Embedable Problem Solving Architectures: A study of integrating OPS5 with GBB

Daniel D. Corkill

March 1989
COINS Technical Report 89-32

Submitted to the Third Workshop on Blackboard Systems to be held at IJCAI-89,
Detroit, Michigan, August 23, 1989.

Embedable Problem Solving Architectures: A study of integrating OPS5 with GBB

Daniel D. Corkill

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

March 1989

Abstract

Typically, problem-solving "shells" are developed with a Ptolemaic view of their universe. While they may have sophisticated interfacing capabilities with subordinate modules, they cannot be used as components of a larger problem-solving system. This paper describes the need and requirements for problem-solving architectures that can be embedded within other architectures and that can coexist with other instances of themselves and of other systems. The additional requirements needed to produce an embedable architecture are minor and the increased applicability of the problem solving architecture is substantial.

A specific case of system embedding arose with the Generic Blackboard Development System (GBB). Initial GBB users coded knowledge sources (KSs) directly in Common Lisp. But why not allow KS writers to code KSs in their favorite AI language? In principle, a KS could be written using any shell or language embedded within GBB. We pursued this idea by embedding the public domain version of OPS5 within GBB. Our observations and the specifications for embedable architectures presented here are the result of generalizing this experience.

This research was sponsored in part by donations from Texas Instruments, Inc., by the National Science Foundation under CER Grant DCR-8500332, and by the Office of Naval Research, under a University Research Initiative Grant (Contract N00014-86-K-0764).

1 Introduction

The specifications for an embedable[1] problem-solving architecture presented here began with a "simple" goal: integrating a popular rule-based AI language (OPS5 [1]) into the Generic Blackboard Development System (GBB) [2,3]. GBB contains a high-performance blackboard database compiler and runtime system that can be extended into a complete blackboard "shell" by the addition of a control shell and knowledge source (KS) representation languages. Initial versions of GBB required application developers to code KSs directly in Common Lisp using functions provided by GBB's blackboard database runtime system. Although Common Lisp is an appropriate language for *computational* KSs, many *knowledge-based* KSs can be more easily expressed using rule- or frame-based AI languages.

Some blackboard systems provide a specialized language for writing KSs [4,5]. We envisioned a different approach with GBB. Why not allow KS writers to code KSs in their favorite AI language? There could be KSs written in native Common Lisp, in OPS5, in PROLOG, and in any number of popular AI shells and languages. In this way, the KS independence of the blackboard paradigm would be extended from the knowledge itself to the language used to codify the knowledge.

Such architectural independence among KSs is a restricted form of the multiarchitecture integration approach exemplified by the ABE project [6]. Instead of ABE's recursively-defined module hierarchy containing *primitive* language framework modules at its leaves, GBB's *KS shell* approach integrates modules as a single-level structure within the blackboard KS framework.[2] This large-grained, KS shell viewpoint reduces the complexity of embedding a problem solver within GBB. From the perspective of the blackboard's KS scheduler, a KS written in OPS5 (or any other embedded language) is not much different than a KS written in Common Lisp. Both are subroutines with specific calling conventions that perform blackboard read/write operations. Once called, an OPS5 KS instance operates as a "regular" OPS5 problem solver, except for commands to access the blackboard and to return control back to the KS scheduler at the end of the KS computation. We felt that this simple subroutine invocation boundary would make embedding a wide range of existing AI architectures within GBB an easier task than the ABE integration effort.

As the scope of AI applications grows beyond restricted domains, the ability to integrate multiple expert systems or problem-solving techniques together is increasingly important. The blackboard framework's cooperating KS model is attracting experimenters who are using the blackboard model as a means of integrating heterogeneous

[1]The form of embedding considered here in which a problem solver is made coresident with other problem solvers in the same Common Lisp environment differs from integrating a C-based AI language as the sole problem-solving component of a general-purpose computing application. The latter meaning is often implied when advertising an expert system as "embedable."

Whether or not the term "embedable" is a suitable extension of English is controversial. We acquiesce to increasingly standard practice by using "embedable" to succinctly capture the notion of a system that can be embedded within a larger system.

