The Formal Specification of ELEKTRA

Iain D. Craig
Department of Computer Science
University of Warwick
Coventry CV4 7AL
UK EC

January 31, 1995

Preface

This document was written in September and October, 1990. Since then, I have intended to edit it and check the specification for correctness (elegance, typographical errors, as well as logical ones), in addition to proving many more results about the specification (the reader will notice that only one or two proofs are included). Unfortunately, time has never been on my side, and there has never seemed to be enough of it to spend on the manuscript.

The specification is now released as a Research Report for two reasons. Firstly, I want to be able to refer to it in other published material. Second, I have been making the specification available to people on a 'per demand' basis: it would appear far easier to make it generally available. It is, however, necessary to state the following caveat about the contents of this document:

> *The specification contained in this document may contain errors, both logical and typographical. To date, no attempt has been made to check the document for correctness in any but the most superficial manner.*

The specification was used as the basis of an implementation of the reflective production system. The system is described in [6, 7].

Chapter 1

Introduction

In a recent book [4], I argued that AI software should receive a formal specification in a way similar to more conventional programs. That book dealt with the blackboard architecture [5, 14, 2, 9] and with my own CASSANDRA architecture [3], and I made the remark that production systems [12, 1, 17] were a little too simple.

The current paper gives a specification of a production rule interpreter, and it is, indeed, rather simple. That having been said, the specification conveys all the benefits of a formal account: it is possible to reason about the software *without* having to go to the expense of implementing it. In addition, the current exercise is an opportunity for me to give a specification of an aspect of the blackboard and CASSANDRA architectures that turned out to be surprisingly difficult to account for in [4]: condition-matching. It has to be stated here that the blackboard architecture places different demands upon matching than does the production architecture: in blackboard systems, matching is a form of procedure call (in fact, a call on a user-defined procedure), whereas one has complete control over matching in a production system since it is simply "ordinary" structure matching.

Because the basic architecture *is* so simple, this paper concentrates on an extension of the "classical" production system: this is concerned with the provision of *meta-level* capabilities. This becomes important in the context of *reflective* or *introspective* systems, and ELEKTRA is intended to be one of these. The formal specification of an architecture that supports meta-level inference affords the benefit of providing an additional way of ensuring that what is being done makes sense. In addition, there is the methodological point that one wants to be absolutely clear about *what* the desired system is to do, and what facilities one is going to need: formal specification helps here, too.

The specification is in Z [15, 16]. There is no particular reason for this, except familiarity. An earlier version of the specification was undertaken in VDM [10, 11] but was discarded because it seemed a little unclear (probably my fault).

Chapter 2

Basic Production Architecture

2.1 Introduction

In this part, the basic specification of the production rule architecture is given. The specification covers all aspects apart from I/O. It also provides an account of terms and relations: this will be useful in part two when meta-level facilities are included. This section also covers initialization and provides alternative conflict resolution mechanisms.

2.2 Terms and Atoms

In this section, we specify the basic data types and operations that are needed to provide relations (atoms). Relations are used as basic condition elements and are the units out of which rule conditions are constructed. A relation is an atom in the logical sense: they are atomic formulae.

Relations are defined in the usual way from relation symbols and terms. We begin by defining terms and functions over them. To begin, we define the basic atomic types that we will need:

$$[CONST, VAR, FUN, REL]$$

where $CONST$ is the set of constant symbols, VAR the set of variable names, FUN is the set of function symbols, and REL the set of relation symbols.

We next define the $TERM$ type. This definition gives the structure of terms and is recursive (a Z free type):

$$\begin{aligned} TERM ::= \; & const\langle\langle CONST\rangle\rangle \mid \\ & var\langle\langle VAR\rangle\rangle \mid \\ & funct\langle\langle FUN \times \operatorname{seq} TERM\rangle\rangle \end{aligned}$$

(Note that there is no problem with the definition of *funct*: a sequence is always *finite*.) This definition provides the recursive structure of terms: a term can be a constant symbol (zero-ary function), a variable, or a function symbol applied to a sequence n terms (note that the arguments to a function are considered to be *ordered*—the ordering can be arbitrary, but must be canonical in the sense that two different orderings are considered to have different denotations).

We now define a number of functions over objects of type *TERM*. These functions have a variety of uses, particularly when the meta-level is given. The functions are defined *axiomatically*.

The function*farity* gives the arity of a term:

$$farity : TERM \rightarrow\!\!\!\!\!+ \ \mathbb{N}$$

$$farity(const\,c) = 0$$

$$farity(funct\,f\,t) = \#t$$

Note that *farity* is a *partial* function: we impose the condition that variables have no arity.

To define the next three functions, it is useful to have the following abbreviations:

$$VARS == \mathbb{P}\ VAR$$

$$FUNS == \mathbb{P}\ FUN$$

$$CONSTS == \mathbb{P}\ CONST$$

The function *varsinterm* returns the set of variables that are components of a given term:

$$varsinterm : TERM \rightarrow VARS$$

$$varsinterm(const\,c) =$$

$$varsinterm(var\,v) = \{v\}$$

$$varsinterm(funct\,f\,t) = \bigcup\{i : 1 \mathinner{..} \#t \bullet varsinterm(t(i))\}$$

The function *funsymsinterm* returns the set of function symbols that are in a term:

$$funsymsinterm : TERM \rightarrow FUNS$$

$$funsymsinterm(const\,c) =$$

$$funsymsinterm(var\,v) =$$

$$funsymsinterm(funct\,f\,t) = \{f\} \cup \bigcup\{i : 1 \mathinner{..} \#t \bullet funsymsinterm(t(i))\}$$

constsinterm is similar to *funsymsinterm*, but returns the set of constant symbols:

$$\begin{array}{|l} constsinterm : TERM \rightarrow CONSTS \\ \hline constsinterm(const\,c) = \{c\} \\ constsinterm(var\,v) = \\ constsinterm(funct\,f\,t) = \bigcup\{i : 1 \mathrel{..} \#t \bullet constsinterm(t(i))\} \end{array}$$

Next, we define a type to represent a sequence of objects of type *TERM*. This type is used to represent the arguments to a relation.

$$STERMS == \mathrm{seq}\ TERM$$

This type should not be confused with

$$TERMS == \mathbb{P}\ TERM$$

which is a *set* of terms (it is used below).

We can now define the type for relations:

$$RELN == REL \times STERMS$$

The basic functions that we define over *RELN* are as follows. First, we define a function which will return the relation symbol of a relation:

$$\begin{array}{|l} relsym : RELN \rightarrow REL \\ \hline \forall\, r : RELN \bullet \\ \quad relsym(r) = first(r) \end{array}$$

Next, we define the function to return the arguments of a relation:

$$\begin{array}{|l} args : RELN \rightarrow STERMS \\ \hline \forall\, r : RELN \bullet \\ \quad args(r) = second(r) \end{array}$$

Finally, we define the *arity* function:

$$\begin{array}{|l} arity : RELN \rightarrow \mathbb{N} \\ \hline \forall\, r : RELN \bullet \\ \quad arity(r) = \#args(r) \end{array}$$

For the next function, we need to assume that the relation has an arity strictly greater than zero:

$$\begin{array}{|l} aterm : RELN \times \mathbb{N} \rightarrow TERM \\ \hline \forall\, r : RELN;\ i : \mathbb{N} \bullet \\ \qquad aterm(r, i) = args(r)(i) \end{array}$$

aterm returns the n^{th} argument of the relation.

For the following definitions, we must assume that r always has arguments. The first function, *relvars* returns the variables that are the arguments to a relation:

$$\begin{array}{|l} relvars : RELN \rightarrow VARS \\ \hline \forall\, r : RELN \bullet \\ \qquad relvars(r) = \\ \qquad\qquad \bigcup\{i : 1 \,..\, arity(r) \bullet varsinterm(aterm(r, i))\} \end{array}$$

relconsts returns the constant arguments:

$$\begin{array}{|l} relconsts : RELN \rightarrow CONSTS \\ \hline \forall\, r : RELN \bullet \\ \qquad relconsts(r) = \\ \qquad\qquad \bigcup\{i : 1 \,..\, arity(r) \bullet constsinterm(aterm(r, i))\} \end{array}$$

relfuncts returns all the function symbols that are arguments:

$$\begin{array}{|l} relfuncts : RELN \rightarrow FUNS \\ \hline \forall\, r : RELN \bullet \\ \qquad relfuncts(r) = \\ \qquad\qquad \bigcup\{i : 1 \,..\, arity(r) \bullet funsymsinterm(aterm(r, i))\} \end{array}$$

Next, we define three relations. They are all variations on the mentioning relation: each is true if and only if its second argument (of whatever type) appears amongst the arguments of the relation.

$$\begin{array}{|l} relmentionsconst : RELN \leftrightarrow CONST \\ \hline \forall\, r : RELN;\ c : CONST \bullet \\ \qquad c \in relconsts(r) \end{array}$$

$$
\begin{array}{|l}
relmentionsfunsym : RELN \leftrightarrow FUNCT \\
\hline
\forall r : RELN; f : FUNCT \bullet \\
\quad f \in relfuncts(r)
\end{array}
$$

The next relation is a predicate: it is true whenever its argument has a non-zero arity:

$$
\begin{array}{|l}
hasargs : RELN \\
\hline
\forall r : RELN \bullet arity(r) > 0
\end{array}
$$

The function *relterms* returns all the subterms of a relation:

$$
\begin{array}{|l}
relterms : RELN \rightarrow TERMS \\
\hline
\forall r : RELN \bullet \\
\quad relterms(r) = \\
\qquad \bigcup\{i : 1 \mathbin{..} arity(r) \bullet subeterms(aterm(r, i))\}
\end{array}
$$

Before defining *subterms*, we define an extra function which will be of help later:

$$
\begin{array}{|l}
sseqterms : STERM \rightarrow TERMS \\
\hline
sseqterms(\langle\rangle) = \\
\\
sseqterms(h \frown t) = \\
\quad subterms(head(h \frown t)) \cup sseqterms(tail(h \frown t))
\end{array}
$$

We can prove a result about *relterms* and *sseqterms*. For all r,

$$\vdash relterms(r) = sseqterms(args(r))$$

We can now define *subterms*. We give two definitions and and an outline proof of their equivalence.

$$
\begin{array}{|l}
subterms : TERM \rightarrow TERMS \\
\hline
subterms(constc) = \{(constc)\} \\
\\
subterms(varv) = \{(varv)\} \\
\\
subterms(functft) = \\
\quad \{(functft)\} \cup \bigcup\{i : 1 \mathbin{..} \#t \bullet subterms(t(i))\}
\end{array}
$$

$$
\begin{array}{|l}
subterms_1 : TERM \rightarrow TERMS \\
\hline
subterms_1(const c) = \{(const c)\} \\
subterms_1(var v) = \{(var v)\} \\
subterms_1(funct f t) = \{(funct f t)\} \cup sseqterms(t)
\end{array}
$$

Theorem 1 *For all t,*

$$\vdash subterms(t) = subterms_1(t)$$

PROOF (Outline—proof by induction.) If $t =$ *(const c)*, $subterms(t) = subterms_1(t)$. If $t =$ *(var v)*, $subterms(t) = subterms_1(t)$.
Finally (induction step), we have $t =$ *(funct f t)*, and we need to prove that:

$$sseqterms(t) = \bigcup\{i : 1 \mathrel{..} \#t \bullet subterms(t(i))\}$$

There are three cases.
Case 1. $\# t = 0$, so

$$
\begin{aligned}
& sseqterms(t) = \\
& \quad sseqterms(\langle\rangle) \\
& \quad = \\
& \quad = \bigcup\{i : 1 \mathrel{..} 0 \bullet subterms(t)\} =
\end{aligned}
$$

Case 2. $\#t = 1$, so:

$$
\begin{aligned}
& sseqterms(t) \\
& \quad = sseqterms(\langle t \rangle) \\
& \quad = \bigcup\{i : 1 \mathrel{..} 1 \bullet subterms(t)\} \\
& \quad = subterms(t)
\end{aligned}
$$

Case 3. $\#t > 1$, so:

$$
\begin{aligned}
& sseqterms(t) \\
& \quad = sseqterms(head(t) \frown tail(t)) \\
& \quad = \bigcup\{i : 1 \mathrel{..} \#t \bullet subterms(t(i))\} \\
& \quad = subterms(t(1)) \cup \{i : 2 \mathrel{..} \#t \bullet subterms(t(i))\} \\
& \quad = subterms(head(t)) \cup \{i : 2 \mathrel{..} \#t \bullet subterms(tail(t))\}
\end{aligned}
$$

□

We can use either version of the subterm function, and we will just write *subterm* without bothering to be clear about which version we imply.

Finally, we define:

$$
\begin{array}{|l}
relhasterm : RELN \leftrightarrow TERM \\
\hline
\forall\, r : RELN;\ t : TERM \bullet \\
\qquad t \in relterms(r)
\end{array}
$$

2.3 Matching

The core of a production system is its match routine. The matcher can be a routine like the RETE algorithm [8], the TREAT algorithm [13] or unification. Here, we do not specify what the particulars of the matcher should be: instead, we merely define its functionality.

In order to specify the matcher, we need to define:

$$BINDINGS == VAR \nrightarrow TERM$$

This type represents the variable bindings that are manipulated by the matcher. The matcher will be considered to be a *relation*, and will relate two sets of bindings.

