Achieving Flexibility, Efficiency, and Generality in Blackboard Architectures

Daniel D. Corkill, Kevin Q. Gallagher, and **Philip M. Johnson**

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

Abstract

Achieving flexibility and efficiency in blackboard-based AI applications are often conflicting goals. Flexibility, the ability to easily change the blackboard representation and retrieval machinery, can be achieved by using a general purpose blackboard database implementation, at the cost of efficient performance for a particular application. Conversely, a customized blackboard database implementation, while efficient, leads to strong interdependencies between the application code (knowledge sources) and the blackboard database implementation. Both flexibility and efficiency can be achieved by maintaining a sufficient level of data abstraction between the application code and the blackboard implementation. The abstraction techniques we present are a crucial aspect of the generic blackboard development system GBB. Applied in concert, these techniques simultaneously provide flexibility, efficiency, and sufficient generality to make GBB an appropriate blackboard development tool for a wide range of applications.

I. Introduction

Blackboard architectures, first introduced in the Hearsay-II speech understanding system from 1971 to 1976 [Erman *et al.*, 1980], have become popular for knowledge-based applications. The interest in the generic blackboard control architecture of BB1 [Hayes-Roth, 1985] is but one example of the increasing popularity of blackboard architectures. The blackboard paradigm, while relatively simple to describe, is deceptively difficult to implement effectively for a particular application. As noted by Nii [Nii, 1986], the blackboard model with its knowledge sources (KSs), global blackboard database, and control components does *not* specify a methodology for designing and implementing a blackboard system for a particular application.

Historically, most blackboard-based systems have been built from scratch, implementing the blackboard model according to the criteria that appeared most appropriate for the particular application. Some implementations were built for execution *efficiency*, with considerable effort placed on providing fast insertion and retrieval of objects on the blackboard. The KSs and control components in these implementations were so tied to the underlying blackboard database that making modifications to the blackboard structure or insertion/retrieval strategies was difficult. Other implementations were designed with *flexibility* in mind. These applications were built on top of a general-purpose blackboard database retrieval facility (for example, a relational database system [Erman *et al.*, 1981]). While these implementations could be restructured relatively easily, their inefficiency in accessing objects on the blackboard made them slow. Finally, a few implementations were simply built in a hurry, with little effort toward achieving either flexibility or efficiency.

In this paper, we concentrate on the two conflicting issues of flexibility and efficiency of blackboard systems. We show that by appropriately hiding information between three phases of blackboard system development—blackboard database specification, application coding (KSs and control components), and blackboard database implementation—it is possible to achieve both flexibility *and* efficiency. This principle of blackboard data abstraction is an integral design principle of the generic blackboard development system GBB [Corkill *et al.*, 1986]. Abstraction also makes GBB sufficiently general for use in a wide range of applications. Although we describe the benefits of blackboard abstraction in the context of GBB, these abstractions are appropriate for any blackboard development environment.

II. On Flexibility and Efficiency

Flexibility in a blackboard system is the ability to change the blackboard database implementation, the insertion/retrieval strategies, and the representation of blackboard objects without modifying KS or control code and vice-versa. Flexibility is important for two reasons. First, the application writer's understanding of the insertion/retrieval characteristics and the representation of blackboard objects may be uncertain and therefore subject to change as the application is developed. Second, even after a prototype of the application has been completed, the number and placement of blackboard objects as the application is used may differ from the prototype. This again requires changes to the blackboard representation in order to achieve the desired level of performance. Therefore, it is important that the blackboard implementation provides enough flexibility to allow these changes without significant changes to the KSs, the control code,

This research was sponsored in part by the National Science Foundation under CER Grant DCR-8500332, by a donation from Texas Instruments, Inc., by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under Contract NR049-041, and by the National Science Foundation under Support and Maintenance Grant DCR-8318776.

or to the blackboard database implementation machinery. With sufficient flexibility it is possible to actually "tune" the blackboard representation to the specific characteristics of the application.

Efficiency in the insertion and retrieval of blackboard objects is an equally important design goal. Typically, improving the execution efficiency of blackboard systems is achieved through improvements to the quality and capability of the control components. Reducing the number of "inappropriate" KSs that are executed (by making more informed scheduling decisions) can significantly reduce the time required to arrive at a solution. Making appropriate control decisions should never be neglected in the development of an application. In this paper, however, we assume that a high-quality control component and high-quality KSs will be written by the application implementer. We will focus on the remaining source of execution inefficiency—the cost of inserting and retrieving objects from the blackboard.

A. The Need For Blackboard Database Efficiency

Why are we placing such an emphasis on the efficiency of the blackboard database? In addition to inserting new hypotheses on the blackboard, KSs perform *associative retrieval* to locate relevant hypotheses that have been placed on the blackboard by other KSs. This need for KSs to *locate* appropriate information on the blackboard is often overlooked in casual discussions of blackboard-based systems. A KS is typically invoked by one or more triggering *stimulus* objects. The KS then looks on the blackboard to find other objects that are "appropriately related" to the stimulus object. Each KS thus spends its time:

1. retrieving objects from the blackboard based on their "location" on the blackboard;
2. performing computations using existing objects (to determine new blackboard objects to create);
3. creating and placing these new objects onto the blackboard.

