ELEKTRA: A REFLECTIVE PRODUCTION SYSTEM

Iain D. Craig

Department of Computer Science

University of Warwick
Coventry CV4 7AL
UK EC

ABSTRACT

This paper provides an overview of the ELEKTRA production rule interpreter. Although of an outwardly traditional nature, ELEKTRA differs from other interpreters by providing considerable support for meta-level inference. The paper describes the representations employed in the system, together with the interface to user-defined code. It also describes ways in which the control problem in production systems can be solved by increasing use of rules. Finally, the reflective properties of the system are introduced and examples are given. These examples centre around the ability of ELEKTRA to encode and to model rule interpreters of different kinds. It is shown that the interpreter on which the entire ELEKTRA system runs can be implemented as ELEKTRA rules. It is also shown that any rule interpreter which is written in the form of ELEKTRA rules can be interpreted by itself.

1. INTRODUCTION

This paper presents an overview of the ELEKTRA system. ELEKTRA is a production rule interpreter of an outwardly traditional kind. The interpreter provides extensive support for meta-rules and meta-level constructs. Although the interpreter allows users to run object-rules alone using a default interpreter loop similar in kind to that employed by, for example, OPS5 (Forgy, 1981), its full power is only appreciated when meta-rules are employed. It is tempting to say that ELEKTRA encourages the user to augment a problem-solving system by the use of meta-rules.

The ELEKTRA system is constructed in such a way that it supports reflective processing: that is, ELEKTRA programs (sets of production rules) can reason about themselves in a theory-relative, causally-connected fashion. In terms of production systems, this means that production rules are allowed to match other rules and to inspect and reason about their contents. The execution of some rules, in particular those rules which are assigned the status of meta-rules, causes changes to the system which are directly felt at the object level. This goes beyond the normal property of production systems by which the execution (or 'firing') of one rule alters working memory contents, thus enabling some other rules, while disabling others, even though this is one of the main ways in which the causal connection between levels is achieved.

In ELEKTRA, the causal connection can be achieved in a variety of different ways. For example, the selection of rules to match and/or fire based on their content can profoundly alter the behaviour of the system in the solution of a problem: the ways in which behaviour is altered are comparatively well-known (see, for example, Davis, 1980). Such content-based reasoning can be achieved in a number of ways in ELEKTRA. These ways range from the simple pre-compiled meta-rule approach to be found in, for example, CHRYSALIS (Terry, 1983), in CENTAUR (Aikens, 1983), or in Davis' TEIRESIAS (1976) at one end, to a runtime search for rules that mention some specific object or construction.

The concept of levels is important in ELEKTRA. In (Davis, 1980), Davis distinguishes between object-rules and meta-rules. Object-rules are just the ordinary rules that are used in the course of solving problems. Meta-rules, on the other hand, are rules that are able to reason about rules. This two-level organisation is extended in ELEKTRA to allow an infinite number of levels. Thus, in ELEKTRA, one can have meta-rules, meta-meta-rules, meta-meta-meta-rules, and so on, *ad infinitum*. Meta-meta-rules are rules which reason about meta-rules; meta-meta-meta-rules are rules which reason about meta-meta-rules.

The second important way in which the causal connection is achieved between levels is that ELEKTRA provides facilities which support the construction of rule interpreters as collections of rules. That is, a new rule interpreter can be added to ELEKTRA by defining a suitable collection of rules. These rules are then executed by the ELEKTRA system to provide a new interpreter. There is nothing mysterious in the way in which the rules are interpreted when the ELEKTRA meta-rule interpretation mechanism is understood: interpreter rules are merely meta-rules of a particular kind. What makes them special is the way in which they manipulate rules at lower levels.

At their simplest, interpreters are just meta-rules[1] that engage in condition matching and action execution; in addition, they also implement a form of conflict resolution, just so that the usual control loop found in production rule interpreters can be imitated. The conflict-resolution process itself can be represented as a collection of rules that are fired when needed in order to select a rule (or small collection of rules) to be fired next. This interpreter design is the most obvious, and the most conservative: it represents one point on the spectrum of possible rule interpreters. Meta-rules, like object-rules, are fully able to inspect and update working memory: this property allows the construction of situation-specific rule interpreters—in other words, rule interpreters can be chosen to suit the current needs of the problem-solving process.

The meta-level facilities in ELEKTRA that we have concentrated on so far have been related to the interpretation problem for production rules. Meta-rules have other functions in ELEKTRA: the conflict-resolution problem can be tackled using rules even when the rule interpreter is not implemented as a collection of rules. This use of meta-rules captures the use highlighted by Maes (1988b) and the use in SOAR (Laird, 1987). This latter use is similar to that described by Davis (1980), and perhaps, for this reason, conflict-resolution has dominated the popular interpretation of meta-rules. As the example of rule interpreters has indicated, there are other uses for meta-rules, some of which we will discuss below.

Since this paper is an overview of the ELEKTRA system, we will not spend any more time discussing the theoretical properties of meta-rules. Instead, we will move on to a description of ELEKTRA. The description covers the basic architecture (Section Two) and the representations which the system uses (Section Three). In Section Four, we describe the facilities which support meta-level processing in ELEKTRA; in Section Five, we briefly consider the ways in which rule interpreters can be constructed. Section Five is brief because we discuss the construction of interpreters elsewhere (Craig, 1991c). The last Section summarises the main points of the paper.

---

1. Here, we will use the term 'meta-rule' to refer to a rule that resides at any level above that of the object level. Thus, by this convention, a meta-meta-meta-rule will simply be referred to as a 'meta-rule'. Where confusion is likely to occur because of this convention, we will make explicit the level at which the rule occurs by using the correct number of 'meta's.

A formal specification of ELEKTRA has been undertaken and was reported by Craig (1991a). Every aspect of the implementation (apart from input/output and unification) was covered in that specification. The formal specification exercise assisted the entire project by making the control theory in ELEKTRA explict; it also helped by serving to identify the primary constructs in the ELEKTRA architecture.

To end this introduction, it is worth making an historical observation. What is now ELEKTRA is the result of a number of years' effort. The work began in 1983 with the construction of a very simple rule interpreter which was able to interpret production rules: this was a simple form of the rule-interpretation property of ELEKTRA. In 1987, an augmented version of that original interpreter was sketchily described in a research report (Craig, 1987). The current work extends and amplifies these previous activities.

2. ARCHITECTURE AND CONTROL

In this Section, the ELEKTRA architecture is described. The simplest characterisation of the architecture is that it is a forward-chaining production rule system of the OPS5 type. This characterisation is not, however, adequate. The ELEKTRA system is composed of two main components: a rule compiler and a runtime system. In this Section, we concentrate on the latter, but it is useful to have at least an outline understanding of the former.

The rule compiler[2] analyses each production rule that is loaded into the system. Strictly, it should be stated that the rule compiler is run before any rules are entered because the output of the rule compiler is essential to the efficient running of the runtime system. The role of the rule compiler is to analyse the components of rules. The rule format that is used by ELEKTRA is of a standard kind: rules contain a condition-part and an action-part. The condition-part is a set of condition-elements, each of which has an internal structure. Similarly, the action-part is a sequence of elementary actions each of which performs the kind of operation that is familiar from other production systems—addition of elements to working memory, deletion of such elements, printing messages, and so on. The rule compiler extracts the components of condition-elements and elementary actions and creates assertions which are eventually loaded into working memory. These assertions state where relations and constants, for example, are used by rules. By adopting this approach, it is possible for meta-rules to determine the contents of object-rules by simply engaging in a normal pat-

2. To be strictly correct, the rule compiler should be called the 'rule analyser' because all it does is analyse rules and record the results of this analysis. There is no transformation of rules into some simpler or more efficient form: indeed, the results of rule analysis swell working memory by a large factor—this is because information is redundantly stored. As will be seen below, the analysis process affords benefits that are far greater than the disadvantages of a large working memory.

tern-matching process. Since meta-rules are subjected to the analysis process, one can see that meta-rules can engage in reasoning activities with themselves as subjects: we will return to this below.

The rule compiler is the first main component of the ELEKTRA system. The results of its analysis are stored in working memory and can be used by rules during problem solving. Although the rule compiler is strictly outside of the basic architecture, the result it generates enable other parts of the system to function more effectively.

