Rule Interpreters In ELEKTRA

Iain D. Craig

Department of Computer Science

University of Warwick
Coventry CV4 7AL
UK EC

ABSTRACT

In this paper, we describe a number of rule interpreters, each of which is presented in production rule form. Each of the interpreters can be executed as an ordinary ELEKTRA ruleset which interprets all or some of the rules in an ELEKTRA system. The interpreters, therefore, override the default behviour of the ELEKTRA interpreter. Some of the interpreters that we describe are capable of interpreting themselves: others can be combined into larger, reflective interpreters. The aim of the paper is not, however, to show how sophisticated reflective processing can be achieved in ELEKTRA, but, rather, to show that ELEKTRA is powerful enough to support a wide variety of different interpreters without requiring any changes to its code. None of the interpreters requires additional system code in order to work: ELEKTRA provides all of the facilities used by the rulesets as part of its standard library. This paper is offered as additional evidence of the enormous power of the ELEKTRA system.

1. INTRODUCTION

In previous papers (Craig, 1991a, 1991b), we introduced the ELEKTRA system, and, in Craig (1991b), promised a paper giving details on the construction and operation of various rule interpreters written in the form of ELEKTRA rules. This paper is the one that we promised: it shows how a number of different interpreters can be constructed, all of them being expressed as collections of ELEKTRA production rules. In some cases, we only give an outline of the full interpreter—this is to insulate the reader from the details of ELEKTRA's operation. For the most part, the interpreters that we describe are complete: all that is required is that they be encoded in the ELEKTRA rule formalism and then executed[1].

This paper contains four main descriptions of rule interpreters:

- A number of forward-chaining interpreters similar to those described in Craig, 1991b.
- A simple backward-chaining interpreter.
- A content-directed backward-chaining interpreter.
- A mixed forward-and-backward interpreter similar to that described by Nilsson (1982).

Along the way, we discuss refinements and extensions to these interpreters. We will also show (in outline) how ELEKTRA can operate with more than one active interpreter.

The emphasis in this paper is on how to write rule interpreters in ELEKTRA. Although we will refer quite often to the system's reflective properties, they are not the primary focus of this work.

The plan of the paper follows the list above. In the next section, we describe each of the interpreters in turn. Section three describes how many, different interpreters can be active at any one time. Finally, we offer a brief summary of this work and make suggestions for future work: in particular, we will be concerned with the paucity of the production rule representation.

Before continuing, we must stress the fact that the interpreters we describe below are not to be considered as the definitive versions: they represent points on a continuum, for ELEKTRA allows each one to be implemented in a variety of ways. We have, therefore, chosen simple interpreters with the aim of demonstrating some of the things that are *possible* in ELEKTRA.

---

1. Since this paper was written (September, 1990), a new version of ELEKTRA has been implemented in CommonLISP. The new implementation is still derived from the formal specification in Craig, 1990a, but differs in the names of some interpreter functions and in the range of primitives it provides. For a very brief outline of the new system, see the Postscript below.

2. RULE INTERPRETERS

In this section, we present the rule interpreters listed in the introduction. Each interpreter is given in a pseudo-code form: this is aimed at facilitating comparison with other approaches. As far as conversion to the ELEKTRA rule-format is concerned, the process is comparatively simple because the pseudo-code is, in any case, very close to ELEKTRA's input format.

2.1 Forward-Chaining

ELEKTRA was initially designed to be forward-chaining. It should come as no surprise that forward-chaining interpreters are relatively easy to construct in ELEKTRA. Here, we first describe a simple interpreter which only executes object-rules. Then, we describe a more powerful one that will execute all rules that reside at a level lower than that on which the interpreter rules are to be found. Finally, we present the version which interprets itself. As the reader will note, there is not much difference between the last two versions (a simple test is the only difference).

2.1.1 Simple forward-chaining

Here, we present the simplest form of forward-chaining interpreter. It is similar to the one presented in Craig (1991b). The interpreter consists of two rules: one to detect termination, the other to perform interpretation. The interpreter only deals with object-rules: i.e., this is a two-level interpreter with the interpretation rules only matching, selecting and executing object-rules.

```
MR1: if match-condition(goal-statement) then stop!

MR2: if object-rules(r) and
        match-conditions-set(r, ri) and
        longest-condition(i, ri)
       then
        execute-instance-actions(ri)
```

where

- `match-condition(goal-statement)` is an executable relation in one argument. `goal-statement` is a user-supplied (not necessarily pattern-matched) atom. `match-condition` is satisfied whenever goal-statement is satisfied (it can be satisfied via the pattern-matcher—i.e., resides in working memory—or represents a call to some procedure). An alternative to the use of `match-condition` is to encode the goal in the form `goal-statment(atom)`, where `atom` is an arbitrary ELEKTRA working memory structure: the intention is that goal-statement is pattern-matched in the usual way. In this case, `MR1` will simply match its condition-element against working memory as normal: the cost is that one or more object-rules must add an element of the correct form to working memory.

- `stop!` is a system-provided primitive which halts execution of the main interpreter loop (for details, see Craig, 1991b or Craig, 1991c).
- `object-rules(rules)` is a system-provided executable relation. It binds `rules` as its output: the output is the set of all object-rule identifiers currently in production memory. This relation is clearly of use to meta-rules when it is required that they inspect *all* the object-rules that are currently in the system.
- `match-conditions-set` is an executable relation provided by the system. Its first argument must be bound to a set of rule identifiers (of any level), and it returns a set of rule instances via its second argument. The rule instances represent all those rules which are elements of the first set whose condition-parts have been satisfied by the current state of working memory. `match-conditions-set` calls the ELEKTRA matcher on the condition-part of each rule named by an element of the set bound to the first argument.
- `longest-condition` is assumed to be a user-supplied executable condition which takes a set of rule instances as its first argument and returns the instance with the shortest condition-part in its second argument. The details of longest-condition are not relevant here (for a discussion, see Craig, 1991b): the essential point is that an arbitrary conflict-resolution procedure can be included in the interpretation rules.
- `execute-instance-actions` is a system-provided action. It executes the action-part of the rule instance which is supplied as its argument. All of the elementary actions in the instance are executed: the variables which they contain obtain their values from the variable bindings which form part of a rule instance.