To begin the specification, we define:

$$
\begin{array}{|l}
tmatch : (STERMS \times STERMS \times BINDINGS) \leftrightarrow BINDINGS
\end{array}
$$

tmatch is responsible for matching two terms. It also relates two sets of bindings: one which represents the input bindings (the bindings before the match has been performed) and the output bindings (which contains the results of matching).

We define the matcher as:

$$
\begin{array}{|l}
match : (RELN \times RELN \times BINDINGS) \leftrightarrow BINDINGS \\
\hline
\forall\, r_1, r_2 : RELN;\ b : BINDINGS \bullet \\
\qquad (\exists\, bdgs : BINDINGS \bullet \\
\qquad\qquad (relsym(r_1) = relsym(r_2) \wedge \\
\qquad\qquad\qquad tmatch(args(r_1), args(r_2), b, bdgs)) \vee \\
\qquad\qquad (relsym(r_1) \neq relsym(r_2) \wedge \\
\qquad\qquad\qquad \mathrm{dom}\, bdgs =))
\end{array}
$$

The matcher relation *match* is defined only for objects of type *RELN*. All objects of this type are *positive*—that is, they are un-negated. In the conditions of production rules, we need both positive and negative (negated) relations, so we define the following type:

$$ATOM ::= pos\langle\langle RELN\rangle\rangle \mid$$
$$\qquad neg\langle\langle RELN\rangle\rangle$$

ATOM is the type from which we will construct rule conditions below. We define two predicates over *ATOM*:

$$\begin{array}{|l} negatedAtom : ATOM \\ \hline \forall\, a : ATOM \bullet \\ \quad (pos a) = false \\ \quad (neg a) = true \end{array}$$

and:

$$\begin{array}{|l} positiveAtom : ATOM \\ \hline \forall\, a : ATOM \bullet \\ \quad positiveAtom(a) \equiv \neg\ negatedAtom(a) \end{array}$$

With *ATOM* defined, we can now give the specification of the match relation for objects of type *ATOM*. This relation will be the one that we use in specifying the match process for rule conditions.

$$\begin{array}{|l} matchatom : (ATOM \times RELN \times BINDINGS) \leftrightarrow\ BINDINGS \\ \hline \forall\, a : ATOM;\ r : RELN;\ b_1 : BINDINGS \bullet \\ \quad (\exists\, b_2 : BINDINGS \bullet \\ \qquad (matchatom(pos(r_1), r_2, b_1, b_2) \equiv \\ \qquad\quad match(r_1, r_2, b_1, b_2) \wedge\ b_1 = b_2) \vee \\ \qquad (matchatom(pos(r_1), r_2, b_1, b_2) \equiv \\ \qquad\quad match(r_1, r_2, b_1, b_2) \wedge \\ \qquad\quad \#\,\mathrm{dom}\, b_2 > \#\,\mathrm{dom}\, b_1) \vee \\ \qquad (matchatom(neg(r_1), r_2, b_1, b_1) \equiv \\ \qquad\quad match(r_1, r_2, b_1,))) \end{array}$$

It is the case that the output bindings can contain the same variable bindings as the input bindings (first disjunct), or there may be additional bindings on output (second disjunct). The third disjunct represents the case in which the atom is negated: in this case, the match is successful if and only if the match of the atom's relation fails (represented by returning the empty bindings—see *match* above).

It should be noted that *matchatom* matches an *ATOM* with a *RELN*. We assume that all items in working memory (see below) contain objects of type *RELN*: in other words, we do not allow negated relations to appear in working memory (this is a variation on the so-called "closed-world" assumption).

2.4 Actions

In a production system, actions cause changes to the state of working memory. We begin by defining the type for action:

$$Act == \{Add, Del, Print, Read, \ldots\}$$

We do not give a complete list of possible actions. Objects of type Act are, in effect, action tags which tell the interpreter what to do. We define the $Action$ type and make the last sentence clearer:

$$Action == Act \times RELN$$

The component of type Act is used to direct the interpreter: it is a kind of op-code.

We define two functions which access objects of type $Action$ and return their components:

$$\begin{array}{|l}
actofaction : Action \rightarrow Act \\
\hline
\forall\, a : Action \bullet \\
\qquad actofaction(a) = first(a)
\end{array}$$

and

$$\begin{array}{|l}
relofaction : Action \rightarrow RELN \\
\hline
\forall\, a : Action \bullet \\
\qquad relofaction(a) = second(a)
\end{array}$$

We can define all the term accessing functions that we require by composing the functions defined for $RELN$ and $ATOM$ with the two functions we have just defined. The definitions are simple and we omit them here.

2.5 Working Memory

In this section, we specify the *working memory*. This is the central database that is maintained by the production system. All productions are matched against working memory during the normal execution cycle. If we add meta-level features, working memory remains the central database, although productions need not be entirely matched against its contents.

In order to specify working memory, we need to define a type that will be of use later:

$$[RULEID]$$

This type represents the set of identifiers that the user can assign to rules in the system (identifiers are assigned before the rules are added to the system).

Working memory is composed of *elements*. Working memory elements are defined by:

$$WMElem == RELN \times (RULEID \times \mathbb{N})$$

Each working memory element contains the following components:

1. a relation (object of type *RELN*) which represents the information that the rules have stored in working memory;

2. the identifier of the rule which placed the element into working memory (the component of type *RULEID*), and

3. the time at which the element was added to working memory (its creation time—this is the component of type $\mathbb{N}$).

Time in the production system is measured by counting the interpreter cycles: the time tag in working memory elements is, therefore, the number of the cycle on which the element was created.

We define a number of functions for the manipulation of working memory elements.

$$\begin{array}{|l}
mkwmemelement : RELN \times RULEID \times \mathbb{N} \rightarrow WMElem \\
\hline
\forall\, r : RELN;\ rid : RULEID;\ tm : \mathbb{N} \bullet \\
\quad mkwmemelement(r, rid, tm) = (r, (rid, tm))
\end{array}$$

mkwmemelement is the function which creates an object of type *WMElem* from its components.

$$\begin{array}{|l}
wmclause : WMElem \rightarrow RELN \\
\hline
\forall\, w : WMElem \bullet \\
\quad wmclause(w) = first(w)
\end{array}$$

wmclause returns the relation stored in a working memory element.

$$\begin{array}{|l}
wmsysdata : WMElem \rightarrow (RULEID \times \mathbb{N}) \\
\hline
\forall\, w : WMElem \bullet \\
\quad wmsysdata(w) = second(w)
\end{array}$$

$wmsysdata$ returns the information about the creation of an element. This information is placed in the working memory element by the system, hence the name of the function. This function is useful in its own right, and for defining the two following functions.

$$
\begin{array}{|l}
wmcreator : WMElem \rightarrow RULEID \\
\hline
\forall\, w : WMElem \bullet \\
\quad wmcreator(w) = first(wmsysdata(w))
\end{array}
$$

$$
\begin{array}{|l}
wmcreationtime : WMElem \rightarrow \mathbb{N} \\
\hline
\forall\, w : WMElem \bullet \\
\quad wmcreationtime(w) = second(wmsysdata(w))
\end{array}
$$

We can now move on to the definition of operations over the working memory. We begin with the definition of the schema that represents working memory:

$$
\begin{array}{|l}
\textit{WMEM} \\
\hline
elems : \mathbb{P}\; WMElem \\
\hline
\end{array}
$$

We assume that $\Delta WMEM$ and $\Xi WMEM$ are defined by convention.

An initialized working memory contains no elements, so we have:

$$
\begin{array}{|l}
\textit{InitWMEM} \\
\hline
WMEM \\
\hline
elems = \\
\hline
\end{array}
$$

The next operation adds a working memory element to the working memory:

$$
\begin{array}{|l}
\textit{AddWMEM} \\
\hline
\Delta WMEM \\
e? : WMElem \\
\hline
elems' = elems \cup \{e?\} \\
\hline
\end{array}
$$

We also need an operation to remove an element from memory:

$$
\begin{array}{|l}
\textit{DelWMEM} \\
\hline
\Delta WMEM \\
e? : WMElem \\
\hline
elems' = elems \setminus \{e?\} \\
\hline
\end{array}
$$

The next three schemata access information contained in working memory.

$$
\begin{array}{|l}
\hline
\textit{ElemsCreatedAt} \\
\Xi \textit{WMEM} \\
tm? : \mathbb{N} \\
els! : \mathbb{P}\ \textit{WMElem} \\
\hline
\forall\ wme : \textit{WMElem} \mid \\
\qquad wme \in elems \land \\
\qquad tm? = wmcreationtime(wme) \bullet \\
\quad wme \in els! \\
\hline
\end{array}
$$

The predicate is equivalent to:

$$
\begin{array}{l}
\{wme : \textit{WMElem} \mid wme \in elems \land \\
\qquad tm? = wmcreationtime(wme) \bullet \\
\quad wme\}
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ElemsCreatedSince} \\
\Xi \textit{WMEM} \\
tm? : \mathbb{N} \\
els! : \mathbb{P}\ \textit{WMElem} \\
\hline
els! = \\
\quad \{wme : \textit{WMElem} \mid wme \in elems \land \\
\qquad wmcreationtime(wme) \geq tm? \bullet \\
\qquad wme\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{ElemsCreatedBy} \\
\Xi \textit{WMEM} \\
cr? : \textit{RULEID} \\
els! : \mathbb{P}\ \textit{WMElem} \\
\hline
\forall\ wme : \textit{WMElem} \mid \\
\qquad wme \in elems \land \\
\qquad cr? = wmcreator(wme) \bullet \\
\quad wme \in els! \\
\hline
\end{array}
$$

The predicate is equivalent to the set:

$$
\begin{array}{l}
\{wme : \textit{WMElem} \mid wme \in elems \land \\
\qquad cr? = wmcreator(wme) \bullet \\
\quad wme\}
\end{array}
$$

Finally, we have:

$$\begin{array}{|l}
\hline \textit{ElemsCreatedByTime} \\
\Xi \textit{WMEM} \\
tm? : \mathbb{N} \\
els! : \mathbb{P}\ \textit{WMElem} \\
\hline
els! = \\
\quad \{wme : \textit{WMElem} \mid wme \in elems \land \\
\qquad\quad tm? \geq wmcreationtime(wme) \bullet \\
\qquad wme\} \\
\hline
\end{array}$$

Notice the relationship between this schema and *ElemsCreatedSince*. *ElemsCreatedByTime* returns all those elements whose creation was *before* the specified time; *ElemsCreatedSince* returns those elements that were created after the specified time.

We now define a schema which represents the operation of creating a working memory element and then adding it to working memory. This operation is the one that is executed by rules when they perform an *Add* operation.

$$\begin{array}{|l}
\hline \textit{AddWMElem} \\
\Delta \textit{WMEM} \\
r? : \textit{RELN} \\
rid? : \textit{RULEID} \\
crtm? : \mathbb{N} \\
\hline
\exists\ e? : \textit{WMElem} \mid \\
\qquad e? = mkwmelement(r?, rid?, crtm?) \bullet \\
\quad \textit{AddWMEM} \\
\hline
\end{array}$$

Notice that we include *AddWMEM* and hide its $e?$ variable by quantification. We employ a similar trick in the definition of the deletion operation:

$$\begin{array}{|l}
\hline \textit{DelWMElem} \\
\Delta \textit{WMEM} \\
r? : \textit{RELN} \\
rid? : \textit{RULEID} \\
crtm? : \mathbb{N} \\
\hline
\exists\ e? : \textit{WMElem} \mid \\
\qquad e? = mkwmelement(r?, rid?, crtm?) \bullet \\
\quad \textit{DelWMEM} \\
\hline
\end{array}$$

2.6 Rules and Production Memory

In this section, we give the general specification of the production rule type *Rule*. The type that we define is general in the sense that object- and meta-rules have the same form. Since we make a distinction between object- and meta-rules, we need to define a type which will allow us to differentiate between them:

$$RULETYPE == \{OBJECT, META\}$$

We make no distinction between meta-rules and meta-...-meta-rules: there is a reason for this, as will become clear.