[2]Our proposed embedable architecture interface specifications (discussed in Section 4) are similarly related to ABE's *black box* (BBOX) framework.

problem solvers. To date, such integration efforts have typically involved distributed machines connected by a communication protocol suggestive of a blackboard system. Such a "paste-up" approach is inappropriate as a general technique. Use of heterogeneous problem solvers (represented as independent KSs) should not require networked KS execution merely to circumvent the inability of problem solvers to be coresident. Such neworking is both excessive and ignores the important issue of opportunistic control of individual KS instances. In general, the problem to be addressed is one of embedability—not distributed blackboards. Although embedding requires the individual KS shells to be coresident in the same environment (in our case, a single Common Lisp image), the basic cooperating KS problem-solving model and control machinery operate normally. It is this approach to heterogeneous problem solving that is considered in this paper.

With our vision of GBB KS shells in place, we began modifying OPS5 to become GBB's first KS shell.[3] As with many goals that are simple in principle, achieving an integration of OPS5 with GBB required dealing with a number of implementation incompatibilities. Our efforts have resulted in both a successfully integrated OPS5/GBB KS shell and an improved understanding of the requirements for embedding a problem solving system within another problem solving system. Our experiences and recommendations should be useful to anyone contemplating a similar integration effort.

2 OPS5 and GBB

What would a GBB/OPS5 KS instance look like? Each GBB/OPS5 KS instance would have its own private working memory (WM) and a knowledge base (KB) shared with other instances of the same KS. Its WM would be appropriately initialized with KS stimulus data and then control would be transferred to OPS5. At this point, one or more OPS5 KB rules have been triggered by the insertion of the stimulus data. Some of these rule activations might include retrieving other objects from the blackboard and placing them or some of their attributes in WM. The OPS5 KS instance would also likely create or modify objects on the blackboard. Finally, it might return one or more values back to GBB's KS scheduler indicating the completion status of the KS.[4]

Note that the WM of OPS5 is conceptually and implementationally distinct from the blackboard (Figure 1). The private computations held in WM should not be seen by other instances of OPS5 KSs or by any other KSs in the blackboard application. Similarly, it is inappropriate to run every blackboard modification through the RETE network of every OPS5 KS instance. Finally, the OPS5 inference engine and the blackboard's KS scheduling cycle should be completely independent.

Depending on the control shell used, instantiating and invoking a GBB/OPS5 KS can require a number of KS activities. Our initial integration efforts focused on GBB's "simple shell" control shell that implements the precondition/action KS invocation model first used in the Hearsay-II speech understanding system [7]. In the "simple shell,"

[3]OPS5 was selected as the initial KS shell for a number of reasons including popularity and publicly-available Common Lisp source code.

[4]For example, in GBB's "simple shell" control shell, a KS can return a special termination value indicating to the shell that a solution has been found and that KS instance invocations should cease.

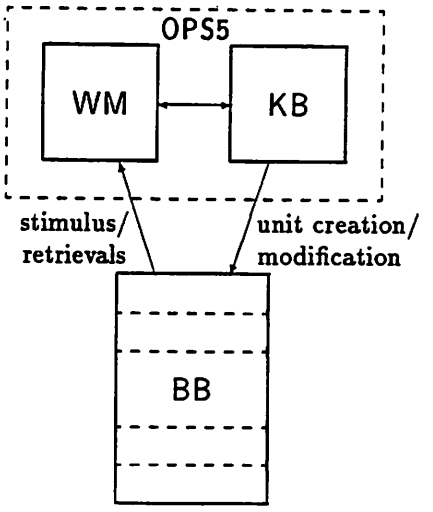

Figure 1: GBB's Blackboard and OPS5's Knowledge Base (KB) and Working Memory (WM).

a KS is declared as two distinct but related modules: a precondition procedure and an action procedure. The precondition procedure is invoked on a *stimulus* blackboard object that matches a defined condition (representing the types of blackboard events of interest to the KS). The precondition procedure typically uses the stimulus object as a context for searching for other relevant blackboard objects. If sufficient data are found, the precondition procedure returns a local estimate of the importance of scheduling the action portion of the KS.

The GBB/OPS5 KS shell allows either the precondition procedure, the action procedure, or both to be coded in OPS5. The two procedures are effectively distinct modules (with distinct KBs) coupled only by a data passing convention in the *stimulus/response* frame of the KS instance.

Another way of looking at the relationship between GBB and OPS5 is to consider OPS5's three major components: the OPS5 inference engine and support code, the KB, and the WM. If any KS precondition procedure or action procedure uses the GBB/OPS5 KS shell, the OPS5 inference engine must be loaded into Common Lisp.[5] Each precondition procedure or action procedure requires a separate instance of an OPS5 KB to be defined and maintained within GBB. Finally, if GBB/OPS5 precondition or action procedures can be interrupted or executed in parallel, every active GBB/OPS5 KS instance (initiated but not completed) requires a separate instance of an OPS5 WM. These relationships are illustrated in Figure 2.