The basic ELEKTRA system is the runtime system. The runtime system is composed of the following parts:

- *Working memory*. This is where the results of problem-solving activity are stored. Working memory, as in all production systems, is a public database. Working memory can be inspected and updated by all rules in the system.
- *Production memory*. This is the store for production rules. In the current implementation of ELEKTRA, there are no provisions for the creation of new rules, so production memory can be thought of as read-only.
- *Instance memory*. This corresponds to the conflict set in other forward-chaining interpreters. When a production rule's condition has been satisfied by working memory contents, an instance is created for it and stored in instance memory: the instance records the identifier of the rule and the variable bindings that were generated by the matching process.

These three components are standard in forward-chaining systems. Working memory can be thought of as a short-term store, and production memory as a long-term one. In most systems, instance memory is present only as an implementational necessity: in ELEKTRA, it serves a functional role.

ELEKTRA's basic architecture is, in many ways, conventional and uncontroversial. As has been stressed above, the three main stores are to be found in the majority of forward-chaining systems. Where ELEKTRA is less conventional is in the way in which the interpreter loop is represented and in the way in which it is an open system. To begin with, the default interpreter loop is described.

The default interpreter loop implements a conventional *match-deliberate-act* cycle. The loop is adequate for executing some kinds of meta-rules, as will be explained. Just in case the reader is unfamiliar with the conventional interpreter cycle, it will briefly be explained.

The default loop is divided into three phases. During the match phase, the conditions of all the rules in production memory are matched against working memory to determine which rules have satisfied conditions. To perform the match, each condition-element is matched against all working memory elements. If the condition-element is negated, the matching process terminates whenever there is a matching working memory element or when all of working memory has been unsucessfully searched. In the former case, the match fails; in the latter it suceeds. If the condition-element is un-negated (positive), the matching process terminates either when all working memory elements have been unsuccessfully tested (in which case, the match fails), or when a working memory element matches against the condition element (a successful match). When all of the condition-elements in a rule's condition-part have been successfully matched, the condition-part is said to be *satisfied*, and an instance of the rule is created and stored in instance memory. If any of the rule's condition-elements fail, no instance is created.

The next phase is the deliberation phase. A procedure is executed to select one or more rules to be executed. The procedure operates on instance memory and can use a variety of techniques to determine which rule or rules to execute. Typically, these techniques rely upon purely syntactic or structural metrics: for example, select the rule with the longest condition. When the deliberation phase terminates, there should be at least one rule instance to execute. If there are no instances, the main loop terminates because there is an unsolvable control problem: there is nothing to do, in other words.

During normal operation, the deliberation process yields at least one rule whose action is to be executed. The action execution process involves executing each elementary action in a rule's action-part. Usually, elementary actions are considered to execute from left to right. The action-execution process requires the variable bindings that were generated during the match phase: these are stored in the rule instance records held in instance memory. The bindings are used to supply values to pattern-match variables that occur in elementary actions: the values supply a context within which actions are executed.

The act phase alters working memory contents by adding or deleting elements. The rewriting of elements is accomplished by first deleting and then adding elements. By executing actions, working memory changes and allows other rules to fire. At the end of the act phase, instance memory is emptied and the main loop repeats.

The assumption behind the default interpreter loop (an assumption common to all production systems) is that the alteration of working memory brings the system closer to the goal state. Eventually, one or more elements will appear in working memory that satisfy the goal that the system has been set. When this occurs, the entire system terminates.

Termination in ELEKTRA (normal termination, that is, and not the error terminations we have mentioned above) is achieved by executing a special elementary action, the *stop* action. The stop action sets a termination flag which halts the default interpreter loop. The stop action can appear in the action-part of any rule: when the rule's condition-part is satisfied by working memory, the stop action will be executed and processing will halt. The provision of a *stop* action is the most flexible way to achieve termination, and variants of it appear in most production rule interpreters. In a sense, termination can be thought of as being a knowledge-directed, or failure-directed, operation.

The default interpreter loop is adequate for systems that use only one level (only employ an object-level, in other words). It is also adequate for systems that engage in the most basic forms of content-directed reasoning. In the latter, meta-rules have conditions which inspect the working memory state and propose rules to match. Those rules are matched and instances are created and placed in instance memory. A conflict resolution procedure is then invoked to perform the deliberation part of the main cycle, and one or more rules are selected for execution. In a program such as this, meta-rules compete with object-rules for processor time: only any cycle, it is not guaranteed that a meta-rule will be fired (executed), or that one of the object-rules chosen by a meta-rule will be fired. It is, though, relatively easy to make such stipulations in ELEKTRA: each rule is tagged with the level at which it resides. It then becomes a straightforward matter for the conflict-resolution procedure to begin by selecting rules which reside at a particular level before engaging in the selection of which rule (or rules) actually to fire.

The default loop is moderately flexible, but this flexibility is only in terms of what the deliberation module, the conflict-resolution process, is able to do. A logical extension of the conflict-resolution scheme outlined at the end of the last paragraph is one which selects object-rules only on alternate cycles; on the other cycles, meta-rules are preferred if any are available (if there are none, then object-rules must be fired in order to make progress). This results in a 'cycle-stealing' interpreter[3] which fires object-rules, and then fires meta-rules. The meta-rules decide what to do next, and the object-rules do it. This contrasts with the original meta-and-object-rule interpreter which makes no distinctions between the levels at which rules resides. In that interpreter, the only difference between object- and meta-rules was in their behaviour: meta-rules had no special properties as far as the interpreter is concerned.

With ELEKTRA, we can actually obtain far greater flexibility than is possible with the default loop. The additional flexibility is achieved at the cost of having to alter the behaviour of that loop. In our experimental studies, we have found that it is possible to remove the relatively complex loop and to turn the interpreter into a very simple procedure. We now turn to these possibilities: they are what make ELEKTRA a little unusual.

3. The term 'cycle stealing' is suggested by the property that meta-rules 'steal' interpreter cycles from the object-rules which actually do the problem-solving work—meta-rules 'only' make control decisions (in fact, they can also do far more than simply making control decisions).

The simplest way of altering the default behaviour of the interpreter is to omit the call to the conflict-resolution module. As was stated above, this module is responsible for selecting one or more rules to fire (it implements the *deliberate* part of the default cycle). The call to the conflict-resolution module can be replaced by a call to a set of production rules: these rules (which are of a form identical to that of ordinary, problem-solving, rules) perform the conflict resolution process. What makes the conflict-resolution rules special is that they can directly inspect and update the contents of instance memory. This is made possible by the fact that ELEKTRA allows production rules to execute the primitives that the architecture provides for manipulating instance memory[4].

In ELEKTRA, rule instances and sets of rule instances are treated just like the other basic data types (atoms, lists and sets), which makes it possible to write rules that deal with rule instances as if they were just ordinary objects. In fact, rule instance manipulating procedures are part of the *standard* library of primitives that the architecture supports.

The way in which these special, instance-manipulating, rules are treated depends upon the way in which the rule-based conflict-resolution module is structured. Perhaps the simplest way is as follows. When the conflict-resolution module is called, it retrieves all those rules which deal with rule instances. This can be achieved in one of two ways:

(i) The instance-manipulating rules are accessed by name. In other words, the identifiers of the rules which manipulate instances are stored somewhere that the conflict-resolution module knows about. An ELEKTRA primitive is then called to perform the actual retrieval from production memory.

(ii) The instance-manipulating rules are accessed via a search of production memory. This search involves finding all those rules which mention one or more of the system-defined conditions that relate to instances or involves finding all those rules which mention an instance-manipulating action (an action that manipulates instance memory would also be a plausible search key).

Either of these options will yield those rules in the system that deal with instances. Once the rules have been gathered, their conditions can be evaluated by directly calling the system matcher. The matcher, in its default configuration, returns a set of rule instances (this is where the instances came from in the first place). When the instance-manipulating rules have been matched, therefore, there will be a new set of rule instances to deal with: in other words, there is another conflict-resolution problem to solve. In this context, there are again

---

4. The interface is, in fact, very simply provided because ELEKTRA contains powerful facilities for executing arbitrary code in rule conditions and actions. This combined with the procedures that are required by the basic architecture is how rule instances and instance memory can be manipulated by rules themselves.

options, some of which are more interesting than others.