We define the *Rule* type as follows:

$$\begin{array}{l} Rule == \\ \quad RULEID \times \\ \qquad (RULETYPE \times \\ \qquad\quad ((SituationFluent \times RuleAction) \end{array}$$

where

$$SituationFluent == \operatorname{seq} ATOM$$

is the type that defines the condition-part of a rule, and where

$$RuleAction == \operatorname{seq} Action$$

We define a number of obvious functions over *Rule*. We present the functions without comment.

$$\begin{array}{|l} ruleid : Rule \rightarrow RULEID \\ \hline \forall\, r : Rule \bullet \\ \quad ruleid(r) = first(r) \end{array}$$

$$\begin{array}{|l} ruletype : Rule \rightarrow RULETYPE \\ \hline \forall\, r : Rule \bullet \\ \quad ruletype(r) = first(second(r)) \end{array}$$

$$\begin{array}{|l} ruleconds : Rule \rightarrow SituationFluent \\ \hline \forall\, r : Rule \bullet \\ \quad ruleconds(r) = first(second(second(r))) \end{array}$$

$$
\begin{array}{|l}
ruleactions : Rule \rightarrow RuleAction \\
\hline
ruleactions(r) = second(second(second(r)))
\end{array}
$$

The following is a predicate: it is satisfied by a rule that has the correct form.

$$
\begin{array}{|l}
wellformedrule : Rule \\
\hline
\forall\ r : Rule \bullet \\
\quad wellformedrule(r) \equiv \\
\quad\quad (ruleconds(r) \neq \wedge \\
\quad\quad ruleactions(r) \neq \langle\rangle)
\end{array}
$$

We now specify the database in which production rules are stored: this database is called *production memory*. It should be noted that rules can only be added to production memory: they can never be deleted.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ Rules\ }} \\
allrules : \mathbb{P}\ Rule \\
orules : \mathbb{P}\ Rule \\
mrules : \mathbb{P}\ Rule \\
\hline
\forall\ r : Rule \mid r \in allrules \bullet \\
\quad wellformedrule(r)) \\
orules = \{r : Rule \mid ruletype(r) = OBJECT \bullet r\} \\
mrules = \{r : Rule \mid ruletype(r) = META \bullet r\} \\
orules \cap mrules = \\
orules \cup mrules = allrules \\
\hline
\end{array}
$$

We can write the last part of the predicate as:

$$\langle orules, mrules \rangle \ \mathsf{partition}\ allrules$$

allrules contains all of the rules in the system. The two partitions, *orules* and *mrules* contain the object- and meta-rules, respectively. Notice that *orules* and *mrules* must be disjoint (a rule cannot be an object-rule and a meta-rule simultaneously). Also note that all the rules in the system must be well-formed.

Production memory is easily initialized:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad InitRules \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
Rules \\
\hline
allrules = \\
orules = \\
mrules = \\
\hline
\end{array}
$$

Again, we assume that the Δ and Ξ schemata for *Rules* are defined by convention. We now define the operation of adding a new rule to production memory:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AddRule \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta Rules \\
r? : Rule \\
\hline
wellformedrule(r?) \\
allrules' = allrules \cup \{r?\} \\
\\
(ruletype(r?) = OBJECT \wedge \\
\qquad orules' = orules \cup \{r?\} \wedge \\
\qquad mrules' = mrules) \vee \\
(ruletype(r?) = META \wedge \\
\qquad orules' = orules \wedge \\
\qquad mrules' = mrules \cup \{r?\}) \\
\hline
\end{array}
$$

The next two schemata return all the rules of a given type.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ORules \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Xi Rules \\
obj\_rules! : \mathbb{P}\ Rule \\
\hline
obj\_rules! = orules \\
\hline
\end{array}
$$

ORules returns all the object-rules currently in the system.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad MRules \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Xi Rules \\
meta\_rules! : \mathbb{P}\ Rule \\
\hline
meta\_rules! = mrules \\
\hline
\end{array}
$$

MRules returns all the meta-rules currently in the system.

2.7 Conflict Set

The conflict set is a set of rule instances that are under consideration for execution. Conflict set formation is the default behaviour for ELEKTRA, and we give a specification in this section. To begin, we need to define the type that will represent a rule instance:

$$RBIND == Rule \times BINDING$$

This tells us that a rule instance is just a rule together with a set of bindings. Since it is the bindings that make instances different, we could have defined $RBIND$ as a pair whose first component is a rule identifier: it makes no difference how the type is specified (although the way it has been done here does avoid the need to define an operation to retrieve a rule by name from production memory).

We define three functions to help in the specification task. The interpretation of these functions should be clear from their definition, so we do not comment on them.

$$\begin{array}{|l} conflictrule : RBIND \rightarrow Rule \\ \hline \forall\, rb : RBIND \bullet \\ \quad conflictrule(rb) = first(rb) \end{array}$$

$$\begin{array}{|l} conflictbinding : RBIND \rightarrow BINDING \\ \hline \forall\, rb : RBIND \bullet \\ \quad conflictbinding(rb) = second(rb) \end{array}$$

$$\begin{array}{|l} mkconfsetelt : Rule \times BINDING \rightarrow RBIND \\ \hline \forall\, r : Rule;\ b : BINDING \bullet \\ \quad mkconfsetelt(r, b) = (r, b) \end{array}$$

We define the conflict set as the state space:

$$\begin{array}{|l} \textit{ConflictSet} \\ \hline crules : \mathbb{P}\ RBIND \\ \hline \end{array}$$

To initialize, the conflict set is empty:

$$\begin{array}{|l} \textit{InitConflictSet} \\ \hline \textit{ConflictSet} \\ \hline crules = \\ \hline \end{array}$$

To add a rule instance to the conflict set, we have:

$$
\begin{array}{|l}
\textit{AddCS} \\
\hline
\Delta \mathit{ConflictSet} \\
r? : \mathit{Rule} \\
b? : \mathit{BINDING} \\
\hline
\mathit{crules}' = \mathit{crules} \cup \{\mathit{mkconfsetelt}(r?, b?)\} \\
\hline
\end{array}
$$

To delete an instance, we define the schema:

$$
\begin{array}{|l}
\textit{DelCS} \\
\hline
\Delta \mathit{ConflictSet} \\
r? : \mathit{Rule} \\
b? : \mathit{BINDING} \\
\hline
\mathit{crules}' = \mathit{crules} \setminus \{\mathit{mkconfsetelt}(r?, b?)\} \\
\hline
\end{array}
$$

In this schema, it is important that the bindings, $b?$, be supplied because it is the bindings that differentiate rule instances.

The next two operation tests the conflict set. It tests to see whether a particular rule is present.

$$
\begin{array}{|l}
\textit{RuleInConfSet} \\
\hline
\Xi \mathit{ConflictSet} \\
\mathit{rid}? : \mathit{RULEID} \\
\hline
\exists\, \mathit{rb} : \mathit{RBIND} \mid \mathit{rb} \in \mathit{crules} \bullet \\
\quad \exists\, \mathit{rl} : \mathit{Rule} \mid \mathit{rl} = \mathit{conflictrule}(\mathit{rb}) \bullet \\
\quad\quad \mathit{rid}? = \mathit{ruleid}(\mathit{rl}) \\
\hline
\end{array}
$$

The next schema represents the operation of obtaining all the rules currently in the conflict set. It returns the set of rule identifiers. Note that the schema cannot be used to determine how many instances of a particular rule are in the conflict set.

$$
\begin{array}{|l}
\textit{RulesInConfSet} \\
\hline
\Xi \mathit{ConflictSet} \\
\mathit{rids}! : \mathbb{P}\ \mathit{RULEID} \\
\hline
\mathit{rids}! = \{r : \mathit{Rule}; \mathit{rb} : \mathit{RBIND} \mid \\
\quad\quad\quad \mathit{rb} \in \mathit{crules} \wedge r = \mathit{conflictrule}(\mathit{rb}) \bullet \\
\quad\quad \mathit{ruleid}(\mathit{rb})\} \\
\hline
\end{array}
$$

The following schema returns all the instances of a particular rule:

$$\begin{array}{l}
\textit{ConflictInstances} \\
\hline
\Xi \textit{ConflictSet} \\
rid? : RULEID \\
rbs! : \mathbb{P}\ RBIND \\
\hline
\forall\ rb : RBIND \mid rb \in crules \bullet \\
\quad ruleid(conflictrule(rb)) = rid? \\
\qquad \Rightarrow rb \in rbs! \\
\hline
\end{array}$$

The next schema returns *one* instance of a rule from the conflict set:

$$\begin{array}{l}
\textit{ConflictRule} \\
\hline
\Xi \textit{ConflictSet} \\
rid? : RULEID \\
rb! : RBIND \\
\hline
\exists\ rbnd : RBIND \mid \\
\qquad rbnd \in crules \land \\
\qquad rid? = ruleid(conflictrule(rbnd)) \bullet \\
\quad rb! = rb \\
\hline
\end{array}$$

We next specify the match operation for production rules: this, in the default interpreter, creates rule instances that are added to the conflict set. We begin by defining a relation:

$$\begin{array}{l}
matchwme : (ATOM \times BINDINGS \times WMElem) \leftrightarrow BINDINGS \\
\hline
\forall\ a : ATOM;\ inb : BINDING;\ w : WMElem;\ outb : BINDINGS \bullet \\
\quad matchwme(a, inb, w, outb) \equiv \\
\qquad matchatom(a, wmclause(w), inb, outb)
\end{array}$$

This relation serves merely to interface the matcher to working memory.

We now define the function which matches rule conditions:

$$
\begin{array}{l}
matchconds : (SituationFluent \times BINDINGS) \leftrightarrow BINDINGS \\
\hline
\forall s : SituationFluent;\ inbdg? : BINDINGS;\ obdg! : BINDING \bullet \\
\quad (s = \langle\rangle \Rightarrow inbdg? = obdg!) \vee \\
\quad (s = h \frown t \Rightarrow \\
\qquad (\exists w : WMElem \mid \\
\qquad\qquad w \in elems \bullet \\
\qquad\quad \exists b : BINDINGS \bullet \\
\qquad\qquad matchwme(head(h \frown t), inbdg?, w, b) \wedge \\
\qquad\qquad matchconds(tail(h \frown t), b, w, obdg!)))
\end{array}
$$

Note that a condition (situation fluent) with no conditions is assumed to be true in all circumstances—such a rule is always satisfied nomatter what the contents of working memory.

We define the operation for matching rules as:

$$
\begin{array}{l}
\underline{\quad MatchRule \quad} \\
\Xi WMEM \\
r? : Rule \\
inbdg?, obdg! : BINDINGS \\
\hline
matchconds(ruleconds(r?), inbdgs?, obdg!)
\end{array}
$$

The *MatchRule* schema defines an operation that can be used more generally than in the generation of a conflict set. Indeed, we specify it as a separate schema and define a composite schema for matching and conflict set generation. The reason for this is that we can make the latter operation very much more concise because we can determine *a priori* the use to which it will be put: we cannot do this as easily for *MatchRule*. We now give the schema representing the operation of matching rule conditions and placing rule instantiations in a (newly created) conflict set:

$$
\begin{array}{l}
\underline{\quad MatchRulesForConflictSet \quad} \\
\Delta ConflictSet \\
\Xi Rules \\
\Xi WMEM \\
\hline
\exists satis : \mathbb{P}\ Rule \mid satis \subseteq allrules \bullet \\
\quad (\forall r : Rule \mid r \in satis \bullet \\
\qquad (\exists inbdg, outbdg : BINDINGS;\ rb : RBIND \mid \\
\qquad\qquad inbdg = initbdgs \bullet \\
\qquad\quad matchconds(ruleconds(r), inbdg, outbdg) \wedge \\
\qquad\quad rb = mkconfsetelt(r, outbdg) \wedge \\
\qquad\quad rb \in crules'))
\end{array}
$$

where

$$\begin{array}{|l}
initbdgs : BINDINGS \\
\hline
initbdgs = \emptyset
\end{array}$$

initializes the bindings for an entire rule—it represents the empty bindings. Note that in the schema, a new set of bindings is created for each rule: bindings are not carried over between rules. Also note that we have defined the schema in terms of a set of rules called *satis*: this is a subset of *allrules*, so opens the way for the possibility that meta-rules might be directly satisfied by working memory contents. Indeed, we do not even bother to check that we only have object-rules in the conflict set: that is, we are not imposing the condition that only object-rules can be satisfied by working memory contents. This might be the case in some systems, but we are trying to be as general as possible. Meta-rules which deal only with object-rules will not be matched by the above process, it should be noted: below, we will show how to activate meta-rules.

We can now define a composite schema which specifies the basic matching operation:

$$\begin{aligned}
&BMatchRules \mathrel{\widehat{=}} \\
&\qquad InitConflictSet \land MatchRulesForConflictSet
\end{aligned}$$

The definition of *BMatchRules* ensures that a fresh conflict set will be created each time the matcher is called: this is exactly as we require if we are going to implement a conventional production system. We will need to over-ride this schema in some cases when meta-rules are employed in their full generality.

The next operation we need is one to select a rule instance for firing. The specification that we give is very loose: it depends upon a function *choose* which we do not specify further. The type of *choose* is:

$$\begin{array}{|l}
choose : \mathbb{P}\ RBIND \rightarrow RBIND
\end{array}$$

Note that we define choose as a total function because the possibility that the conflict set is empty can be detected elsewhere: this simplifies matters considerably. The selection operation is specified by:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ SelectConflictSet\ }} \\
\Xi ConflictSet \\
rb! : RBIND \\
\hline
rb! = choose(crules) \\
\hline
\end{array}$$

This operation returns only *one* rule instance; if we wanted to return a set of instances (as is done, for example, in SOAR (REFS)), we need to define another selection operation:

$$
\begin{array}{|l}
\hline
\textit{SelectFiringSet} \\
\Xi \textit{ConflictSet} \\
rbs! : \mathbb{P}\ \textit{RBIND} \\
\hline
rbs! = \textit{choose\_set}(\textit{crules}) \\
\hline
\end{array}
$$

where:

$$\textit{choose\_set} : \mathbb{P}\ \textit{RBIND} \rightarrow \mathbb{P}\ \textit{RBIND}$$

We need to ensure, of course, that:

$$rbs! \subseteq \textit{crules}$$

(this can be done by proving a theorem about *choose_set*).

