Formal Specification of AI Systems: Four Case Studies

Iain D. Craig

Department of Computer Science

University of Warwick
Coventry CV4 7AL
UK EC

ABSTRACT

In this paper, we outline four AI systems and their formal specification in Z. Two of the systems (a blackboard framework and the author's CASSANDRA architecture) are of high complexity and are for predominantly real-time, high-reliability applications; the other two are production systems of an unconventional type. The first production system (called ELEKTRA) contains extensive facilities for performing meta-level inference. The second extends ELEKTRA with a frame system and organises rules into rulesets for enhanced modularity (its specification is still underway). Both production systems contain control features that can be accessed by rules: both are reflective systems. To date, the four specifications amount to more than 500 A4 pages, including proofs and explanatory text. The four specifications are related in that solutions to problems encountered in the blackboard and CASSANDRA systems are attempted in the other two specifications. The formal specifications of the second two systems were undertaken because we wanted to experiment with ideas without having to engage in implementation at too early a stage (ELEKTRA was subsequently implemented in Scheme from its Z specification): this turned out to be an excellent strategy. To conclude the paper, we review our approach and consider the general use of formal specifications in the development of AI systems.

1 INTRODUCTION

Knowledge-based systems are typically constructed from two main components: (i) an interpreter (or 'inference engine'[1]) which executes expressions in the representation language, and (ii) the knowledge which is encoded as expressions of the knowledge representation language. Of these two, the first tends to be fairly static (and might be provided by an external agency as a complete package). The knowledge encoded in a system is the more volatile component: it is the encoded knowledge which is augmented and modified during system development and maintenance. It is the interpreter which constrains the behaviour of the system and gives (in a weak fashion) a sense to the knowledge that is encoded in the system: at the very least, it enables the encoded knowledge to be manipulated in 'meaningful' ways.

In real-time and safety-critical knowledge-based systems, the interpreter may, in many cases, be regarded as a software component of a relatively ordinary kind. This suggests that the benefits of formal specification can be obtained for the interpreter component. As the interpreter tends to be a stable component of the system, formal specification can be attempted without risk of having repeatedly to redo the specification and its refinement once an implementation has been achieved. In addition, the success of the entire system depends upon the correct functioning of the interpreter because, in a sense, it enforces meanings. For safety-critical systems, one would like the best possible guarantee of correct functioning, so formal specification appears, again, to be of utility.

Even in more experimental systems, formal specification may still have a useful role to play. This is because a formal specification can be considered to be a formal model of a system (albeit one that does not actually execute). Typically, experimental systems are implemented in order to determine their properties: if a formal specification is undertaken, many of these properties can be derived on an *a priori* basis. The formal specification exercise tends to sharpen one's intuitions about an experimental system, and this, too, can be of great help in system and theory development.

In this paper, we describe four case studies in the formal specification of AI programs (using Z—Spivey, 1988, 1989). Each case study treats the formal specification in a different way, and each system was specified in Z for different reasons (although there are some common themes). The four case studies divide naturally into two families, each of two specifications.

The first family contains specifications of a blackboard system (Hayes-Roth, 1983; Nii, 1986; Engelmore, 1988) and of the author's CASSANDRA architecture (Craig, 1989). Both of these systems are of relatively high complexity and both are indended for use

1. Throughout this paper, we will always use the term 'interpreter' to refer to that part of a knowledge-based system which performs the tasks associated with an inference engine.

in complex domains whose problems require the integration of different kinds of knowledge. The CASSANDRA architecture was designed for use in safety-critical, distributed applications where high reliability is essential (see Craig, 1989, for one such application). The Z specifications of these two systems are to appear in book form (Craig, 1991a).

The second family consists of production rule interpreters, although of a somewhat unconventional kind. The first, called ELEKTRA (Craig, 1991c, 1991d, 1991e), is a forward-chaining system with extensive meta-level facilities. The second, called RPS (1991g), extends ELEKTRA by adding a frame system and rulesets as well as by an enriched control structure. Both ELEKTRA and RPS are reflective systems (Hayes, 1974): that is, they are intended to reason about their own contents and operation. The systems in the second family are experimental, and we are using formal specifications as a means of exploring their capabilities and implications *without* having to engage in an implementation, although the Z specification of ELEKTRA was used to develop an implementation in Scheme (Sussman, 1978).