The simplest option is to use a conflict-resolution module of a standard kind. The set of instances generated by the conflict-resolution rules will typically be smaller than that produced during general problem-solving, so a conflict-resolution procedure will terminate more rapidly on conflict-resolution rules. This option may seem a little puzzling: why should one go to the trouble of implementing conflict-resolution in terms of rules, only to use a conflict-resolution procedure after all? The answer is that a conflict-resolution module that is written in terms of production rules can engage in the whole range of activities that are normal for production rules while performing the conflict-resolution process. In other words, conflict-resolution *rules* can inspect working memory and determine which rules are most appropriate in a given context. This contrasts with the usual form of conflict-resolution process which deals almost exclusively with syntactic properties of rules—condition-length, last firing time, and so on. In addition, a conflict-resolution module which is written as a set of rules allows the possibility of inspecting rules in order to determine what to do next: in other words, content-directed reasoning can be directly applied to the problem of selecting the next rule or rules to fire. Finally, all of the advantages of the modularity of production rules can be applied to the conflict-resolution process (a similar argument was put forward in support of the rule-based scheduler in the AGE system (Nii, 1979; Feigenbaum, 1982)).

The implementation of an explicit conflict-resolution module in terms of rules can be carried one step further than simply matching and executing a set of conflict-resolution rules. This can be done without going to the next logical step which is to implement all control in terms of rules. The intermediate step is to retain the idea of implementing conflict-resolution as a collection of production rules, but removing the conflict-resolution procedure from inside this rule-based conflict-resolution module. In other words, the decision as to which conflict-resolution rules are fired rests again with a set of rules. The conflict-resolution module, in this case, relies upon the fact that, at some stage, there is only one conflict-rule that can be tested and fired: if the condition-part of this rule is not satisfied, it means that the conflict-resolution process is finished for the current interpreter cycle. The entirely rule-based approach depends upon the ability of some rules to reason about the content of other rules: in other words, this approach is based upon the presence of a hierarchy of conflict-resolution rules, some of which are able to reason about the contents of other rules (perhaps using explicit definitional or taxonomic information—such information, if provided, can, of course, be used in the rule-based conflict-resolution module described in the last paragraph). In this approach, the conflict-resolution module executes a simple loop which merely matches and executes a single rule. To see how this is possible, we need to see the more advanced, and to our mind, more powerful and elegant facilities which ELEKTRA contains[5].

Within an ELEKTRA rule, it is possible to access *all* rules in production memory. Because rules are tagged with their level, it is possible to retrieve all the object rules that are stored there, all the meta-rules, all the meta-meta-rules, and so on. Once rules have been retrieved, it is possible to call the matcher explicitly. The matcher can be directly called on a set of rules, say the object rules, or it can be called on the condition-part of a single rule; it can even be called on a single condition-element. In each case, it is necessary for an appropriate set of variable bindings to be supplied to the matcher (variable bindings are a primitive type in ELEKTRA and so can be created, tested and manipulated using system primitives). Depending upon the way in which the matcher is used, it will return a rule-instance or a set of variable-bindings: in the former case, it is not much of a step to see that this mechanism allows rules (at some level of the hierarchy) to engage in the explicit matching of rules and to store rule instances in working memory so that they can be inspected by conflict-resolution rules. This is in contrast with the conflict-resolution module approach that was described above, for, here, there is no need to employ a conflict-resolution module as part of the main interpreter loop. The reason for this is that meta-rules engage in all the processing of rule instances that were originally performed by the conflict-resolution module.

In addition to the features just mentioned, ELEKTRA also gives rules access to the action-execution primitives. This enables rules to execute the actions of other rules: in fact, action execution is just another rule action as far as the ELEKTRA interpreter is concerned. The interpreter allows rules to execute the actions of each rule in a given set of rule instances, execute all the elementary actions of a given rule, and to execute a single action (by the latter, we mean that ELEKTRA provides an action which takes another action as its argument and executes that second action). In the first case, a set of rule instances must also be supplied to the execute action as an argument. In the second case, there are two primitives: one executes the action-part of a rule instance, the other takes a rule and a set of variable bindings as arguments and executes the actions of the rule using the bindings. In the third case, an elemtary action and a set of bindings must be supplied as arguments and the action is executed in the context of the bindings.

In addition, ELEKTRA provides a number of ways in which to construct and manipulate sets. Thus, collections of rules can be treated as sets, or they can be treated as lists (lists in ELEKTRA are used to represent ordered sequences).

---

5. Our position is, quite simply, that there is no need to write conflict-resolution modules if one employs some of the more advanced features of ELEKTRA. In fact, it is possible to use the reflective capabilities of the system in order to perform all aspects of control.

The combination of the primitives for explicitly matching and executing rules with those for representing and manipulating sets and sequences allows the construction of a variety of control structures which are more or less similar to the structure provided by the default interpreter. In addition, the contents of *every* rule in an application system are described by working memory elements: this means that (meta-) rules need only search working memory to obtain information about the contents of other rules. Once rules have been found that satisfy some criterion, it is possible to match them using other rules; it is also possible to perform conflict-resolution (without employing an interpreter-provided procedure) and then to execute the actions of the rules which were selected for firing by the conflict-resolution rules. These facilities are in addition to those normally found in a production rule interpreter: in other words, a meta-rule can perform a context-sensitive control operation which requires reference to the current contents of working memory. This means that control rules can make decisions based upon the contents of working memory *as well as* the contents of the rules in the system: this is not an option that is usually available to conflict-resolution modules that are encoded as implementation procedures. Access to rule contents in ELEKTRA is, in any case, identical to access to any working memory element that was generated during the normal problem-solving cycle—there is no need to write rule conditions in terms of special, executable, conditions that search production memory.

The combination of facilities described above also allows rules to perform the full interpretation task: this is an issue to which we will return below in Section 4 where we discuss the possibilities for reflective processing in ELEKTRA (the issue will be considered in more detail in another paper). The combination of facilities also indicates how the second kind of conflict-resolution module can be structured: it depends upon the existence of meta-rules that select and execute other rules. The selection and execution processes can be directly performed using the primitives that have just been outlined. In order to ensure that only the simplest coded loop is required, the user has to provide one or two top-level rules that control the execution of all the other conflict-resolution rules. These top-level rules use the match and execute primitives, together with the set operations to provide a complete conflict-resolution process that is expressed entirely in terms of production rules[6].

One of the simplest forms that these top-level rules can take is as follows. We provide two rules: one has a condition which is satisfied when the conflict resolution process is complete; the other executes the rules that perform the conflict-resolution process proper. The conflict-resolution rules are structured so that when a final decision has been made, an

6. In the interests of speed, one would probably want to write a few procedures that would be called from the rules using the ELEKTRA execute primitives. This is not logically necessary, however, for ELEKTRA can provide everything that is needed, even though it might run a little slowly.

element is added to working memory which states that the decision has been made: this element (a 'signal') is used to satisfy the first of the top-level rules. The conflict-resolution module implements a simple match-deliberate-act loop in which the two top-level rules are matched and then subjected to a simple selection process, and then executed. The selection process involves examining instance memory and executing the instance that is to be found there. There should only be one instance because the two top-level rules should have complementary conditions. Under this scheme, it is important that the rules which perform the conflict-resolution process do not manipulate instance memory: if they need to store sets of instances, they should use working memory only. In this way, the conflict-resolution module's control loop is considerably simplified.

A variation on the above scheme can be used to implement the default forward-chaining interpreter's main loop. This is one of the more interesting aspects of ELEKTRA, and will not be discussed in this Section. A more detailed discussion can be found below in Section 3, and in Craig (1991c).

To conclude this Section, we briefly describe how a relatively simple form of content-directed reasoning can be performed. This form is related to that described by Davis (1980) and implemented by Terry as part of his Hierarchical Production System (1983). It is also related to the use of meta-rules in Aikens et al.'s CENTAUR (1983) system.

In its simplest form, content-directed reasoning involves the firing of meta-rules that mention object-rules by name. A general form for a content-directed meta-rule of this kind is:

If {some conditions} Then Try rules {a set of rule names}

where **{some conditions}** is some appropriate, legal, condition-part, and where **{a set of rule names}** is a set of identifiers of rules that are present in the system and which are considered by the rule-writer to be relevant to the situation described by the conditions. Rules of this form are created by the human rule-writer, it should be noted. The interest in this example is in showing how ELEKTRA can support the **Try** action. **Try** is usually considered to be an action which involves matching the condition-parts of those rules named by the elements of **{a set of rule names}**. The rules whose names are in this set and whose conditions are satisfied by the current state of working memory can be subjected to the conflict-resolution process and some (or, indeed, all) can be fired to change working memory. The problem comes in implementing the **Try** action.