By a simple reconfiguration, we can change the default behaviour of the interpreter so that it uses one or other of these two rule-selection schemata.

2.8 Action Execution

We now move on to action execution. The execution of actions is what actually changes working memory contents. Every rule that is fired has its actions executed.

We begin by defining the schema which defines how single actions are executed, then we define the schema which represents the execution of the entire action-part of a rule. The reader should note that we have to produce a loose specification for action execution because we have not stated all the actions that are possible (for example, a read action).

$$
\begin{array}{|l}
\hline
\textit{ExecuteAction} \\
\Delta \textit{WMEM} \\
rid? : \textit{RULEID} \\
tm? : \mathbb{N} \\
bdgs? : \textit{BINDINGS} \\
act? : \textit{Action} \\
\hline
\exists\, rl, rl_i : \textit{RELN};\ wme : \textit{WMElem};\ atyp : \textit{Act} \mid \\
\qquad\qquad rl = relofaction(act?) \land \\
\qquad\qquad instantiaterel(rl, bdgs?) \land \\
\qquad\qquad wme = mkwmelement(rl_i, rid?, tm?) \land \\
\qquad\qquad atyp = actofaction(a?) \bullet \\
\qquad (atyp = Add \land \textit{AddWMEM}) \lor \\
\qquad (atyp = Del \land \textit{DelWMEM}) \\
\qquad \lor \ldots \\
\hline
\end{array}
$$

We have indicated where the other actions are performed by ...: this is an incomplete specification, it must be stressed.

Next, we define the *ExecuteActions* schema. This represents the execution of *all* actions in an action-part:

$$
\begin{array}{|l}
\hline
\textit{ExecuteActions} \\
\Delta \textit{WMEM} \\
rb? : \textit{RBIND} \\
\hline
\exists\, r : \textit{Rule};\ rid? : \textit{RULEID};\ a : \textit{RuleAction};\ b : \textit{BINDINGS} \bullet \\
\quad r = conflictrule(rb?) \land \\
\quad b = conflictbinding(rb?) \land \\
\quad a = ruleactions(r) \land \\
\quad (\forall\, act : \textit{Action};\ i : 1 \mathrel{..} \#a \mid \\
\qquad\qquad act = a(i) \bullet \\
\qquad \textit{ExecuteAction}) \\
\hline
\end{array}
$$

Although we needed not to, we have explicitly included Δ *WMEM* in the signature of this schema just so that it would not look odd: the reason it need not be included is that *ExecuteAction* supplies it. Note that the time variable $tm?$ is free in this schema—it is bound externally in the main loop.

We can define the basic match-deliberate-act cycle (the default cycle) as:

$$
\begin{array}{l}
\textit{MatchDecideAct} \mathrel{\widehat{=}} \\
\quad \textit{BMatchRules} \land \textit{SelectConflictSet} \land \textit{ExecuteActions}
\end{array}
$$

2.9 The Default Cycle

In this section, we link everything together and define the basic (default) cycle for the interpreter. The reader should note that the existence of meta-rules can alter the basic cycle: we explain how below.

The first thing we need to do to define the cycle is to give the interpreter some notion of time. We do this by providing a counter to record the number of the current interpreter cycle.

$$\begin{array}{|l}
\textit{Cycle} \\
\hline
\textit{time} : \mathbb{N} \\
\hline
\end{array}$$

The Δ and Ξ forms are defined as by convention.

To initialize the counter, we start time for the interpreter: time starts at zero.

$$\begin{array}{|l}
\textit{InitCycle} \\
\hline
\textit{Cycle} \\
\hline
\textit{time} = 0 \\
\hline
\end{array}$$

On every cycle, the counter is incremented:

$$\begin{array}{|l}
\textit{NextCycle} \\
\hline
\Delta \textit{Cycle} \\
\hline
\textit{time}' = \textit{time} + 1 \\
\hline
\end{array}$$

The creation of working memory elements requires that the time be known:

$$\begin{array}{|l}
\textit{ThisCycle} \\
\hline
\Xi \textit{Cycle} \\
\textit{tm}! : \mathbb{N} \\
\hline
\textit{tm}! = \textit{time} \\
\hline
\end{array}$$

Now we can define the basic cycle body:

$$\textit{BCycle} \mathrel{\widehat{=}} \textit{MatchDecideAct} \wedge \textit{NextCycle}$$

This schema is used in the definition of the default cycle. Note that we need to access the value of *time* inside *MatchDecideAct*—we do this when we define the default interpreter cycle.

The next thing to do is to provide a termination mechanism. This is done using a flag which can be set by productions (or by an external agent—for example, a clock).

$$
\begin{array}{|l}
\hline
\textit{TerminateF} \\
\hline
stop : \{true, false\} \\
\hline
\end{array}
$$

Initially, the termination flag is false:

$$
\begin{array}{|l}
\hline
\textit{InitTerminateF} \\
\hline
\textit{TerminateF} \\
\hline
stop = false \\
\hline
\end{array}
$$

It is necessary to set the flag:

$$
\begin{array}{|l}
\hline
\textit{MkTerminate} \\
\hline
\Delta \textit{TerminateFwherestop} = true \\
\hline
\end{array}
$$

and to be able to determine when it is set (inspect it, in other words):

$$
\begin{array}{|l}
\hline
\textit{Terminate} \\
\hline
\Xi \textit{TerminateF} \\
\hline
stop \\
\hline
\end{array}
$$

(Note that we use the variable *stop* as the entire predicate: the reasons for this should be clear.)

Finally, we define the default cycle:

$$
\begin{array}{l}
\textit{DefaultCycle} \mathrel{\widehat{=}} \\
\quad \forall\, tm? : \mathbb{N};\ \textit{Cycle} \mid \\
\qquad\qquad tm? = time \land \neg\ \textit{Terminate} \bullet \\
\qquad\quad \textit{BCycle}
\end{array}
$$

As promised, we have made the current time available to the schemata withinin the *BCycle* composite. Note that *ThisCycle* is not redundant: meta-rules can access it whenever required. What is the case, however, is that *ThisCycle* is not needed in the definition of the default cycle.

With this definition, we have completed the main part of the interpreter specification. We need to do a little initialization, and then we will have a classical production system.

2.10 Initialization

In this section, we will be concerned with the initialization of the system. This involves loading the rules into production memory, selecting the conflict resolution strategy and setting the system running.

We being with the operation to load a set of rules. The rules come from some external store (say file) and are loaded in one chunk:

$$
\begin{array}{|l}
\textit{LoadRules} \\
\hline
\Delta \textit{Rules} \\
rs? : \mathbb{P}\ \textit{Rule} \\
\hline
\forall\, r? : \textit{Rule} \mid r? \in rs? \bullet \\
\quad \textit{AddRule} \\
\hline
\end{array}
$$

Note three things:

1. The rules that are loaded are automatically partitioned into meta- and object-rules (see the definition of *AddRule*).

2. *Rules* should already have been initialized.

3. The input variable $r?$ in *AddRule* is bound by the quantifier in the predicate of *LoadRules*.

Next, we select the conflict resolution strategy. To do this, we need to tell the system to use a strategy, and then which one to use. The first instruction is necessary when we add meta-level structures to the system. We therefore define:

$$\textit{UseCS} ::= \textit{use} \mid \textit{nouse}$$

UseCS is employed to tell the system whether it should use a conflict resolution strategy. For the object-system (the one we are defining here), the value in *UseCS* that we give to the interpreter will always be *use*.

Next, we need a type to represent the strategy that will actually be used. For the sake of simplicity, we will allow the system to use a strategy that yields one rule, and another strategy that will yield a set of rules. If the aim were to construct a system that seriously used conflict resolution, we would need to include instructions to the effect that specificity or refractoriness, etc., was to be used. We give the definition of the type:

$$\textit{ConflictResult} ::= \textit{none} \mid \textit{single} \mid \textit{set}$$

and note that instead of *single* and *set*, we would have the names of the strategies that we wanted to make available. The intention here is to give enough of the object-system to make the meta-level work, so we ignore the existence of other strategies.

We need to define two variables and operations over them. These variables record the conflict strategy in use and the fact that conflict sets are being constructed.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CSUse \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
csmanip : UseCS \\
\hline
\end{array}$$

The *CSUse* schema is used to tell the system to construct conflict sets. It is initialized as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad InitCSUse \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
CSUse \\
\hline
csmanip = use \\
\hline
\end{array}$$

The initialization schema defines the default behaviour for the interpreter: the default is to construct a conflict set on every interpreter cycle.

We now define a schema to represent the operation of setting the use flag:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad SetCSUse \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Delta CSUse \\
u? : UseCS \\
\hline
csmanip = u? \\
\hline
\end{array}$$

and we define a schema to test the contents of the variable:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CSInUse \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
\Xi CSUse \\
\hline
csmanip = use \\
\hline
\end{array}$$

The *CSInUse* schema is used to determine whether to construct a conflict set: it is a flag, in other words.

Next, we define the schemata which determine which conflict resolution strategy to use. The state schema is called *CSSelect*:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CSSelect \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}} \\
cres : ConflictResult \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \textit{InitCSSelect} \\
\textit{CSSelect} \\
\hline
cres = single \\
\hline
\end{array}$$

The default behaviour for the interpreter is to execute one rule per cycle, so the conflict resolution strategy should return one rule.

The strategy can be set using the following operation:

$$\begin{array}{|l}
\hline \textit{SetCSSelect} \\
\Delta \textit{CSSelect} \\
s? : \textit{ConflictResult} \\
\hline
cres = s? \\
\hline
\end{array}$$

Next, we define three predicates which determine what the conflict resolution module is to do. The aim is that, if the selected strategy is *none*, that conflict resolution should be skipped entirely.

$$\begin{array}{|l}
\hline \textit{SelectSingle} \\
\Xi \textit{CSSelect} \\
\hline
cres = single \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \textit{SelectSet} \\
\Xi \textit{CSSelect} \\
\hline
cres = set \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \textit{NoCSSelection} \\
\Xi \textit{CSSelect} \\
\hline
cres = none \\
\hline
\end{array}$$

To make all of this work, we need to define a schema that will execute a set of rules. We define it thus:

$$\begin{array}{|l}
\hline \textit{ExecuteActionSet} \\
\Delta \textit{WMEM} \\
rbs? : \mathbb{P}\ \textit{RBIND} \\
\hline
\forall rb : \textit{RBIND} \mid rb \in rbs? \bullet \\
\quad \textit{ExecuteActions} \\
\hline
\end{array}$$

We can now define a schema to handle the complete conflict resolution structure that we have just set up. The schema, called *ConflictResolution*, checks both use and strategy flags, initializes the conflict set, performs rule matching, applies the selection (conflict resolution) strategy and executes the rules that the strategy has produced. Alternatively, there is no conflict resolution in use, so the schema does nothing.

$$
\begin{array}{l}
\mathit{ConflictResolution} \mathrel{\widehat=} \\
\quad \neg\ \mathit{CSInUse} \lor \\
\quad (\mathit{CSInUse} \land \\
\qquad (\mathit{BMatchRules} \land \\
\qquad\quad (\mathit{SelectSingle} \land \\
\qquad\quad \mathit{SelectConflictSet} \land \\
\qquad\quad \mathit{ExecuteActions}) \lor \\
\qquad\quad (\mathit{SelectSet} \land \\
\qquad\quad \mathit{SelectFiringSet} \land \\
\qquad\quad \mathit{ExecuteActionSet}) \lor \\
\qquad\quad \mathit{NoCSSelection}))
\end{array}
$$

Now we are in a position to define the fundamental cycle that is executed at the object-level. This cycle can be overridden by meta-level activity, as will be seen. For an object-level only system, the cycle is definitive:

$$
\begin{array}{l}
\mathit{OCycle} \mathrel{\widehat=} \\
\quad \forall\, tm? : \mathbb{N};\ \mathit{Cycle} \mid tm? = \mathit{time} \land \neg\ \mathit{Terminate} \bullet \\
\qquad \mathit{ConflictResolution} \land \\
\qquad\quad \mathit{NextCycle}
\end{array}
$$

We need now to define the initialization sequence for the interpreter and to say how it is started.

$$
\begin{array}{l}
\mathit{InitSys} \mathrel{\widehat=} \\
\quad \mathit{InitWMEM} \land \mathit{InitRules} \land \mathit{InitCycle} \land \\
\qquad \mathit{InitTerminateF} \land \mathit{InitCSUse} \land \mathit{InitCSSelect}
\end{array}
$$

Next, we define the schema for adding the very first working memory element: this will enable the first production to be matched and the main cycle to go into action.

$$
\begin{array}{|l}
\hline
\textit{FirstWMElem} \\
r? : RELN \\
rid! : RULEID \\
crtm! : \mathbb{N} \\
\hline
rid! = system \\
crtm! = 0 \\
\hline
\end{array}
$$

The value *system* indicates that the element was created by the interpreter. We will use this variable again, below.

We now define:

$$AddInitWMElem \mathrel{\widehat{=}} FirstWMElem \land AddWMElem$$

Now we can define the schema which represents the setting of the conflict resolution-related variables and the creation of the initializing working memory element:

$$
\begin{array}{l}
StartSys \mathrel{\widehat{=}} \\
\quad SetCSUse \land SetCSSelect \land \\
\quad\quad AddInitWMElem
\end{array}
$$

Finally, to get the basic production system in its entirety, we compose a number of schemata and define them to be the object-level system:

$$
\begin{array}{l}
ObjectPSystem \mathrel{\widehat{=}} \\
\quad InitSys \land StartSys \land OCyle
\end{array}
$$

The definition of this schema completes the final version of the object-level production system. In the next part, we will extend this specification to include a meta-level in addition to the object-level we have defined above.