The next two sections of this paper describe the systems of each family and their specifications. We explain the reasons for undertaking the specification in each case. We also discuss some of the problems that were encountered as the specifications were developed. Of the two, section three is the longer: this is because the systems described in it will be less familiar to the general reader. Finally, we end with a general discussion of the role of formal specification in the development of AI systems.

2 BLACKBOARD AND CASSANDRA SPECIFICATIONS

In this section, we discuss our specifications of the blackboard and CASSANDRA architectures. The complete specifications are to be published in book form (Craig, 1991a). In each case, we report our reasons for undertaking a formal specification and outline the scope and status of the specification.

2.1 The Blackboard Specification

The blackboard specification was undertaken in order to try to define an interpretation of the architecture. The blackboard architecture, as has been observed by a number of authors (Hayes-Roth, 1983; Nii, 1986), is a relatively informal construct which is open to different interpretations. Our initial goal was to define our interpretation of the architecture. In addition, having presented an *informal* specification in English and CommonLISP pseudo-code (Craig, 1991b), it seemed natural to attempt a specification in Z. Z was chosen for its schema calculus. The schema calculus makes for very much more modular (and perhaps, thereby, understandable) specifications; also, we wanted to engage in a model-oriented for reasons that will become clearer in the next section.

The aim of the exercise was to develop as complete as possible a specification of a blackboard framework (i.e., a blackboard system interpreter), but ignoring interfaces to the user and to external databases and data sources.

In order to produce the specification, we adopted a top-down method, with each main module receiving independent attention. This approach is natural given the modular nature of a blackboard framework, and given our previous experience in constructing three such systems. The particular interpretation of the architecture that we adopted is similar to BB1 (Hayes-Roth, 1985): control is seen as a knowledge-based task which operates on a globally accessible agenda. Knowledge Source activation is via blackboard events.

A specification of each of the main components of the interpreter was developed. That is, we specified the following components in detail:

- The blackboard, its abstraction levels and the entries which reside on abstraction levels.
- The global agenda and the KSARs (Knowledge Source Activation Records) which reside in it.
- Knowledge Source structure and the Knowledge Source database.
- The event system.
- Knowledge Source triggering, precondition and action evaluation.
- The main interpreter cycle.

In each case, we attempted to produce as general a specification as possible. Thus, the specification allows an arbitrary number of abstraction levels on the blackboard (although we allow only one blackboard—panels can be obtained merely by choosing appropriate names for abstraction level); entries are represented as partial mappings, so new attributes can be added with ease.

The specification of each module involved the development of appropriate basic structures and the definition of operations and predicates over them. For example, the blackboard database is considered to be a state space with appropriately defined operations.

The resulting specification is highly modular. The decision to specify the event system as a separate component paid the dividend that, by schema composition, the blackboard, event system and triggering modules could be combined to form a more complex module in a simple fashion. A similar technique was employed in the specification of the main interpreter loop.

The interpretation we chose for Knowledge Sources led us to encounter problems that we do not feel that we have adequately solved as yet. We followed the BB1/BB* (Hayes-Roth, 1985, 1986) model. This model allows arbitrary predicates (expressed as arbitrary pieces of implementation language code) in preconditions; it structures Knowledge Source actions as sets of production rules in which arbitrary code may be employed as condition elements[2].

The need to take into account problem-specific operations turned out to be an aspect of the specification that gave the least satisfaction: the way in which it was solved was simply to state pre- and post-conditions that all predicates must satisfy. We were, however, able to prove a result: that result is simply that, on assumption that no user-supplied predicate which satisfies the invariant will alter the blackboard state.

A similar problem was encountered when we came to specify the scheduler. The account of scheduling that we prefer is akin to Hayes-Roth's Blackboard Control Model (Hayes-Roth, 1985). Because the details of scheduling depend so crucially on the problem to be solved and on the characteristics of the problem domain, we found the scheduler almost impossible to specify in any detail. Indeed, we resorted to defining the scheduler as a function which was applied on each interpreter cycle. Again, we were forced to gloss over details and to take an indirect route in producing a specification. Because of the ambiguity inherent in the scheduler, we were unable to prove termination: without detailed knowledge of the scheduling mechanism, it is impossible to prove that the system will do anything useful! These were other unpleasant aspects of the specification exercise, even though the additional details can be filled in once the details of an application are known and understood.

In addition to developing a specification of the blackboard interpreter, we also proved a number of results about the system. This was part of our intention when we undertook the specification: many properties of blackboard systems form part of the 'folklore' and are not documented in an easily accessible place.

