The Formal Specification
of a
Blackboard Framework

Iain D. Craig

Department of Computer Science

University of Warwick
Coventry CV4 7AL
UK EC

ABSTRACT

The blackboard architecture is a complex, though powerful, model of problem-solving, and opinions vary as to its interpretation. The use of formal specifications for blackboard systems appears warranted by their complexity, their application in real-time and safety-critical domains, and because of the informality of the construct itself. This paper describes a Z specification of a blackboard framework, the aims of the specification, and the method by which it was executed. At present, the specification is only a top-level one (and occupies over 100 A4 pages, including proofs and explanatory text): this has allowed concentration on the interpretation of the architecture, and has allowed the formal proof of a number of properties which have, hitherto, had 'folklore' status. The specification exercise revealed a number of areas in which further work was required. The blackboard specification is one of a number of Z specifications of AI architectures that we have undertaken: the problem areas first identified in the blackboard specification have reappeared in the others, and we suggest ways of solving the problems which are, perhaps, of general utility.

1 INTRODUCTION

As the longer Appendix of Craig (1991a), we gave a specification of a blackboard interpreter[1]. The interpreter was specified in a mixture of English, CommonLISP code fragments and an ADA-like pseudocode. Although the specification is complete in the sense that all interpreter modules are included, we felt that the specification method did not allow us to capture all that we wanted. In particular, we felt that the specification could be interpreted in ways other than that which we intended, although we had taken great care to be as unambiguous as possible. For this reason, we decided to undertake a formal specification of a blackboard architecture: the formal specification appears in another book (Craig, 1991b).

The blackboard architecture, as has been observed by a number of workers (including Hayes-Roth (1983)), is essentially an informal construct. Interpretations of the architecture differ in detail, although there is a broad consensus on what, basically, constitutes a blackboard system. Because of the informality of the construct, and because many of the descriptions of blackboard systems are quite condensed, even the series of papers describing HEARSAY-II (Erman and Lesser, 1975; Hayes-Roth and Lesser, 1977; Lesser and Erman, 1977; Erman *et al.*, 1980), many properties of the architecture remain, more or less, in the folklore. The specification in Craig, 1991a, does little to make properties clear, although it does go a little way towards making the fundamentals clearer.

As part of a formal specification exercise, we wanted to pin our interpretation of the blackboard architecture down: we wanted, therefore, to construct a formal *model* of our interpretation. By adopting a formal specification approach, we hoped to make our interpretation of the architecture clear without having recourse either to implementation language code or to pseudocode, as we had earlier done. In addition, we wanted to prove a number of properties of the architecture in order (i) to show that they are natural properties (i.e., that they follow from our assumptions about the architecture), and (ii) to make them explicitly available in the literature. The properties that we wanted to emphasise concern the top-level specification: in other words, they are not discovered during the refinement of the formal specification, but relate to the model itself.

Having defined the goals of the exercise, we chose the Z specification language (Spivey, 1988, 1989). We did this for three main reasons: (i) Z is a model-oriented specification language (and the concept of a model was important in our conception of the role of specification); (ii) Z contains a schema calculus which allows the combination of small sections of specification into ones of larger scope (and also forces one, if the

---

1. In this paper, we will use the term *interpreter* or *framework* to denote all the basic software that is normally included with a *shell*.

calculus is used properly, to think in modular terms), and (iii) Z is a very rich language and its expressiveness was considered to be a bonus.

The specification that resulted from this exercise appears in Craig, 1991b. The specification is over 100 A4 pages in length and covers all aspects of the interpreter with the exception of interfaces to external databases and data sources. In addition, some aspects of the interpreter had to be omitted (these will be discussed in section 3.5). The specification has not been refined to any extent (although the Appendix to Craig, 1991b, contains a sample refinement): we hope to undertake the refinement in due course, producing an implementation as a result of this second exercise.

In this paper, we will explain what we specified (i.e., our interpretation of the blackboard architecture) and the way in which the specification was performed (these two topics are discussed in section 2 and in sections 3.2 and 3.3). Then, we discuss some of the properties of the interpreter which we formally proved (section 3.4). Next, we consider some of the more important limitations of the specification in its current form (section 3.5). In section 4, we review the work to date and consider some of the issues that arise as a result of employing formal methods in the production of AI software.

