hashing, it is also possible to implement retractions efficiently.

I. INTRODUCTION

One of the activities of a system builder developing any kind of software is *debugging*, "the process of locating, analyzing, and correcting suspected faults" ((IEEE 1989), p. 15). So, first, the system builders notice, one way or another, that there is a manifestation of an error in the program. Then, they debug by finding, and then correcting, the cause(s) of that particular error. In a forward-chaining rule-based language such as CLIPS ((Giarratano 1989), (COSMIC 1989)), a program's data-driven execution affects the process of debugging.

In a CLIPS program, data changes determine what happens next. One of the rules whose left-hand-side conditions are all satisfied will be chosen to have its right-hand-side actions executed. Those actions may change working memory, causing some previously-unsatisfied rules to now be satisfied, and vice versa. So, the choice of which rule to fire determines which rules will have a chance to fire next, and so affects what can happen next. To debug such a program, the system builders need detailed information about what happened during its run, including the *order* in which rules were executed, and *when* certain data were (or were not) in working memory. This *historical* information about a run will be

PRECEDING PAGE BLANK NOT FILMED

necessary to discover why the program executed as it did.

Particularly for large CLIPS program, system builders armed only with those tools currently provided have a tedious job ahead. CLIPS allows one to run a program one rule-firing at a time (or, it allows one to *single-step* through a program), and to check what is in memory or in the *agenda*, the ordered list of rules currently eligible to fire. CLIPS can also be directed to display a trace of rule-firings and/or of working memory changes for system builder use. So, to find out, for example, when a fact was in working memory, we can carefully examine a possibly-long trace of assertions to and retractions from working memory, or we can single-step through the program, and ask to see the current list of facts when we reach various points at which we suspect that the fact is true. To find out why a rule did not fire at a particular time, we can single-step through the program up to that time, and then try to determine, from the agenda and working memory contents, why this rule did not fire then. We can find out such information using the current tools, but we may also easily overlook details in the sheer volume and monotony of the data.

CLIPS' current debugging tools are not very different from those found for other, similar forward-chaining rule-based languages. Evidence that these tools are not sufficient can be found in the current research trying to ease the debugging of large forward-chaining programs. (Domingue and Eisenstadt 1989) presents a graphics-based debugger, while (Barker and O'Connor 1989), (Jacob and Froscher 1990), (Eick et al. 1989), and (Eick 1991) suggest changes to rule-based languages, such as the addition of rule-sets, that might, among other goals, ease debugging, changing, and maintaining rule-based programs.

We would like to explore a slightly different approach to debugging: how explanation can be used in debugging CLIPS programs. The explanation subsystem envisioned will allow system builders to ask questions on the top level of CLIPS about the latest run of a program, which the system then answers, instead of requiring the system builders to run the program again, single-stepping through it, or to pore over the system trace. This explanation, designed with the problems of forward-chaining rule-based program debugging in mind, would be a useful addition to the current debugging facilities provided by CLIPS.

Many questions useful for debugging deal with the aforementioned historical details of a run. For example, system builders might ask what fired rule's right-hand-side actions contributed a particular fact to working memory, allowing it to trigger some other rule. They might ask which fired rules needed a particular fact to be true for their left-hand-side conditions to be satisfied; this gives them an idea of that fact's impact. They might ask why a particular rule did not fire at a certain time, especially if they think that it should have fired then. These questions can be answered using the current CLIPS facilities, by single-stepping through a run or by studying a trace, but the tedium would be reduced if the questions could be directly answered instead. However, one of the first hurdles to answering such questions is determining how to store and maintain a run's historical information.

CLIPS uses an inference network to efficiently match left-hand-side (or LHS) rule conditions to facts; in particular, it uses the Rete algorithm ((Forgy 1982), (Scales 1986)) for this matching. Basically, in the Rete algorithm, a network of all the LHS conditions from all the rules is built, which includes tests both for each LHS condition appearing in any rule, and for certain combinations of conditions within rules. When a fact matches a LHS condition, an *instantiation* — indicating that this condition is satisfied by this fact — is stored in the network; instantiations are also stored for combinations of LHS conditions satisfied by sets of facts. A rule instantiation represents a collection of facts that satisfies all of the rule's LHS conditions. Then, as each new fact is asserted, it is sent through the network. As a result of this propagation, rules may become eligible to be fired, if this fact's assertion causes instantiations of those rules to be created. Likewise, rule instantiations may be removed because of this fact's assertion, if the rule contains a LHS condition requiring that this fact not be true. When facts are retracted from working memory, that is also propagated through the network, causing the removal of instantiations including the now-retracted fact.

In this paper, a generalization of the Rete network, called a *historical Rete network*, is proposed that allows the storing and maintenance of historical information for a single run of a CLIPS program. We have two main objectives in modifying Rete for this purpose: CLIPS programs using the modified network must still run almost as efficiently as before the modifications, so that run-time operation during program development is not overly impeded, and the modified network should allow reasonable maintenance and retrieval of a program run's historical information. Since the Rete network implements rule instantiations, it is feasible to 'tag' condition and rule instantiations with when they occurred during a run. Including this information within the network will allow us to design top-level explanation facilities that can more easily and efficiently answer 'historical' questions about a run, and thus ease the task of debugging for system builders. Storing and maintaining this information is just one of several components in providing such explanation, but it is a necessary and important aspect.

The rest of the paper will be organized as follows. Section 2 briefly introduces the Rete algorithm, then discusses our use of rule-firings as the basis for time, and then describes how time-tags, along with current and past partitions, can be used to to store CLIPS program run history. Since some questions that a system builder might ask would involve knowing what the agenda looked like at a certain time, section 3 covers how an agenda copy may be reconstructed on demand. Section 4 then briefly describes how historical information about a program run may be retrieved in the context of gathering data for answering several different kinds of questions useful for debugging. Finally, section 5 concludes the paper.