Chapter 3

Meta-Level in Z

3.1 Introduction

In this part, we add those operations that are required to augment the object-level system we specified above to include a meta-level. Many of the operations deal with the analysis of rules and addition of working memory elements: this can be seen as a kind of rule-compiler specification.

3.2 Rule Analysis

In this section, we present the specification of the component of the meta-level that analyses rules and generates assertions in working memory from them. This section relies heavily upon the specification of term and relation structure that we presented above.

The idea is that the structure of each rule (object and meta) in the system be expressed in terms of assertions in working memory *as well as* rules in production memory. Thus, as each rule is entered into production memory, it is analysed and assertions are generated. This reduces the amount of search required by meta-rules at runtime. In order to make this work, we need some function definitions.

The first function we define constructs a relation from a relation symbol and a sequence of terms:

$$
\begin{array}{|l}
mkreln : REL \times STERMS \rightarrow RELN \\
\hline
\forall\, r : REL;\ t : STERMS \bullet \\
\qquad mkreln(r, t) = (r, t)
\end{array}
$$

We also need to extract the relation from an atom:

$$
\begin{array}{|l}
atomrel : ATOM \longrightarrow RELN \\
\hline
\forall\, a : ATOM \bullet \\
\qquad atomrel(pos(r)) = r \\
\qquad atomrel(neg(r)) = r
\end{array}
$$

We use this function in creating relations that will be added to working memory by the analyser.

The next function we define returns the action tags from the action-part of a rule:

$$
\begin{array}{|l}
actionacts : Action \longrightarrow \mathbb{P}\, Act \\
\hline
\forall\, a : Action \bullet \\
\qquad actionacts(a) = \{ i : 1 \mathinner{\ldotp\ldotp} \#a \bullet actofaction(a(i)) \}
\end{array}
$$

We need this function in determining which actions are performed by a rule. A similar process is required for conditions: in this case, it is necessary to extract all the relation symbols so that a record can be kept of which rule mentions which relation symbol. Before defining this function, we define a pair of functions which map conditions and actions onto sequences which contain their component relations.

$$
\begin{array}{|l}
actionrelseq : Action \longrightarrow \mathrm{seq}\, RELN \\
\hline
\forall\, a : Action;\ i : 1 \mathinner{\ldotp\ldotp} \#a \bullet \\
\qquad actionrelseq(a)(i) = relofaction(a(i))
\end{array}
$$

$$
\begin{array}{|l}
condrelseq : SituationFluent \longrightarrow \mathrm{seq}\, RELN \\
\hline
\forall\, sf : SituationFluent;\ i : 1 \mathinner{\ldotp\ldotp} \#sf \bullet \\
\qquad condrelseq(sf)(i) = atomrel(sf(i))
\end{array}
$$

To obtain the relation symbols that appear in the condition part of a rule, we apply the following function:

$$
\begin{array}{|l}
condrelations : SituationFluent \longrightarrow \mathrm{seq}\, REL \\
\hline
\forall\, sf : SituationFluent;\ i : 1 \mathinner{\ldotp\ldotp} \#sf \bullet \\
\qquad condrelations(sf)(i) = relsym(condrelseq(sf)(i))
\end{array}
$$

Similarly, for actions, we want to extract the relation symbols that are mentioned by the individual actions:

$$\begin{array}{l}
actionrelations : Action \rightarrow \operatorname{seq} REL \\
\hline
\forall\, a : Action;\ i : 1 \mathrel{..} \#a \bullet \\
\qquad actionrelations(a)(i) = relsym(actionrelseq(a)(i))
\end{array}$$

We can now define the functions which extract constants, function symbols and subterms from sequences of objects of type *RELN*. These functions are used in the processing of both conditions and actions. It should be remembered that not all relations in a condition (or action) can be assumed to have a non-zero arity (some might be proposition symbols, in other words). This complicates the definitions slightly.

$$\begin{array}{l}
relseqconsts : \operatorname{seq} RELN \rightarrow \operatorname{seq}(\mathbb{N} \times REL \times CONSTS) \\
\hline
\forall\, rs : \operatorname{seq} RELN;\ i : 1 \mathrel{..} \#rs \mid \\
\qquad\quad arity(rs(i)) > 0 \bullet \\
\qquad relseqconsts(rs)(i) = (i, relsym(rs(i)), relconsts(rs(i)))
\end{array}$$

Note that a record is kept of where the relation appears in the sequence: this enables additional information to be recorded: it also assists in the definition of the function.

We define similar functions for function symbols and subterms:

$$\begin{array}{l}
relseqfuncts : \operatorname{seq} RELN \rightarrow \operatorname{seq}(\mathbb{N} \times REL \times FUNS) \\
\hline
\forall\, rs : \operatorname{seq} RELN;\ i : 1 \mathrel{..} \#rs \mid \\
\qquad\quad arity(rs(i)) > 0 \bullet \\
\qquad relseqfuncts(rs)(i) = (i, relsym(rs(i)), relfuncts(rs(i)))
\end{array}$$

$$\begin{array}{l}
relseqterms : \operatorname{seq} RELN \rightarrow \operatorname{seq}(\mathbb{N} \times REL \times TERMS) \\
\hline
\forall\, rs : \operatorname{seq} RELN;\ i : 1 \mathrel{..} \#rs \mid \\
\qquad\quad arity(rs(i)) > 0 \bullet \\
\qquad relseqterms(rs)(i) = (i, relsym(rs(i)), relterms(rs(i)))
\end{array}$$

With these definitions behind us, we can define the functions that operate on conditions and actions: these functions are no more than compositions, but we write them out in full for clarity.

$$\begin{array}{l}
condconsts : SituationFluent \rightarrow \operatorname{seq}(\mathbb{N} \times REL \times CONSTS) \\
\hline
\forall\, sf : SituationFluent \bullet \\
\qquad condconsts(sf) = relseqconsts(condrelseq(sf))
\end{array}$$

$$\begin{array}{|l}
condfuncts : SituationFluent \rightarrow \operatorname{seq}(\mathbb{N} \times REL \times FUNS) \\
\hline
\forall sf : SituationFluent \bullet \\
\qquad condfuncts(sf) = relseqfuncts(condrelseq(sf))
\end{array}$$

$$\begin{array}{|l}
condterms : SituationFluent \rightarrow \operatorname{seq}(\mathbb{N} \times REL \times TERMS) \\
\hline
\forall sf : SituationFluent \bullet \\
\qquad condterms(sf) = relseqterms(condrelseq(sf))
\end{array}$$

For actions, we have the following, again observing that not all relations mentioned by actions will have a non-zero arity:

$$\begin{array}{|l}
actionconsts : Action \rightarrow \operatorname{seq}(\mathbb{N} \times CONSTS) \\
\hline
\forall a : Action; i : 1 \mathrel{..} \# a \mid \\
\qquad\qquad arity(relofaction(a(i))) > 0 \bullet \\
\qquad actionconsts(a)(i) = (i, relconsts(relofaction(a(i))))
\end{array}$$

$$\begin{array}{|l}
actionfuncts : Action \rightarrow \operatorname{seq}(\mathbb{N} \times FUNS) \\
\hline
\forall a : Action; i : 1 \mathrel{..} \# a \mid \\
\qquad\qquad arity(relofaction(a(i))) > 0 \bullet \\
\qquad actionfuncts(a)(i) = (i, relfuncts(relofaction(a(i))))
\end{array}$$

$$\begin{array}{|l}
actionterms : Action \rightarrow \operatorname{seq}(\mathbb{N} \times TERMS) \\
\hline
\forall a : Action; i : 1 \mathrel{..} \# a \mid \\
\qquad\qquad arity(relofaction(a(i))) > 0 \bullet \\
\qquad actionterms(a)(i) = (i, relterms(relofaction(a(i))))
\end{array}$$

Using these functions, we can specify the analysis schemata. The reason indices were recorded in the results of the above functions is that we want to record the position of each set of constants, functors and terms in the condition or action: this makes it possible to refer to a particular set. This would not be possible if we omitted the indices because there may be more than one instance of a relation in a condition, and, in any case, position is the only way to refer to an action in an action-part (because the same action tag may appear more than once, and because the same relation may be present in different instantiations). To define the schemata, it is useful to define some more functions which will create the relations that we want to assert in working memory. These relation-creating functions assemble all the relevant data and package it so that the schema need only deal with complete working memory elements.

$$mkcondmenrel : RULEID \times REL \rightarrow WMElem$$

$$\forall\ r : RULEID;\ rl : REL \bullet \\ \quad mkcondmenrel(r, rl) = \\ \quad\quad mkreln(rule\_condition\_mentions\_relation, \\ \quad\quad\quad \langle rl \rangle)$$

$$mkcondmenconstant : RULEID \times CONST \rightarrow WMElem$$

$$\forall\ r : RULEID;\ c : CONST \bullet \\ \quad mkcondmenconstant(r, c) = \\ \quad\quad mkreln(rule\_condition\_mentions\_constant, \\ \quad\quad\quad \langle c \rangle)$$

$$mkcondmenfunsym : RULEID \times FUN \rightarrow WMElem$$

$$\forall\ r : RULEID;\ f : FUN \bullet \\ \quad mkcondmenfunsym(r, f) = \\ \quad\quad mkreln(condition\_mentions\_functionsym, \langle r, f \rangle)$$

$$mkcondmenconst : RULEID \times \mathbb{N} \times REL \times CONSTS \\ \quad \rightarrow WMElem$$

$$\forall\ rn : RULEID;\ i : \mathbb{N};\ r : REL;\ c : CONSTS \bullet \\ \quad mkcondmenconst(rn, i, r, c) = \\ \quad\quad mkreln(condition\_mentions\_constants, \langle rn, i, r, c \rangle)$$

$$mkcondmenfun : RULEID \times \mathbb{N} \times REL \times FUNS \\ \quad \rightarrow WMElem$$

$$\forall\ rn : RULEID;\ i : \mathbb{N};\ r : REL;\ f : FUNS \bullet \\ \quad mkcondmenfun(rn, i, r, f) = \\ \quad\quad mkreln(condition\_mentions\_functors, \langle rn, i, r, f \rangle)$$

$$mkcondmenterms : RULEID \times \mathbb{N} \times REL \times TERMS \\ \quad \rightarrow WMElem$$

$$\forall\ rn : RULEID;\ i : \mathbb{N};\ r : REL;\ t : TERMS \bullet \\ \quad mkcondmenterms(rn, i, r, t) = \\ \quad\quad mkreln(condition\_mentions\_terms, \langle rn, i, r, t \rangle)$$

We define similar functions for actions.

$$actionmentionsrel : RULEID \times REL \rightarrow WMElem$$

$$\forall\, rn : RULEID; r : REL \bullet \\ \quad actionmentionsrel(rn, r) = \\ \qquad mkreln(rule\_action\_mentions\_rel, \langle rn, r\rangle)$$

$$actionmentionsconst : RULEID \times CONST \rightarrow WMElem$$

$$\forall\, rn : RULEID; c : CONST \bullet \\ \quad actionmentionsconst(rn, c) = \\ \qquad mkreln(rule\_action\_mentions\_const, \langle rn, c\rangle)$$

$$actionmentionsfunsym : RULEID \times FUN \rightarrow WMElem$$

$$\forall\, rn : RULEID; f : FUN \bullet \\ \quad actionmentinsfunsym(rn, f) = \\ \qquad mkreln(rule\_action\_mentions\_funsym, \langle rn, f\rangle)$$

$$actmensrel : RULEID \times \mathbb{N} \times REL \rightarrow WMElem$$

$$\forall\, rn : RULEID; i : \mathbb{N}; r : REL \bullet \\ \quad acmensrel(rn, i, r) = \\ \qquad mkreln(action\_mentions\_relation, \langle rn, i, r\rangle)$$

$$actmensconsts : RULEID \times \mathbb{N} \times CONSTS \rightarrow WMElem$$

$$\forall\, rn : RULEID; i : \mathbb{N}; c : CONSTS \bullet \\ \quad actmensconsts(rn, i, c) = \\ \qquad mkreln(action\_mentions\_constants, \langle rn, i, c\rangle)$$

$$actmensfuns : RULEID \times \mathbb{N} \times FUNS \rightarrow WMElem$$

$$\forall\, rn : RULEID; i : \mathbb{N}; f : FUNS \bullet \\ \quad actmensfuns(rn, i, f) = \\ \qquad mkreln(action\_mentions\_constants, \langle rn, i, f\rangle)$$

$$actmensterms : RULEID \times \mathbb{N} \times TERMS \rightarrow WMElem$$

$$\forall\, rn : RULEID; i : \mathbb{N}; t : TERMS \bullet \\ \quad actmensterms(rn, i, t) = \\ \qquad mkreln(action\_mentions\_terms, \langle rn, i, t\rangle)$$