2 THE BLACKBOARD INTERPRETATION

2.1 Introduction

In this section, we briefly outline our interpretation of the blackboard architecture. The interpretation that we prefer is very close to that of Hayes-Roth (1985, 1986): it formed the basis of the NBB component of the BB-SR system (Craig, 1987a) and was the kernel of the specification we gave as the Appendix of Craig, 1991a. The interpretation contains all those components normally associated with blackboard systems. Where the interpretation differs is in the treatment of control (which we consider to be a knowledge-scheduling problem—see Nii, 1986a, 1986b, and Craig, 1991a) and in the structure of Knowledge Sources. We will describe each component of our interpretation in a little detail, but will concentrate on those aspects we have already mentioned.

2.2 The Blackboard and Entries

In our interpretation, the global database (or blackboard) is divided into a number of *panels*. Because we favour the Hayes-Roth (1985) blackboard control model, we normally expect there to be at least two panels: one for control and one for domain problem solving. No a priori upper bound is placed on the number of panels on the blackboard.

Each panel is divided into a number of *abstraction levels*. Again, the number of abstraction levels on each panel is not specified, but we would normally expect there to be at least two (for otherwise, the hierarchy generated by the abstraction levels is redundant). Each abstraction level is associated with a unique identifier: a constraint on abstraction levels is that they belong to one and only one panel. The identifier, of course, does not denote anything in itself—it is merely a way of identifying the abstraction level. In a similar fashion, panels are also associated with a unique identifier.

On each abstraction level, there reside *entries*. In our interpretation, entries are complex data structures and are realised as collections of attribute-value pairs. Access to a value is via the attribute name; clearly, attribute names must be unique within entries. Also, each entry is associated with a unique identifier: the identifier is used as a key when retrieving data from the blackboard. Entries may have attributes whose values represent references to other entries (either on the same or on different abstraction levels—cross-panel references are permitted): this enables 'solution islands' to be developed in a natural fashion.

In our interpretation, a fundamental constraint on entries is that each entry resides on *one and only one* abstraction level: that is, an entry can neither migrate from one abstraction level to another, nor can an entry reside on more than one abstraction level.

This point is fundamental because, as we have argued elsewhere (in Craig, 1991a, and in Craig, 1987b, in particular), the uniqueness of each abstraction level depends upon the entries which reside upon it. This is a logical point, and not merely an implementational one. The argument that we have to support this point is that the attributes which form entries determine the representational vocabulary that defines each abstraction level, and we assume that it is this vocabulary which defines the objects that can be represented at each level of abstraction[2].

2.3 Knowledge Sources

The interpretation of Knowledge Sources that we prefer is the following. Each Knowledge Source is composed of four main components. They are:

(i) An event-based *trigger*.
(ii) A state-based *precondition*.
(iii) A state-based *obviation condition*.
(iv) An *action* composed of one or more production rules.

Both the trigger and precondition must have been satisfied before the action can be executed. Before execution of the action, the obviation condition is evaluated: if it is satisfied, the Knowledge Source action is not executed. Obviation conditions are an additional check on the relevance of Knowledge Sources to the current blackboard state.

2. In HEARSAY-II, all abstraction levels shared the same attributes, it will be remembered, and so there is at least one counter-example to our position. In HEARSAY-II, functional role appeared to define each abstraction level and its contents: what our position amounts to is the reification of representational vocabulary.

In order to understand the last paragraph, it is necessary to make a distinction between the blackboard state and events. The distinction is standard, but is worth repeating for reasons of clarity.

The *blackboard state* is, quite simply, represented by the entries that reside on it. In other words, the state is composed of the entries that are to be found at all abstraction levels and the values of their attributes. When an entry is added to the blackboard or is modified, the blackboard state changes. Preconditions and obviation conditions examine the blackboard state: that is, they examine entries at a variety of abstraction levels and test the values stored in them (the state also includes the network of entries that comprise solution islands).

An *event*, on the other hand, is caused by the alteration or creation of a single entry. In our interpretation, there are three primitive operations on entries:

(i) A *new* operation which adds a completely new entry to be added to the blackboard.
(ii) An *add* operation which adds a new attribute-value pair to an already existing entry.
(iii) A *modify* operation which changes the value stored in a particular attribute of a particular entry that is already on the blackboard.