II. MODIFICATIONS TO THE RETE NETWORK

A. Introduction to Rete Networks

Before discussing the necessary structural changes, we will briefly review 'normal' Rete networks. (For a fuller description, see (Forgy 1982), (Scales 1986), and (Gupta 1987).) In a Rete network, there are three kinds of memory nodes: alpha nodes, beta nodes, 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; the alpha node stores an instantiation for each fact matching this condition. A beta node contains instantiations representing two or more consistently-satisfied LHS conditions from a particular rule (or rules), and a production node holds instantiations that satisfy all of the LHS conditions of a rule.

In a Rete network, two alpha nodes representing rule conditions are joined into a beta node, containing instantiations that consistently represent both conditions being true. Then that beta node is (typically) joined with another alpha node into another beta node, containing instantiations that consistently represent these three conditions being true, and so on until all of a rule's LHS conditions have been represented, at which point, instead of leading to a beta node, a beta node and alpha node are joined into a production node, containing 'complete' rule instantiations that are eligible to fire. And, when it is being built, as conditions are added to the network, each condition appears in the network only once; if it is used in several rules, then that alpha node has a number of successors, and likewise, if a set of conditions appears in several rules, the section of the Rete network leading to a beta node representing that set of conditions may also be shared among several rules.

Figure 1 shows a simplified Rete network for a single CLIPS rule, rule-13, given in Table 1. (We will use facts, instead of fact-ids, in instantiations in most of the figures, for greater clarity.) Each of the three LHS conditions in rule-13 has a corresponding alpha node that tests for matches and stores an instantiation of each matching fact. So, we see in Figure 1 that working memory fact (*p* 1 3) matches rule-13's LHS condition (*p* *?X* *?Y*), that fact (*q* 3 5) matches (*q* *?Y* *?Z*), and that fact (*r* 1 7) matches (*r* *?X* *?Q*). The first two conditions are then joined into a beta node, which tests if any of the facts matching those two conditions are compatible, and then stores any combinations passing the test. (*p* 1 3) and (*q* 3 5) both match their respective conditions with *?Y* = 3, so an instantiation for that pair is stored in the beta node. Then, that beta node is joined with the remaining condition, and since this is the last condition, any instantiations resulting from these compatibility tests will be instantiations for rule-13, stored in a production node. The variable *?X* is 1 in both the beta node's only instantiation and in (*r ?X ?Q*)'s only instantiation, so they can be combined into a compatible instantiation for the entire rule, and so an instantiation is stored in rule-13's production node.

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

Table 1. A CLIPS rule

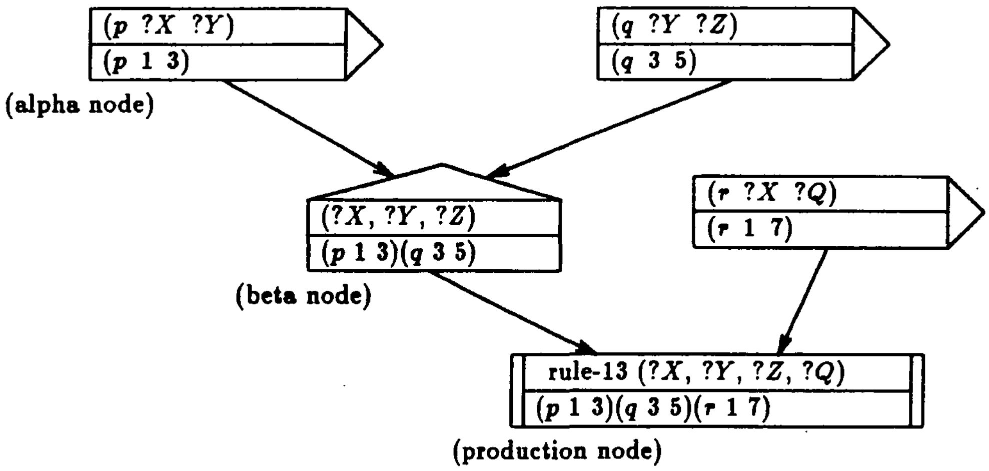


Figure 1. A 'regular' Rete network

In general, when a fact is asserted, it is added to the working memory element (or wme) hash table, which stores all the facts currently in working memory. Each wme hash table entry includes pointers to all of the LHS conditions (or, alpha nodes) in the Rete network that match this fact; these pointers allow us to avoid another search of all the conditions if the fact is later retracted. The newly-asserted fact is then compared to every alpha node, and, if the fact matches that LHS condition, an instantiation is stored in the alpha node, and a pointer to this alpha node is stored in the fact's wme hash table entry. We then visit all of that alpha node's successors, seeing if the new instantiations in a node result in new consistent instantiations at the succeeding node. Any new rule instantiations added to production nodes are added to the agenda.

CLIPS uses an *agenda*, a priority queue containing the currently-eligible rule instantiations in the order that they should be fired (if they stay on the agenda long enough); the one on top of the agenda will be the next chosen to fire. An agenda of size *n* does not necessarily display the next *n* rules that will be fired, however, because each rule-firing has the potential to change working memory, causing rule instantiations to join and to be deleted from the agenda. CLIPS uses the following conflict resolution strategy (COSMIC 1989):

(1) The instantiations are ordered by the *salience*, or priority, of the rules involved; the instantiation with the highest salience is on top of the agenda.
(2) If instantiations have the same salience, then the one which became true most

recently is preferred over earlier ones by being placed nearer to the top of the agenda.

B. Time-Related Considerations

Before we can modify Rete to store historical information, we need to decide on some time scale, so that we can store the data needed to determine when and in what order rules fired, and when facts were (and were not) in working memory. To further facilitate debugging, we would like a time-unit that is central to CLIPS program operation. Conceptually, rule-firings are the major units of action in a CLIPS program. Computations are done when a rule fires, and the computations performed are those of the selected rule.