To use some of these functions, we need to extract *sets* of constants and function symbols from sequences of relations. We do this using the following functions:

$$\begin{array}{|l}
constsetinrseq : \mathrm{seq}\, RELN \rightarrow CONSTS \\
\hline
\forall\ rs : \mathrm{seq}\, RELN;\ i : 1 \,..\, \#rs \mid arity(rs(i)) > 0 \bullet \\
\qquad constsinrseq(rs) = \bigcup\{ i : 1 \,..\, \#rs \bullet relconsts(rs(i))\}
\end{array}$$

$$\begin{array}{|l}
funsymsinrseq : \mathrm{seq}\, RELN \rightarrow FUNS \\
\hline
\forall\ rs : \mathrm{seq}\, RELN;\ i : 1 \,..\, \#rs \mid arity(rs(i)) > 0 \bullet \\
\qquad funsumsinrseq(rs) = \bigcup\{ i : 1 \,..\, \#rs \bullet relfuncts(rs(i))\}
\end{array}$$

The first schema we define is *AssertRuleConstants*:

$$
\begin{array}{|l}
\hline
\textit{AssertRuleConsts} \\
\Delta \mathit{WMEM} \\
r? : \mathit{Rule} \\
\mathit{cnd} : \mathit{SituationFluent} \\
\mathit{act} : \mathit{Action} \\
\mathit{rid} : \mathit{RULEID} \\
\mathit{cndconsts} : \mathrm{seq}(\mathit{REL} \times \mathbb{N} \times \mathit{CONSTS}) \\
\mathit{actconsts} : \mathrm{seq}(\mathbb{N} \times \mathit{CONSTS}) \mathit{cconsts}, \mathit{aconsts} : \mathit{CONSTS} \\
\hline
\mathit{rid} = \mathit{ruleid}(r?) \\
\mathit{cnd} = \mathit{ruleconds}(r?) \\
\mathit{act} = \mathit{ruleactions}(r?) \\
\mathit{cndconsts} = \mathit{condconsts}(\mathit{cnd}) \\
\mathit{actconsts} = \mathit{actionconsts}(\mathit{act}) \\
\mathit{cconsts} = \mathit{constsetinrseq}(\mathit{condrelseq}(\mathit{cnd})) \\
\mathit{aconsts} = \mathit{constsetinrseq}(\mathit{actionrelseq}(\mathit{act})) \\
\forall\, i : 1 \mathrel{..} \#\mathit{cndconsts} \bullet \\
\quad (\exists\, e : \mathit{WMElem} \mid \\
\qquad\quad e = \mathit{mkcondmenconst}(\mathit{rid}, \mathit{cndconsts}(i)) \bullet \\
\qquad \mathit{AddWMEM}) \\
\forall\, i : 1 \mathrel{..} \#\mathit{actconsts} \bullet \\
\quad (\exists\, e : \mathit{WMElem} \mid \\
\qquad\quad e = \mathit{actmensconsts}(\mathit{rid}, \mathit{actconsts}(i)) \bullet \\
\qquad\qquad \mathit{AddWMEM}) \\
\forall\, c : \mathit{CONST} \mid c \in \mathit{cconsts} \bullet \\
\quad (\exists\, e : \mathit{WMElem} \mid \\
\qquad\quad e = \mathit{mkcondmenconstant}(\mathit{rid}, c) \bullet \\
\qquad \mathit{AddWMEM})) \\
\forall\, c : \mathit{CONST} \mid c \in \mathit{aconsts} \bullet \\
\quad (\exists\, e : \mathit{WMElem} \mid \\
\qquad\quad e = \mathit{actionmentionsconst}(\mathit{rid}, c) \bullet \\
\qquad \mathit{AddWMEM}) \\
\hline
\end{array}
$$

With a little judicious hiding, we have the schema we need (we perform the hiding as an explicit operation in order to avoid writing quantifiers at the top-level of the *AssertRuleConsts* schema's predicate):

$$
\begin{array}{l}
\mathit{RuleConstants} \mathrel{\widehat{=}} \\
\qquad \mathit{AssertRuleConsts} \\
\qquad\quad \setminus (\mathit{cnd}, \mathit{act}, \mathit{rid}, \mathit{cndconsts}, \\
\qquad\qquad \mathit{actconsts}, \mathit{cconsts}, \mathit{aconsts})
\end{array}
$$

The next schema the we define specifies the operation of extracting the relation symbols from a rule and adding them to working memory.

$$
\begin{array}{l}
\textbf{RuleRels} \\
\hline
\Delta \mathit{WMEM} \\
r? : \mathit{Rule} \\
\mathit{condrels}, \mathit{actrels} : \mathrm{seq}\, \mathit{REL} \\
\mathit{rid} : \mathit{RULEID} \\
\hline
\mathit{rid} = \mathit{ruleid}(r?) \\
\mathit{condrels} = \mathit{condrelations}(\mathit{ruleconds}(r?)) \\
\mathit{actrels} = \mathit{actionrelatons}(\mathit{ruleactions}(r?)) \\
\forall\, i : 1 \mathrel{..} \#\mathit{condrels} \bullet \\
\quad (\exists\, e : \mathit{WMElem} \mid \\
\qquad\qquad e = \mathit{mkcondmenrel}(\mathit{rid}, \mathit{condrels}(i)) \bullet \\
\qquad \mathit{AddWMEM}) \\
\forall\, i : 1 \mathrel{..} \#\mathit{actrels} \bullet \\
\quad (\exists\, e_r : \mathit{WMElem} \mid \\
\qquad\qquad e_r = \mathit{actionmentionsrel}(\mathit{rid}, \mathit{actrels}(ii)) \bullet \\
\qquad \mathit{AddWMEM}) \\
\forall\, i : 1 \mathrel{..} \#\mathit{actrels} \bullet \\
\quad (\exists\, e_{rw} : \mathit{WMElem} \mid \\
\qquad\qquad r_{rw} = \mathit{actmensrel}(\mathit{rid}, i, \mathit{actrels}(i)) \bullet \\
\qquad \mathit{AddWMEM}) \\
\hline
\end{array}
$$

We engage in some hiding to derive the final schema:

$$\mathit{RuleRelations} \mathrel{\widehat=} \mathit{RuleRels} \setminus (\mathit{condrels}, \mathit{actrels}, \mathit{rid})$$

To process the function symbols in a rule, we use the following schema:

$$
\begin{array}{l}
\textit{RuleFuns} \\
\hline
\Delta \mathit{WMEM} \\
r? : \mathit{Rule} \\
\mathit{condfns} : \mathrm{seq}(\mathbb{N} \times \mathit{REL} \times \mathit{FUNS}) \\
\mathit{actfns} : \mathrm{seq}(\mathbb{N} \times \mathit{FUNS}) \\
\mathit{allcondfns}, \mathit{allactfns} : \mathit{FUNS} \\
\mathit{rid} : \mathit{RULEID} \\
\hline
\mathit{rid} = \mathit{ruleid}(r?) \\
\mathit{condfns} = \mathit{condfuncts}(\mathit{ruleconds}(r?)) \\
\mathit{actfns} = \mathit{actionfuncts}(\mathit{ruleactions}(r?)) \\
\mathit{allcondfns} = \mathit{funsymsinrseq}(\mathit{condrelseq}(\mathit{ruleconds}(r?))) \\
\mathit{allactfns} = \mathit{funsymsinrseq}(\mathit{actionrelseq}(\mathit{ruleactions}(r?))) \\
\forall\, i : 1 \mathinner{\ldotp\ldotp} \#\mathit{allcondfns} \bullet \\
\quad (\exists\, e : \mathit{WMElem} \mid \\
\qquad\qquad e = \mathit{mkcondmenfunsym}(\mathit{rid}, \mathit{condfns}(i)) \bullet \\
\qquad \mathit{AddWMEM}) \\
\forall\, i : 1 \mathinner{\ldotp\ldotp} \#\mathit{allactfns} \bullet \\
\quad (\exists\, e : \mathit{WMElem} \mid \\
\qquad\qquad e = \mathit{actionmentionsfunsym}(\mathit{rid}, \mathit{allactfns}(i)) \bullet \\
\qquad \mathit{AddWMEM}) \\
\forall\, i : 1 \mathinner{\ldotp\ldotp} \#\mathit{condfns} \bullet \\
\quad (\exists\, e : \mathit{WMElem} \mid \\
\qquad\qquad e = \mathit{mkcondmenfun}(\mathit{rid}, \mathit{actfns}(i)) \bullet \\
\qquad \mathit{AddWMEM}) \\
\forall\, i : 1 \mathinner{\ldotp\ldotp} \#\mathit{actfns} \bullet \\
\quad (\exists\, e : \mathit{WMElem} \mid \\
\qquad\qquad e = \mathit{actmensfuns}(\mathit{rid}, \mathit{actfns}(i)) \bullet \\
\qquad \mathit{AddWMEM}) \\
\hline
\end{array}
$$

To obtain the interface, we again do some hiding:

$$
\begin{array}{l}
\mathit{RuleFunctors} \mathrel{\widehat{=}} \\
\quad \mathit{RuleFuns} \setminus (\mathit{condfns}, \mathit{actfns}, \mathit{allcondfns}, \mathit{allactfns}, \mathit{rid})
\end{array}
$$

We engage in similar operations for terms:

$$
\begin{array}{|l}
\hline
\textit{RuleTrms} \\
\Delta \textit{WMEM} \\
r? : \textit{Rule} \\
\textit{rid} : \textit{RULEID} \\
\textit{ctrms} : \mathrm{seq}(\mathbb{N} \times \textit{REL} \times \textit{TERMS}) \\
\textit{atrms} : \mathrm{seq}(\mathbb{N} \times \textit{TERMS}) \\
\hline
\textit{rid} = \textit{ruleid}(r?) \\
\textit{ctrms} = \textit{condterms}(\textit{ruleconds}(r?)) \\
\textit{atrms} = \textit{actionterms}(\textit{ruleactions}(r?)) \\
\forall\, i : 1 \mathrel{..} \#\textit{ctrms} \bullet \\
\quad (\exists\, e : \textit{WMElem} \mid \\
\qquad\qquad e = \textit{mkcondmenterms}(\textit{rid}, \textit{ctrms}(i)) \bullet \\
\qquad\quad \textit{AddWMEM}) \\
\forall\, i : 1 \mathrel{..} \#\textit{atrms} \bullet \\
\quad (\exists\, e : \textit{WMElem} \mid \\
\qquad\qquad e = \textit{actmensterms}(\textit{rid}, \textit{atrms}(i)) \bullet \\
\qquad\quad \textit{AddWMEM}) \\
\hline
\end{array}
$$

$$\textit{RuleTerms} \mathrel{\widehat{=}} \textit{RuleTerms} \setminus (\textit{rid}, \textit{ctrms}, \textit{atrms})$$

To complete the processing of rules, we want to assert the rule type, the conditions and the actions, as well as the individual acts that are performed. We define the obvious schemata to do this. With each schema, we will define a function to create the working memory element.

We begin with rule types.

$$
\begin{array}{|l}
\textit{mkruletypeelem} : \textit{RULEID} \times \textit{RULETYPE} \rightarrow \textit{WMElem} \\
\hline
\forall\, \textit{rn} : \textit{RULEID}; \textit{rt} : \textit{RULETYPE} \bullet \\
\quad \textit{mkruletypeelem}(\textit{rn}, \textit{rt}) = \\
\qquad \textit{mkreln}(\textit{rule\_type}, \langle \textit{rn}, \textit{rt} \rangle)
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{RuleType} \\
\Delta \textit{WMEM} \\
r? : \textit{Rule} \\
\hline
\exists\, \textit{rid} : \textit{RULEID}; e : \textit{WMElem} \mid \\
\qquad \textit{rid} = \textit{ruleid}(r?) \land \\
\qquad e = \textit{mkruletypeelem}(\textit{rid}, \textit{ruletype}(r?)) \bullet \\
\quad \textit{AddWMEM} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
mkruleconds : RULEID \times SituationFluent \rightarrow WMElem \\
\hline
\forall rn : RULEID; sf : SituationFluent \bullet \\
\quad mkruleconds(rn, sf) = \\
\quad\quad mkreln(rule\_conditions, \langle rn, sf \rangle)
\end{array}
$$

$$
\begin{array}{|l}
mkcondelem : RULEID \times \mathbb{N} \times ATOM \rightarrow WMElem \\
\hline
\forall rn : RULEID; i : \mathbb{N}; a : ATOM \bullet \\
\quad mkcondelem(rn, i, a) = \\
\quad\quad mkreln(rule\_condition, \langle rn, i, a \rangle)
\end{array}
$$

$$
\begin{array}{|l}
\hline
RuleConditions \\
r? : Rule \\
\hline
\exists sf : SituationFluent; rid : RULEID \mid \\
\quad\quad sf = ruleconds(r?) \land \\
\quad\quad rid = ruleid(r?) \bullet \\
\quad (\exists e : WMElem \mid \\
\quad\quad\quad e = mkruleconds(rid, sf) \bullet \\
\quad\quad AddWMEM) \land \quad (\forall i : 1 \mathrel{..} \#sf; a : ATOM \mid \\
\quad\quad\quad a = sf(i) \bullet \\
\quad\quad (\exists e : WMElem \mid \\
\quad\quad\quad\quad e = mkcondelem(rid, i, a) \bullet \\
\quad\quad\quad AddWMEM)) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
mkruleacts : RULEID \times RuleAction \rightarrow WMElem \\
\hline
\forall rn : RULEID; a : RuleAction \bullet \\
\quad mkruleconds(rn, a) = \\
\quad\quad mkreln(rule\_actions, \langle rn, a \rangle)
\end{array}
$$

$$
\begin{array}{|l}
mkactelem : RULEID \times \mathbb{N} \times Action \rightarrow WMElem \\
\hline
\forall rn : RULEID; i : \mathbb{N}; a : Action \bullet \\
\quad mkcondelem(rn, i, a) = \\
\quad\quad mkreln(rule\_action, \langle rn, i, a \rangle)
\end{array}
$$

$$
\begin{array}{l}
\textit{RuleActions} \\
\hline
r? : \textit{Rule} \\
\hline
\exists\ \textit{acts} : \textit{RuleAction};\ \textit{rid} : \textit{RULEID} \mid \\
\qquad\qquad \textit{sf} = \textit{ruleactions}(r?) \land \\
\qquad\qquad \textit{rid} = \textit{ruleid}(r?) \bullet \\
\qquad (\exists\ e : \textit{WMElem} \mid \\
\qquad\qquad\qquad e = \textit{mkruleacts}(\textit{rid}, \textit{acts}) \bullet \\
\qquad\qquad \textit{AddWMEM}) \land \qquad (\forall\ i : 1 \mathrel{..} \#\textit{sf};\ a : \textit{Action} \mid \\
\qquad\qquad\qquad a = \textit{acts}(i) \bullet \\
\qquad\qquad (\exists\ e : \textit{WMElem} \mid \\
\qquad\qquad\qquad\quad e = \textit{mkactelem}(\textit{rid}, i, a) \bullet \\
\qquad\qquad\qquad \textit{AddWMEM})) \\
\hline
\end{array}
$$