Each of these primitive operations causes a blackboard event: the events have the same names as the operations which cause them. There is no concept of entry deletion in our interpretation (and most blackboard systems do not engage in entry deletion): this is because the implications of a deletion are hard to determine in the general case, although a method has been proposed by Craig (1989). The three primitive operations are performed by the actions in the production rules which comprise the action-part of a Knowledge Source: they are the *only* operations which cause events.

When a production rule action executes one of these primitive operations, a blackboard event is caused. The event is associated with a variety of information: for example, the abstraction level on which the operation was performed, the name of the entry involved (in the case of a *new* operation, a unique entry identifier is generated, and it is this newly generated identifier which is associated with new events), the time at which the event occurred and the name of the attribute which was added or modified (in the case of *add* and *modify* events). The trigger of a Knowledge Source responds to events and has access to this information (in addition, it can read the values of attributes in the entry which caused the triggering event).

The only component of a Knowledge Source which can alter the blackboard state is the action. Triggers, preconditions and obviation conditions may only read the blackboard: they may not write on it. In a similar fashion, the condition-parts of the rules which comprise a Knowledge Source's action have read-only access to the blackboard.

Communication between triggers, preconditions, obviation conditions and actions is essential, so our interpretation of the Knowledge Source structure includes local variables for this purpose. Unlike BB1 (Hayes-Roth, 1985, 1986), our interpretation restricts the scope of Knowledge Source variables: they are only in scope within the Knowledge Source in which they are defined. In BB1, Knowledge Source variables seem to be globally visible. Our approach has all of the advantages of lexical scoping. If information (for example, the value stored in a particular attribute) needs to be passed from the trigger to the action of a Knowledge Source, local variables are used. In other words, the context within which a Knowledge Source action executes is represented by the values of local variables.

The final component of a Knowledge Source is the action. As we have stated, actions are composed of sets of production rules. The condition-parts of rules may only read the blackboard; the action-parts can read as well as write to the blackboard (all writing *must* be done by means of the primitive, event-causing, operations). Our interpretation of the architecture imposes no ordering on the evaluation and execution of rules: our NBB implementation used a simple, sequential order, although we are open to the idea that a particular control regime might be applied in a situation-specific fashion.

2.4 Control

Control is effected as a knowledge-based activity in our interpretation. In other words, we prefer knowledge-scheduling to event-scheduling: this makes our interpretation closer to HEARSAY-II and BB1 than to AGE (Nii and Aiello, 1979) or HASP/SIAP (Feigenbaum *et al.*, 1982). Control is performed by special control Knowledge Sources which respond to events on the control panel and cause alterations to its state (or to domain panels—our interpretation is neutral in this respect). We will refer to the generic process of control as *scheduling*.

At the lowest level of the control structure, there is a *global agenda*. The agenda is an ordered queue of *Knowledge Source Activation Records* (or *KSARs* for short). KSARs are the basic unit of scheduling: when a Knowledge Source is to be executed, the execution mechanism uses information that is stored in the KSAR which represents one of the Knowledge Source's instances.

When a Knowledge Source triggers as a result of a blackboard event, a KSAR is created for it and placed in the agenda. The KSAR contains information about the triggering event (and it also contains the bindings of the Knowledge Sources local variables). Each KSAR also contains a reference to the Knowledge Source which it instantiates. Thus, the scheduling problem is really one of selecting a Knowledge Source instance for execution: in other words, control is not about executing Knowledge Sources, but their instances—instances represent situation-specific incarnations of Knowledge Sources.

In order to enable the scheduler to choose a KSAR to execute, each KSAR contains scheduling information. In our interpretation, most of the scheduling information is inserted into KSARs by control Knowledge Sources; some scheduling information is, however, inserted by the mechanism which creates KSARs.

The scheduling problem therefore reduces to one of searching the currently available KSARs in the agenda to find the one whose scheduling information determines that it is the best to execute. This, of course, requires the application of control strategies, for it is these that determine how to evaluate the information contained in each KSAR.

Although our preferred interpretation requires knowledge-scheduling of an explicit kind (via a control panel and control Knowledge Sources), it is possible to use the structures that we have concentrated on above in producing other kinds of scheduler. In particular, in Craig, 1991a, we indicate that the specification can be adapted to incorporate an *intelligent scheduler* of the kind used in HEARSAY-II. In other words, our interpretation is broad enough to allow other interpretations of the control problem.