Of course, **Try** is not inherently difficult to implement when one has the implementation language available, but most interpreters lack the facilities needed to make the implementation possible without considerable amounts of work. ELEKTRA is an exception to this, and the implementation and use of **Try** is particularly easy: in fact, **Try** can be implemented as a rule without too much trouble. The implementation as a rule takes us outside the scope of this Section, so we ignore it here. Instead, we will show, in rough terms, how to implement **Try** using only the primitives that ELEKTRA provides.

The nasty fact about **Try** is that it requires the matcher to be called from *inside* an action. For ELEKTRA, this causes no problems.

As has been stated, the matcher procedure can be called directly from any ELEKTRA rule. In other words, the matcher is always visible to applications: this makes the implementation of **Try** far easier than it would otherwise be. ELEKTRA's action-execution primitives are also visible, and can therefore be called from a procedure which implements **Try**. Two of the sources of difficulty are removed. The third main problem derives from the fact that rule instances are usually buried deep inside the production rule interpreter: as we have seen, instances are just another data type as far as ELEKTRA is concerned. Direct access to rule instances and to sets of rule instances makes the task of implementing **Try** considerably easier: it is not necessary in ELEKTRA to store instances in instance memory—this has been suggested above. Even if instances are stored in instance memory, the form of the **Try** primitive makes a search through instance memory easy (as long as there are no other instances of the rules being **Try**-ed currently in the memory).

The fourth difficulty with **Try** is that conflict-resolution is needed if **Try** is used in a forward-chaining system: this is where ELEKTRA's ability to incorporate new primitives is useful. ELEKTRA provides the user with facilities to incorportate any executable code in system and to install it as a primitive. When code has been installed as primitive, it can be executed from rules. Thus, if one really wanted a conflict-resolution procedure to be called from within **Try**, it can be done, even if **Try** is implemented as a rule. The user can also define a **Try** procedure and install it as part of the ELEKTRA system: the primitive does not have to be provided as part of the system distribution kit.

The outline for the implementation of a **Try** primitive is as follows:

(i) For each rule identifier in the set supplied as the argument to **Try**, retrieve the corresponding rule and match its conditions. If the match fails, repeat. If the match succeeds, store the rule instance generated by the rule in a temporary instance memory.

(ii) Either:

- (a) Place all of the rule instances in instance memory.
- (b) Call a conflict-resolution procedure on instance memory. This can be the standard one, or it can be one which, for example, removes all rule instances other than those whose rule is named in the argument to **Try** and stores them in a temporary instance memory, calls the standard conflict-resolution procedure and stores the

selected instance or instances in a temporary memory. The procedure must then replace all of the instances which it initially removed.

Or:

Use meta-rules to select one or more rules to fire;

Or:

Apply a conflict-resolution procedure to the instances in the temporary instance memory generated in step (i).

The meta-rule approach involves matching the meta-rules against the contents of the temporary memory. This is made possible by the fact that instances are first-class objects and can be handed directly to rules (this is done by binding the instances to a variable which is then used by the rule—this is the standard way to communicate between implementation code and ELEKTRA rules).

(iii) Either:

(a) Use meta-rules to execute the actions of the rule or rules chosen in step (ii); or

(b) Call the interpreter's action-execution procedures to execute the actions.

The reader should note that we have deliberately given hints as to how to implement **Try** in terms of meta-rules. This is so that it is possible to appreciate just how far the rule interpretation process can be integrated with other system functions. In a 'conventional' implementation of this primitive, it is to be expected that the options that employ procedural code would be adopted for reasons of speed.

Note also that the **Try** primitive can make use of existing parts of the rule interpreter: ELEKTRA is constructed as a 'toolkit' in addition to being a working program. In order to obtain the more interesting possibilities inherent in ELEKTRA, it is, in fact, essential to consider it as something closer to a library than to a simple program.

The **Try** primitive shows how, by means of the definition of relatively simple procedures (the **Try** procedure *is* quite simple, given the ELEKTRA primitives), the interpreter can be extended. **Try**, as we have described it, does something that has proved to be fairly difficult to achieve: it performs the conditional firing of rules. This has been cited by Cohen et al. (1989) as one of the failings of meta-rules:

> *Meta-rules give control a "one step at a time" or reactive flavor that, when implemented in a medical expert system, failed to capture some aspects of diagnostic expertise. For example, it was difficult to formulate meta-rules that had contingent actions on their right-hand sides—to say "do test A and if the answer is positive do test B otherwise do test C".*

The **Try** primitive that we have described above goes some way to achieving this property.

It conditionally performs actions from a set. The actions which are executed from the set depend upon the satisfaction of the conditions of the rules in which they are to be found. **Try**, as it stands, does not implement an **if...then...else** construct, but that was not the aim. A primitive similar to **Try** can be used to implement **if...then...else**, and the exercise is not difficult. Indeed, a general-purpose conditional of this form can be defined using ELEKTRA's primitives for matching sets of condition-elements: this allows an arbitrary condition to be represented.

The definition of **Try** involves the use of ELEKTRA's meta-level facilities. As will be seen in Section 4, these are comprehensive in scope. Before describing these facilities, it is necessary to understand the representations used by ELEKTRA: it is to the description of the representation that we now turn.

3. REPRESENTATIONS

The representations employed by ELEKTRA are, by and large, of a highly conventional nature. We spend time describing them for reasons of completeness, and because a knowledge of the representations used is helpful in understanding ELEKTRA's reflective capabilities.

The emphasis in this Section is on the representations used for condition-elements and for actions: overall, the representation is, quite obviously, production rules. The fine-structure of the representations used in the system are important to ELEKTRA's reflective capabilities because they enable the system to contain representations of itself. The Section is in two general sub-sections: the first, and longest, deals with the representation chosen for condition-elements. Sub-section one is longer because the condition-element representation is used in the construction of the elementary actions which rules execute when fired. The second sub-section discusses the range of actions that are provided by the basic system. We believe that the repertoire of actions is a full part of the representation system because, in a production system, as in many other computational systems, the concept of action is fundamental to the operation of the system (this is a point made by Batali, 1988, and one to which we will return at a later date).

3.1 Production Rules and Conditions

The overall representation, as we have stated, is that of 'production rules'. We briefly describe the format and interpretation of rules in order that the reader have a context within which to place the descriptions of the next two sub-sections. In ELEKTRA, rules have four components:

- A *rule identifier* (which must be unique). This is a symbolic identifier assigned to the rule by the user. Rule identifiers are used as search keys by a variety of ELEKTRA primitives. The rule compiler complains if it encounters two rules with the same identifier, but other than this no checks are made by the system.

- A *level tag*: an integer indicating the level at which the rule resides in the hierarchy of levels. A zero tag indicates that the rule belongs to the object-level; a tag with value greater than zero indicates that it is a meta-rule. For example, a rule with tag equal to three is a meta-meta-meta-rule.
- A *condition-part*. Condition-parts are ordered sequences of condition-elements. The interpretation of condition-parts is that they are conjunctions of positive and negative condition-elements. A positive condition-element is simply a condition-element (see below for details). A negative condition-element has the symbol *not* as its prefix, so the element[7]:

  (not (r a b c))

  is a negative condition-element, whereas:

  (r a b c)

  is a positive one. The empty condition-part denotes the value **true**: it represents the rule condition which is satisfied by all working memories.
- An *action-part*. Action-parts are composed of ordered sequences of elementary actions. It is the elementary action which is executed by the interpreter to cause changes to working memory, print messages, load files and so on. The format of elementary actions varies with the task they perform: some take a condition-element as argument, while others take a number of arguments of different types.

Positive condition-elements have a form which is identical to that of positive literals in resolution-based theorem-provers. Here, we will refer to the form of a positive condition-element as an *atom*. Negative condition-elements will be referred to as either *negated atoms* or *molecules*. The representation we have chosen deliberately follows the traditional logical form for atoms. The definition of an atom is as follows.

An *atom* is composed of a *relation symbol* followed by zero or more arguments, each of which must be a *term*. The syntax actually employed in ELEKTRA is Cambridge Polish. A relation symbol is represented as a LISP symbol. In the example above:

(r a b c)

is an ELEKTRA atom, and:

(not (r a b c))

---

7. In all examples, we use a LISP syntax—the current version of the sytem is written in LISP.

is a molecule (negated atom).