A variant of this interpreter has already been examined in some detail (Craig, 1991b). The interpreter merely shows the general form for forward-chaining interpreters: it must contain a mechanism for matching rule conditions, a mechanism for performing some kind of conflict-resolution and a mechanism for executing actions. In addition, termination must be detected at the application level and the appropriate signal given to the ELEKTRA base interpreter (via `stop!`).

The interpreter given above differs from the one presented by Craig (1991b) in that the above only interprets object-rules. We can make it more powerful by allowing it to interpret meta-rules. The alteration to the interpreter that is required is as follows:

```
MR2a: if all-rules(r) and
        f-subset(r, lower-level-than-me, mr) and
        match-conditions-set(mr, ri) and
        longest-condition(i, ri)
      then
        execute-instance-actions(ri)
```

The differences are as follows. The first condition-element now becomes `all-rules`: this system-provided condition-element returns the identifiers of all the rules currently in production memory. Since the set bound to `r` will contain the identifiers of rules that reside at all levels of the hierarchy, they must be filtered: this is what `f-subset(r,lower-level-than-me,mr)` does. `f-subset` is a general set-manipulating function that is provided by ELEKTRA[2]: its first argument is a set, its second a predicate and its third is an output set (the form given above is the characteristic relation of the function). The predicate, in this case (which must be executable code) compares the level tag of the rules in `r` and is satisfied if and only if the value of the tag is less than the level at which `MR2a` resides. The result, `mr`, is the set of the identifiers of all those rules currently in production memory which reside at a lower level in the rule hierarchy than does interpreter rule `MR2a`. `f-subset` returns, as its last argument, the subset of its first argument which satisfies its second argument. It should be noted that, in this case, the level at which `MR2a` resides is 'compiled' into `lower-level-than-me`.

This version of the interpreter will allow the interpretation of both object- and meta-rules. It interprets all rules that are lower in the rule hierachy than the two interpretation rules. In order to make this interpreter work, it is necessary to replace the default ELEKTRA main loop with a new loop which simply executes the rules at the level on which the interpreter resides: more details on this can be found in Craig, 1991b.

The next step is to allow the interpreter to interpret itself. This is very easily accomplished: the second condition-element (the element whose relation symbol is `f-subset`) is merely removed. The instance of `mr` in match-conditions-set must be changed to `r` so that all rules are matched. As has been argued in Craig, 1991b, the resulting interpreter is such that it interprets itself.

2.2 Backward-Chaining

In this sub-section, we describe two simple backward-chaining interpreters. Both interpret-

---

2. The CommonLISP version employs CommonLISP set-manipulation primitives: the `f-subset` primitive must be defined by the user in the latest version.

ers are based on AND-trees. The approach that we have adopted (OR-trees) is the simplest possible: expansion to AND/OR-trees is possible, although we do not pursue this. The reader is warned at this stage that the interpreters are of limited power and that they are not further developed. This is because we have a bias to forward-chaining and find it more interesting than backward-chaining (it can generate many more blind alleys in the search space, for one thing, so there is a more accute control problem).

The simulation of backward-chaining in forward-chaining interpreters is a fairly well-known process, and we include the interpreters below only for the sake of completeness. We describe a content-directed backward-chaining interpreter just to show, in general, how it might be done. We begin with the basic interpreter and then extend it to cater for content-directed reasoning. In the statement of the rules which follows, for reasons of brevity and clarity, we have decided to abbreviate some of the conditions and actions In each case, we give either the necessary pseudo-code or a detailed explanation of the condition-element or action in question: this is so that the reader will be better placed to see how the abbreviations can be inserted into the rules. We follow our usual format and present the rules before explaining their components and their operation.

```
MR1: if current-goal(g) and
       not(expanded(g)) and
       one-condrules(r) and
       match-condition-against-wme(r,g,ri) and
       not(empty-set(ri))
     then
       make-expanded(g);
       execute-instances-actions(ri)

MR2: if current-goal(g) and
       not(expanded(g)) and
       goal-statement(g,gs) and
       match-wm(gs)
     then
       mark-solved(g)

MR3: if current-goal(g) and
       expanded(g) and
       goal-id(g, gi) and
       findall(gs,[solved(gs) and dominator(gs,gi)],sg) and
       not(empty-set(sg))
```

```
then
 mark-solved(g)
```

```
MR4: if top-goal(tg) and
        goal-id(tg, tgi) and
        expanded(tg) and
        findall(g,[solved(g) and dominator(g,tgi)],sg) and
        not(empty-set(sg))
      then
        mark-solved(tg);
        mark-solved-top-goal(tg);
        stop!
```

where:

- `current-goal` is a working memory element that refers to the goal the system is currently trying to satisfy (see below for discussion).
- `goal-id` is a two-place executable condition which returns the identifier of the goal node as its second (rightmost) argument (see below).
- `expanded(g)` is a condition-element which is true if and only if `g` is not marked as being 'expanded' (see `make-expanded` below).
- `one-condrules` is a one-place executable condition which binds its single argument to the set of those rules in production memory which have a condition-part of length one (see below).
- `match-conditions-against-wme(ruleidset,wme,ri)` is an executable condition which matches the (single-element) condition-parts of the rules whose identifiers are elements of `ruleidset` against the working memory element `wme` (which must be an atom). Rule instances are created for each of the rules whose (single-element) condition-part is satisfied by `wme`: the instances are elements of the set that is bound to `ri` — `ri` is an output variable.
- `mark-expanded` is an action which marks the current goal as being expanded. Expanded goals are not further developed by the system: in other words, once a goal has been expanded, rules which apply to it are not further considered for execution. This explains why a number of the above rules contain the condition-element `not(expanded(g))` where `g` is the current goal.
- `goal-statement` extracts the statement of a goal from the its goal node (see below).
- `match-wm(g)` is an executable condition-element (an executable relation) which attempts to match `g` against the contents of working memory. `g` can be positive or negative. If the match succeeds, `match-wm(g)` is true: it does not return any bindings. In the case of `MR2`, the bindings that are generated by `match-wm` are, in any case, not required

because the intention is to discover whether the element mentioned by the current goal (`g`) is already an element of working memory—if it is, no more rules should be matched against it because it is a given.