3 THE SPECIFICATION

3.1 Introduction

In this section, we outline our approach to the specification of a blackboard interpreter that conforms to our interpretation of the architecture. In the next sub-section, we state the extent of the specification and mention those aspects of the resulting Z specification which differ from the interpretation outlined in the previous section. In section 3.3, we describe our principal concerns and explain how we approached the specification. Then, in section 3.4, we describe some of the properties that we have proved as a result of the specification exercise. We conclude the section with a review of some of the problems that we encountered: we view the problems as being as important as the overall success of the specification exercise.

3.2 Specification Scope

The scope of the Z specification was all the components described in the last section. In other words, we intended to specify all parts of the blackboard interpreter that did not deal with external interfaces (to build a blackboard application, it is frequently necessary to provide interfaces to external databases and to external data sources, as well as to computational structures that process the solution when it has been found—user interfaces and so on). Because we wanted the specification to reflect our interpretation of the architecture, we considered that external interfaces of all kinds would distract the reader of the final document.

More specifically, we gave a formal specification of the following components of the interpreter:

(i) The blackboard, its abstraction levels and the entries which reside on those abstraction levels.
(ii) The event system.
(iii) Knowledge Sources.
(iv) KSARs and the global agenda.
(v) Knowledge Source actions: i.e., the rules which are contained therein.
(vi) The module which supports Knowledge Source triggering.
(vii) The control loop.

The term *event system* refers to that component of the interpreter which processes blackboard events. The event system generates KSARs and records the information which describes the triggering context (the abstraction level on which the event occurred, the time it occurred, and so on). This information is generated by the triggering module: the triggering module searches Knowledge Sources and evaluates their triggers. The control loop was specified as an attempt to collect the module specifications into one, consistent whole.

As can be seen, all of the components listed above correspond directly to the modules identified in section 2 above. In this sense, the Z specification covered all of the interpreter. However, as it turned out, the Z specification does not reflect *all* of our interpretation: in particular, we were forced to treat the scheduler in a way that is quite dissimilar to our interpretation, and we decided to omit obviation conditions (the book in which the specification appears is partly oriented as a tutorial and obviation conditions contribute nothing that is conceptually new). The reasons why the scheduler turned out to be very different are given in section 3.5 below.

3.3 Approach

The basic approach we adopted in the development of the formal specification was that of giving each main module a separate specification. In other words, we adopted an essentially top-down approach to the specification process: each major module is identified by the architecture, so, in a sense, all that remained was to refine each module.

Some of the modules clearly required a state-based representation: this is because these modules involve the storage of information of different kinds. A state space was defined for the blackboard database and for the agenda; other modules did not store information and, instead, represented conceptual structures which needed to be made explicit (the specification of Knowledge Sources being an example of this second kind of structure).

The specification of the blackboard required the representation of abstraction levels and entries. Both abstraction levels and entries were represented by partial maps: this allows information to be added or deleted easily (there are operations on partial maps which correspond directly to those one requires for addition and deletion).

In earlier versions of the specification, we represented entries independently of abstraction levels. This proved to be a difficult strategy because it posed problems when it came to combine the specifications of these components. In the final version, we represent entries together with abstraction levels. Each entry has a unique identifier, and each abstraction level also has one. The two sets of identifiers form the domains of the maps which represent the two structures.

The blackboard state is represented by a set (of abstraction level identifiers) and by the mapping from abstraction level identifiers to entries. The entries are represented as a mapping from entry identifier to another map whose domain is the set of attributes and whose range is the set of attribute values. The representation that we chose makes the proof of a variety of different properties quite easy. The representation is also natural in the sense that it corresponds to one's intuitions about the way in which the blackboard operates; in addition, because the structures employed were primitives of the Z language, proofs of properties were rendered less complex (i.e., axiomatic properties of the structures could be employed directly without the need for additional lemmata and theorems).

Around the state space, a number of operations was defined. The operations are for adding new entries, modifying attribute values, adding attribute-value pairs, and so on. An operation to initialise the blackboard was defined: the initialisation operation initialises the blackboard with its full complement of abstraction levels. As part of the state space definition, constraints were defined: these constraints are aimed at ensuring that the blackboard state is always consistent.