The proofs that we produced fell into two categories: the first deals with operations defined over the various system components, the second with more obscure properties. In the first category, for example, there is the proof of the proposition that the immediate addition of a new attribute-value pair to a newly created entry is identical in result to creating the entry with the additional pair as an element of the attribute-value set that is used to initialise the entry. In the second category, there is a proof of the proposition that every entry resides on one and only one abstraction level—this property is only occasionally stated with enough emphasis: in order to see that it must be the case, one has to think for a while about the status of entries.

The complete specification covers all aspects of the interpreter. It amounts to over one

2. We also need to give a semantics—this has been done in outline for RPS (Craig, 1991g) and for CASSANDRA (unpublished).

hundred pages of A4 paper, including proofs and explanatory text. The version which will appear as Craig, 1991a, is intended for tutorial use (as an example of a large specification), so some of the explanatory text could be removed. In addition, the specification concentrates on the top-level, although we have gone a little way to producing a refinement. Overall, the specification exercise proved to have some utility: for one thing, it showed that a formal specification with relatively complete coverage *could* be produced, and that such a specification could be used to pin down an interpretation of the architecture, as well as allowing important properties to be proved.

In the near future, we hope to undertake a complete refinement of this specification down to the code level.

2.2 The CASSANDRA Specification

The specification of the author's own CASSANDRA architecture (Craig, 1989) was undertaken immediately after the completion of the blackboard specification outlined above (and is also documented in Craig, 1991a). The CASSANDRA architecture is a derivative of the blackboard architecture, so it seemed plausible that the specification could employ many of the structures developed in the earlier exercise: this turned out to be an important factor in shortening the resulting document, even though many of the inherited blackboard components need to be altered in fairly obvious ways.

The reason for giving CASSANDRA a formal specification was that we wanted a complete definition of the 1989 version of it in as unambiguous a form as possible. As has been noted, the blackboard architecture has remained an informal construct for many years, and we view this as a weakness. We wanted CASSANDRA to avoid problems of interpretation, so a formal definition was undertaken. Because of the different nature of the exercise, we felt free to be less strict than with the blackboard interpreter: this is the reason that we merely adopt blackboard component specifications and leave the reader to make the necessary changes—they are all quite simple, as has been noted.

The CASSANDRA architecture represents one approach to building distributed blackboard systems. The architecture's primary component, the *Level Manager*, is, basically, one abstraction level which is equipped with a local control component. Level Managers communicate with each other by sending messages along uni-directional channels when solving problems.

The fact that messages are sent to and received from asynchronously operating processes entails that CASSANDRA systems are prone to deadlock and livelock problems: this is felt most acutely in the way in which Knowledge Sources operate. In the definition of the architecture, we altered the semantics of Knowledge Sources so that communications could be incorportated in a relatively safe way.

Because the CASSANDRA architecture is so closely related to the blackboard architecture, we concentrated on those aspects which are CASSANDRA-specific. In other words, we concentrated on the communications system and on the interpretation of Knowledge Sources. The specification of the communications system was essentially straightforward: it amounted to the definition of message formats, queues and operations to send, receive and route messages.

The specification of Knowledge Sources was considerably more complex. Although the CASSANDRA architecture employs event-based Knowledge Source triggering, precondition evaluation divides into local and non-local preconditions. Local preconditions test the state of the entries that are local to a Level Manager. Non-local preconditions require messages to be sent. In fact, non-local preconditions divide into two types: those which wait for messages to arrive, and those which send messages and then wait for a reply. (Knowledge Source actions are only allowed to send messages—they may not wait for replies). The interaction with the communications system has the entailment that a precondition may wait an indefinite time before evaluating to either true or false. While a precondition is waiting for a message, the architecture states that it has a undefined truth-value.

The Z specification of preconditions deals with structures that are needed to support the interaction with the communications system, structures for maintaining waiting preconditions, and the evaluation process once a message has been received. Luckily, it is possible to define primitive operations for message processing, so the difficulties encountered with arbitrary predicates do not reoccur in this case (although they do in the case of local preconditions).

The specification of the 1989 version of CASSANDRA is relatively complete. Where necessary, we imported the names of schemata developed during the blackboard specification (in some cases, we gave the redefined form of blackboard schemata in order to be more complete). A specification of the Level Manager, including its control structure, is part of the full document. We believe that the specification, although not totally complete, is adequate to define the limits of the 1989 version of the architecture. From this specification, it is quite possible to move on to the refinement and implementation phase.