- `mark-solved(g)` is an action which marks the goal named by `g` as being solved. Any goal which is marked as solved is already marked as expanded (but not vice-versa — expanded goals are not necessarily solved). Solved goals are used in directing the interpreter rules to examine other goals. The marker is also used as a form of 'success-popping': that is, the propagation of solution information back up the goal tree.
- `findall` is a set-forming executable condition-element[3]. `findall` takes three arguments. The first argument must be bound to a set: the elements of the set must be such that they can be pattern-matched. The second argument is a predicate which is used to test the elements of the first argument. Those elements of the input set for which the predicate is satisfied become elements of the third argument (which is the output variable for the condition-element). In this case, the predicate to `findall` is a conjunction of simpler predicates: clearly, the predicate is satisfied whenever all of its conjuncts are satisfied. The output variable in the instance of findall to be found in rule `MR3` is `sg`; that in rule `MR4` is also named `sg`.
- `empty-set` is an executable relation which is true if its argument is bound to the empty set.
- `top-goal` is a normal (i.e., pattern-matched) working memory element which represents the goal the system is to satisfy. All `top-goal`s are also `goal`s.
- `make-solved-top-goal` is an action which marks the working memory element bound to its single argument as being solved. This action is for the convenience of the user and has no impact on the operation of the system because, in `MR4`, it is immediately followed by `stop!` — it is `stop!` which halts the system.
- Semi-colon (`;`) is the pseudo-code operator that we use to denote the concatenation of rule actions.

We now give the definitions of those rule elements which we abbreviated above. We use the symbol '`:=`' as the definition sign. It should be noted that '`:=`' is not a symbol of the ELEKTRA rule-language, nor of the pseudo-code: instead, it is a symbol in the informal meta-language that we adopt when explaining ELEKTRA constructs.

-
```
one-condrules(r) := rules(rs) and
                    f-subset(rs,[l rl. length(action-part(rl)) = 1)],r)
```

3. In the CommonLISP version, the `findall` primitive does not have this form: the form given above is easy to define, however.

where we have expressed the predicate that `f-subset` expects as its second argument as a `l`-expression (to write this in a form which ELEKTRA will accept, the `l`-expression *must* be expressed as a *named* LISP function).

- `goal-id` is defined as follows. Each goal (whether top or current) has an associated identifier. `goal-id` retrieves the identifier from working memory.
- `mark-solved`. This action deletes the goal named in its argument from working memory and replaces it with a goal which has an extra argument: this argument is bound to the constant `solved`. `make-solved-top-goal` is defined analogously.
- `dominator`. When a goal is added to working memory, it is always the descendent of another goal: in other words, it is always a sub-goal of some other goal. The identifier of the goal is recorded in each of its sub-goals in order to allow success-propagation back to the root goal. The identifier of the goal from which a sub-goal was derived (by the application of *exactly one* rule) is recorded in the dominator field of the sub-goal. The `dominator` relation extracts this information from the sub-goal and binds it to dominator's rightmost argument.

In order to implement these rules, it is necessary to complicate the representation somewhat. The presentation of the rules has been, the most abstract one possible. In order to engage in an implementation, the dominator and identifier arguments to the goal relation must be included. In other words, a goal has the general form:

```
goal(goal-statement,identifier,dominator)
```

where `goal-statement` is an atom, `identifier` is the goal element's identifier (which is generated somehow), and `dominator` is initially bound to some meaningless value.

The (seemingly most sensible) representation of `current-goal` is, therefore:

```
current-goal(goal)
```

where `goal` is identical in form to the general form we gave above. This leads to the following definitions of `goal-id` and `dominator`:

```
goal-id(goal(gs, i, d), i)
```

and:

```
dominator(goal(gs, i, d), d)
```

We can define `goal-statement` as:

```
goal-statement(goal(gs,i,d), gs)
```

(this is required in rule MR2).

If we adopt the alternative, and make `current-goal` just bind the identifier of the goal whose satisfaction is currently being attempted, we need to add an extra condition-element to each rule which accesses the current goal. The extra condition-element merely uses pattern-matching to bind the first and third arguments of goal (the middle one is bound by `current-goal`). The details of this modification should be immediate.

We can define mark-solved as:

```
mark-solved(goal(gs,i,d,e)) :=
        delwm(goal(gs,i,d,e));
        addwm(goal(gs,i,d,expanded,solved))
```

where `e` is the expanded argument (note that `mark-solved` also marks the goal is being `expanded`). We can define `expanded` as:

```
expanded(goal(gs, i, d)) :=
        delwm(goal(gs, i, d));
        addwm(goal(gs, i, d, expanded))
```

The changes needed if `current-goal` binds an identifier should be clear, as should the general scheme for implementing `mark-solved` for top-level goals and for implementing `make-solved-top-goal`.

With these technicalities out of the way, we can move on to describing how the rules that we presented above operate. Once we have described them, we will point out some interesting facts about them. The description that we give here is at the same level of generality as the initial presentation of the rules.

The idea behind the backward-chaining interpreter is as follows. An initial goal is accepted by the system which then tries to decompose the goal into sub-goals until all of the sub-goals match working memory elements. In other words, the initial goal is successively decomposed into an AND tree of sub-goals, the leaves of which are working memory elements. The leaves are added to working memory before any rules are applied. In order to simplify matters, the interpreter constrains the problem-solving rules to having *one* condition-element in their condition-parts.