A similar approach was adopted for the agenda. The specification treats the agenda as a priority queue (which is, really, what it is). The state space represents the state of the queue at all times. Operations were specified to add and delete KSARs, and to clear the agenda of all KSARs. An initialisation operation was defined: this set the agenda to the empty state.

Knowledge Sources were treated rather differently. These structures was represented as a state space (although a state space representation was assumed for the Knowledge Source database—i.e., that component of the interpreter which contains the Knowledge Sources and which is searched by the trigger module). Instead, this type of structure was represented as a tuple (a product, in other words), and selector and access functions were defined over them using axiomatic definitions. The tuple representation was chosen because the contents of a Knowledge Source do not vary with time: once Knowledge Sources have been defined and loaded into the system (into the Knowledge Source database, that is), they remain constant. A tuple representation was also adopted for the production rules that form Knowledge Source actions: this was for an identical reason.

Finally, KSARs were represented as partial maps. This is because we interpret them as collections of attribute-value pairs. The attributes which KSARs contain are different from those which entries contain, so there is no overlap between entries and KSARs. Each KSAR can be thought of as a state space, although we did not treat them as such in the specification. Operations were defined over the KSAR type in order to access and update the mappings. Many of the results which we proved about entries carried over to KSARs in a natural fashion because of the similarity of representation.

We began the specification exercise with the definitions of the state space and operation schemata for each of the above modules. We added schemata to represent the various events that could occur and then used schema composition to generate new schemata for the representation of compound operations. One such compound operation is the creation of a KSAR as the result of a blackboard event: this operation was defined by composition.

Once the fundamental modules had been specified in this fashion, we concentrated on the development of schemata to represent the complete triggering operation, the execution of Knowledge Source actions. The triggering module required the definition of more operation schemata and a considerable amount of schema composition. The execution module required the specification of a control loop (using universal quantification for we adopted an interpretation in which *all* rules whose condition-parts are satisfied by the blackboard state are fired). For the definition of the action execution module, we ignored the presence of local variables (see below, section 3.5).

The modules were then connected by means of composition or by quantification. At a relatively late stage of the specification, we defined a collection of schemata for entry identifier generation: once defined, we combined them with the schemata that represent the primitive entry creation operation.

In general, we found that, once the fundamental modules had been specified, the additional operation schemata that we needed were relatively easy to define. As a routine part of the specification exercise, we had to revise previously completed parts of the specification in order to make the definition of the later schemata easier (or, in some cases, possible).

In general, we chose representations from the repertoire of basic constructs offered by Z. One of the main goals of the exercise was to specify the interpreter in the most general way possible. This was reflected in our choice, for example, of infinite sets rather than finite ones: we did not want to have to worry about finiteness in stating the specification. Furthermore, we also wanted to avoid forcing implementation decisions: this is reflected in our frequent choice of conjunction over sequential composition in schema composition. In other words, we did not want to enforce one particular order in which the final operations would be performed (this was, in some cases, motivated by the fact that such a choice was purely arbitrary).

3.4 Properties and Proofs

The specification reported in the main part of Craig, 1991b, is a top-level one, although some refinement is presented in the Appendix. We concentrated on the top-level for a number of reasons, the primary one being that we wanted to give a formal proof of as many properties of the interpreter as possible. In other words, we wanted to concentrate on the properties of the specification as a model of the blackboard architecture, and refinement would have been, for this purpose, something of a distraction.

We wanted to engage in formal proof because a good many properties of the blackboard architecture are part of the folklore. What we wanted to do was to make properties explicit and, furthermore, to show that they are held by the specification. Otherwise put, we proved properties that directly resulted from our formal model of the blackboard architecture.

The proofs that are included in the book fall into two broad categories. The first category contains results that are general in scope: in a way, they are proofs of properties of our interpretation of the architecture. Because our interpretation conforms, in broad terms, with the majority of others, these are properties of any blackboard system. The second category contains results that are specific to our model (i.e., to our Z specification). The results in the second category relate to the particular definitions that we made in the specification: they might not carry over to all other specifications without modification. In other words, the results in the second class deal, rather more, with the representations that we chose, even though the properties are, arguably, as general as those in the first category. Another characterisation of the division that we feel exists is that the first category contains results of wider scope than the second: the second category tends to contain results about a very few (typically one or two) Z schemata, while the first category contains results about a very much larger part of the interpreter.