3 ELEKTRA AND RPS

In this section, we concentrate on our efforts to specify two production rule interpreters. The interpreters are for production systems of a relatively unconventional kind. Our aims in developing these specifications are many and varied, but they were both undertaken as attempts to use the specification as a model of an experimental system; in other words, we used formal specification *instead of* implementation. That having been said, we must record the fact that the first specification (of the ELEKTRA system) was refined down to the level of Scheme code to produce a working implementation. The second specification (of the RPS system) is not yet complete, although its state is such that we can report on it in some detail.

3.1 ELEKTRA

The ELEKTRA system (Craig, 1991c, 1991d, 1991e) is basically a forward-chaining production rule interpreter. The Z specification is to be found in Craig (1991c). When the constructs that facilitate meta-level reasoning are removed, ELEKTRA is a rather conventional system.

The basic interpreter consists of the following modules:

- A working memory.
- A production memory.
- A pattern matcher (unification).
- A conflict set.
- A conflict resolution module.
- A (default) main loop.

Of these modules, the conflict resolution module is not part of the specification[3]: in the implemented version, we used an extremely simple conflict resolution strategy (one which merely chooses that rule instance which was first inserted into the conflict set), although we did specify the strategy in Z before implementing it. In addition, we used a version of unification based on the VDM specification in Jones (1990, Chapter 5), so only an interface (represented as a relation) appears in the report (Craig, 1991c).

The formal specification falls into two parts: the first contains specifications of all the modules listed above. In addition, the specification contains an abstract syntax for condition-elements and rule actions: the abstract syntax defines a representation based on the syntax of first-order logic. Connectives (with the exception of negation) and quantifiers are implicit: connectives are provided by the standard interpretation of production rules (see, for example, Waterman, 1978).

Once again, the specification treats each main data structure as a state space and defines operations over these state spaces. Rules are treated as tuples and not as state spaces.

As part of the specification exercise, we defined and enforced a strict distinction between rules and their instances. In ELEKTRA, a rule is a complex structure that contains, essentially, pattern templates. A rule instance, on the other hand, is represented as a pair which consists of the rule identifier (which must be unique) and a set of variable bind-

3. We did, in fact, specify a couple of relatively simple conflict resolution strategies, in order to discover how difficult they were to specify. The distinction between rules and their instances (see below) served us well.

ings. The rule identifier is not used as the criterion of identity in the conflict set—rule identifiers *together with* variable bindings form the criterion. The distinction between rules and rule instances makes the operation of the conflict set and of conflict resolution strategies easier; it also makes clear what gets executed in order to change working memory state.

A central part of the specification of the conflict set schemata involved the definition of operations to manipulate the conflict set (i.e., operations to clear it, add an instance, search for one or more instances, remove an instance, find a rule in production memory given an instance of it, and so on): these operations were deemed to be useful when constructing actual conflict resolution modules, and in giving the meta-level access to the interpreter's control structures.

The main interpreter schema contains the global state. This is represented by working memory, production memory and the conflict set. The specification as a whole contains schemata for accessing and updating working memory and for adding and retrieving rules from production memory. In addition, a number of counters are specified: the counters represent the passage of time in the system (rule instances are, for example, time-stamped).

Thus far, ELEKTRA is highly conventional: it contains no features or facilities that are in any way novel. This was a deliberate design decision: we wanted ELEKTRA to be useble as an ordinary production system. It is when the meta-level is added that matters become rather more interesting.

The specification of ELEKTRA followed the same path. We began by specifying *only* the object-level structures—these were necessary, given the decision mentioned in the last paragraph. Once the object-level only system had been specified, we extended it by adding the meta-level. The result is, essentially, a second system, although the second specification is wholly compatible with the first, so the schemata defined to provide an object-level only control structure apply to the full meta-level version.

In producing the second version of the system, we had to make provision for meta-rules. At the start of the exercise, we had decided that we would make no *a priori* restriction on the number of meta-levels and to use a numerical tag to differentiate between levels (a tag value of zero indicates object-level, a value of one indicates meta-level, a value of two indicates meta-meta-level, and so on). In altering the specification to cater for meta-rules at a potentially infinite number of meta-...-levels, we decided to separate meta-rules from object-rules. This required the definition of a separate variable in the global state schema and the introduction of an invariant which stated that each rule could not reside on more than one level.