For each goal (initial or sub-goal), the condition-element is matched against that goal. If the condition matches, the rule's action is executed. Action execution leads, as normal, to the addition or deletion of working memory elements. The interpreter rules require that all goal-setting (i.e., all insertions of `goal` elements into working memory) be done by rule actions. Rule actions, therefore, must have access to the identifier of the goal which satisfied their condition: thus it is necessary for goal elements to contain their identifier because the goal-setting actions must set the dominator argument of the goal node which they add to working memory. To make this easier and less error-prone, a special action can be defined by the user for problem-solving rules (i.e., rules whose condition-part is matched against the goal tree, and whose action adds a new `goal` element to working memory). This action might be defined (in Scheme) as[4]:

4. For a variety of reasons, it is recommended that this action be defined as a user-defined action: i.e., as a procedure.

```
(define (add-goal goal-statement dominator)
  (let ((goal-node-id (gensym "goal")))
     (addwm
       `(goal ,goal-statement ,goal-node-id ,dominator))
```

where all Scheme constructs are as defined in the standard (Scheme, 1989).

The restriction on the length of problem-solving rule condition-parts is to make the interpreter easier to define. Each rule can match one and only one sub-goal. Problem-solving rules can, however, set an arbitrary number of sub-goals: this is because their action-part is unrestricted.

The rules that constitute the interpreter form two sets: the first (rules `MR1` and `MR2`) is concerned with sub-goal expansion, and the second (`MR3` and `MR4`) with termination.

Rules `MR1` and `MR2` determine what should be done when there is a currently unsatisfied sub-goal. `MR2` has a condition-part which detects that there is a currently unsolved sub-goal. Its condition-part attempts to match the goal-statement of the sub-goal with working memory contents. If there is a match, the sub-goal has been solved: this is because it is one of the givens of the problem. The action-part of `MR2` marks the sub-goal as solved.

`MR1`, on the other hand, is concerned with what to do when a sub-goal has a statement which is not already an element of working memory. It finds all rules in production memory that have exactly one condition-element (using `one-condrules`): in other words, it finds all the problem-solving rules. Next, using `match-condition-against-wme`, it determines which of the problem-solving rules match the goal. The way in which this process is represented in the statement of rule `MR1` shows that the condition-parts of rules that match sub-goals should have the form `goal(gs,i,d)`—it is, of course, possible for them just to have the form `gs`, for it, too, is a legal condition-element as far as ELEKTRA is concerned. Using the first representation for condition-elements, it is also possible to use an ELEKTRA-provided condition to obtain all the rules which mention goals in their condition-parts. This takes the form:

```
findall-wm-elems(condition-mentions-relation(ruleid,goal),ruleid,es-
et)
```

where `findall-wm-elems` is an executable condition which binds `eset` to all those working memory elements which satisfy the relation which is the first argument to the condition. The relation `condition-mentions-relation` is generated by the ELEKTRA rule compiler. In the present case, `condition-mentions-relation` asserts that the rule named ruleid has a condition-part which contains at least one occurrence of the relation symbol `goal` (it also asserts that `goal` is the relation symbol of an atom which does not appear negated in the rule). The the variable `ruleid` is used to tell `findall-wm-elems` which elements to collect into `eset`. The rule condition has to be modified so that the conditions of the rules named in eset are matched: there is an ELEKTRA primitive to do this.

Once the rules have been found and matched, their actions are executed to generate more sub-goals. The sub-goals state that the goal which was current when `MR1` was fired is their dominating sub-goal. `MR1` marks the sub-goal mentioned by its first condition-element as `expanded`. An expanded sub-goal is one to which all applicable problem-solving rules have been applied. When a sub-goal is marked as being `expanded` no other rules can be applied to it, and the focus of attention within the system shifts to other sub-goals.

The other pair of rules is concerned with success propagation and with termination. `MR3` can be fire whenever a sub-goal has been marked as solved. It checks that the goal `g` has been expanded (i.e., that `g` directly matches some working memory element). Next, it uses the ELEKTRA `findall` primitive to construct the set of sub-goals which are dominated by `g` and which are maked as being solved (`findall` was described above). The last part of `MR3`'s condition-part checks that `sg` is not bound to the empty set: that is, it checks that there are sub-goals of `g`. The action-part of MR3 sets `g` to be solved. What this rule does, therefore, is to traverse the sub-goal tree checking that there are sub-goals *all* of whose sub-goals are solved (i.e., match working memory elements). When it encounters such a sub-goal, it marks it as solved. The intention is that `MR3` will fire for every solved sub-goal and will propagate the solved marker up to the topmost goal in the tree.

The reader should recall that `mark-solved` also marks a terminal sub-goal as `expanded`: this prevents any rules matching against it, thus causing additional branches in the sub-goal tree. This property of `mark-solved` ensures that once a terminal sub-goal has been detected, its dominating sub-goal will become eligible for marking as solved (by satisfying expanded in `MR3`'s condition-part). This protocol ensures that all sub-goals will eventually become closed and that the solved marker will propagate back to the root of the tree if and only if all sub-goal nodes are marked `expanded` and their children marked `solved`[5].

The operation of `MR4` is similar to that of `MR3`, but it deals with the topmost node in the goal tree: that is, with top-goal. The condition-part of `MR4` finds the top goal and checks that it has been expanded (i.e., that there has been some problem-solving activity since the goal was set). Next, it finds all the sub-goals in the tree that are immediately dominated by the top goal. If these sub-goals are all marked solved, the set `sg` is not empty, and the entire condition-part is therefore satisfied. `MR4`'s action marks the top goal as solved and places another marker on the top-goal working memory element. Finally, it halts the system by executing `stop!` — a solution has been found.

---

5. This property of the rules can be proved by induction—it is left as an exercise.

The marking of the working element that represents the top goal can take one of two forms, depending upon whether the top goal is represented by a separate element—i.e., we have a pair of elements, either of the form:

```
top-goal(goal(gs, tgi...))
goal(gs,tgi...)
```

or of the form:

```
top-goal(tgi)
goal(gs, tgi,...)
```

where `gs` is the goal statement (an atom) and `tgi` is the name of the top goal. In either case, the marking of the top goal as being solved has no impact upon the problem-solving behaviour of the interpreter: the marking is there only for the convenience of the user.