The properties that we proved include the following:

1. Entry creation followed by the addition of a completely new attribute-value pair to the entry is identical in effect to entry creation. When an entry is created, a set of initial attribute-value pairs must be supplied to the creation operation. The addition of a new attribute-value pair (by the *add* primitive) merely adds that new pair to the set of attribute-value pairs in the entry. Thus, the composition gives exactly the same result as creating the entry with the new pair in the initialising set of attribute-value pairs. A similar result was proved for entry creation followed by attribute modification.

2. The addition of an attribute-value pair to an entry which already contains that pair leaves the entry identical. (If the value is different, an error should result, but, in the interests of simplicity, we ignored errors. As a consequence, an attempt to add a pair whose attribute is already merely falsifies the operation schema's predicate.)

3. All entry identifiers generated by the system are unique. We defined a collection of schemata which specify the identifier generation process. This proof is of the correctness of that generator process. This result was used in the proof of the next property.

4. For any pair of abstraction levels, the sets of identifiers of the entries which reside on those abstraction levels are disjoint. In other words, an entry identifier is to be found on one and only one abstraction level. This was proved using the uniqueness result for entry identifiers and using the fact that the entry creation schema employs only system-generated identifiers (users are not allowed to generate entry identitfiers for themselves).

   This disjointness result is the closest that we can come to proving that every entry resides on one and only one abstraction level. The reason for this is that there is no criterion of identity for entries other than the identity of their identifiers (see the next sub-section for details).

5. Knowledge Sources can only be triggered as the result of executing one of the blackboard update primitives (i.e., the *new*, *add* and *modify* primitives).

6. Knowledge Source preconditions leave the blackboard state invariant. The proof of this result rests upon an assumption which we could not directly prove from the specification. The result we actually proved is that no sequence of preconditions alters the blackboard state provided that each precondition leaves the state invariant. (We will discuss this result in some more detail below.)

Results 1, 2 and 3 fall into the second category of result (although result 3 is debatable). The other results fall into the first category. These examples should give a feeling for the kinds of proposition that we could prove.

In addition to these proofs, we occasionally prove propositions about objects that we defined axiomatically. The proofs of axiomatic definitions were undertaken to ensure correctness of the definitions concerned.

In the book, a considerable number of results are stated without proof. This is because we have given an analogous proof earlier in the text: the relevant changes are immediate. In one case, the case of triggering, we only give an outline proof: this was because a complete, formal, proof would have been excessively long.

We were pleasantly surprised to find that the proofs were all, by and large, quite straightforward. Apart from standard predicate calculus, all that was required was the occasional induction and use of the axioms given by Spivey (1989). The most complicated proof (uniqueness of entry identifiers on the blackboard) required an indirect proof of some complexity: even so, it amounts to little more than two pages of typeset A4.

One property that we did not attempt to prove was termination. We could have proved that the interpreter will terminate when the agenda becomes empty—this would have been a relatively simple exercise—but we could not prove a more general termination property because we could not specify the details of the scheduler: this is a point to which we will return in the next sub-section.

3.5 Problems

Although the Z specification was successful, it raised a number of problems. In this sub-section, we will consider those problems which relate to our specification.

In the specification, we concentrated on those components of the interpreter that are essential to the operation of the system. We ignored Knowledge Source variables because they required the syntax and semantics of Knowledge Sources to be defined. At the time we undertook the specification, we wanted to ignore these issues and to concentrate, instead, on the basic software (we have now turned to the problem of syntax and semantics in our later specifications). This turned out not to be very satisfactory, and led to complications. In particular, it is hard to see, from just the Z specification, how the various parts of Knowledge Sources communicate. Also, it made the task of specifying preconditions and rule conditions much more difficult, because we had to make them as general as possible and to use structures which might not, at first sight, seem very natural.

A second, Knowledge Source-related problem was a direct consequence of our interpretation of preconditions and rule conditions. In our interpretation, preconditions and rule conditions are implementation-language code, and there is no restriction on them (apart from the injunction that they may not alter the blackboard state—although they may side-effect non-blackboard variables and databases). Under this interpretation, it is not possible to specify the necessary code. This, then, represents a hole in the Z specification.