In ELEKTRA, a *term* is either a *constant*, a *variable* or a *functional term*. A constant is any legal LISP atomic value, but symbols do not need to be quoted (in other words, numbers, symbols, strings and **nil** are allowed as constants). ELEKTRA variables are represented as LISP symbols which have a '**?**' prefix: thus, **?output** is a valid ELEKTRA variable.

Functional terms are composed of a functor symbol followed by zero or more terms. A functor symbol is any legal LISP symbol. There is no restriction on the nesting depth of functional terms in atoms. ELEKTRA does not require that the set of relation and functor symbols be distinct (this would not make sense for, sometimes, it is necessary to employ object-language relation symbols as meta-language functor-symbols—see below, footnote 13 in particular[8]). An example functional term is:

(f a1 a2 a3)

The definitions of atom and term closely follow those in logic. The roles played by relations, constants, variables and functional terms is similar to that in logic.

The definitions given above are adequate for an elementary understanding of ELEKTRA condition-parts. A condition-part is just a sequence of atoms or molecules. The atoms or molecules in a condition-part are interpreted as forming a conjunction, and variables are understood to be universally quantified (functional terms can therefore be interpreted as being Skolem terms which represent existentially quantified terms). The molecules in a condition-part are interpreted using a version of the negation-as-failure rule, so the closed world assumption obtains. The interpretation of condition-parts is general: nothing that we have to say below contradicts it.

The atoms and molecules defined above can be thought of as being *matched* condition-elements. That is, the pattern-matcher[9] is used to determine whether they are satisfied by the contents of working memory. In order to obtain an increase in power and flexibility, ELEKTRA also allows two other kinds of condition-element, neither of which is tested using the pattern-matcher. In other words, ELEKTRA allows procedural attachments to be made to condition-elements. The idea is that, when the matcher encounters condition-elements of these kinds, it refrains from calling the pattern-matcher, but, instead, executes them using the variable bindings currently in force as a context. The two additional kinds of condition-element are:

---

8. Another way of saying this is that ELEKTRA uses a *reified* representation.

9. The pattern-matcher used by ELEKTRA is unification—this is because variables can occur in both patterns.

• User-defined relations.
• User-defined conditions.

Below, we will use the *term executable* condition-element to refer to both user-defined relations and user-defined conditions.

These are explained as follows. A *user-defined relation* is a condition-element very much like an atom. It is used for its truth-value. The matcher executes the procedure associated with its relation symbol. The values which are input to this procedure are derived from the bindings of the (pattern-match) variables which appear in the instance of the relation. If the relation is satisfied by the bindings, the matcher continues to the next condition-element; if it is not satisfied, the entire condition-part is not satisfied and the matching process terminates for that rule. An example will clarify this. Assume that the symbol **>** is bound to the LISP greater-than relation[10]. Assuming that the variables **?x** and **?y** are bound to **2** and **1**, respectively, the user-defined relation:

(> ?x ?y)

will be satisfied—true, in other words. The variable bindings are unaffected by the call to the attached procedure.

The second kind of user-defined condition-element is called a *user-defined condition*. The difference between user-defined conditions and user-defined relations is that the former can return values (indeed, this is their primary use), while the latter cannot. User-defined conditions also have relation symbols which are attached to procedures. In addition to determining a truth-value, a user-defined condition binds an additional variable. An example will make the difference between the two kinds of user-defined condition-element clear. Assume that the relation symbol **+** is bound to the LISP addition function and that **?x** and **?y** are bound to **2** and **4**, respectively. The user-defined condition:

(+ ?x ?y ?out)

when evaluated in the context of these bindings will cause **?out** to be bound to **6**.

Simply put, user-defined conditions are the characteristic relations of functions. The main use of user-defined conditions is the execution of functions from inside rule conditions. The set and list operations that ELEKTRA provides are attached to the LISP procedures which actually compute the desired object. When the procedure returns with a value, that value is always bound to the rightmost argument in the condition-element: thus, the value returned by the LISP addition procedure is bound to the pattern-match variable **?out**. The value re-

---

10. **>** is actually an ELEKTRA primitive which is bound to the appropriate LISP relation.

turned by user-defined conditions is always bound to the rightmost variable (which we call the *result* variable). In the current implementation[11], the result variable should not be bound: this is because ELEKTRA, at present over-writes the binding with the result of the function—when more memory is available, this restriction will be removed, thus affording the opportunity to perform tests in user-defined conditions.

By use of this general-purpose procedural attachment, ELEKTRA provides a great many executable condition-elements as part of the system. All list-manipulating functions and predicates are executable from ELEKTRA: this means that ordinary LISP-style list manipulation comes with the system. ELEKTRA also gives rules access to every LISP primitive. In addition, a number of set-manipulation operations and predicates are also provided as standard. This means that the operations over sets of rule instances, which we described in the last Section, are easily encoded as rules.

Users define executable condition-elements by defining appropriate LISP functions and then loading the function and declaring it to ELEKTRA. The declaration specifies what sort of executable condition-element it is: condition or relation. Thereafter, whenever the matcher encounters the LISP function identifier as a relation, the appropriate code is executed. At present, it is not possible to use executable conditions for their effect in terms: that is, functional terms are not evaluated. The reason for this is lack of space: this restriction, too, will be lifted in the future[12].

Before moving on, it is worth noting that the declaration procedures mentioned above are also available as executable code which can be called from within ELEKTRA rules. This permits users to write rules which conditionally load condition-elements (and actions) as and when they are required. In addition, most of the predicates and relations that are defined inside the ELEKTRA interpreter are defined as executable, so significant parts of the interpreter code can be called from rules (as can a number of procedures which are best considered as actions): the full implications of this will be drawn out in the next Section when we consider the reflective capabilities of ELEKTRA. One fairly immediate use of this facility is in the inclusion of the predicates, relations and operators that are defined inside the ELEKTRA interpreter for the testing and manipulation of variable bindings and rule instances: those facilities which are needed to implement conflict-resolution in the examples of the last Section are provided by procedural attachment of the kind just described.

3.2 Rule Actions

---

11. In Scheme on a 1Mbyte Apple Macintosh Plus.
12. Lambda expressions will be another, future, extension.

Like condition-elements, there are two fundamentally different kinds of rule action: those which are provided as part of the ELEKTRA system and those which are supplied by the user. Both kinds are, in fact, added to the interpreter by the same mechanism: that is, a declaration must which notifies to the system the name of the procedure which implements the action is notified to the system.

ELEKTRA provides most of the elementary actions which are familiar from other systems: for example, it contains elementary actions to add an atom to working memory, to delete one instance of a given atom from memory, to delete all instances of an atom from working memory. It also provides actions to print a message on the user's terminal and to perform a variety of file-handling tasks (for example, it contains actions to load working memory from a file, store working memory in a file, to add the contents of a file to working memory, and actions to load and store the contents of production memory). At present, the input-output actions are the least developed of those provided by the system: this is because (i) they are easy to implement, and (ii) the focus of attention in the project was on other things.

The basic ELEKTRA interpreter provides a number of actions which are not normally found in production systems. For example, there are actions to manipulate instance memory and to create rule instances from a rule identifier and a set of bindings. In addition, there are actions which cause the execution of rule actions. These actions allow the execution of all the actions in a set of rule instances, the execution of the action of one rule instance, the execution of the actions of a rule (this action takes a complete rule and a set of bindings as inputs), and there is an action which takes an action and a set of bindings and executes the action in the context generated by the bindings. This last set of actions allows ELEKTRA to support rules which interpret other rules: in other words, it allows the construction and use of meta-rules which implement rule interpreters.

3.3 The Rule Compiler and Representations

As has been stated above, the rule compiler is used to analyse all rules in the system. The rules are created in a file and are then submitted to the compiler for analysis. The output of the compiler is a set of working memory elements which can be loaded into the runtime system (this is optional for the user can decide not to load the contents of the file if they are not needed). The working memory elements which are generated by the rule compiler describe every rule in the file being analysed. Each run of the compiler generates a new file, so each set of working memory elements can be loaded independently of the others. In a crude fashion, this allows large rulesets to be constructed in an incremental fashion without incurring an enormous overhead at load-time.