GBB's "simple shell" buffers all *blackboard events* (the triggers for KS precondition procedure invocation) until the end of the currently executing procedure. This means that "simple shell" KS procedures written in OPS5 are not interrupted by the execution of precondition procedures. This is not the case with all GBB control shells. To support

[5]Since both GBB and OPS5 are written in Common Lisp, a tight coupling (desired for efficiency) requires they reside in the same Common Lisp heap.

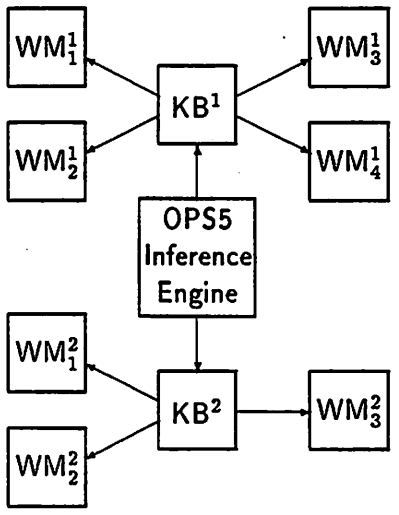

Figure 2: Instances of the three major OPS5 Components.

multiple, interruptable procedures, additional embedding efforts are required. We define three increasingly powerful embedding levels based on KB and WM instance execution relationships:

1. **Serially reusable:** A *serially-reusable* embedable system must be able to be invoked on an instance of a KB with an appropriately initialized WM. This is the minimum required for integration in GBB.

2. **Interruptable:** An *interruptable* embedable system must be able to be suspended while another instance of a KS and WM are processed.[6] The computation can be resumed when the second instance is completed or interrupted. An interruptable system requires the ability to save and restore the execution context of the system.

3. **Interactive:** An *interactive* embedable system is able to asynchronously receive and assimilate changes to its WM (either created directly by another process or in response to triggering events). Interactive systems operating on a single processor would have their executions time-shared by a process scheduler. In a multiprocessing setting, multiple interactive systems could be executing simultaneously. In addition to managing the KB and WM information, this capability level requires care in implementing the inference engine.

We consider only serially-reusable embedding in the remainder of the paper.

[6]This issue is local to the internals of the KS shell. Problems associated with suspending/resuming KS instances in general are another matter.

3 The OPS5 Modifications

As with most AI shells and languages, the publicly-available OPS5 implementation[7] does not support multiple instances of the KB or WM. We collapsed all global information associated with a KB or WM instance into one of two global "context" structures to simplify context switching.[8] This included moving property list information into the appropriate global context. Once this effort was complete, running OPS5 on a particular KB and WM instance simply requires binding the appropriate context structures to these two global variables.

A means for defining each KB instance was also needed. Each KB ruleset is named and is compiled, stored, invoked, and potentially redefined using its name. In essence, we modified OPS5 to support a *library* of named, independent KB rulesets. Defining a particular KB ruleset merely involves enclosing the rules within a KB definition form naming the ruleset:

(ops-define-kb *name* (*rule*$_1$ *rule*$_2$ ... *rule*$_n$))

When invoked, a GBB/OPS5 KS execution needs to interact with GBB's runtime blackboard routines. Since both GBB and the OPS5 implementation were in Common Lisp, interaction might appear to be a simple issue. In fact, even with OPS5's external routine capabilities, interacting with GBB required substantial modifications.

Consider GBB's retrieval operation **find-units**. **Find-units** represents a blackboard retrieval pattern as a nested list of specifications and values. OPS5 does not provide support for list objects, neither internally nor as part of its external routine interface. How could a GBB/OPS5 KS construct a retrieval pattern?

GBB's **find-units** function also returns a list of the retrieved blackboard objects. This need for a list datatype could be circumvented by entering each of the returned objects as a separate WM element. This approach hides the common retrieval relationship among the returned objects. Another approach would use an OPS5 vector-attribute to contain the returned items. The disadvantage with this approach is lack of control over the number of retrieved elements and therefore a potentially unbounded WM element size.

Since lists were needed to represent GBB patterns anyway, we decided to extend OPS5 to include a "list" datatype that was atomic from the perspective of OPS5, but could be passed as a list to GBB. We then added the following "pseudo-list" operators to OPS5: **$cons**, **$first**, **$list**, **$quote**,[9] and **$rest**.