The Transparent Rule Interpreter (TRI), a graphical debugger for forward-chaining rule based languages described in (Domingue and Eisenstadt 1989), uses rule-firings as the 'time' scale in its 'musical score' framework for graphically representing forward-chaining execution. Also, note that the existing debugging aids within CLIPS are rule-firing based. As previously mentioned, CLIPS allows system builders to run a program one rule-firing at a time, in order to more closely examine its execution while debugging; the single step in this single-stepping process is one rule-firing. And, the CLIPS *(run)* command concludes by printing out how many rules have fired during the program execution, and if one chooses to display rule-firing information during program execution, then each is numbered by its order of occurrence within the run. The rule-firings are considered a measure of how much or how little has occurred. Therefore, it is quite natural to use rule-firings as a time basis. A counter starts at zero at the beginning of each run and is incremented with each rule-firing. This also has the useful feature of being comparable between runs; for example, running the same program with the same data twice, the fifth rule to fire does so at the same 'time' in both runs — at time counter value five — which can make it easier to compare and contrast, for example, two runs of the same program using slightly different sets of facts.

C. Historical Rete Networks

Historical Rete networks, proposed by this paper, differ from classical Rete networks in two major respects: each instantiation stored within the network has a *time-tag*, which gives the period(s) of time that the instantiation was in effect, and each memory node has its contents partitioned into two sets: a *current* partition, containing all instantiations currently in effect, and a *past* partition, containing all instantiations in effect earlier in this run.

A *time-tag* is a set of one or more intervals stored with a fact or instantiation, which gives the time period(s) during a run that the fact or instantiation was in effect. For brevity, we will use 'true' to describe a fact in working memory, an instantiation representing a condition or conditions satisfied by working memory, and an eligible rule instantiation. (Note that, because of *refraction* ((Brownston et al. 1985), pp. 62-63), a rule instantiation that fires becomes ineligible, even if its RHS actions do not cause any of its LHS conditions to become unsatisfied, until at least one of its facts is retracted and asserted again.)

This time-tag is different from the time tag mentioned in (Brownston et al. 1985), p. 43, because that time tag is associated just with facts, and not also with instantiations as ours is, and it consists of only one integer, representing when that fact joined working memory or was last modified. The time-tags we use store more information, about both facts and instantiations. An *interval* is a component $(x\ y)$ in a time-tag, in which x was the time when that fact or instantiation became true and y was the time when it became no longer true — when the fact was retracted from working memory, when one or more conditions represented by an instantiation were no longer satisfied, or, for a rule instantiation, when it left the agenda. An *open interval* indicates that the fact or instantiation is still true; we write such an interval as $(x\ *)$.

Time-tags are found in the wme hash table and the historical Rete network. Each wme hash table entry now also includes the time-tag for that fact. When a fact is retracted from working memory, its entry is not removed from the table; instead, the open interval in its time-tag is closed with the current time. So, the wme hash table stores all the facts that are *or have been* in working memory during this program run. We can tell if a fact is currently true by simply seeing if the last interval of its time-tag is open. (The wme hash table entry should probably also store all of the fact-ids that a fact has had during a run.)

The time-tags give the time period(s) during a run that a fact or instantiation was true, and so they are part of the run's historical information. The partitions, on the other hand, serve a very different purpose: they allow the basic Rete network *operation* to be surprisingly unchanged by the addition of time-tags and the resulting effect that instantiations do not leave the historical Rete network. (Instantiations that no longer hold have their time-tags' intervals closed, but those instantiations are not actually removed from the network.) If we keep a memory node's no-longer-true instantiations in a past partition, then the instantiations in each memory node's current partition are exactly those that would appear in the corresponding 'normal' Rete network memory node. This, then, allows most historical Rete network operations to take place as in a 'normal' Rete network: the actions that involve *all* instantiations in normal Rete now involve all instantiations *in current partitions only* in historical Rete.

```
(defrule rule-1
  ?p_addr <-(p ?X ?Y)
  (q ?Y ?Z)
  (r ?X ?W)
=>
  (assert (r ?X ?Z))
  (retract ?p_addr))

(defrule rule-2
  (r ?X ?W)
  (s ?Z ?X)
=>
  (assert (q =(*?W ?X) ?Z)))
```

Table 2. Rules for Historical Rete Example

Figure 2 shows a historical Rete network as it would be right before time 4 for the initial facts given and for the rules in Table 2. Following the chronology shown in Figure 2, one can see how the facts propagate through the historical Rete network, how the time-tags are set, and how instantiations come and go (and move from current to past partitions). As shown, the instantiation of rule-2 matching facts (*r* 4 6) and (*s* 2 4) will be the next to fire, at time 4. The action is basically the same as a classical Rete network, 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.