The next part of the specification involved defining schemata that provide operations to support the meta-level.

Now, one interpretation of meta-rules is that they inspect object-rules—in ELEKTRA, this generalises so that rules at a given level can inspect rules at all lower levels—Davis, 1980, for example. Quite clearly, inspection of rules involves matching and analysing them. This is a different mechanism from that normally used to match working memory elements. Rather than complicate the system, we decided that a pre-processor was necessary: the pre-processor (called, somewhat grandiosely, the 'Rule Compiler') analyses rules and generates working memory elements which describe rule structure and contents. By loading rule analyses into working memory, a single matching process can be used—this simplifies the structure (and use) of the system.

The first part of the specification of the meta-level in ELEKTRA consisted, therefore, of the specification of the rule compiler: this turned out to be a relatively simple exercise given the abstract syntax. The rule compiler was always intended to be used off-line, so we made use of working memory initialisation schemata from the object-level only version to define the interface between the rule compiler and the system.

The next major part of the meta-level specification required us to define operations that could be executed from within rules. Sets, bags and lists were the obvious data structures that needed to be added to the system. We defined these (although only sets appear in Craig, 1991c, for reasons of space) and also defined an extensible interface so that such procedural elements could be called from rule conditions (the interface required the definition of an escape mechanism so that the pattern matcher would not be called).

Sets, bags and lists were important in the meta-level because we wanted ELEKTRA to be able to reason about its own control behaviour. More precisely, we wanted the system to be able to support rule interpreters that were different from the interpreter that actually runs the system. Since this point needs some explanation, we dwell on it in order to clarify matters.

In ELEKTRA, the default behaviour of the system is forward-chaining (with or without conflict resolution). It is well known that a forward-chaining system can emulate a backward-chaining one. However, the existence of a meta-level entails that rules can be chosen for firing on bases different from the simple syntactic methods employed by general-purpose conflict resolution modules. Since a forward-chaining system describes a universal computing maching (a Post system, in fact), it is possible to define interpreters as sets of production rules (in other words, ELEKTRA is a *universal* Post system). This observation is the crucial one for understanding how ELEKTRA can support interpreters that differ from forward-chaining while using that approach as its default mechanism. In practical terms, the implementation of a rule interpreter consists of the execution of a set of forward-chaining rules which select other rules and execute them (see Craig, 1991d, for more information and some example interpreters).

It should be clear that part of the ELEKTRA meta-level support must consist not only of facilities to analyse rules, but also to give meta-rules access to the condition-matching and action-execution primitives. Furthermore, if control is genuinely to the subject of reasoning, provision should be made to give access to the conflict set and to rule instances—the latter are particularly important because the action-execution operations defined in the object-level only system operate *only* on rule instances. The major part of the specification of the meta-level version of ELEKTRA consisted of making these operations visible to rules (the schemata were defined as part of the earlier exercise, and the escape mechanism was also defined, so the task was relatively straightforward). In addition to the obvious operations, we added some more in order to provide additional functionality: the additional operations extended the range of options so that, for instance, a single condition element could be tested against working memory (the basic visible operation tests conjunctions of elements)[4].

The control loop of the specification proved to be the hardest part. This is because the user is intended to have a great many options when using the system. For example, the user may want to start a session with ELEKTRA using the conflict set and then turn it off; alternatively, meta-rules may store rule instances in the conflict set, in local variable bindings or in working memory elements, and the option must be signalled to the basic interpreter so as to prevent it exhibiting its default behaviour. Because of the modularity of Z, we found that the various choices could be catered for with relative ease (and a major problem was in deciding which choices could reasonably be excluded). Also, we had to provide options affecting the level at which ELEKTRA began matching rules: this, too, turned out to be relatively simple. Eventually, we completed a control loop for the meta-level which provides a reasonable set of features.

The most significant difficulty that we encountered in the ELEKTRA specification derived from the fact that ELEKTRA is intended to be an *open* structure. By this, we mean that the user (or user-supplied rules) have access to every part of the system's internals. As has been seen, access to all the main data stores and control structures was provided. This gives the specification the feel of a description of a kit of parts which need to be assembled before (or even during) use: this would be fine, were it not for the fact that only certain combinations make real sense. What we have, we believe, is an adequate specification (the Scheme implementation witnesses that fact) of some of the ways in which ELEKTRA can be configured.