The external routine interface in OPS5 is cumbersome to use from a Common Lisp environment. In particular, it requires the subroutine to explicitly manage the passing of values through OPS5's *result element*. We added a new GBB/OPS5 operator, **cl-call**, that automatically performs result element value management, allowing any number of evaluated arguments to be passed to an external Common Lisp function. Multiple return

[7] All remaining references to OPS5 pertain to this public version.

[8] The original implementation contained well over 200 global variables! Many were eliminated using Common Lisp lexical binding techniques.

[9] OPS5's // operator also functions as **$quote** for pseudo-lists.

values are automatically placed into the result element for extraction within OPS5. **Cl-call** transparently supports the GBB/OPS5 pseudo-list datatype extensions.

A new operator **return-values** was added that terminates GBB/OPS5 recognize-act cycle and returns multiple values to the calling routine. The value `nil` is returned if the last recognize-act cycle evaluates an OPS5 **halt** operator or if no further rules remain to be fired.

Table 1 contains an example of a GBB/OPS5 precondition procedure for a simple synthesis KS. The example illustrates the use of **cl-call**, pseudo-list, and **return-value** operators.[10] For comparison, the same precondition coded in Common Lisp is shown in Table 2.

Tracing and debugging a GBB/OPS5 KS execution is also an issue. The OPS5 implementation provides its own interactive command loop, watch, and trace facilities. When OPS5 is used as an embedded system, these facilities are disabled by default. What is needed is a way to selectively enable them on a GBB/OPS5 KS instance-specific basis. We extended the OPS5 **initialization** command to specify whether or not each invocation of a named KB should enter OPS5's interactive command loop for debugging purposes.[11] The **initialization** command is also used to control the tracing (watch) level and whether history recording (for "backing up") is enabled.

There are two disadvantages with our simple extension. First, a single KB instance can be invoked many times during in a blackboard application (once per KS instance). Only a particular invocation may need to be debugged. We should extend our KB-specific command loop entry to be conditional on the initial contents of WM. The second disadvantage is that the debugging information is kept within the KB instance itself. To enable/disable debugging, the desired KB(s) must be individually edited or redefined. A more global specification of debugging needs is more appropriate. Given the specific requirements of individual shell and language debugging and tracing tools, however, it appears difficult to find an acceptable "common denominator" encompassing all tools. A better approach is to have each shell provide a facility for entering/modifying the debugging and tracing specifications for its KB instances (see next section). We have not yet made this modification to GBB/OPS5.

4 A Proposal: An Interface for Embedable Architectures

What have we learned from our modifications of OPS5? What generalizations can we make for repeating the procedure with another AI shell or language?

To review, OPS5 required the following modifications to be embedded within GBB:

- It must be able to define and maintain multiple, independent KB instances.

[10] Because it performs counting, this example is better suited to Common Lisp than OPS5.

[11] If the OPS5 debugging command loop is enabled, the OPS5 invocation does not immediately return when a **return-values** or **halt** command is evaluated or when there are no further rules to execute. Instead an explicit exit command must be entered. This requirement is especially useful when determining why rule firing stagnated.

```
(ops-define-kb SYNTHESIS-KS-PRECONDITION
  ((literalize count count)
   (literalize belief value)
   (literalize supporting-hyps hyps)
   (literalize counting-hyps hyps)

   (startup (watch 0) (disable back) (disable halt) (disable break))

   (p INITIAL
   ;; When triggered with a stimulus-hyp, locate other hyps on the same
   ;; blackboard level with a value within 3 of the stimulus-hyp's
   ;; value.  Prepare to count them and determine the maximum belief
   ;; value.
      (stimulus <stimulus-hyp>)
      -->
      (bind <found-hyps>
            (cl-call
               find-units
               hyp
               (cl-call make-paths :unit-instances <stimulus-hyp>)
               ($list :ELEMENT-MATCH :within
                      :PATTERN-OBJECT
                      ($list :INDEX-TYPE // (:DIMENSION value :TYPE :point)
                             :INDEX-OBJECT (cl-call hyp$value <stimulus-hyp>)
                             :DELTA // ((value 3))))))
      (make supporting-hyps ^hyps <found-hyps>)
      (make counting-hyps ^hyps <found-hyps>)
      (make count ^count 0)
      (make belief ^value 0))

   (p COUNT-FOUND-HYPS
   ;; Determine the number of supporting hyps and the maximum belief
   ;; among the supporting hyps:
      { <Counting-hyps>
           (counting-hyps ^hyps {<hyps> <> nil}) }
      { <Count>
           (count ^count { <value> <=> 0 }) }
      { <Belief>
           (belief ^value { <belief-value> }) }
      -->
      (modify <Belief>
              ^value (cl-call max (cl-call hyp$value ($first <hyps>))
                                  <belief-value>))
      (modify <Count> ^count (compute <value> + 1))
      (modify <Counting-hyps> ^hyps ($rest <hyps>)))

   (p FAIL
   ;; Return 0 indicating failure to instantiate the KS:
     (count ^count <= 2) --> (return-values 0))

   (p SUCCEED
   ;; Return the max belief value as the KS instance rating and the
   ;; stimulus/response frame data:
      (counting-hyps ^hyps = nil)
      (count ^count > 2)
      { <Supporting-hyps>
           (supporting-hyps ^hyps { <hyps> <> nil }) }
      (belief ^value <belief>)
      (stimulus <stimulus-hyp>)
      -->
      (return-values <belief> ($list <stimulus-hyp> <hyps>)))))
```