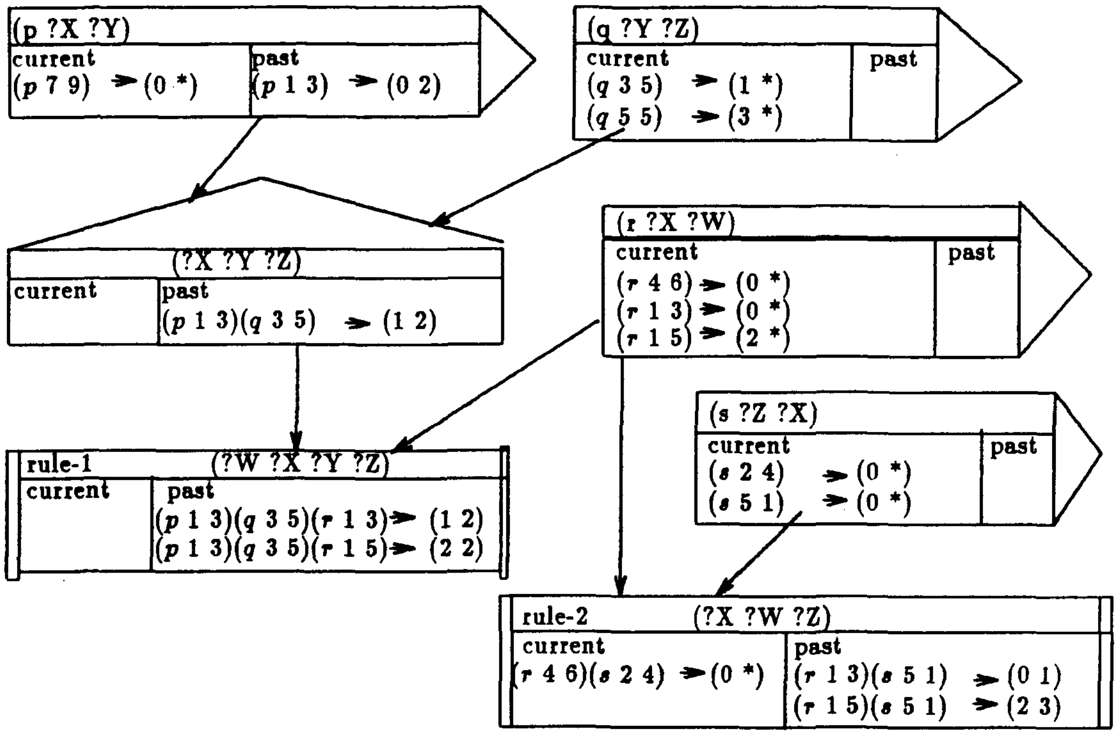
Conceptually, a historical Rete network will look like figure 2; however, for performance reasons, we will likely implement it slightly differently; for example, we will very likely incorporate hashing into it. Hashing has been proposed for Rete networks to improve performance (for example, in (Gupta et al. 1988)); it will be useful for historical Rete networks as well. In particular, past partitions of nodes should be hashed, so that a particular past partition entry can be found in constant time (in the average case).

When a fact is asserted into a historical Rete network at time counter value *t*, it has the time-tag (*t* *) added to its wme hash table entry and also to any new instantiations resulting from its propagation through the historical Rete network. The propagation through the historical Rete network is essentially the same as for a "classical" Rete network, except that

(a) each new instantiation that results is placed in the corresponding memory node's current partition (instead of in its 'only' partition, in the normal case),

(b) beta tests are performed for instantiations in current partitions (but these current partitions contain the same instantiations as the 'only' partitions in the normal case), and

(c) (as already mentioned) each instantiation that results from asserting this fact has the time-tag interval $(t\ *)$ appended to it.



time 0:
(initialise working memory)
assert (p 1 3)
assert (p 7 9)
assert (r 4 6)
assert (r 1 3)
assert (s 2 4)
assert (s 5 1)

time 1:
FIRE rule-2 (r 1 3)(s 5 1)
ASSERT (q 3 5)

time 2:
FIRE rule-1 (p 1 3)(q 3 5)(r 1 3)
ASSERT (r 1 5)
RETRACT (p 1 3)

time 3:
FIRE rule-2 (r 1 5)(s 5 1)
ASSERT (q 5 5)

Figure 2. A Historical Rete Network

When asserting a fact, the computational cost of keeping historical information is quite low. The computations for deciding if an instantiation has to be propagated are still the same as in the original network. The only additional computational overhead comes from adding the time-tag intervals to each fact and instantiation, and from updating any node hash tables being used as necessary.

Retracting a fact from the historical Rete network at time counter value t has a few additional differences compared to its "classical" Rete counterpart. In a "classical" Rete network, the wme hash table entry for the fact to be retracted is found, and the pointers within this entry to every alpha node matching this fact are traversed in turn. From each alpha node matching this fact, we remove any instantiations making use of the retracted fact, and then continue to all the nodes reachable from this alpha node, searching for, and removing if found, any instantiations using the fact being retracted. When done with all of that, the fact's wme hash table entry is deleted.

In a historical Rete network, the process is basically the same — we find the fact's wme hash table entry, follow its pointers to the alpha nodes matching this fact, and search from each of these alpha nodes for all of the instantiations using this fact. Where we searched all the instantiations in the "only" partition of each encountered memory node in the "classical" case, now we search only the instantiations in the current partition (which has the same collection of instantiations as the classical Rete network's only partition, for each memory node). Instead of deleting those instantiations that make use of the fact being retracted, we move them from the current partition to the past partition of their memory nodes, and close the open interval in their time-tags with the current time-counter value. (For example, if the open interval was $(a\ *)$, and the current time counter value is t, then the time-tag interval becomes $(a\ t)$.) Finally, when done with all that, instead of removing the wme hash table entry, we merely close the open interval in its time-tag with t.

Comparing retraction in the historical and regular cases, most of the differences are minor: the entry is removed from the wme hash table in the regular case, but just has its open time-tag interval closed in the historical case, and each instantiation using the retracted fact is deleted from the network in the regular case, but is moved from the current to the past partition of its memory node in the historical case, with its time-tag also closed with the current time counter value. However, there is a bit more work than might be apparent in moving an instantiation from the current to the past partition. To keep things most straightforward if an instantiation is true for more than one period of time (for example, if a rule contains a negated condition that is true, then false, then true again), we would like to keep no more than one entry per instantiation in a memory node's past partition; this will require a search of the past partition. If a previous instance of this instantiation is found, we append the newly closed time-tag to it, and if not, then we add a new entry with the instantiation and its newly closed time-tag. Fortunately, if we hash the past partitions, then this search will take constant time, in the average case.