Even a cursory inspection of the four rules shows that the two pairs are not in competition with each other until at least one sub-goal has been maked solved. This requires that at least one branch of the sub-goal tree be expanded as far as possible: this is because, conceptually, all rules which match a sub-goal are fired in parallel to generate new, unsolved and unexpanded sub-goals. Once a sub-goal has been marked expanded *and* solved, the three rules `MR1`, `MR2` and `MR3` compete with each other: `MR4` can never compete until the entire tree has been solved (because of the backward propagation of the solved markers). As we described above, the `solved` markers propagate back towards the root of the tree, so none of the children of the root node (the top goal) are marked until the sub-trees which they dominate have been marked: this prevents the entire condition-part of `MR4` from being satisfied.

During the expansion of the tree (that is, while no sub-goals are marked `solved`), rules `MR1` and `MR2` compete with each other. At any one time, there may be many instances of both of these rules which can be fired. There will be as many instances of `MR2` as there are sub-goals whose statement matches a working memory element, and there will be as many instances of `MR1` as there are sub-goals which have been matched by rule condition-parts. It should be noted that there may be a non-empty intersection of these two sets: that is, a sub-goal might match a working memory element *and* might also match the condition-part of some rule. If no sub-goals have statements which match working memory elements, `MR2` will not be instantiated (because its condition-part cannot be satisfied). When there is a non-empty intersection, there is a control problem.

Now, it is clearly possible to tolerate the non-determinacy which arises when we allow `MR1` and `MR2` to execute at random. If the interpreter just fires the first rule that is satisfied, non-determinacy will result. The base interpreter might, for example, employ a conflict-resolution strategy which prefers the longest condition-part, in which case `MR1` will be fired more frequently than `MR2`. If `MR1` fires more often, there is a risk that the system will generate more sub-goals than necessary. As a first attempt, we can leave `MR1` and `MR2` to fire on a random basis: as long as `MR2` is eventually fired, the `solved` marker will propagate back to the top goal. Even if we allow the three rules `MR1`, `MR2` and `MR3` to fire in a non-deterministic fashion, the `solved` marker will still propagate. Depending upon how the problem-solving rules are set up, and, depending upon the particular selection of rules, there will be a combination of behaviours: sometimes the system will expand sub-goals, sometimes it will set a `solved` marker in a terminal node, and sometimes a `solved` marker will propagate back towards the top goal. As we have argued above, this will lead, eventually, to termination (assuming that the problem-solving rules are not degenerate[6]). Since `MR4` can only fire under certain, special, circumstances, when the time comes for it to be executed, it will not compete with any of the other meta-rules.

The interpreter for backward-chaining can be executed in a number of ways. The simplest is just to use the ELEKTRA default loop. The rules which implement the backward-chaining interpreter are, themselves, forward-chaining rules, so the default loop will cope with them quite adequately. Another, and more radical solution, is to use the rule-based forward-chaining interpreter described in section 2.1.1. The simple interpreter described in that section interprets forward-chaining rules, so the backward-chaining interpreter can be executed using the rules in 2.1.1 quite naturally. Content-directected forward-chaining interpreters can also be used to control backward-chaining. These are all simple instances of the ways in which ELEKTRA can support multiple, simultaneous interpreters.

The backward-chaining interpreter that we have described above is not the best possible: it is, in any case, only an extended example. However, we can make improvements to it as it stands (we will return to the question of its limitations below). One improvement is to reduce the amount of matching that `MR1` performs: this allows us to show how another of ELEKTRA's meta-level facilities can be used. The other improvement that we consider concerns selecting rules that will close branches in the sub-goal tree at the earliest possible time. In order to make these improvements, we will use a form of content-directed reasoning. ELEKTRA's rule compiler generates working memory elements which describe the structure and content of the rules in the system: we make use of this to make the improvements.

We begin with the case of reducing the number of matches. This concerns `MR1`. As it stands, this rule will match the condition-part of *every* rule of the correct form (i.e., will

---

6. Here, we intend the term 'degenerate' to denote the property of producing (in the limit) a potentially infinite sub-goal tree.

match the condition-parts of all rules whose condition-part contains exactly one element). The amount of work can be reduced by, quite simply, only matching those rules whose unique condition-element contains a relation symbol which is identical to that in the statement of the current goal. In order to write the rule, we need to make use of the output of the rule compiler (which is in working memory), and to use an ELEKTRA-supplied executable condition; we also need to extract the goal statement from the goal—this can be done by pattern-matching. The rule that we want is:

```
MR1a: if current-goal(goal(gs, i, ...)) and
         goal-id(g,i) and
         one-condrules(ocr) and
         predicate(gs, pred) and
         findall(rule,condition-mentions-relation(rule,pred),grules) and
         match-condition-against-wme(grules, gs, ri)
  then
         make-expanded(goal(gs,i,...));
         execute-instances-actions(ri)
```

where `predicate(r,p)` is an executable condition which expects `r` to be bound to a predication (an atom) and which binds `p` to the relation symbol of the predication, and where `condition-mentions-relation` is a compiler-generated working memory element.

Rule `MR1a` will match only those rules whose condition-element contains the same relation symbol as the sub-goal which is to be satisfied. It does this by first obtaining the relation symbol from the goal statement (via `predicate`) and then using `findall` to search working memory for those compiler-generated condition-elements which state that the relation symbol appears in the condition-part of some rule (the relation symbol may not appear within the scope of a negation—this is ensured by the definition of `condition-mentions-relation`). This solution exchanges a call to the matcher with a rule condition-part as argument for a call to the matcher inside `findall`. In order to match rules, the matcher must extract condition-elements: in this solution, the extraction step is omitted. In addition, the content-directed version given as `MR1a` is clearer because it explicitly states which rules are of interest: this change may well pay dividends if content-directed reasoning were to be applied to rule `MR1a`.

The second alteration requires that the actions of all applicable rules be examined to see whether they add a goal which matches an existing working memory element—in other words, which match a given. If there are such rules, they can be deleted from the rules to be fired, and the sub-goals with which their condition-parts match can be marked as `solved`. In order to implement this change, `MR1a` needs to be split into a rule which performs the content-directed step given above but which then adds the chosen rules to working memory (the chosen rules must have been matched before the addition is made, of course). A second rule then performs the search through the chosen rules, marking all of the solved sub-goals. The rest of the chosen rules (those which do not assert a goal which matches a given) can be executed in order to generate new sub-goals. The second of this pair of rules can replace `MR2` since it, essentially, performs the same task. We omit the definition of this rule because it is straightforward.