The working memory elements that are generated by the rule compiler are in the format specified above in Section 3.1: in other words, the rule compiler generates a set of atomic formulae for each file it analyses. Because these atomic formulae are loaded into working memory, they can be matched by rules in the usual fashion. This gives rules access to descriptions of other rules (including themselves) and it also makes a runtime search of production memory unnecessary because the usual matcher can be employed to retrieve components of rules.

The main types of working memory elements that the rule compiler generates fall into three groups: elements dealing with condition-parts, elements dealing with action-parts and miscellaneous. We briefly describe them now.

1. Elements describing condition elements:
   (i) **condition-mentions-constant**. For each constant in each condition-element, one of these working memory elements is generated. The element denotes the fact that the constant appears in the condition-part of the rule named by the element. The general form[13] is:

   (condition-mentions-constant rulename constant)

   Where **rulename** is the name of the rule being analysed and **constant** is the constant itself. All constants are named by working memory elements of this kind: that is to say, each condition-element is fully expanded and all constants—including those which are the arguments of functional terms that are deeply nested—are extracted and working memory elements created for them.

   (ii) **condition-mentions-relation**. For each relation symbol in each condition-element, one of these working memory elements is generated. The working memory element denotes the fact that the relation symbol is to be found in the condition-part of the rule named by the element. The format is identical to that of **condition-mentions-constant**.

   (iii) **condition-mentions-functor**. A working memory element of this form is created for each functional term in each condition-element of a rule's condition-part. The format is identical to that of **condition-mentions-constant**. Every functional term is mentioned in the working memory image that results from analysis: that is, all functional terms, nomatter how deeply nested, are included.

---

13. Here, we need to point something out. The working memory elements described in this list belong to the *meta-level*. Each instance of an object-level object, such as a constant, should be *quoted* in the assertions. However, they are not—one can consider constants, relation and function symbols, atoms, molecules, and so on as being implicitly quoted. The reasons why quoting is not used are many and various: one is the fact that the rule compiler analyses meta-rules as well as object-rules. When analysing a meta-rule, an occurrence of a constant in one of these structural assertions should be quoted, but in a way different from the way [illegible] operators.

(iv) **condition-mentions-functor-symbol**. A working memory element of this form is created for each functor symbol that appears in each of the condition-elements of a rule's condition-part. Every functor symbol, nomatter how deeply nested, is included in the working memory image that results from rule compiler analysis.

(v) **condition-mentions-atom**. A working memory element is generated for each atomic formula that appears in the condition-part of each rule. The format is similar to that of the above element formats.

(vi) **condition-mentions-negated-atom**. A working memory element is generated for each negated atom (molecule) in the condition-part of each rule. The format is similar to that of the above formats.

(vii) **rule-conditions**. This working memory element type has the form:

(rule-conditions rulename condition-part)

where **rulename** is the name of the rule being analysed and **condition-part** is that rule's entire condition-part. One element of this type is generated for each rule. This is an example of the redundancy in the resulting ELEKTRA system: this redundancy is amply compensated by the reduced runtime taken to perform meta-level operations.

2. Working memory elements describing rule action components:

(i) **action-mentions-action**. This oddly named working memory type represents the fact that an elementary action is used in the named rule. The general form is:

(action-mentions-action rulename elem-act)

where **rulename** is the name of the rule and **elem-act** is the name of the elementary action. For example, assume that **rule1** adds an element to working memory in its action-part. To do this, it must use the **addwm** elementary action. The rule compiler will generate the following working memory element to record this:

(action-mentions-action rule1 addwm)

(ii) A set of element types similar to types (i) to (iv) in (1) above. The difference between the former and the latter is that the symbol **'condition'** is replaced by the symbol **'action'**: for example, for actions we have **action-mentions-constant** instead of **condition-mentions-constant**. The elements of these types denote the fact that the constant, relation symbol, etc., is mentioned in the action-part of the rule in question.

(iii) **action-mentions-action-clause**. An element of this type is generated for each action which takes an atom as argument. An example is:

(action-mentions-action-clause rule1 (p a b c))

where **rule1** is the name of the rule and **(p a b c)** is an atom.

(iv) **rule-actions**. This is analogous to **rule-conditions**, and contains all the actions in the action-part of the named rule.

3. Miscellaneous.

(i) **rule-level**. This has the form:

(rule-level rulename level-num)

where **rulename** is the name of the rule and **level-num** is the number of the level at which the rule resides. **level-num** must be a zero or positive integer: if it is not, the rule compiler generates an error.

The possibility that a rule system contains working memory elements that describe the rules themselves leads to the addition of a number of executable condition-elements: these deal with set-formation. In particular, it is often useful to find a set of working memory elements which satisfy some predicate or relation. In ELEKTRA, the **findall** relation has been included to perform this task. Its semantics are similar to the **findall** to be found in many Prolog systems (see, for example, Clocksin and Mellish, 1981). In the current implementation, **findall** is restricted to a single predicate or relation (to a single atom or molecule), but, in later versions, conjunctions will be permitted.

At present, the rule compiler does not generate working memory elements to record the presence of variables in condition- and action-parts. This was because it was considered that variables were, after all, only place-holders for values. The addition of variable assertions is an extremely simple matter given the way in which the rule compiler is structured, and may be included at a later data.

In addition, the rule compiler does not record the locations of the various objects that it encounters during its analysis. For the first version of the system, it was considered enough to extract the information held in rules in the form described above: it is always possible for the user to construct rules which look for the atom which contains a particular combination of constants. Also, given the limited memory available when the first version was constructed, it was considered that location data would cause an excessive memory load: when ELEKTRA is ported to a larger system, this data might be generated.

The rule compiler, as has been stated, generates working memory elements that represent assertions about the structure and contents of rules. These assertions are of supreme importance in providing ELEKTRA with reflective capabilities. It is to this issue that we now turn.

4. REFLECTION IN ELEKTRA

ELEKTRA, as the title of this paper states, is a *reflective* production rule interpreter. A reflective system is one that is able to represent and reason about its own structure, actions and representations. ELEKTRA, we claim, implements the first two of these properties: the third is only partially implemented (the reason for this being the lack of an adequate theory). It can be argued that ELEKTRA is, in any case, only capable of exhibiting procedural reflection (we do not entirely agree with this, but an account would be difficult and has not been fully formulated). In this Section, we describe some of the features of the ELEKTRA system which lead to reflective behaviour and we concentrate on procedural reflection. Readers who are unfamiliar with the concept of reflection should consult Smith (1982) or Maes (1988a).

At the very least, a procedurally reflective system should be capable of reasoning about the operations of its interpreter. As part of this, it should be able to access and manipulate its interpreter's data structures as first-class objects. This is required so that there be a causal connection between the running program and the representation of the interpreter which the program maintains: the representation of the interpreter must be connected to the actual interpreter—perhaps the simplest way of doing this is to provide the program (the object-level program, that is) with a representation of the interpreter which actually *is* the interpreter.

In 3-LISP (Smith, 1982), the system is organised so that the coded interpreter is at the top of a hierarchy of meta-...-meta-descriptions; the meta-level is a version of the interpreter written in 3-LISP. The close connection between the running program and the 3-LISP interpreter is achieved by having the 3-LISP interpreter execute the object-level: the 3-LISP interpreter is, itself, implemented in a more conventional dialect of LISP. Reflection occurs in 3-LISP whenever the running program at the object-level executes a reflective procedure: reflective procedures are able to execute meta-level programs, thus causing changes to the interpreter state. By this means, object-level programs are able to change the state of the interpreter and also to cause different parts of the interpreter to be exercised.

The situation in ELEKTRA is somewhat different because ELEKTRA, unlike 3-LISP, is a system in which assertions are possible. Smith, in his thesis (Smith, 1982), describes 3-LISP as 'merely algebraic' and states that, because it lacks quantification, its structures cannot have assertional force. ELEKTRA is determinedly assertional in nature. Because it is assertoric, ELEKTRA has to take a slightly different position on reflection than does 3-LISP, even though many of the implementation techniques are the same. In particular, ELEKTRA needs to maintain a *model* of the interpreter and the programs that run on it: this model must be updated at runtime whenever decisions are made. The model that ELEKTRA can maintain must contain assertions about various aspects of its operation[14].

Reflection in ELEKTRA can be effected in a way similar to that in 3-LISP and other reflective systems. The meta-level is the basic construct which supports reflection, and reflection exploits the power afforded by meta-level processing. As has been seen, ELEKTRA provides a great number of facilities to assist meta-level reasoning, perhaps the most immediately important of them being the rule analyses generated by the rule compiler. In addition, we have also seen how ELEKTRA provides the user with access to the interpreter's functions and data types: in particular, the system gives rules direct access to the interpreter. This direct access is what makes procedural reflection possible in ELEKTRA.