3.2 RPS

The RPS system (Craig, 1991g) is currently under specification and has not yet been fully documented. RPS is a development of ELEKTRA. RPS extends ELEKTRA by the inclusion of the following additional structures:

4. The problem is one of matching quoted expressions, in other words. There is clearly no problem with matching expressions at the same level as the rule—that is what the matcher is for; the problem is that of matching expressions lower in the hierarchy.

- A declarative database organised as frames.
- An agenda-based control structure.
- The use of tasks as the primary unit of control.

The additional structures make for a system which is reminiscent of a blackboard system and which is also reminiscent of Lenat's EURISKO (Lenat, 1983).

RPS is not a mere extension of ELEKTRA, despite what the above list suggests: RPS is a re-working of some of the ideas in ELEKTRA using different organising and interpreting principles. The new system is still indended to be a reflective one, but it organises its activities in terms of tasks: a task must be performed in order for the system to do anything. In RPS, a task is a collection of rules, together with an objective that they must achieve[5]: one of the most important tasks that RPS must perform its its own interpretation—we have identified the basic operations that the basic *RPS-task* must perform in executing the other code. The basic interpreter task is similar to the three fundamental Knowledge Sources in BB1.

A problem that we found with ELEKTRA (and which seems to be endemic to production systems that do not provide higher-level structuring facilities) is that it became hard to determine what rules did: in RPS, we hope to have solved this by collecting rules into tasks. Since the tasks that need to be performed will vary with time, we allow tasks to be created at runtime—rules construct tasks from other rules—although the default is for the system to execute pre-defined tasks. Within a task, there can be rules at a variety of levels: some tasks will perform meta-level operations, whereas others will perform object-level tasks only. We allow all tasks to contain a mixture of object- and meta-rules: sometimes, the meta-rules will interpret the object-rules in a task in a way similar to ELEKTRA's interpreters. However, since we now view interpretation as a separate type of task, tasks can refer to and use other tasks. We believe that this cleans up the structure of ELEKTRA in a crucial way.

A second way in which RPS differs from ELEKTRA is in the structure of its rules. The ELEKTRA rule structure was the conventional bipartite one: each rule had a condition-part and an action-part.

From our experiments with ELEKTRA, it became clear that rules were used in very different ways, even though they had a uniform structure. This occurs because of the different ways in which the interpreters treat rules. As a result, we have decided to structure RPS rules as frame-like objects: this allows different interpreters to define the attributes that they access when interpreting rules (this is similar to Lenat's treatment of rules in EURISKO). A classic example of what we want for RPS is the distinction between an approximate and a definitive test of relevance. Some rule interpreters may want a

5. Tasks are not necessarily viewed as executing concurrently, although this is a possibility. Certainly, many tasks can be simultaneously active: whether this is implemented as parallelism is another matter.

rough test of relevance before considering a rule for execution; others may be more cautious and require a stronger test; yet another example is the *unless* condition that has been proposed by a number of authors.

The fundamental point is that, because ELEKTRA allows rule interpreters that interpret conditions in different ways, it becomes impossible to tell what a condition actually means. In RPS, we want to be able to give a semantics to the representations that are used in the system.

The desire to include, reason about and to enforce semantics is one of the principal aims of RPS. We want to enforce semantics because we want to avoid the problems we encountered in the blackboard specification (and in the construction of a number of blackboard applications): entry attributes are too often considered to be symbols which do not have a fixed interpretation. In Craig, 1991b, we propose a framework similar to that adopted for BB* which would enforce the semantics of attributes. In RPS, because it is a reflective system, what is the equivalent of attribute semantics needs to be available to the system itself.

Although we do not intend RPS to be a real-time interpreter, nor yet one which operates in safety-critical domains, we want to be able to experiment with aspects of reflection that may well have some relevance to these areas: for example, it would be of considerable utility if a system could diagnose and even repair its own faults; the ability of a system to reason about its own limitations and capabilities has already been noted.

The enforcement of semantics also impacts upon working memory. In RPS, we have decided to organise working memory as an abstraction hierarchy. The objects which reside on each abstraction level are (instances of) frames. In ELEKTRA, working memory elements are instances of relations. Nothing in ELEKTRA, however, really says what a relation symbol means: it is just a symbol which can be pattern matched (or, in some cases, executed). In RPS the idea is that, because working memory elements are objects with considerable internal structure, they can be described and defined: this information can then be made available to the system in its inferential processing[6]. A particularly simple example of the kind of issue that can be handled is semantic well-formedness: a constraint can be easily applied in the RPS architecture which forbids the co-occurrence of two particular attributes in the same working memory structure.