With these changes, we have defined two backward-chaining rule interepretes in the ELEKTRA rule formalism. Both of these interpreters can be executed by the ELEKTRA interpreter, thus converting ELEKTRA into a form of backward-chaining system. The backward-chaining interpreters that we have defined are capable of executing meta-rules, as well as object-rules, provided that they are in the format that we have defined for all backward-chaining rules. If the user wants to ensure that only object-rules are interpreted, the necessary change is simple: the condition-part which finds all one-condition rules needs to be augmented by the addition of a subset-forming condition: the subset required is just that subset of all the one-condition rules which reside at level one of the hierarchy. As we have seen, it is easier just to try all backward-chaining rules, but ELEKTRA provides facilities for making the interpreter more specific. Indeed, there is nothing to prevent the definition of a backward-chaining interpreter which interleaves the interpretation of object- and meta-rules on adjacent cycles.

The remarks of the last paragraph show that ELEKTRA can, in fact, support a backward-chaining interpreter of some considerable power. This is the case even though we have made restrictions on the form of the rules that can be interpreted by our backward-chaining system. It is possible to remove these restrictions, although it makes the definition of the interpreter considerably more complex: indeed, to our mind, the interpreter becomes rather inelegant. In fact, we consider the backward-chaining interpreters presented above to be rather ugly[7]. The interpreters, as they stand, cannot, of course, interpret themselves: this is because they are written as *forward-chaining* rules. To obtain a backward-chaining interpreter capable of interpreting itself would require us to write the interpreter as backward-chaining rules and then to execute it on the interpreter we defined above (or one very similar to it, perhaps one including OR branches). One reason for our dissatisfaction with the above interpreters is, quite simply, that we cannot, without additional effort, turn them into

7. This, it must be stressed, is a purely personal view. We can prove some rather nice results about the system—for example, termination properties. It is the general approach to backward-chaining that we find inelegant, not the process itself.

reflective interpreters as we can with all forward-chaining interpreters. Of course, if we took time to construct a backward-chaining interpreter that manipulated an AND/OR tree, we would be able to implement the interpreter in itself and thereby obtain the full power of the reflective capabilities of ELEKTRA. The interpreters that we have defined only operate on AND trees, while the interpreter contains implicit OR branches: the basic representation that we need is, therefore, absent.

This failing of the backward-chaining interpreter should not be seen as a limitation on ELEKTRA: we *can*, if we so wish, extend the interpreters of this section—to do so only requires more effort than we are prepared to make for the purposes of this exercise. The reflective backward-chaining interpreter can then be run on top of the ELEKTRA default loop, or can, itself, be run on an interpreter composed of forward-chaining rules—the choice is open. The second approach seems to be very much more interesting, although we have not followed it up.

A deeper problem concerns the way in which the backward-chaining interpreter is controlled. We have suggested that no control need be included in the system (in which case it makes non-deterministic choices) or that conflict-resolution might be used to choose between firing `MR1` and `MR2` at any point in the development of a solution tree. Another alternative is to place an extra level of rules above the backward-chaining interpreter: this additional level can perform a variety of control functions using forward-chaining rules. We leave this suggestion as an area for further investigation.

Before closing this section, there are some more points that seem relevant to the backward-chaining interpreters that we have described. In particular, there is the observation that the relationship between the backward-chainer and the other rules that they system might contain. In other words, it is feasible to incorporate mixed control in one system. We will develop this theme in the next section, but, before moving on, the reader is invited to reflect on these possibilities.

3. MULTIPLE INTERPRETERS

In this section, we consider the opportunities for incorporating more than one interpreter in a running ELEKTRA system. By the incorportation of multiple interpreters, we do not mean simply the inclusion of different interpreter rulesets, with only one of them being used. On the contrary, we intend that an ELEKTRA system can contain a number of different interpreters, each one of which is used for part of the time the system is running. That is to say, the execution of the interpreters is interleaved over the course of the system's execution. Right at the start of this section, we admit that the area of multiple, concurrently active interpreters is not an area we have investigated in any depth.

We can begin the discussion by considering the backward-chaining interpreters that we described in the last section. The intepreters were constructed from forward-chaining rules. In effect, they modify the default behaviour of ELEKTRA only when they are executing. In addition, they only operate on rules of a restricted kind. It is immediately clear that a system may contain different kinds of rules: those which satisfy the constraint on backward-chaining rules (i.e., the constraint that rules only contain a single element in their condition), and those which do not. The rules which do not satisfy the backward-chaining constraint will not be considered by the backward-chaining interpreter, and so are outside its jurisdiction, so to speak. In other words, rules which do not conform to the constraint cannot be interpreted by the backward-chaining interpreter and will be interpreted by some other mechanism. One can easily imagine a system in which there are some backward-chaining rules and other, ordinary rules, and in which the backward-chaining interpreter is executed by the default ELEKTRA loop. Since the non-backward-chaining rules are of a form suitable for execution by the default loop, they will, at some stage, be interpreted and will cause changes to working memory. In other words, the ELEKTRA system can (and probably will) interleave the execution of backward-chaining rules with the execution of normal, forward-chaining ones: backward-chaining rules will only be executed when the backward-chaining interpreter causes ELEKTRA's behaviour to deviate from the norm (which, when using the default interpreter, is forward-chaining). Otherwise put, ELEKTRA will backward-chain only when executing the backward-chaining interpreter.

Now, it is interesting to note that forward-chaining meta-rules can interact with backward-chaining rules. This is possible because the basic format of all rules is the same: the backward-chaining constraint merely restricts the length of the condition-part of backward-chaining rules (it also constrains the condition-element of all backward rules to be positive—we have not included negated condition-elements in our considerations for reasons of simplicity). Because the format is the same, backward rules pass through the rule compiler in the usual way and meta-rules can inspect their contents. This entails that what are intended to be backward rules can, in fact, be used in a forward direction, provided they are required to do so by meta-rules (also, the default interpreter will treat them as normal rules unless it is prevented from doing so). Thus, we can, essentially, make 'puns' on the interpretations of rules[8]. However, the primary importance of this observation is that meta-rules can interact with the backward-chaining interpreter in a variety of ways and can, therefore,

---

8. In the period since this paper was written, such 'punning' uses have taken on the appearance of an important point about representation — meaning as use.

influence its behaviour.