Procedural reflection in a rule-based system entails that the system be able to reason about why it took one decision as opposed to another. This ability imposes the requirement that the system have a theory or model of itself so that it can perform such reasoning. This reasoning has two sides: the first is to reason retrospectively on the decisions which led to a particular action; the other is to reason about which decision to take next. Both sorts of reasoning are possible in ELEKTRA, although the facilities to support the former are not in a developed state. The current version of the system concentrates on the second kind of reasoning: essentially, it concentrates on a form of reasoning about what to do next. In other words, ELEKTRA primarily concentrates on the problem of the control of inference.

In the SOAR system (Laird, 1987), this is also the focus of attention: reflection can occur in SOAR when there is more than one production rule that is applicable to the current goal. As we noted in Section Two, this is a version of the conflict-resolution problem, and we have seen how ELEKTRA rules can be used to perform conflict-resolution in a variety of ways. The conflict-resolution problem is also briefly examined by Maes (1988b), giving the impression that this is the only form of reflective behaviour that is reasonable in a rule-based system. On the contrary, ELEKTRA shows that a range of reflective behaviours is possible.

In one form, ELEKTRA can combine the conflict-resolution process with content-directed reasoning. This is done by employing meta-ules to implement conflict-resolution: these meta-rules are applied to a set of object-rule instances that is generated by meta-rules which examine the current solution state and find those rules which are applicable to it. As part of this process, it is necessary for the object-rules which have been found by the search rules to be evaluated: that is, for their conditions to be matched. This, too, can be performed by meta-rules.

The meta-rules that are needed to perform these actions can be summarised as follows (they fall into four sets):

---

14. The phrasing of this sentence is deliberate: the basic ELEKTRA interpreter does not perform reflection. All reflective processing must be explicitly included in the form of rules. The reason for this is that ELEKTRA is intended to be a general-purpose tool. In addition, by making reflection, in a sense, optional, it is possible to experiment with different approaches. Also, ELEKTRA's reflective mechanisms must be implemented as user-supplied rules because it enables the system to reflect on its own reflective processes.

(i) A collection of meta-rules which examine the current state and search production memory for applicable rules.

(ii) A collection of meta-rules which match the condition-parts of the rules found in (i) against working memory. The result of the match process is a set of rule instances which can be stored in working memory in preparation for set (iii).

(iii) A collection of meta-rules which select one or more instances to execute. The chosen instances can either be directly executed, or else can be stored in working memory ready for set (iv).

(iv) A collection of meta-rules to execute the rule instance or instances selected in set (iii) and, possibly, to record the results.

In sets (i) and (iii), it is necessary to have some rules which will fire when no rules are applicable or when no rules are selected: these rules might cause the system to revert to its default behaviour, for example (such a change in behaviour can be controlled by ELEKTRA rules).

The set of four sets just described needs a little augmentation in order to test for termination, but, in outline, they describe a complete rule interpreter which performs content-directed inference control. It must be emphasised that the four collections of rules together constitute a rule interpreter: in other words, they define a rule interpreter which is different from the default one provided by the ELEKTRA system. The rules themselves need to be controlled: this can be done by the default interpreter, together with appropriate use of signals, for example.

The default interpreter, however, need not be used. In fact the set of meta-rules described above (the union of the sets described in (i) to (iv)) form an adequate basis for *their own interpretation*. The construction of sets of ELEKTRA rules which are able to interpret themselves is a relatively simple exercise (once the problem is comprehended and once the ELEKTRA mechanisms are understood). In order to make this clearer, we give an example.

It is possible to construct an interpreter for a forward-chaining rule interpreter in terms of a set of ELEKTRA rules (in fact, two rules are needed). To see how to do this, we need to consider how a forward-chaining interpreter works.

Forward-chaining executes a basic match-deliberate-execute loop. During the first phase, all rules are matched against the contents of working memory: those rules whose condition-part is satisfied are stored in instance memory. During the second phase, one or more instances is selected for execution. In the final phase, the chosen rule or rules is executed and the loop repeats. The loop terminates when a special rule which sets the loop termination flag is fired. We can express this loop in terms of two rules as follows (the rules are written in pseudo-code so that the reader is insulated from the details of the ELEKTRA implementation):

MR1: if is-satisfied(goal) then stop!

MR2: if all-rules(r) and

```
        match-conditions-set(r,i) and
        longest-condition(i,ri)
    then
        execute-instance-actions(ri)
```

where:

- **stop!** is a system-provided action which sets the system termination flag.
- **all-rules** is an executable condition (provided by ELEKTRA) which binds all the rules in production memory to its (output) argument.
- **match-conditions-set** is an executable condition (provided by ELEKTRA) which matches the conditions of all the rules bound to its first argument. The instances created as a result of successful matches are bound to the second argument (**ri**).
- **longest-condition** is an executable condition (not provided by ELEKTRA). The condition expects a set of rule instances as its first argument. It sorts the rules in order of condition-part length (number of condition-elements) and binds the instance which has the longest condition-part to its second argument as a singleton set.
- **execute-instance-actions** is a system-provided action which executes the action-part of the rule instances in the set which is bound to its argument. The bindings needed by the individual action-parts are extracted from the corresponding rule instances.

This example shows how, in principle, it is possible to encode a simple rule interpreter as a pair of ELEKTRA rules. The example is a bit of a cheat because it requires the user to define the **longest-condition** executable condition-element: in other words, a little procedural representation is required to make the interpreter work as it stands. The reason for using **longest-condition** is that it makes the example easier to understand because both rules are apparently simple meta-rules. However, it must be noted that the interpreter defined above can, in fact, also execute meta-rules—this means that the level to which the two rules are assigned (they should be assigned to the same level) varies with what else is in the system[15].

The interpreter defined by these two rules would, if used, be a little stupid. It would tend to diverge from promising regions of the solution: this is because longest-condition does not, really, implement a particularly powerful conflict-resolution strategy. However, with some augmentation, it is possible to construct more complex and powerful interpreters based on the two rules given above. In a companion paper (Craig, 1991c), we give examples of interpreters which can be defined in terms of ELEKTRA rules: some of these interpreters provide behaviours which are radically different from the forward-chaining default of the ELEKTRA interpreter proper.

The ELEKTRA system loop required to implement the above two-rule interpreter is simple:

until stop? do

15. Strictly speaking, these two rules should be assigned to a level whose number is one greater than the highest level number of the rules which they are to interpret.

```
                match rules at level n;
                execute rule instance in instance-memory
        end until;
```

where **stop?** is an ELEKTRA primitive which is used to control the execution of the system (it tests a flag set by **stop!**). An assumption of this loop is that there will only ever be one rule instance in instance memory: this is reasonable because the condition-parts of **MR1** and **MR2** can never be simulataneously true. The system loop has to be told where to look for the rules it is to match: this is the point of the first call inside the loop. Inside the loop, no rules at a level other than that determined by **n** will be matched by the loop (although they may be matched and executed by rules that are interpreted by other rules).

Now, one of the more interesting properties of the interpreter that we have defined above it that it is able to interpret itself. In fact, the way we have set the rules up in the example will ensure this! The reason is that the first condition-element of **MR2** returns all the rules in production memory—this includes **MR1** and **MR2**, of course (note that there is absolutely no need to alter the main loop to obtain this behaviour). As an immediate consequnce, these two rules will interpret themselves. Sometimes, they will be executed, and sometimes they will not. Which rules will be chosen will depend upon what **longest-condition** finds in the instance set. If, instead of longest-condition, we had made the last condition-element of **MR2**'s condition-part **shortest-condition**, the chances that **MR2** would be executed by **MR2** would be increased (unless the rules in production memory all have extremely short condition-parts, which would be very strange for a realistic system). Even with **shortest-condition**, it is not necessarily the case that the two-rule interpreter will engage in viciously circular processing: in other words, it is not necessarily the case that **MR2** will spend all of its time matching, choosing and executing **MR2** with no progress being made.