Table 1: The Synthesis Precondition in GBB/OPS5.

```
(defun SYN-KS-PRECONDITION (stimulus-hyp)
  (let ((supporting-hyps
         (find-units
           'hyp
           (make-paths :unit-instances stimulus-hyp)
           '(:ELEMENT-MATCH :within
             :PATTERN-OBJECT (:INDEX-TYPE (:DIMENSION value :TYPE :point)
             :INDEX-OBJECT ,(hyp$value stimulus-hyp)
             :DELTA ((value 3)))))))
    (cond ((> (length supporting-hyps) 2)
           (values (mapc-max 'hyp$belief supporting-hyps)
                   (list stimulus-hyp supporting-hyps)))
          (t 0))))
```

Table 2: The Synthesis Precondition in Common Lisp.

- It must be able to be called as a subroutine with a particular instance of its KB on an appropriately initialized WM. It may be required to return values to its caller.
- It must be able to "call-out" to other modules (many shells already have this capability). A well integrated call-out capability supports an extensible "foreign" data structure capability (including lists!) and allows the results of a call-out to be used as part of its inference mechanism.[12]

The following proposal describes interface requirements for serially-reusable embedable systems. This proposal extends each system to include a KB *library* facility for managing and invoking instances of the system with a specified KB. By extending each system to meet these interface specifications, the details of managing KBs are encapsulated within the system itself. Such modularity is important, especially if proprietary KB representation mechanisms are to be used within a larger system.

***xxx*-initialize**

This function performs all initializations for shell *xxx* that are independent of a particular KB instance. This function is called once (no matter how many KB instances of the shell are to be used), and must be called before any of the following interface functions. With some shells, this function may be unnecessary.

***xxx*-define-kb** *kb-name* [*kb-declarations*] [*compile?*]

This function instructs shell *xxx* to create and initialize a new entry in its KB library named *kb-name*. If the shell supports declarative initialization of its KB, the optional *kb-declarations* value can be used to pass the initialization data to the shell. If *compile?* is true and the shell supports KB compilation, the KB representation is compiled.

***xxx*-load-kb** *kb-name filename*

[12]The OPS5 and GBB/OPS5 implementations do not allow external calls to be in the left-hand side of rules, an inconvenience.

This function instructs shell *xxx* to load KB data from file *filename* into its KB library entry named *kb-name*. It is an error if *name* has not been previously defined using *xxx*-**define-kb**.

xxx-**edit-kb** *kb-name* [*compile?*]

This function instructs shell *xxx* to provide interactive editing support for its KB library entry named *kb-name*, if the shell supports interactive KB editing. If *compile?* is true and the shell supports KB compilation, the KB representation is compiled at the completion of editing.

xxx-**exit-kb** *name initializations*

This function performs cleanup activities after shell *xxx* has been invoked in a serially-reusable fashion.[13]

The function can return multiple values when shell *xxx* completes. The mechanism for specifying these values is dependent upon the shell.

xxx-**save-kb** *kb-name filename*

This function instructs shell *xxx* to save KB data from its KB library entry named *kb-name* into file *filename*. It is an error if *name* has not been previously defined using *xxx*-**define-kb**.

xxx-**delete-kb** *kb-name*

This function instructs shell *xxx* to delete all traces of its KB named *kb-name*. The result is as if KB *name* had never been defined.

xxx-**invoke-kb** *name initializations*

This function instructs shell *xxx* to begin executing with the KB named *name* and a new WM instance initialized according to *initializations*.