Under these circumstances, forward-chaining rules can be used to update the state of working memory by engaging in forward problem-solving. That is to say, ELEKTRA can be made to operate in a combined forward-backward mode simply by loading forward-chaining rules together with the backward-chaining interpreter and the backward rules. A similar situation arises when an interpreter other than the default loop is used: it, too, will dictate a behaviour for ELEKTRA which is distinct from that dictated by the backward-chainer.

An interesting question, at this point, is how to control the various behaviours when more than one interpreter ruleset is loaded. The solution to the problem of control with multiple interpreters depends upon a variety of factors, the main one of which being the state of the solution process. Thus we can immediately see that rules at a level higher than the interpreters can be used to determine the appropriate behaviour of the system. These latest rules can either be in the form of 'strategy' rules, or can form another level of interpretation. In either case, when a decision has to be made, the default behaviour will be to reflect to a higher level in order to decide what to do next.

In the case of a system which combines forward- and backward-chaining, if backward-chaining has just been used to reduce some goal and there are no working memory elements which match the terminal nodes of the sub-goal tree and there are no more rules which are capable of expanding the tree, it is apparent that the system should engage in a period of forward-chaining in order to generate new working memory elements. It would appear to be better to encode this strategy as a general control routine for mixing forward- and backward-interpreters: this is because a variety of factors enter into the decision. It can also be the case that a particular interpretative strategy needs to be interrupted and another resumed: this again, appears to have a demon-like quality which can be represented as a relatively loosely coupled set of interpretational rules. In the cases just cited, it has to be admitted that the distinction between interpreter and strategy is quite fine: in any case, one can view any interpreter which is implemented as a ruleset as being an implementation of a particular control strategy (more will be said on this in the next section).

Matters become rather more interesting when the top-level interpreter (say the one which decides when to go in a forward direction and when to go backwards) is reflective. We have seen that one aspect of reflection is the ability of an interpreter to interpret itself. If, for example, a reflective interpreter controls forward- and backward-chaining sub-interpreters (i.e., interpreters at lower levels of the hierarchy), it will be able to use them in making its own decisions. In this case, the interpreter reflects one level and invokes the forward- and backward-chainers on the problem of which to use next—forward- or backward-chaining. As we have repeatedly stated, ELEKTRA supports reflection of this kind: this entails that the use of two or more sub-interpreters can be made in coming to a decision about which interpreter to use next. Now this decision can be motivated by a great variety of factors, but one which has not been previously mentioned is the use of *definitional information* in interpreters.

Usually, as we have seen, conditions and actions are just treated as symbols, some of which just ‘happen’ to have procedural attachments. In ELEKTRA, it is perfectly possible to introduce definitions of the kind that we have employed in the informal meta-language we used to define various constructs above. This kind of information (which can be encoded as working memory elements or as production rules) can be accessed by a change to the condition-part interpreter that ELEKTRA uses. The default behaviour is simply to pattern-match conditions if they are not known to have procedural attachments: in the latter case, the default is to evaluate. This process can be altered and the matcher intercepted. What is required is a collection of meta-rules to perform condition-matching and action-execution. One simple example is as follows.

The new matcher is arranged so that, first, it has rules which check to see if there is a definition: if there is a definition for, say, a particular condition-element, the definition is expanded[9]. The expansion can lead to another phase of definition expansion, or, alternatively, to the use of the ELEKTRA default. If no definition exists, the default interpretation mechanism is used. A similar process can be defined for actions. The definition expansion process can involve arbitrary amounts of problem-solving activity—for example, a definition may cause backward-chaining in order to obtain a complete expansion.

The definitional mechanism requires that ELEKTRA reflect in order to match condition-elements and to execute actions: the whole having the flavour of a re-write process. The point of including a brief description of definition processing is that it can be incorporated into the general interpretational mechanisms used in a rule-based interpreter. This allows the interpreter to engage in inference not only on the objects which appear in rules, but also on the structures which comprise the definitions of these objects. In a relatively weak sense, ELEKTRA can make use of this facility in enriching the interpretational process with information of a more ‘semantic’ nature.

An abstract example of the use of definitions in the control of interpreters is as follows. A choice has to be made as to which interpreter to use next. A particular structure has been developed in working memory and the control interpreter is aware of its presence. The definition mechanism can be used to determine whether the structure can be decomposed, and

9. Although definitions have only a limited power, as shorthands, they save time. In all versions of ELEKTRA, it is possible to introduce new definitions at runtime, provided the appropriate LISP definitions are loaded — this, too, can be performed dynamically.

if so, whether it leads to new results. The reflective interpreter reflects and uses the definition. At the higher level, the interpreter has access to information which is contained in the definition and which, when combined with the interpreter's information about the various options it has, leads to the selection of a particular interpreter. With the decision made, the control interpreter reverts to its previous level, selects an interpreter and problem-solving continues.

By the abstract nature of the last paragraph, it can be seen that the proposals that we have made in this section are highly tentative and in need of further work. It does remain the case, though, that ELEKTRA is quite capable of engaging in the higher-level control processes which have been outline in this section: it is capable of engaging in this kind of processing *without modification*—this is because the kind of behaviour we have been discussing is implicit in ELEKTRA's structure.

4. CONCLUSIONS

This paper has investigated the construction of a number of different kinds of rule interpreter using the ELEKTRA rule formalism. When suitably encoded (and the encoding is almost the same as the pseudo-code we have given[10]), each interpreter can be executed as a set of ELEKTRA rules: in other words, ELEKTRA does not need modification in order to execute the complete range of interpreters we have presented. The default loop which is provided by the basic ELEKTRA interpreter can be replaced in order to execute the rulesets, but even this is not necessary (even though we recommend it because the default loop contains checks that are duplicated by the rulesets). All of this adds weight to the claim that ELEKTRA is a powerful system which is capable of self-interpretation (the rulesets presented in section 2.1 represent variations on the default interpreter loop).