Here is a simple example, to demonstrate the use of partitions. Let i be an instantiation that was true between times a and b and between times c and d. When it became true at time a, its time-tag was $(a\ *)$, and it clearly belonged in the current partition of the appropriate memory node. When it became no longer true at time b, its interval was closed, resulting in the time-tag $(a\ b)$, and i was removed from its memory node's current partition and moved into its past partition. At time c, it became true again.

There are a number of possibilities about how to proceed; we will leave the *past* instance of this instantiation in the past partition and create a current instantiation in the current partition. Keeping two copies of an instantiation when it is true and has been false in the past, one in each partition, simplifies run-time operation of the historical Rete network. It allows us to assume that instantiations in current partitions have exactly one interval, which is known to be open, in their time-tags. It also lets us avoid accessing the past partition when adding an instantiation. However, as mentioned, the past partitions will be accessed when instantiations are removed from current partitions, so that the newly-closed time-tag can be appended to an existing entry, if one exists. That way, each instantiation has at most one entry in its memory node's past partition, whose time-tag includes all of the time periods that this instantiation was true, instead of having a past partition entry for each interval that the instantiation was true.

So, assuming that i is the only instantiation in its memory node, then after time b and before time c the memory node's partitions are as shown in Table 3:

current: —
past: i, (a b)

Table 3. After time b, before time c

Now, when i becomes true again at time c, the past instance of the instantiation will be in the past partition, and the current instance will be in the current partition, as shown in Table 4:

current: i, (c $*$)
past: i, (a b)

Table 4. After time c, before time d

Finally, when i becomes false again at time d, then the 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, as shown in Table 5:

current: —
past: i, (a b)(c d)

Table 5. After time d

These preliminary intuitions suggest that the addition of time-tags and current and past partitions does not fatally increase the overhead of asserting facts to and retracting facts from the historical Rete network. They also suggest that we will meet our goal of keeping the run-time operation of the historical Rete network reasonably close to that of the original version, while still allowing historical information to be maintained within. The major cost will be the storage of the historical information.

III. AGENDA RECONSTRUCTION

Since the agenda determines which rule fires next, its changing contents and their order are part of a run's historical information. And, for answering certain debugging-related questions, the agenda's state at a particular time will, indeed, be needed. For example, consider the question "Why did rule X not fire at time T?". Using the historical information in the historical Rete network, we can find out, from rule X's production node, if any instantiations of rule X were true — and thus eligible to fire — at time T (Any rule instantiation whose time-tag has an interval containing T was eligible at time T.) However, once we find that it was true then, we need the agenda from that time to obtain further details about why rule X did not fire. For example, with the agenda, we can see how many other rule instantiations were above rule X's highest instantiation. Such details may make it easier to determine what would be needed for rule X to fire at time T.

Since past states of the agenda may be useful in debugging a CLIPS program, we need to determine how to handle agenda history. We would like to avoid storing a copy of the agenda for every value of the time counter, because the potentially large number of instantiations in common between 'consecutive' agenda copies makes this seem like a poor use of space. It seems preferable to store enough information to reconstruct an agenda copy when desired. This reconstructed copy could be used by an explanation system to answer questions, could be printed for direct system builder use, or could be 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.

Our current plan is to use information stored with only moderate redundancy to reconstruct the agenda at a particular time reasonably quickly. The needed information will be stored in an *agenda-changes list*, containing a chronological list of all changes made to the agenda during a program run. Three kinds of changes are possible: a rule instantiation can be added (an ADD), a rule instantiation can be removed to be fired (a DEL/FIRE), and a rule instantiation can be removed because at least one of the rule's LHS conditions is no longer satisfied by working memory (a DEL/REMOVE). Each change also includes the time of that change, even though the list is ordered, for easy searching for changes from a particular time period. And, finally, each agenda-changes entry also contains some representation of the instantiation being added, deleted/fired, or deleted/removed. Notice that, for any single time counter value, there will be exactly one DEL/FIRE entry, and zero or more DEL/REMOVE's and ADD's.

To construct a copy of the agenda as it was at time counter value T, we basically search the agenda-changes list entries from time 0 to time T. Then, for each ADD, we see if it was still on the agenda at time T, and if so, we add it to the agenda copy being constructed. We must start at the beginning of the agenda-changes list each time because a very-low-priority rule may be instantiated from the very beginning of a run, but not fired for a very long time because other rules always take priority. However, we can, of course, safely ignore all changes made to the agenda after time T.

To see if an ADD'ed instantiation from the agenda-changes list was removed from the agenda before time T, we think the best approach will be to search for the instantiation's entry in the past partition of its rule's production node. With hashed past partitions, this should normally take constant time. If found, then we find the time-tag interval beginning with the time of this ADD (after all, this rule instantiation was added to the agenda at this time because it had become eligible). If this interval was closed before time T, then this instantiation was not on the agenda at time T, and we do not need to add it to our agenda copy. Otherwise, it was still eligible then, and we should add it. (If the instantiation is not found in the past partition, then it is in the current partition, in which case, being still true currently, it also was true at time T, and should be added to the agenda copy. Notice, however, that this can only occur if an agenda copy is being constructed during a run, for example while single-stepping through it.)

Using the production node in this way should take less time than other alternatives, which involve possibly time-consuming searches of other, non-hashed data structures. For example, we could add every ADD'ed instantiation encountered to the agenda copy, and then could remove any that left before time T as we come to their DEL/FIRE's or DEL/REMOVE's. However, that would involve a search of the agenda copy every time that an instantiation had to be removed. Similarly, for each ADD'ed instantiation, we could search down the agenda-changes list to see if it has a DEL/FIRE or DEL/REMOVE before time T; but, this would involve searching down the agenda-changes list for each ADD'ed instantiation.