The point of this example has not been to show a realistic interpreter, but to add weight to the claim that ELEKTRA can support rule interpreters that are written as rules. It also shows that rule interpreters can be interpreted by themselves. This, as it has been stated, does not seem particularly remarkable: after all, production rules are Post systems, and Post systems are Turing equivalent. Turing machines are able to model Turing machines, so production rules ought to be able to model production rules. This is indeed the case, and without this property, ELEKTRA could not have been constructed. What is of interest is that, even with the two-rule interpreter, the ELEKTRA interpreter is minimal. In fact, it turns out not to be the smallest ELEKTRA base interpreter which can be constructed: the simplest ELEKTRA interpreter matches and interprets one rule, and that rule makes no mention of the **stop?** condition.

Even though the two-rule interpreter is so simple, it has the capability to be as powerful as some of the early production rule interpreters (the one used by Waterman in his Ph.D.

had a conflict-resolution strategy that used simple rule order in production memory). The next step is, of course, to improve conflict-resolution. This can be done by matching, selecting and executing other rules: in other words, the condition-element which performs conflict-resolution can be removed and rule-evaluation can be put in its place. The introduction of a set of conflict-resolution rules involves more of a control problem because the interpreter must ensure that conflict-resolution rules are matched: this presents no theoretical problems, however. What it does suggest is that, as the interpreter becomes increasingly complex, it is necessary to go up one or more levels in the hierarchy and construct another, simpler, interpreter which controls the behaviour of the interpreter that we are really interested in. As this process continues, a level is reached where the interpreter has a complexity about the same as the two-rule one that we presented above. In a system which is structured along these lines, reflection passes not to one but through *many*, qualitatively different, interpreters until a simplest and most basic one is encountered (in fact, the simplest and most basic interpreter will be the ELEKTRA main loop, or base interpreter, and, as we have seen, when the control task is handed over to rules, the base interpreter becomes simpler because it has less to do).

The interpreters that we have seen so far (including the outline of the content-directed one) perform a kind of *compulsory reflection*: that is, user-supplied rules have no opportunity to *decide* whether the level above should be invoked. This is, quite clearly, because the interpreters are responsible for matching, deliberating and for acting. In other words, meta-rules interpret object-rules, so any time a decision has to be made, reflection occurs. It is, though, possible to engage in a rudimentary form of selective reflection in ELEKTRA: this is done in the obvious way by means of inter-level signals. Since this is so obvious, we will not discuss it any further.

It must be stressed that the interpreters that have been presented in this paper have been of an extremely simple kind. ELEKTRA provides very much more powerful facilities than those which have been used in the examples: for example, the contents of rule instances can be accessed and updated by rules, and this makes for extremely fine-grained and sophisticated control (on a par with the environment and continuation manipulations in 3-LISP). In a general overview document such as this, it is not possible to go into the details of such schemes—the reader would be expected to have a detailed knowledge of ELEKTRA's primitives which it is not possible to impart in a paper such as this. Also, we have not explored the various ways in which records can be kept of the various decisions that are made by interpreters: it is probably the case that there are as many ways as one can imagine.

Finally, it is worth noting that the reflective capabilities that we have discussed here, with the possible exception of the content-directed interpreter we outlined above, have been focussed on the interpretational aspects of ELEKTRA. That is, we have concentrated almost entirely on how to construct rule interpreters that are built from rules. We have seen how, in the limit, rules can interpret themselves, and this is an extremely interesting property because the main interpreter loop contains no interpretational code other than a couple of calls to very basic procedures which do no more than implement matching and action execution—in other words, no choices are made by the main loop[16]. In a companion paper to this (Craig, 1991c), we investigate rule interpreters in more detail and examine more properties of ELEKTRA.

5. CONCLUSIONS

In this paper, we have presented an overview of ELEKTRA, a production rule interpreter which allows reflective rules to be constructed. We have explained the basic control behaviour of production systems, and have shown how ELEKTRA allows control to be expressed in terms of rules. Next, we explained the basic representations employed in ELEKTRA, and then moved on to show how reflection is possible in the system. The discussion of reflection concentrated on the implementation of interpreters in the form of rules, and we showed how an interpreter can interpret itself. As part of that outline demonstration, we showed how the normally complex control loop of a production rule interpreter can be simplified to the extent that it contains no decision-making code: all decisions, including the most basic ones are made by the production rules that run on the interpreter. In a companion paper, we will investigate the construction of a number of rule interpreters of different kinds in ELEKTRA and we will amplify the discussion of the last Section (Craig, 1991c).

In its current form, ELEKTRA has been implemented in MacScheme on a 1MByte Apple Macintosh Plus. The memory on the machine is insufficient to allow extended experiments, but it is enough to implement some simple rule interpreters: an increase in memory will allow larger rulesets to be constructed and used in experiments. In the future, it is hoped that a version will become available in CommonLISP or in ML.

A formal specification of ELEKTRA has been prepared using the Z specification language (Craig, 1991a). The current implemetation was undertaken on the basis of that specification.

Much work remains to be done on ELEKTRA. In particular, the issue of declarative reflection must be tackled. Just as the formal specification of ELEKTRA gave us insights into the control theory, so, we hope, a formal semantics of ELEKTRA will help us in the task of in-

16. The main loop of BB1 (Hayes-Roth, 1985) is of equal simplicity: it repeatedly executes three knowledge source actions until told to stop. The BB1 main loop also makes no decisions of its own.

vestigating declarative reflection. A more immediately practical aspect of the work that is now to be done concerns hybrid reasoning: in the companion paper to this (Craig, 1991c), we will show how ELEKTRA can simultaneously execute more than one interpreter. Given the fact that the inclusion of other forms of representation (for example, frame systems or even theorem-provers!) in ELEKTRA is a relatively easy task, different forms of representations can be catered for, each of which poses its own, special, control problems. Even within a uniformly rule-based program, ELEKTRA allows radically different behaviours to be manifested at the same time. This is an area in which more work is required. A further area in which work is needed is to make ELEKTRA a single-calculus system: at present, it uses the dual-calculus approach.

REFERENCES

Aikens, J.S., Prototypical Knowledge for Expert Systems, *Artificial Intelligence Journal*, **20**, 1983.

Batali, J., Reasoning about Self-Control, in *Maes*, 1988a.

Clocksin, W.F. and Mellish, C.S., *Programming in Prolog*, Springer-Verlag, Heidelberg, Germany, 1981.

Cohen, P.R., DeLisio, J. and Hart, D., *A Declarative Representation of Control Knowledge*, COINS TR 89-15, Department of Computer and Information Science, University of Massachusetts at Amherst, Amherst, MA, USA, 1989.

Craig, I.D., *SeRPenS - A Production Rule Interpreter*, Research Report No. 97, Department of Computer Science, University of Warwick, 1987.

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

Craig, I.D., *Meta-knowledge and Introspection*, Research Report, *in prep.*, Department of Computer Science, University of Warwick, 1991b.

Craig, I.D., *Rule Interpreters in ELEKTRA*, Research Report, *in prep.*, Department of Computer Science, University of Warwick, 1991c.

Davis, R., *Applications of Meta-level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases*, Memo HPP-76-7, Computer Science Department, Stanford University, 1976.

Davis, R., Meta-rules: Reasoning about Control, *Artificial Intelligence Journal*, **15**, 1980.

Feigenbaum, E., Nii, H.P., Anton, J.J. and Rockmore, A.J., Signal-to-Symbol Transformation: HASP/SIAP Case Study, *AI Magazine*, **3**, 1982.

Forgy, C.L., *OPS5 User Manual*, Computer Science Department, Carnegie-Mellon University, 1981.

Hayes-Roth, B., A Blackboard Model for Control, *Artificial Intelligence Journal*, **26**, 1985.

Laird, J.E., Newell, A. and Rosenbloom, P.S., SOAR: An Architecture for General Intelligence, *Artificial Intelligence Journal*, **33**, 1987.

Maes, P. and Nardi, D., Meta-level Architectures and Reflection, Elsevier Publishers, Amsterdam, 1988.

Maes, P., Issues in Computational Reflection, in *Maes*, 1988.

Nii, H.P. and Aiello, N., AGE: A Knowledge-based Program for Building Knowledge-based Programs, *Proc. IJCAI-6*, 1979.

Smith, B.C., *Reflection and Semantics in a Procedural Language*, LCS Technical Report TR-272, MIT, Cambridge, MA, 1982.

Terry, A., *The CHRYSALIS Project: Hierarchical Control of Production Systems*, Memo HPP-83-19, Computer Science Department, Stanford University, 1983.