In addition, we have briefly and speculatively considered the possibility that a number of different rule interpreters (each encoded as a ruleset) can be present and active within an ELEKTRA system at the same time. This capability is, if anything, enhanced by ELEKTRA's reflective properties because reflection can be used as a powerful control mechanism, as is well known. It must be stated that, at the moment, we have been unable to experiment with multiple interpreters: this is because of the limited memory available to the current implementation.

Although we have emphasised reflection in a number of places in this paper, and we have

10. A little more work is required to make these examples run in the CommonLISP version of the system: this is because the latest version does not include as many primitives as did the original Scheme one.

given a relatively detailed account of it elsewhere (Craig, 1991b), this paper has not concentrated upon the control issues raised by reflection. Instead, we have deliberately confined ourselves to the description of the various interpreters in an attempt to convince the reader of ELEKTRA's power. Reflection will be treated in a great more detail at a later date.

To conclude, we want to give a defence of our use of the term 'interpreter' in the context of this paper. To some, it might appear that we are dealing with different control strategies. To our way of thinking, the interpreter is the single item in a system which enables strategies to be implemented. Without an interpreter of a particular kind, it is not possible to use some strategies. Interpreters, in other words, are what make strategies possible. Now, what we have described above is a series of different general-purpose mechanisms which can be used to alter the default behaviour of the ELEKTRA interpreter. Each of these mechanisms can be used to implement other behaviours of a more limited kind. One example is the forward-chaining interpreter we defined in section 2.1: this, as we stated in section 2.2, can be used to execute our simple backward-chaining mechanism. This entails that the interpreter of section 2.1 can enable ELEKTRA to behave in a backward fashion: within the backward-chaining interpreter, we can employ a number of different strategies as the problem demands. The presence of a backward-chaining interpreter facilitates the articulation of those strategies (since forward-chaining is universal, we cannot claim that the backward-chainer makes them possible—it only makes the task of representing and implementing them easier). This example serves to illustrate the distinction which we see between strategy and interpreter: the distinction is one of generality. Also, in another sense, some strategies are special-purpose (in the sense that they apply to some problems and not to others): what we have done is to define general-purpose and domain-independent mechanisms with which to articulate domain-specific principles. This also applies to the content-directed interpreters: all we have done is to define general mechanisms which support content-directed reasonsing—we have said nothing about what is to comprise the content (indeed, we could couple content-directed inference with the definitional mechanisms we briefly discussed in the last section).

POSTSCRIPT

Since this paper was written (September, 1990), a second implementation of ELEKTRA has been completed. The new implementation is in CommonLISP and not Scheme: the reasons for this are that it is expected that the new implementation will be more portable than the old one and that the new implementation has made full use of hash-tables and CommonLISP structures (the result is a somewhat faster system). The CommonLISP version replaces the older one.

Like the Scheme version, the CommonLISP version of ELEKTRA was derived from the formal specification in (Craig, 1991a), but differs from the older version in a number of respects. Although the new version implements all the previous functionality, the treatment of primitives reflects a different approach. In the Scheme version, a library of some dozens of primitives (like `findall`) was provided; in addition, it provided a rich variety of set-manipulation operators. The CommonLISP version contains a smaller library. A significant part of the CommonLISP library contains interfaces to useful CommonLISP primitives and also makes the major routines of the ELEKTRA interpreter available to rules. The set operations that were part of the Scheme implementation are now absent: users must define their own set-manipulating operations in terms of the primitives supplied as part of CommonLISP and install them in the system (which is not a difficult process).

The diference between the libraries is compensated by the existence of far more 'hooks' into the basic interpreter code. The new version of the system gives the user far more access than did the old one, and it also makes visible some of the set-forming operations that appear in the Z specification. The Scheme version implemented the formal specification very faithfully: the new version, although still derived from the Z specification, it turns some structures that ought to be hidden (if one observes the letter of the Z schemata) into visible ones — in some cases, it turns the internal set-manipulations into separate schemata. The result of this is that the user now has access to more of the data structures that the interpreter maintains.

In addition, the interpreter provides more high-level functions to perform searches. For example, the `findall` primitive can now be defined in terms of a general-purpose interpreter function that performs matching on working memory[11]. The function takes an arbitrary LISP predicate as input and returns a list of all those elements that satisfy the predicate. In order to gain the full power of the interpreter's matching algorithm, this function can be called with a predicate which calls the working memory element matcher: this matcher executes user-defined condition elements. Similar functions are provided for the other main structures in the interpreter (in particular, sets of rule instances, the conflict set and the various structures that comprise production memory).

A final difference between the Scheme and CommonLISP implementations is that the CommonLISP one forbids rule actions directly to modify variable bindings: this appears to be more in keeping with the spirit of production systems (there is, however, a backdoor

11. This is just one of a number of system-defined procedures for searching and comparing working memory elements.

method for performing such dirty tricks).

The CommonLISP implementation does not use packages to any great extent. This gives the user freedom to choose and access any interpreter function or macro.

The result of these changes means that users have to do a little more work when constructing their own condition-elements and actions. The facilities that ELEKTRA now provides should make the process easier than previously. As applications are developend in ELEKTRA, a library of useful user-defined conditions and actions will be built up: this is intended to make life easier all round.

REFERENCES

Craig, 1991a. Craig, I.D., *The Formal Specification of ELEKTRA*, Research Report, Department of Computer Science, University of Warwick (*in prep.*), 1991.

Craig, 1991b. Craig, I.D., *ELEKTRA: A Reflective Production System*, Research Report 184, Department of Computer Science, University of Warwick, 1991.

Nilsson, 1982. Nilsson, N.J., *Principles of Artificial Intelligence*, Springer-Verlag, Berlin, 1982.

Scheme (1989) Clinger, W. and Rees, J. (eds.) *Revised$^{3.99}$ Report on the Algorithmic Language Scheme*, AI Laboratory, MIT, Cambridge, MA, 1989.