We collect all of the above schema to define a new operation:

$$
\begin{array}{l}
\textit{AnalyzeRule} \mathrel{\widehat=} \\
\qquad \textit{RuleConstants} \land \\
\qquad \textit{RuleFunctors} \land \\
\qquad \textit{RuleTerms} \land \\
\qquad \textit{RuleRelations} \land \\
\qquad \textit{RuleConditions} \land \qquad \textit{RuleActions} \land \\
\qquad \textit{RuleType}
\end{array}
$$

To process the contents of rules and place them in working memory, we need the following schema:

$$
\begin{array}{l}
\textit{ProcessRuleComponents} \\
\hline
rs? : \mathbb{P}\ \textit{Rule} \\
\hline
\forall\ r : \textit{Rule} \mid r \in rs? \bullet \\
\qquad \textit{AnalyzeRule} \\
\hline
\end{array}
$$

We have now defined the schemata that analyse all the rules in a given ruleset. This completes the definition of the rule analyzer component of ELEKTRA.

3.3 Operations and Bindings

In this section, we will be concerned with providing a number of operations that turn out to be useful for systems that use meta-rules. Some of the operations we define in outline only, because it is the case that there are so many operations that a complete account would rapidly become tedious (in addition, we do not pretend to know them all).

We begin with a simple operation: it retrieves a rule by name. To make the operation robust, we first define a schema which is true whenever there is a rule in production memory with a given name. To make best use of this schema, we must assume that rules have unique names.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{KnownRule}} \\
\hline
\Xi Rules \\
rid? : RULEID \\
\hline
\exists\, r : Rule \mid r \in allrules \bullet \\
\qquad rid? = ruleid(r) \\
\hline
\end{array}
$$

The retrieval operation is simple:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FindRuleByName}} \\
\hline
\Xi Rules \\
rid? : RULEID \\
r! : Rule \\
\hline
\exists\, r : Rule \mid r \in allrules \bullet \\
\qquad ruleid(r) = rid? \land r! = r \\
\hline
\end{array}
$$

We now define the *GetRule* schema as:

$$
\begin{array}{l}
GetRule \mathrel{\widehat=} \\
\qquad (KnownRule \land FindRuleByName) \lor \qquad \neg\ KnownRule
\end{array}
$$

We can extend the test schema idea and define two schemata (these can be used from within rules):

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{KnownObjectRule}} \\
\hline
\Xi Rules \\
rid? : RULEID \\
\hline
\exists\, r : Rule \mid r \in orules \bullet \\
\qquad rid? = ruleid(r) \\
\hline
\end{array}
$$

and

$$
\begin{array}{|l}
\hline
\textit{KnownMetaRule} \\
\Xi Rules \\
rid? : RULEID \\
\hline
\exists\, r : Rule \mid r \in mrules \bullet \\
\qquad rid? = ruleid(r) \\
\hline
\end{array}
$$

The next major task that faces us is to account for user-defined relations and actions. These are needed to interface user-defined code to the production system. This turns out to be a fairly tricky operation to provide: in particular, it requires that the relation or action be known to the system and that there be code to interpret it. In addition, the arguments to the user-define relation or action must be checked, and this involves manipulating variables. We will explain the last problem in more detail below. To begin with, we will define the two sets of tables that are needed to support the inclusion of user-defined code.

We being with the definition of a table that tells the system whether a relation is known to it: the idea is that if a relation is in the table, it is a *user-defined* one and is, therefore, to be interpreted.

$$
\begin{array}{|l}
\hline
\textit{ExecRels} \\
er : \mathbb{P}\, REL \\
\hline
\end{array}
$$

The table is represented as a set of relation symbols. If a relation symbol is in the set, it is assumed to be user-defined, and therefore executable.

The initialization schema is obvious:

$$
\begin{array}{|l}
\hline
\textit{InitExecRels} \\
ExecRels \\
\hline
er = \\
\hline
\end{array}
$$

The table has to be set up at some stage, so we define:

$$
\begin{array}{|l}
\hline
\textit{SetExecRels} \\
\Delta ExecRels \\
rels? : \mathbb{P}\, REL \\
\hline
er' = rels? \\
\hline
\end{array}
$$

Sometimes, a new user-defined (executable) relation will be added during a run of the system, so we define:

$$
\begin{array}{|l}
\hline
\textit{AddExecRel} \\
\Delta \textit{ExecRels} \\
\textit{rel}? : \textit{REL} \\
\hline
\textit{er}' = \textit{er} \cup \{\textit{rel}?\} \\
\hline
\end{array}
$$

Finally, we define the test on *ExecRels*:

$$
\begin{array}{|l}
\hline
\textit{ExecutableRelation} \\
\Xi \textit{ExecRels} \\
\textit{rel}? : \textit{REL} \\
\hline
\textit{rel}? \in \textit{er} \\
\hline
\end{array}
$$

The schemata for user-defined actions are directly analogous: we give them without comment.

$$
\begin{array}{|l}
\hline
\textit{ExecActs} \\
\textit{ea} : \mathbb{P}\ \textit{Act} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{InitExecActs} \\
\textit{ExecActs} \\
\hline
\textit{ea} = \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{SetExecActs} \\
\Delta \textit{ExecActs} \\
\textit{acts}? : \mathbb{P}\ \textit{Act} \\
\hline
\textit{ea}' = \textit{acts}? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{AddExecAct} \\
\Delta \textit{ExecActs} \\
\textit{act}? : \textit{Act} \\
\hline
\textit{ea}' = \textit{ea} \cup \{\textit{act}?\} \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\textbf{ExecutableAct} \\
\hline
\Xi ExecActs \\
act? : Act \\
\hline
act? \in ea
\end{array}
$$

Next, we define a set of functions and relations over bindings. We introduced bindings in the first part of this paper in order to account for the matching process: here, we need to extend the operations that are available to us. The reasons for this will become clearer when we give a fuller account of user-defined conditions and actions.

We begin with a predicate: the predicate is true if and only if a variable (its first argument) is bound in a set of bindings (its second argument):

$$
\begin{array}{l}
bound : VAR \leftrightarrow BINDINGS \\
\hline
\forall\, v : VAR;\ b : BINDINGS \bullet \\
\qquad v \in \operatorname{dom} b
\end{array}
$$

varval returns the value to which a variable is bound: the precondition on this function is that the variable is actually bound.

$$
\begin{array}{l}
varval : VAR \times BINDINGS \rightarrow VAL \\
\hline
\forall\, v : VAR;\ b : BINDINGS \bullet \\
\qquad varval(v, b) = b(a)
\end{array}
$$

The *addbinding* function adds a variable and a value to a set of bindings. The precondition is that the variable is not already bound:

$$
\begin{array}{l}
addbinding : VAR \times VAL \times BINDINGS \rightarrow BINDINGS \\
\hline
\forall\, var : VAR;\ vl : VAL;\ b : BINDINGS \bullet \\
\qquad addbinding(var, vl, b) = b \cup \{var \mapsto vl\}
\end{array}
$$

The final function is *rebind*. This function rebinds a variable in a set of bindings: that is, it replaces the value to which the variable is bound. Clearly, the variable must previously have been bound.

$$
\begin{array}{l}
rebind : VAR \times VAL \times BINDINGS \rightarrow BINDINGS \\
\hline
\forall\, var : VAR;\ vl : VAL;\ b : BINDINGS \bullet \\
\qquad rebind(var, vl, b) = b \oplus \{var \mapsto vl\}
\end{array}
$$

Now, in order to execute a relation or action, it is necessary to check that the relation or action is actually known to be executable. Then matters differ: for actions, it is simply a matter of applying the procedure which implements the action to its argument or arguments. In the case of relations, each argument must be checked to see whether it is a variable, constant or function. In the case of functions, a check must be made of its arguments, and so on recursively. In the case of variables, it is necessary to check whether a variable is bound or not in the current binding set. This becomes important when relations are implemented as procedures. Typically, constraints are imposed on the variables that appear in executable relations: for example, all but the rightmost must be bound prior to call. For example, imagine that we have an executable relation *set_union* to represent the operation of uniting two sets. We can write this as:

$$set\_union(s_1, s_2, s_3)$$

We can insist that s_1 and s_2 be bound so that no search is made (after all, we are not specifying Prolog). The third argument can be bound or not: if it is unbound, it represents the result of the union—this can be written as:

$$set\_union(s_1, s_2, s_1 \cup s_2)$$

If s_3 is bound, it represents a set, and the object of calling *set_union* is to check that the union of s_1 and s_2 is the same as s_3—that is:

$$s_3 = s_1 \cup s_2$$

Since we are defining the executable relations, we are at liberty to place such interpretations upon them. It clearly becomes necessary to check variables.

In what follows, we will be building up to the definition of schemata for interpreting relations and actions. We will concentrate on relations because they are harder. As has been stated above, we cannot give a full specification because the user may want to add new relations: what we will give is a skeleton, together with a list of conditions on executable relations. This specification will, naturally, be in terms of some example relations. The relations that we will define will be:

- set union and intersection;
- condition element and situation fluent testing.

Set operations are very important at the meta-level, and the meta-level also needs to be able to test the conditions in object-rules.

First, we need a schema to check the relation symbol to see whether it is executable. The schema that we give below builds on *ExecutableRelation* by extracting the relation symbol from the atom and then testing it:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ExecCond \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
a? : ATOM \\
\hline
\exists\, r : REL \mid \\
\quad\quad r = relsym(atomrel(a?)) \bullet \\
\quad\quad\quad\quad ExecutableRelation \\
\hline
\end{array}
$$

The next schema extracts the relation symbol and its argument: all of the arguments are extracted, and no attention is paid to the arity of the relation—that is handled by the primitive code which executes the relation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ExecRelAndArgs \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
a? : ATOM \\
rel! : REL \\
rargs! : STERMS \\
\hline
rel! = relsym(atomrel(a?)) \\
rargs! = args(atomrel(a?)) \\
\hline
\end{array}
$$

We have to take into account the fact that a condition element may be negated: we will give the account of this in each schema, and will pass the original atom to the schema in order to make the test.

The next thing we shall do is give examples of operation schemata that implement executable relations. We being with the *SetUnion* schema. We will define all the schemata and will assume that the temporary variable t is hidden at some later stage.

$$
\begin{array}{l}
\hline
\textit{SetUnOp} \\
a? : ATOM \\
b? : BINDINGS \\
res! : BINDINGS \\
rargs? : STERMS \\
t : VAL \\
\hline
bound(rargs?(1), b?) \\
bound(rargs?(2), b?) \\
t = varval(rargs?(1)) \\
\quad \cup varval(rargs?(2)) \\
(bound(rargs?(3), b?) \wedge \\
\quad (bound(rargs?(3), b?) \wedge \\
\quad (t = varval(rargs?(3), b?) \wedge res! = b?) \vee \\
\quad t \neq varval(rargs?(3), b?) \wedge res! = emptybindings)) \\
\vee \\
\quad (\neg\ bound(rargs?(3), b?) \wedge \\
\quad\quad res! = addbinding(rargs?(3), t, b?)) \\
\hline
\end{array}
$$

where

$$
\begin{array}{l}
emptybindings : BINDINGS \\
\hline
emptybindings =
\end{array}
$$

In *SetUnOp*, the variable t must be hidden when we come to define the interface: this is to ensure that the local state remains hidden and that no external operations can rely upon it. In addition, the negation of $a?$ is handled by checking the result bindings ($res!$): if they are not empty, they are made empty so that the negative result is propagated outwards. In the case in which $res!$ is empty, we want to signal failure by returning the original bindings (this represents the negation of a negative result). We handle negations by defining a schema below.