The ratio of items 1 and 3 over item 2 defines the amount of time the KS spends interacting with the blackboard versus the amount of time the KS spends performing computations. The larger this *interaction/computation ratio* is, the more that blackboard efficiency issues will dominate performance. The ratio of item 1 over item 3 defines the *read/write ratio* of blackboard interactions for the KS. This ratio can be used to aid the selection blackboard implementation and retrieval strategies.

Note that associative retrieval is central to the blackboard paradigm. Associative retrieval is used to provide *anonymous communication* among KSs by allowing KSs to look for relevant information on the blackboard rather than receiving the information via direct invocation by other KSs. Yet the blackboard provides more than this anonymous communication channel among KSs. Objects on the blackboard often have significant *latency* between the time they are placed on the blackboard and the time they are retrieved and used by another KS. If it were not for this latency, the blackboard could be "compiled away" into direct calls among KSs by a configuration-time compiler. This latency in blackboard objects indicates that the blackboard also serves as a global *memory* for the KSs. Objects are held on the blackboard to be used when and if they are needed by the KSs. Without the blackboard each KS module would have to maintain its own copy of objects received from other modules. Whether the memory is globally shared (on the blackboard) or private, an efficient means of scanning the remembered objects is still required.

The amount of time a KS spends creating and scanning for objects versus performing other computations (the interaction/computation ratio) varies greatly between different applications and even between different KSs in a single application. Of course, the greater this ratio the more significant the efficiency of the blackboard implementation becomes. Experience with the Hearsay-II speech understanding system [Erman *et al.*, 1980] and the Distributed Vehicle Monitoring Testbed (DVMT) [Lesser and Corkill, 1983] demonstrates that blackboard performance has a significant effect on system performance in these applications.

If the underlying hardware provided true associative retrieval, these efficiency issues would become irrelevant and the implementer would only need to write the application KSs and control code. However, the present hardware situation requires that the associative retrieval of blackboard objects be simulated in software by appropriate retrieval strategies on the blackboard database.

B. Basic Blackboard Operations

Before we continue, it is useful to describe in more detail the blackboard operations that are typically required to support an application.

Insertion: When a blackboard object is created, it must be placed onto the blackboard. Placement onto the blackboard involves creating one or more *locators*, pointers that are used to retrieve the object. In the simplest situation where blackboard objects are merely pushed onto a list, the single locator is the list pointer. With retrieval strategies supporting efficient retrieval of objects based on complex criteria, multiple locators are used. These locators are determined based on attribute values of the object.

Merging: When placing an object onto the blackboard, it can be important to determine if an "identical" object already exists on the blackboard. The semantics of identity depend on the application, but an example is two hypotheses created by different KSs that differ only in their belief attribute. Often it is desirable that hypotheses on the blackboard be unique; that is, no identical hypotheses be created on the blackboard. Instead, the two hypotheses should be merged into a single blackboard object that reflects the two by merging their belief attributes into a single attribute value in the existing hypothesis.

Merging can be handled in two ways. One approach is to have all KSs avoid creating identical hypotheses by checking for an existing hypothesis before creating a new one. If an existing hypothesis is found, its attributes are updated by the KS. The second approach builds an application-specific merging capability into the basic blackboard object insertion machinery.

Retrieval: Retrieval involves searching the blackboard for objects that satisfy a set of constraints specified in a retrieval pattern. Retrieval can be broken down into two steps. The first step determines a set of locators (based on the retrieval pattern) that contain pointers to potentially desirable objects. The second step eliminates those candidates from the first step that do not satisfy the constraints of the retrieval pattern. Since this elimination process can be computationally expensive, an efficient retrieval strategy is one where the first step substantially reduces the number of candidates. In order to implement an efficient, yet flexible, retrieval strategy the constraints must be expressed declaratively so that they may be examined by the blackboard implementation machinery to determine the appropriate set of locators to use in the retrieval.

Deletion: Deleting an object from the blackboard requires removing it from the locators which point to it. Since other blackboard objects may contain *links* pointing to the deleted object, these links must also be found and eliminated. For example, if links are maintained as bidirectional pointers (as is the case in GBB), deleting these links is simply a matter of traversing all links from the deleted object and then eliminating the inverse links.

Repositioning: If the attributes that determine the object's locators (such attributes are termed *indexing attributes*) are modified, the locators may also need to be changed (deleting some and adding others) to maintain consistency in the blackboard database. In many applications, all indexing attributes are static—only the values of the other attributes (such as belief) are allowed to change. Domains involving objects that move over time, however, are examples of situations where the positioning of objects may need to be modified during the course of problem solving.