We wanted, though, to give as complete a specification of Knowledge Sources as possible. We chose to ignore the individual predicates that comprise preconditions and rule conditions and opted, instead, to give a specification of both rule conditions and Knowledge Source preconditions. To do this, we employed the standard interpretation from production rules: a sequence of condition-elements in a production rule is interpreted as a conjunction. We therefore considered Knowledge Source preconditions and rule conditions to be sequences of predicates. Over and above this, we made the assumption that all condition-elements in rules and all Knowledge Source preconditions are benign—i.e., they do not alter the blackboard state. We backed this up by the definition of an invariant which every user-supplied predicate must satisfy: the proof of satisfaction is the job of the predicate-specifier. With the invariant and the definitions of the structures for preconditions and rule conditions, we could prove that no sequence of predicates which satisfy the invariant will ever alter the blackboard state.

Our approach to preconditions and rule conditions is hardly ideal. In fact, it seems to be highly inelegant. There is, we contest, no alternative because we are unable to give specifications of *all* possible predicates that might be used in a working system. The approach that we have taken seems, therefore, to be the best that we can do. We could, of course, have adopted a different interpretation for preconditions and rule conditions (and also for obviation conditions): we could, for example, have specified pattern-matching for these conditions. Unfortunately, preconditions, etc., in real systems, must, very often, have access to software components that are external to the blackboard interpreter, so the problem will be raised later.

The next major problem came with the scheduler. Scheduling is very much a domain- and problem-specific activity. Furthermore, an application might be developed using the blackboard control model, then to be converted to a system that uses an intelligent scheduler for reasons of speed. In order to specify the scheduling component, it would be necessary to give a totally general specification, and, has we have suggested, this cannot be done. We were therefore forced to represent the scheduler as an uninterpreted function symbol. Clearly, this prevents proofs of termination. This is a second hole in the Z specification.

Finally, we encountered problems with entries. As we have remarked, we could only give an identity criterion based on entry identifier. There is no reason, in our specification, that two entries with different identifiers, but with the same set of attribute value pairs, should not be considered different. In other words, the attribute-value pairs that appear in an entry do not contribute to the criterion of identity. This appears to be absurd. However, on reflection, it is anything but absurd for it reveals a rather deeper problem. Although it seems reasonable to insist on the identity of two entries which are composed of identical attribute-value sets, what happens if the two sets differ in only one value—say, one attribute of one entry has a value that is slightly different from the corresponding attribute in the other entry (for example, the *a* attribute in one entry has the value 1 and the other instance of the attribute has the value 1.1)? Clearly, a criterion of identity must be defined. This, again, seems to be problem- or domain-specific: it does not seem to be the case that we can give a generally applicable criterion. Rather than impose possibly inappropriate (or just plain wrong) criteria, we decided to adopt the identifier identity as the criterion for entries. This problem, unlike the others is of a more conceptual nature (a more detailed version of the argument appears in Craig, 1991a).

The last problem is not really a problem with the specification, but with its scope. For reasons identical to those which prevented us from specifying the predicates which comprise rule conditions and Knowledge Source preconditions, we did not attempt to specify any external interfaces for the interpreter. Real blackboard systems frequently need to access external databases, display information on consoles, or send data along external data links. External interfaces are not part of the blackboard architecture and are there to make systems work properly, so we also felt able to ignore them. In any case, the sheer variety of interfaces that could be required seems to preclude a formal specification in general terms, although, for any particular application, the interfaces can, and should, be formally specified.

4 REVIEW AND CLOSING REMARKS

4.1 Review

In this paper, we have reported on the specification of a blackboard interpreter. The interpreter covers the majority of our interpretation of the blackboard architecture and is intended to serve as a model of that interpretation. Some compromises had to be made during the specification exercise just so that it would remain of a reasonable size; other compromises had to be made because we would otherwise have had to try to specify code that we had no knowledge of (preconditions). In its current form, the specification occupies over 100 A4 typeset pages (prepared using Spivey's *fuzz* package) and contains considerable amounts of explanatory text: the book in which the specification appears is intended to be partly tutorial in nature, so some of the text explains Z constructs.

The specification concentrates on the top-level: this is because we wanted to make our interpretation of the blackboard architecture as unambiguous as possible. One level of refinement of the blackboard database appears in the Appendix of Craig, 1991b. We hope to be able to refine the remainder of the specification and generate code from it in due course. As part of the realisation exercise, we intend to give a tighter specification of Knowledge Sources and to try to solve the problems with preconditions and rule condition-elements.