The adoption of a frame system also allows variation in the methods by which structures in working memory are accessed. In particular, information may be fetched from other parts of the frame hierarchy when a working memory fetch is performed. This suggests that working memory structures can refer to other information: for example, contextual information. This additional information can be accessed as a one-step pro-

6. Similar things can be done in a logic or in a production system: below, we explain why we use structured objects.

cedure if the access method enables it. Another way of viewing this property of working memory is that pieces of procedural code can be used to access working memory and that these code pieces can be given an explicit definition, and, hence be subjected to inference. In ELEKTRA, the system relies on a uniform access method (in ELEKTRA it is a linear search which uses unification as its test): this entails that 'implicit' information must be derived either as the result of inference or else as the result of multiple working memory fetches.

This, then, is the background to RPS. To date, we have completed only a part of the specification. The current state of the specification is as follows.

We have specified the entire frame system. This involves the specification of the frame database, access operations on frames, frame addition operations and slot inheritance (single inheritance of a traditional kind). We have not yet specified demons such as **if-needed** and **if-added** methods: we are not yet convinced of their utility within the context of RPS.

Because of its procedural interpretation, we have also given a denotational (standard) semantics (in Z) for a provisional rule language. The rule language is used to define the rules that RPS will use in problem-solving. The language contains predicates, relations and actions as well as local variables. An essential component of the language is the way in which it treats the state of the frame database: this serves the role of the store in the semantics. At the time of writing, the rule language is still an experimental aspect of the system.

The remaining components are all relatively simple to specify, and we are leaving them until we have a better understanding of the ways in which rules and frames interact.

In the work on RPS, we are concentrating on using the formal specification as a way of experimenting with the design and with the concepts in an AI system. To date, we have found that it has been an extremely useful way of working. The approach we are taking to the design is not that of concentrating on structures which are easy to specify: instead, we are using the formal structures to help us determine what useful structures can be defined. Utility is an informal concept which is based upon the implications of properties of those structures which we have considered to date. As with ELEKTRA, we are also using the Z specification to develop a formal model of the system. However, the emphasis is now rather more on semantics than on implementation structures: the relative lack of semantics was one of the important lessons of the ELEKTRA experiment[7]. This fact is demonstrated most clearly in our need to give a denotational semantics for rules and their component structures: the denotational semantics, when

7. We have, in fact, attempted a denotational semantics of ELEKTRA. The result was unpromising because the semantics looked like a much more abstract version of the operational semantics: what appears necessary is a semantics which more closely resembles a model of a first-order theory.

complete, will help in defining what rules do—in other words, the semantics will help us in producing models of rules. To date, we are amply satisfied with the use of Z in specifying the program and in defining the semantics.

4 CONCLUSIONS

In this paper, we have briefly surveyed our use of the Z specification language in the specification of AI systems. We have completed specifications of three systems and have a fourth in a relatively advanced stage of development. In the case of the ELEKTRA system, we found that the formal specification greatly assisted in the production of code: the programming task would have been somewhat more difficult if we had not bothered to produce and refine a Z specification as an initial step. For ELEKTRA, we found (as we had expected) that time required to produce the Z specification was worthwhile because it meant that we had already spent time in analysing the system, in identifying problem areas, and in finding acceptable solutions to problems.

For the blackboard interpreter specification, we found the entire exercise extremely useful in that it enabled difficult concepts to be better understood (dispite our considerable experience in the field). The blackboard specification has been refined in some areas, and we hope to refine the entire specification down to code level, the end result being a complete implementation based upon a formal derivation process.

In our more recent Z specifications, we have extended our goals. In addition to specifying the code that implements a system or program, we are now interested in the problem of semantics. In the blackboard and CASSANDRA specifications, we found that it was hard to deal with matters of attribute interpretation; in addition, we found the purely operational view of the systems somewhat restrictive. This can be seen most clearly in the way in which Knowledge Sources were treated in the blackboard and CASSANDRA specifications. In the ELEKTRA and RPS specifications, we are interested in using the formal specification as a basis for models that the programs themselves maintain and manipulate: this clearly entails a semantic component in the model because we would like to capture the 'intended' interpretation.