A pleasant advantage to handling the ADD's in chronological order is that they can be added to the agenda copy using the same means that the system adds instantiations to the agenda during run-time, using CLIPS' conflict resolution strategy. For each ADD'ed instantiation that was still on the agenda at time T, we start at the top of the agenda copy, and compare the salience of the 'new' instantiation to that of each of the instantiations on the copy in turn, until reaching one whose salience is the same as or less than the 'new' instantiation's salience. The 'new' instantiation will be placed directly above that instantiation. We can safely stop searching down the copy after reaching one with the same salience because, since we are handling the ADD's in chronological order, recency dictates that this new instantiation, joining the agenda later than any of those already in the copy, should go on top of those with the same salience.

Here is a simple example of agenda reconstruction. We have an agenda-changes list as shown on the left in Figure 3, in which we use letters to represent rule instantiations. To make the figure easier to read, current and past partitions are not indicated. Assume that the saliences of the instantiated rules labelled by A, B, C, F, and G are all zero, and that those of D and E are 1.

Figure 3. Data for Agenda Reconstruction Example

Agenda-changes list:

0	ADD	A
0	ADD	B
0	ADD	C
1	DEL/FIRE	C
1	ADD	D
1	DEL/REMOVE	A
1	ADD	E
2	DEL/FIRE	E
3	DEL/FIRE	D
3	ADD	F
3	ADD	G
4	DEL/FIRE	G

(labels used for instantiations, for easier reading)

Production Nodes from Historical Rete Network:

A, time-tag:(0 1)

B, time-tag:(0 *)

C, time-tag:(0 1)

D, time-tag:(1 3)
E, time-tag:(1 2)

F, time-tag:(3 *)
G, time-tag:(3 4)

(partitions left out, for easier reading)

To reconstruct the agenda from right before time 2, before E fired, we start at the top of the agenda-changes list; the first entry is the addition of A. We check its entry in its rule's production node, and see that it was removed from the agenda at time 1, and so does not belong in the copy. The next entry is the addition of B at time 0. B does not have a past partition instance at this time, as it is still on the agenda, and so it was also on the agenda at time 2; B becomes the first instantiation in 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 entry after that, the addition of D. D is not removed until time 3, so it belongs on the copy. D's rule's salience of 1 is greater than B's, which is 0, so D is placed on top of B, as shown in Table 6:

D — salience: 1
B — salience: 0

Table 6. D added to agenda copy

The next entry is ignored, since it is a DEL/REMOVE. For the next, E should be in the copy, since it is still on the agenda at time 2; its salience is the same as D's, but E is more recent, and so E goes on top of D, as shown in Table 7. The next entry occurred at time 2, and so, since we want the copy to be as the agenda was right *before* time 2, we stop now. The final agenda copy is the one shown in Table 7. If one writes out the agenda from time 0 onward, adding and deleting as specified, one sees that this is, indeed, the state of the agenda as it was right before time 2.

E	salience: 1
D	salience: 1
B	salience: 0

Table 7. *E* added to agenda copy

One might question why the DEL/FIRE's and DEL/REMOVE's are kept in the agenda-changes list at all, since they are ignored during agenda reconstruction. They are worth keeping for when the states of the agenda *over a period of time* are desired. To observe the agenda between times T and U, having DEL/FIRE's and DEL/REMOVE's in the agenda-changes list keeps the system from having to reconstruct an agenda copy for time T, to then completely reconstruct another for time $(T+1)$, and so on through time U, starting over at the beginning of the agenda-changes list each time. With the DEL/FIRE's and DEL/REMOVE's, the system can instead reconstruct the agenda for time T, then apply each of time $(T+1)$'s agenda-changes list entries to the copy, resulting in time $(T+1)$'s agenda, and so on to time U, updating the previous time value's copy instead of completely rebuilding it each time. Similarly, the DEL/FIRE's and DEL/REMOVE's allow us to show what happened to the agenda, and in what order, during a single value of the time counter, if that is desired by the system builder.

IV. USING THE HISTORICAL RETE NETWORK TO ANSWER QUESTIONS FOR DEBUGGING

We will now consider how a CLIPS program run's historical Rete network and associated data structures can be used by an explanation subsystem in answering several types of questions useful for debugging. We will not discuss all the aspects involved in answering these questions, but will instead concentrate on which historical data should be collected for use in eventually answering them, and how to obtain this information.

A. When was fact *F* in working memory?

This question is very simple to answer using the stored historical information, and is also very useful for debugging. For example, if F is a control fact, asserted to indicate that the program has identified a particular sub-situation, then knowing when F was in working memory lets the system builders know when that sub-situation was considered during the run, if ever. If a discovered fault involves F, then knowing when F was in working memory lets the system builders concentrate on those time periods in their subsequent efforts. If F was *not* in working memory during the run, that may also be important, especially if the system builders think it should have been; its not being in working memory may explain why certain actions were not done. In addition, knowing when F joined working memory makes it easy to find out which rule asserted it (if it was not asserted directly by the system builder); we just find out what rule fired at that time, and double-check that its RHS actions could, indeed, assert F. (We may have a separate rule-firings list, or we can find out what rule fired at time T by looking for the DEL/FIRE entry with that time in the agenda-changes list.)

To find out when fact F was in working memory is even easier than finding out what rule (probably) asserted it, since we plan to include time-tags in the wme hash table. All we need to do is access fact F's wme hash table entry, and copy the time-tag from that entry (and probably the corresponding fact-id(s) as well). For example, if F's time-tag is $(a\ b)(c\ d)$, then F was in working memory from the ath rule firing to the bth rule firing, and then again from the cth rule firing to the dth rule firing. Since CLIPS associates a different fact-id with each new assertion of a fact (following a retraction), it would also be potentially useful to the system builders to include with each time period the fact-id assigned to the fact during that period.