The schema for set difference is similar to *SetUnOp*:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad SetDiffOp \quad}} \\
a? : ATOM \\
b? : BINDINGS \\
res! : BINDINGS \\
rargs? : STERMS \\
t : VAL \\
\hline
bound(rargs?(1), b?) \\
bound(rargs?(2), b?) \\
t = varval(rargs?(1)) \\
\quad \setminus varval(rargs?(2)) \\
(bound(rargs?(3), b?) \land \\
\quad (bound(rargs?(3), b?) \land \qquad (t = varval(rargs?(3), b?) \\
\qquad \land\ res! = b?) \lor \\
\quad t \neq varval(rargs?(3), b?) \\
\qquad \land\ res! = emptybindings)) \\
\lor \\
\quad (\neg\ bound(rargs?(3), b?) \land \\
\qquad res! = addbinding(rargs?(3), t, b?)) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad NegateExecCond \quad}} \\
a? : ATOM \\
b? : BINDINGS \\
res! : BINDINGS \\
\hline
negatedAtom(a?) \land res! \neq emptybindings \\
\quad \Rightarrow\ res! = emptybindings \lor \\
\quad negatedAtom(a?) \land res! = emptybindings \\
\qquad \Rightarrow\ res! = b? \\
\hline
\end{array}
$$

We conjoin and hide to obtain the interface schema:

$$
\begin{array}{l}
SetDifference \mathrel{\widehat=} \\
\quad SetDiffOp \setminus (t) \land NegateExecCond
\end{array}
$$

We use a similar technique to define *SetUnion*:

$$
\begin{array}{l}
SetUnion \mathrel{\widehat=} \\
\quad SetUnOp \setminus (t) \land NegateExecCond
\end{array}
$$

The reader might think that, because the two schemata are so similar, that it might be possible to separate the predicate into a number of relations: the problem comes when checking the arguments to see whether they are bound—we cannot know in advance what the arity of the relation is going to be. We have therefore played safe and have defined the complete operation in one schema.

The next schema that we define is the one which matches the condition part of a rule (an object rule, by default).:

$$
\begin{array}{|l}
\hline
\textit{MatchConditionElementsOp} \\
\Xi \textit{Rules} \\
b? : \textit{BINDINGS} \\
res! : \textit{BINDINGS} \\
rargs? : \textit{RULEID} \\
\hline
\exists\ inbdg, obdg : \textit{BINDINGS};\ r : \textit{Rule} \mid \\
\qquad\qquad inbdg = b? \land res! = obdg \land \\
\qquad\qquad r \in allrules \land ruleid(r) = varval(rargs?(1)) \bullet \\
\qquad \textit{MatchRule} \\
\hline
\end{array}
$$

With the operation to match a rule, we need explicitly to search working memory: if we had defined the object-system matcher a little differently, this would not have been needed. However, since we only need to define this operation once, it appears not to matter all that much.

$$
\begin{array}{|l}
\hline
\textit{MatchConditionElementOp} \\
b? : \textit{BINDINGS} \\
res! : \textit{BINDINGS} \\
rargs? : \textit{STERMS} \\
\hline
bound(rargs?(1), b?) \\
\exists\ w : \textit{WMElem} \mid \\
\qquad\qquad w \in elems \bullet \\
\qquad matchwme(varval(rargs?(1)), b?, w, res!) \\
\hline
\end{array}
$$

For these last two schemata, we have no hiding to perform, so we can define the interface directly in terms of the operation and the negation-handling schema:

$$
\begin{array}{l}
\textit{MatchConditionElements} \mathrel{\widehat=} \\
\qquad \textit{MatchConditionElementsOp} \\
\qquad\qquad \land\ \textit{NegateExecCond}
\end{array}
$$

$$
\begin{array}{l}
\textit{MatchConditionElement} \mathrel{\widehat=} \\
\qquad \textit{MatchConditionElementOp} \\
\qquad\qquad \land\ \textit{NegateExecCond}
\end{array}
$$

We are now in a position to define the outline schema for the dispatcher: this schema represents the operation which examines the relation symbol and then decides which operation to perform:

$$\begin{array}{|l}
\hline
\textit{DispatchConds} \\
b? : \textit{BINDINGS} \\
a? : \textit{ATOM} \\
\textit{rel}? : \textit{REL} \\
\textit{rargs}? : \textit{STERMS} \\
\textit{res}! : \textit{BINDINGS} \\
\hline
(\textit{rel}? = \textit{setunion} \land \textit{SetUnion}) \lor \\
(\textit{rel}? = \textit{setdifference} \land \textit{SetDifference}) \lor \\
(\textit{rel}? = \textit{matchcond} \land \textit{MatchConditionElement}) \lor \\
(\textit{rel}? = \textit{matchconds} \land \textit{MatchConditionElements}) \lor \\
\qquad \ldots \\
\hline
\end{array}$$

We cannot complete the definition of this schema because we need many more operations. The schema has as inputs all the variables that are required by the operation schemata (including the atom so that the negation test can be performed). Its output is *res*!.

We can now collect all of the schemata for executing user-defined relations and define a schema to represent the operation.

$$\begin{array}{l}
\textit{ExecuteCondition} \mathrel{\widehat=} \\
\quad \textit{ExecCond} \land \textit{ExecRelAndArgs} \land \\
\qquad \textit{DispatchConds}
\end{array}$$

We need to define a schema to represent the new operation of executing a condition element as well as matching it against working memory: we leave this as an exercise (it is relatively simple—it involves redefining the matcher given in the first part and introducing a disjunction). We will call this schema *MatchOrExecuteConds*.

The schemata to execute user-define actions are similar to those above, so we do not extend this report by their inclusion. Once the above has been understood, the definition of action schemata should be relatively easy.

3.4 Execution

The last task we need to perform is that of defining the way in which the enriched production interpreter executes the rules that it contains. We can rely to a certain extent upon the operations that we defined in part one. What we need to do is to extend the original execution cycle to include meta-rules. It turns out that this poses no problems: however, we might want to execute *only* meta-rules on the interpreter, and leave object-rule interpretation to meta-rules. If we are interested in matching object- and meta-rules together and placing them in the conflict set, we need only use the cycle given in part one.

There is more than one way to achieve this, depending upon whether the conflict set is going to be used. If the conflict set is to be used and if it will contain only meta-rules, we can use a variation on *MatchRulesForConflictSet*:

$$
\begin{array}{l}
\textbf{MatchMetaRulesForConflictSet} \\
\hline
\Delta ConflictSet \\
\Xi Rules \\
\Xi WMEM \\
\hline
\exists\, satis : \mathbb{P}\, Rule \mid satis \subseteq mrules \bullet \\
\quad (\forall\, r : Rule \mid r \in satis \bullet \\
\qquad (\exists\, inbdg, outbdg : BINDINGS; rb : RBIND \mid \\
\qquad\qquad inbdg = initbdgs \bullet \\
\qquad\quad matchconds(ruleconds(r), inbdg, outbdg) \wedge \\
\qquad\quad rb = mkconfsetelt(r, outbdg) \wedge \\
\qquad\quad rb \in crules'))
\end{array}
$$

The variation is simply to restrict the rules in *satis* to meta-rules: this will ensure that only meta-rules are to be found in the conflict set. Conflict resolution can then be applied in the normal way, and meta-rules will be assigned the task of interpreting object-rules. The basic cycle can be redefined to give:

$$
\begin{array}{l}
BMatchMetaRules \mathrel{\widehat{=}} \\
\qquad InitConflictSet \wedge MatchMetaRulesForConflictSet
\end{array}
$$

and:

$$
\begin{array}{l}
MetaMatchDecideAct \mathrel{\widehat{=}} \\
\quad BMatchMetaRules \wedge SelectConflictSet \wedge ExecuteActions
\end{array}
$$

and finally:

$$
BMetaCycle \mathrel{\widehat{=}} MetaMatchDecideAct \wedge NextCycle
$$

It is possible to do away with the conflict set completely. This involves the direct execution or forcing the execution of one root rule. This rule is charged with interpreting all other rules. One schema to perform this operation is shown below (it is only one of a number of alternatives):

$$
\begin{array}{l}
\textit{ExecuteFromMetaRule} \\
\hline
\Delta \mathit{WMEM} \\
\mathit{rid}? : \mathit{RULEID} \\
\hline
\exists\, r : \mathit{Rule};\ b, \mathit{res} : \mathit{BINDINGS};\ \mathit{rb} : \mathit{RBIND};\ \mathit{rargs} : \mathit{STERMS} \\
\quad \mathit{rel} : \mathit{REL} \bullet \\
\quad \mathit{GetRule} \wedge \\
\quad \mathit{rel} = \mathit{matchconds} \wedge \\
\quad b = \mathit{initbindings} \wedge \\
\quad \mathit{rargs}(1) = \mathit{ruleconds}(r) \wedge \\
\quad \mathit{MatchConditionElements} \wedge \\
\quad \mathit{rb} = \mathit{mkconfsetelt}(r, \mathit{res}) \wedge \\
\quad \mathit{ExecuteActions} \\
\hline
\end{array}
$$

Note that this schema merely uses operations that we have defined elsewhere, and it also constructs a conflict set element to pass to the *ExecuteActions* schema.

We now have three ways in which rules can be tested, selected and executed: this complicates the main cycle because choices have to be made. The complication is not conceptual, and rests upon an obvious disjunctive definition of the *Cycle* schema. Note that time is handled in exactly the same way as it was in the object-interpreter given above. We leave it to the reader to fill in the missing details, and pass on to the initialization phase of the interpreter.

3.5 Initialization

Initialization in ELEKTRA is a little more complex when the meta-level is included: the main source of complication is that rules must be analyzed and placed in working memory. Apart from this, all other operations are identical. It should be noted that the analysis of rules applies to *all* rules, and, after analysis, the rules are loaded into production memory as usual—this implies (correctly) that an additional conjunct must be added to the initialization schema in order to convert it to the initialization routine for the meta-level system.

3.6 Differences

In this section, we want briefly to make a few remarks about the additional structure that is imposed on the ELEKTRA interpreter by the inclusion of a meta-level. In particular, we want to make some notes about values and variables.

Above, we have assumed that there is some uniform variable domain. The elements of this domain are bound to variables. Quite obviously, we need to give additional structure to VAL before the specification is complete. This is because we can bind rules, constants, relations, conditions, actions, and so on.

Next, we have assumed that variables are just variables: we have not made a distinction between object- and meta-variables. We might want to define:

$$VAR ::= metavar\langle VARNM\rangle \mid objvar\langle VARNM\rangle$$

where

$$[VARNM]$$

is a domain of variable names. The separation of variables into meta- and object-variables brings a number of advantages: for example, it might be possible to avoid the use of unification as the main matching procedure (this procedure is indicated by the fact that working memory can contain elements with variables in them), although it will still have to be used in certain cases. In addition, the distinction brings added clarity to the definition of rules. We have not made the distinction above because it was considered to complicate the specification in an inessential way.

Bibliography

[1] Anderson, J.R., *The Architecture of Cognition*, Harvard University Press, London, 1983.

[2] Corkhill, D.D., Gallagher, K.Q. and Murray, K.E., GBB: A Generic Blackboard Development System, *Proc. Fifth National Conference on Artificial Intelligence (AAAI-86)*, pp. 1008-14, 1986.

[3] Craig, I.D., *The CASSANDRA Architecture*, Ellis Horwood, Chichester, UK, 1989.

[4] Craig, I.D., *The Formal Specification of Advanced AI Architectures*, Ellis Horwood, Chichester, UK, 1991.

[5] Craig, I.D., *Blackboard Systems*, Ablex Publishing Corp., Norwood NJ, to appear 1994.

[6] Craig, I.D., A Reflective Production Rule System, *Kybernetes*, to appear 1994.

[7] Craig, I.D., Rule Interpreters in ELEKTRA, *Kybernetes*, to appear 1994.

[8] Forgy, C.L., RETE: A fast algorithm for the many pattern/many object pattern matching problem, *Artificial Intelligence Journal*, Vol. 19, pp. 17-37, 1982.

[9] Hayes-Roth, B., A Blackboard Model for Control, *Artificial Intelligence Journal*, Vol. 26, pp. 251-322, 1985.

[10] Jones, C. B., *Software Development: A Rigorous Approach*, Prentice-Hall, Hemel Hempstead, 1980.

[11] Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall, Hemel Hempstead, 1986.

[12] Newell, A., Production Systems: models of control structures, *in* Chase, W.G. (ed.), *Visual Information Processing*, Academic Press, New York,. 1973.

[13] Miranker, D.P., *TREAT: A New and Efficient Match Algorithm for AI Production Systems*, Pitman, London, 1990.

[14] Morgan, T. and Engelmore, R., *Blackboard Systems*, Addison-Wesley, Wokingham, UK, 1988.

[15] Spivey, J.M., *Understanding Z*, Cambridge University Press, 1988.

[16] Spivey, J.M., *The Z Notation: A Reference Manual*, Prentice-Hall, Hemel Hempstead, 1989.

[17] Waterman, D.A. and Hayes-Roth, F. (eds.), *Pattern-Directed Inference Systems*, Academic Press, New York, 1978.