Although we have found Z to be quite capable of formalising a semantics, there are difficult, conceptual problems that remain to be solved. To date, we have concentrated on a version of semantics that deals exclusively with the behaviour of the systems in purely computational terms (although we have not attempted to specify time-space behaviour). The rich variety of mathematical concepts that the Z language contains has certainly made this task easier than it would have been in the more austere setting of pure, first-order logic (whether typed or not). What we really do need is a form of 'natural' semantics which will relate the formal structures in the program to the world—in a sense, this is the old problem of how applied mathematics has a sense. We have argued elsewhere that a different conception of semantics is required (Craig, 1991f), but our investigations are, as yet highly incomplete. However it turns out, what *is* clear is that any reflective system must be able to reason about a great many things: our work on formal specification in this area has shown that formal specifications are a useful tool in developing models that programs can manipulate. At the very least, we may have found a method for developing systems that are able to reason about their own behaviour in case of failure.

In conclusion, we state two things. Firstly, we believe quite strongly that formal specification is an *extremely* useful tool in the development of those components of AI systems that are more or less stable (interpreters, that is). Second, because we have only worked on the specification of interpreters (for our concern has, for part of the time, been with the control behaviour and with trying to make control behaviour as rich as possible), we have ignored the specification of representation languages. The rigorous specification of representation languages is crucial because it relates that which is encoded to some external world. This, second, aspect is what we are beginning to tackle in the RPS project. As we have noted, the semantic issues are complex and the relationship between the knowledge encoded in a system, and the computational behaviour the system is expected to exhibit, is only part of the general problem of correctness in AI systems.

REFERENCES

Craig, I.D., *The CASSANDRA Architecture*, Ellis Horwood, Chichester, England, 1989.

Craig, I.D., *The Formal Specification of AI Systems*, Ellis Horwood, Chichester, England, *in press*, 1991a.

Craig, I.D., *Blackboard Systems*, Ablex Publishing Corp., Norwood, NJ, *in press*, 1991b.

Craig, I.D., *The Formal Specification of ELEKTRA*, Department of Computer Science, University of Warwick, *in prep.*, 1991c.

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

Craig, I.D., *ELEKTRA: A Reflective Production System*, Department of Computer Science, University of Warwick, *in prep.*, 1991e.

Craig, I.D., *Meta-Knowledge and Introspection*, Department of Computer Science, University of Warwick, *in prep.*, 1991f.

Craig, I.D., *The Formal Specification of RPS*, Department of Computer Science, University of Warwick, *in prep.*, 1991g.

Davis, R., Meta-rules: Reasoning about Control, *AI Journal*, Vol. 15, pp. 179-222, 1980.

Engelmore, R. and Morgan, T. (eds.), *Blackboard Systems*, Addison-Wesley, Wokingham, England, 1988.

Hayes, P.J., *The Language GOLUX*, Department of Computer Science, Essex University, 1974.

Hayes-Roth, B., *The Blackboard Architecture: A General Framework for Problem Solving?* Report No. HPP-83-30, Heuristis Programming Project, Computer Science Department, Stanford University, 1983.

Hayes-Roth, B., A Blackboard Model for Control, *AI Journal*, Vol. 26, pp. 251-322, 1985.

Hayes-Roth, B., Garvey, A., Johnson, M.V. and Hewett, M., *BB*: A Layered Environment for Reasoning about Action*, Technical Report No. KSL 86-38, Knowledge Systems Laboratory, Stanford University, 1986.

Jones, C.B. and Shaw, R.C. (eds.), *Case Studies in Systematic Software Development*, Prentice Hall, Hemel Hempstead, England, 1990.

Lenat, D.B., Theory Formation by Heuristic Search. The Nature of Heuristics II: Background and Examples, *AI Journal*, Vol. 21, pp. 31-60, 1983.

Nii, H.P., The Blackboard Model of Problem Solving, *AI Magazine*, Vol. 7, No. 2, pp. 38-53, 1986.

Spivey, J.M., *Understanding Z*, Cambridge Tracts in Theoretical Computer Science No. 3, CUP, 1988.

Spivey, J.M., *The Z Notation*, Prentice Hall, Hemel Hempstead, England, 1989.

Sussman, G.J. and Steele, G.L., Jr., *The Revised Report on Scheme*, AI Laboratory Memo 452, MIT, 1978.

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