There is also the possibility of giving the system builders the time periods in a form other than the relative rule-firings; for example, the time-periods could be expressed as the actual rule instantiations that fired, or as the actions that were done. However, whether this would be better for debugging, how such an answer can be expressed clearly, and how the system builders can perhaps be allowed to specify dynamically which they would prefer, are still open questions.

B. What facts matched LHS condition *L*, and when?

This question could be useful for debugging when the system builder is interested in strange program behavior related to a particular LHS condition. If the system builder considers this condition to be key to one or more rules' firing, then knowing what satisfied it, and when, may shed light on why rules containing this LHS condition did or did not fire. The facts which satisfied *L*, and when each was in working memory, can be used in subsequent questions; and, if *L* was never satisfied, then that in itself may reveal to the system builders why certain actions were not performed by the program.

Given the historical Rete network, answering this question will be easy, as long as *L* is indeed a LHS condition in one of the rules. We access this LHS condition's alpha node within the historical Rete network; each instantiation stored in that alpha memory, whether in the present or past partition, corresponds to one fact that matched *L*. Moreover, each instantiation contains the time-tag giving when that fact matched *L*. So, this alpha node's contents constitute the data for answering this question.

C. What fired rule instantiations' LHS's included fact *F*?

This question allows system builders to easily find out which rule firings actually made use of a particular fact. For example, this could reveal, if *F* should not have been in working memory, just how much "damage" it caused, in terms of instantiations firing that should not have fired. The answer to this question can also make visible some of the effects of a previous instantiation firing: if one fired instantiation asserts fact *F*, and the system builders want to see if that action directly contributed to other actions, then the answer to this question gives them that information. And, if *F* is a control fact, inserted specifically to control what rule is to fire in a particular scenario, then if the answer to this question is that *no* rule fired using this fact, that could indicate to the system builders (1) that the fact was never in working memory, or (2) that the rules had an error (for example, a typo, or a missing field) in the condition that was supposed to correspond to that fact, or even (3) that the rules that were supposed to assert *F* did not do so, or asserted it incorrectly.

Although not directly requested, the answer should include the time periods that this fact was in working memory. This might be useful, because the system builders might notice, for example, that during one period the fact contributed to several rules' firing, and in another it did not. We should also include the time of firing of each rule instantiation that used this fact: this lets the system builders know when in the run this particular fact played a role, and it gives them the time information in case they want to follow up this question with one about why a certain rule instantiation did or did not fire at one of those time values. And, if the system builders choose to single-step through the run again, they know which rule-firings to pay particular attention to.

To collect the information for answering this question, we start at the wme hash table entry for fact *F*, and copy the time periods that *F* was in working memory. Then, using the hash table entry's pointers to all alpha nodes matching this fact, we travel in turn to each such alpha node.

From each alpha node that matches *F*, we travel to all of the production nodes reachable from that alpha node. (An alpha node may lead to more than one production node because, as stated earlier, if a LHS condition is used in more than one rule, its alpha node is shared in the historical Rete network.) At each of these production nodes, we search its past partition for instantiations containing F. Each found is a rule instantiation that used F, and is no longer eligible — now, we need to see if it actually fired. After all, it may have left the agenda because one of its LHS conditions became unsatisfied before it could fire. We check every time value that it left the agenda — for example, if its time-tag is $(s\ t)(w\ x)$ then it left the agenda at times t and x — and see if the rule instantiation that fired at that time is, indeed, this instantiation. If so, then we have found an instantiation that fired, and that used F, and so it should be added to the list-in-progress of instantiations to be included in the answer. We continue in this way until we have checked all the past partition instantiations in production nodes reachable from alpha nodes for LHS conditions matching F. At that point, we have collected all of the fired rule instantiations that included fact F, and we can present them to the system builder in some reasonable form.

D. Why did rule X not fire at time T?

This type of question has the potential to be particularly useful for the purpose of debugging. Once system builders notice that a rule that they thought should fire at a particular time did not, having this question available — even with only low-level suggestions for what caused the "error" — could save them much tedium in terms of single-stepping through a run to the time of interest, poring over a long trace of rule firings, and/or adding print statements to particular rules. Knowing why a rule did not fire may lead directly to an error that kept a rule from firing, or it may indicate gaps in the system's rules (or data). If a particular unsatisfied LHS condition kept it from firing, then the system builders might immediately notice any obvious typos as soon as they are shown that condition. Or, the condition might cause the system builders to think of a fact that they thought was true, and that should have satisfied this condition; that might suggest what question they should ask next. (For example, they might ask what rules have RHS actions that could have asserted that particular fact. This follow-up question does not involve historical information about the run, but it is useful in this particular scenario. Historical information would come into play again with the likely-follow up to that question: why the rules that could have asserted that fact did not themselves fire.) If, on the other hand, the rule was eligible to fire at time T, but another fired instead, then knowing the relative saliences of the fired rule and rule X might give the system builders clues that salience-adjusting is needed, or might point out a rule (or rules) that should not have been eligible at time T, but were.

Again focusing on what historical data should be collected and how it can be obtained, we first reiterate 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 of these is the case, we start by going to rule X's production node in the historical Rete network. We check the time-tags of all of the instantiations in this production node, in both the current and past partitions, and see which, if any, contain intervals including T. Each such instantiation was eligible to fire at time T, and we should add it to a list of instantiations of rule X that were eligible to fire at that time.

When we are done checking all of rule X's production node's instantiations, the list of eligible instantiations built will determine what we do next. If this list is empty, then rule X did not fire because it was not eligible to fire at that time; we should next determine what LHS conditions were not satisfied then. If this list is not empty, then rule X did not fire because none of its instantiations were on top of the agenda right before time T; we should next determine, in this case, why other instantiation(s) preceded rule X's instantiations on the agenda.