The function can return multiple values when shell *xxx* completes. The mechanism for specifying these values is dependent upon the shell.

xxx-**debug-kb** *name*

As with GBB/OPS5, tracing, stepping, and other debugging techniques are important. This function causes shell *xxx* to ask the use how invocations of its KB library entry named *kb-name* should be conditionally traced, single-stepped, etc. The details are specific to the particular shell.

A system supporting this interface can be quickly embedded in another architecture (such as GBB). Here is an example of the control shell interface for a GBB/OPS5 precondition procedure invocation:

[13]The need for this capability was pointed out by an anonymous reviewer.

```
(defun RUN-OPS5-PRECONDITION (ks stimulus)
    (ops5-invoke-kb ks
       '(make (stimulus ,stimulus))))
```

From within the embedded system the following capabilities are required:

- **The ability to call external routines.** In GBB's case, this includes the ability to construct and receive list data structures for interfacing to GBB runtime blackboard retrieval routines.
- **The ability to return values to the calling routine.** In GBB's "simple shell" case, precondition procedures must return a KS rating value and (optionally) a stimulus/response frame data structure. KS action procedures can return a termination indicator informing the control shell to terminate its KS scheduling cycle.

The details of these two capabilities are specific to the particular shell or language. This interface proposal merely requires that the capabilities be present.

Although the proposed interface specifications have been developed specifically for GBB and OPS5, we believe that the issues of managing knowledge base libraries, invoking and exiting a shell, and debugging apply to the general problem of integrating heterogeneous problem solvers into larger systems. Of course, we have not validated this belief.

5 Summary and Status

Even the best problem-solving architecture can be improved by making it embedable. We began by considering the requirements for embedding OPS5 within the Generic Blackboard Development System (GBB). Although modifying the publicly-available OPS5 implementation was an effort, designing a shell with embedding requirements in mind will not significantly complicate its implementation or reduce its efficiency. The OPS5 modifications have resulted in a well-integrated GBB KS shell.

The OPS5 modification experience also formed the basis for our proposed embedable architecture interface specifications. We are integrating other AI languages into GBB, using the above interface specifications as a common protocol.

Finally, the lessons learned from embedding OPS5 have also been applied to GBB itself. Multiple independent blackboards and control shells can now be embedded in a higher-level architecture if needed. We have not used this capability however.

References

[1] C. L. Forgy. OPS5 reference manual. Technical Report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1981.

[2] Daniel D. Corkill, Kevin Q. Gallagher, and Kelly E. Murray. GBB: A generic blackboard development system. In *Proceedings of the National Conference on Artificial Intelligence*,

pages 1008–1014, Philadelphia, Pennsylvania, August 1986. (Also published in *Blackboard Systems*, Robert S. Engelmore and Anthony Morgan, editors, pages 503–518, Addison-Wesley, 1988.).

[3] Daniel D. Corkill, Kevin Q. Gallagher, and Philip M. Johnson. Achieving flexibility, efficiency, and generality in blackboard architectures. In *Proceedings of the National Conference on Artificial Intelligence*, pages 18–23, Seattle, Washington, July 1987. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 451–456, Morgan Kaufmann, 1988.).

[4] Alan Garvey, Michael Hewett, M. Vaughan Johnson, Robert Schulman, and Barbara Hayes-Roth. *BB1 User Manual.* Knowledge Systems Laboratory, Departments of Medical and Computer Science, Stanford, California 94305, Common Lisp edition, October 1986. (Published as Working Paper KSL 86-61, Knowledge Systems Laboratory, Departments of Medical and Computer Science, Stanford University, Stanford, California 94305.).

[5] L. Baum, R. Dodhiawala, and V. Jagannathan. Boeing Blackboard System, version 1.0. Technical Report BCS-G2010-31, Boeing Computer Services, P.O. Box 24346, Seattle, Washington 98124, July 1986.

[6] Frederick Hayes-Roth, Lee D. Erman, Scott Fouse, Jay S. Lark, and James Davidson. ABE: A cooperative operation system and development environment. In Mark Richer, editor, *AI Tools and Techniques.* Ablex Publishing Corporation, 1988. (Also published in *Readings in Distributed Artificial Intelligence*, Alan H. Bond and Les Gasser, editors, pages 457–489, Morgan Kaufmann, 1988.).

[7] Lee D. Erman, Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *Computing Surveys*, 12(2):213–253, June 1980.