We view the exercise as being quite successful. At the very least, it shows that the formal specification of AI software is possible. The blackboard interpreter was the first large scale specification that we undertook in Z, and we are relatively happy with the outcome. We have now undertaken more Z specifications of AI software. The other specifications are of the author's (1989) CASSANDRA architecture (which is the second large specification in the book), of a new production rule interpreter called ELEKTRA (Craig, 1991c, 1991d, 1991e), both of which have been completed. In addition, we are undertaking a new specification of an AI system, called RPS: RPS is a production system that incorporates a rich control structure and a declarative database organised as frames. All of these specifications are in Z (a review of all four specifications can be found in Craig, 1991f), but we have also prepared a specification of CASSANDRA in VDM for comparison purposes: as a result of the VDM exercise, we have concluded that we prefer Z and will use it in all future specifications.

Although we have been generally pleased with the results of the specification exercises (and we hope to undertake more work in the area, including formal derivation of code), we have encountered a number of problems with the formal specification of AI programs.

We believe that we need a formal semantics *in addition to* the formal specification of the code, and we need formal guarantees that the code respects the semantics. The newer specifications (those of ELEKTRA and RPS) are attempts at solving some of the problems we encountered when doing the blackboard specification. In the RPS specification, we will include a denotational semantics; the VDM specification of CASSANDRA contains a detailed standard semantics for Knowledge Sources—we intend translating the VDM specification into Z and extending the semantics. The focus of our work is increasingly turning to the issue of formal semantics of AI systems.

4.2 Closing Remarks

We believe that formal specification is a valuable tool for AI, particularly where systems are complex or intended for safety-critical applications. The guarantee that the software works properly is only part of the problem, for knowledge-based systems rely on encoded knowledge (which must be shown to be 'correct' in some sense). AI systems also rely on the correct manipulation or transformation of their encoded knowledge (in other words, a 'soundness' result is required—completeness and consistency may be considerably harder, or even impossible, to prove for some problems). In other words, the correctness of the inference procedures needs to be shown as well as the correctness of the basic software. The work that has been reported in this paper only deals with the software that underpins those components which represent and manipulate knowledge. What is required, as we mentioned above, is a semantic account or specification of the knowledge representation, and that account must be matched with the specification of the software: this work remains to be done, as far as we know.

REFERENCES

Craig, I.D., *The BB-SR System*, Department of Computer Science, , University of Warwick, Research Report No. 94, 1987a.

Craig, I.D., *The Blackboard Architecture: A Definition and Its Implications*, Department of Computer Science, University of Warwick, Research Report No. 98, 1987b.

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

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

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

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

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

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

Craig, I.D., *Formal Specification of AI Systems: Four Case Studies*, Department of Computer Science, University of Warwick, *in prep.*,1991f.

Erman, L.D. and Lesser, V.R., A Multi-level Organization for Problem-Solving Using Many Diverse Cooperating Sources of Knowledge, Proc. IJCAI-75, 1975.

Erman, L.D., Hayes-Roth, F., Lesser, V.R. and Reddy, D.R., The HEARSAY-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty, *ACM Computing Surveys*, **12**, No. 2, pp. 213-53, 1980.

Feigenbaum, E., Nii, H.P., Anton, J.J. and Rockmore, A.J., Signal-to-signal Transformation: hasp/siap Case Study, *AI Magazine*, **3**, pp. 23-35, 1982.

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.

Hayes-Roth, F. and Lesser, V.R., Focus of Attention in the HEARSAY-II Speech Understanding System, *Proc. IJCAI-77*, pp. 27-35, 1977.

Lesser, V.R. and Erman, L.D., A Retrospective View of the HEARSAY-II Architecture, *Proc. IJCAI-77*, pp. 790-800, 1977.

Nii, H.P. and Aiello, N., AGE: A Knowledge-based Program for Building Knowledge-based Programs, *Proc. IJCAI* **6**,pp. 645-655, 1979.

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

Nii, H.P., Blackboard Systems Part Two: Blackboard Application Systems, *AI Magazine*, **7**, No. 3, pp. 82-106, 1986b.

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.