If the rule was not eligible to fire at time T, we will traverse the historical Rete network backwards from rule X's production node, searching the nodes corresponding to rule X's LHS conditions. These alpha and beta memory nodes will be searched for instantiations current at time T: both current and past partitions of each node will be searched, and each instantiation's time-tag will be checked to see if T lies within any of its intervals.

If a beta node is found to have no instantiations that were true at time T, then the corresponding inter-condition test was not satisfied then. Likewise, if an alpha node is found to have no instantiations that were true at that time, then no fact matched this LHS condition then. Eventually, in this way, we build a list of unsatisfied LHS conditions and inter-condition tests, and this list will be used in constructing the answer, so that the system builders will know which conditions of rule X were not satisfied at time T; these need to be satisfied, if rule X is to even be eligible to fire then.

In the case that the rule was eligible to fire at time T, we could simply report what rule instantiation did fire then; however, that would not give such potentially-useful information as, for example, just where in the agenda rule X's instantiations were at time T. We can gather additional details in the following way. First, we reconstruct a copy of the agenda from right before time T, using the already-discussed agenda reconstruction algorithm. Then, we search down the agenda copy from the top to find out how far down the agenda the highest instantiation of rule X was, and also how many of the instantiations above rule X's highest one had higher saliences; such instantiations will always be chosen to fire before rule X's, if all are on the agenda concurrently. The remainder of the instantiations above rule X's highest one are there because they joined the agenda more recently — for rule X to fire before these, those rules will have to be instantiated earlier, or rule X will have to be instantiated later. If there are too many instantiations above rule X's to readably present them all to the system builders, then we can still at least tell them how many were above it, and how many of those had higher salience; and we should include the instantiation on the agenda top, and its salience, in any case.

In these examples, the historical information in the historical Rete network and associated data structures 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. Storing historical information within the network makes it easy to update as the run proceeds, and leaves it where it can be easily obtained for different kinds of debugging purposes — such as different kinds of questions — after the run concludes. With the system able to answer such questions, the system builder will be able to easily obtain historical information about a run.

V. CONCLUSIONS AND FUTURE WORK

We have proposed that historical information about a run of a CLIPS program be stored within a historical Rete network used to match CLIPS rules' LHS's to working memory facts. This paper has explained how a Rete network can be modified for this purpose, and how the historical information thus stored can subsequently be used by a question-answering system to be designed to help with debugging a CLIPS program.

It should be feasible to store and maintain this historical information without degrading CLIPS run-time performance too badly. In fact, it is noteworthy that it can be integrated so easily into the Rete network. The basic Rete propagation is almost unchanged; the only 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 to perform these additional duties should be quite reasonable. Keeping an agenda-changes list as well will also allow reasonable agenda reconstruction whenever the system builder (or question-answering system) needs a past agenda state.

Maintaining this information is a necessary first step for providing explanation to help with debugging CLIPS, since CLIPS, being a forward-chaining language, requires temporal information to determine why a program behaved as it did. 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.

Future work will include actually implementing these ideas, to see if integrating this information into the Rete network works as well in practice as it appears that it should in principle. Such investigation may also reveal ways to reduce the space needed to store historical data. Then, we will build the question-answering system described, on the top-level of CLIPS. We will determine what types of questions are useful for debugging CLIPS programs, and will design a system that can answer such questions, using the historical information stored in the historical Rete network and associated data structures. Empirical experiments will also be needed for evaluating the effectiveness of the resulting explanations in helping with debugging. Both debugging and testing require knowledge of what has occurred during a program run — therefore, additional future work may also include using this stored historical information in the development of further testing and debugging tools for CLIPS programs. As the size of CLIPS programs increases, such software engineering tools will become more and more necessary. These methods for maintaining and storing run history, and the explanation that they will facilitate, should help in developing these future CLIPS programs.

REFERENCES

Barker, V. E. and O'Connor, D. E. (1989). Expert systems for configuration at Digital: XCON and beyond, *CACM*, Vol. 32, No. 3 (March), pp. 298-318.

Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming Expert Systems in OPS5*, Addison-Wesley.

COSMIC (1989). *CLIPS Reference Manual*, Version 4.3, Artificial Intelligence Section, Lyndon B. Johnson Space Center, COSMIC, 382 E. Broad St., Athens, GA., 30602.

Domingue, J. and Eisenstadt, M. (1989). A new metaphor for the graphical explanation of forward-chaining rule execution, *Proceedings of IJCAI '89*, Detroit, Michigan, pp. 129-134.

Eick, C. F. (1991) Design and implementation of a rule-based forward chaining language that supports variables, encapsulation, and operations, submitted for publication to *IEEE Transactions on Knowledge and Data Engineering*, February 1991.

Eick, C. F., Yao, C., and Fu, H. (1989). More flexible use of variables in rule-based programming, *Proceedings of the 2nd Int. Symposium on Artificial Intelligence*, Monterrey, Mexico.

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

Giarratano, J. (1989). *CLIPS User's Guide*, Version 4.3, Artificial Intelligence Section, Lyndon B. Johnson Space Center, COSMIC, 382 E. Broad St., Athens, GA., 30602.

Gupta, A., Forgy, C., Kalp, D., Newell, A., and Tambe, M. (1988). Parallel OPS5 on the Encore Multimax, *IEEE International Conference on Parallel Processing*, pp. 271-280.

Gupta, A. (1987). *Parallelism in Production Systems*, Morgan Kaufmann.

IEEE (1989). *IEEE's Software Engineering Standards*, IEEE.

Jacob, R. and Froscher, J. (1990). A software engineering methodology for rule-based systems, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 2 (June), pp. 173-189.

Scales, D. (1986). Efficient matching algorithms for the SOAR/OPS5 production system, Knowledge Systems Laboratory, Stanford University, Report No. KSL 86-47 (June).