III. Past Practice

In this section, we characterize approaches that have been used to implement associative retrieval in blackboard systems.

A. The Unstructured Blackboard

A simplistic approach to building a blackboard application is to represent each blackboard level as an unstructured list of the objects residing on that level. KSs add a new object to the blackboard by simply pushing it onto the appropriate list. Retrieval is performed by having the KS scan the list for objects of interest.

This approach only appears to be simple, as there is no work to implementing the blackboard implementation machinery (global variables serve quite nicely). Actually, all the effort has been shifted into the KSs. Each KS must worry about the entire retrieval process, and since each object on the blackboard level must be tested for appropriateness, the KS must perform this test as efficiently as possible. Each KS may also need to worry about merging blackboard objects; avoiding the creation of a blackboard object that is semantically equivalent to an existing object. If merging is not performed, KSs must consider the possibility that semantically equivalent objects may be retrieved from the blackboard. Insertion, deletion, and repositioning of blackboard objects must also be directly handled by the KSs as well.

B. The General-Purpose Kernel

In this approach, a general-purpose blackboard database facility is provided to the KS and control component implementers. The facility supports blackboard object retrieval based on the attributes of the objects. In its most general form, all attributes of the objects may be used as retrieval keys (for example, blackboard objects may be stored in a relational database). The application implementers retrieve objects by writing queries in the retrieval language. This approach provides a very flexible development environment, but the unused generality of the blackboard database implementation poses severe time/space performance penalties.

C. The Customized Kernel

As noted above, the use of a general-purpose retrieval strategy for all blackboard applications is a source of inefficiency. Retrieval of blackboard objects in a particular application may be made significantly faster using a specialized retrieval mechanism. Furthermore, retrieval of different classes of blackboard objects within a single application may be best achieved using different retrieval strategies. One solution is to custom-code the appropriate retrieval strategy for each situation. In this approach an insertion/retrieval kernel is written that is tailored to the situations that arise in a particular application. When a KS needs to locate blackboard objects, it invokes kernel functions to perform an initial retrieval from the blackboard and then uses procedural "filters" to identify which returned objects are actually of interest. This approach is significantly more efficient then the general-purpose approach when the kernel functions significantly prune the number of blackboard objects that need to be filtered by the KS. However, it poses a number of disadvantages:

- A new customized kernel must be written to suit the different insertion/retrieval characteristics of each application.
- If the kernel is found to be inappropriate to the application, due to incorrect intuition during the initial design or to changing application characteristics, it must be rewritten.
- The KS code is directly coupled to the particular kernel. The code must be written with the knowledge of which attributes are matched by the kernel code and which attributes must be filtered by the KS. Changing the kernel attributes requires rewriting the KSs.
- The kernel code is tied to the blackboard representation. Changes to the blackboard representation require modifications to the kernel code.
- The KS and kernel code is tied to the structure of blackboard objects. Changes to the representation of attributes require code modifications.

In short, although the custom-coded kernel approach can provide efficient insertion and retrieval of blackboard objects, that efficiency comes at the cost of inflexibility to changes in the KS and control code and to changes in the blackboard and object representation.

IV. Blackboard Database Abstraction in GBB

By appropriately combining a number of blackboard data abstraction techniques, it is possible to "have your cake and eat it too" with respect to flexibility and efficiency. The generic blackboard development system GBB [Johnson *et al.*, 1987] provides the application implementer and blackboard database administrator with distinct, abstract views of the blackboard. Developing an application using GBB involves three separate, but interrelated phases:

blackboard & blackboard object specification: This phase involves describing the blackboard structure (the blackboard hierarchy), the structure of each blackboard level, the attributes associated with each class of blackboard objects (called *units* in GBB), and the mapping of units onto blackboard levels (called *spaces* in GBB).

application coding: This phase involves writing KSs and control code in terms of the blackboard and blackboard object specifications. Application code deals with the creation, deletion, retrieval, and updating of units. Retrieval is specified by patterns based on the structure of the relevant blackboard space(s).

blackboard database implementation specification: This phase involves specifying the blackboard database implementation and retrieval strategies. The locator data structures appropriate for the particular characteristics of the application are specified in this phase. These specifications are also made in terms of the blackboard structure and unit specifications.

By maintaining an abstracted view of the blackboard, the details of decisions made in each of the three phases can be hidden until they are combined in GBB's code generation facility.

A. Abstracting the Blackboard

In GBB, each blackboard space is a highly structured n-dimensional volume. Space dimensionality provides a *metric* for positioning units onto the blackboard in terms that are natural to the application domain. Units are viewed as occupying some n-dimensional extent within the space's dimensionality.

For example, in a speech understanding system, one of the dimensions of a blackboard space could be *utterance time*. In the domain of vehicle tracking, a space might contain the dimensions *sighting time*, *x-position*, and *y-position*. In GBB, such dimensions are termed *ordered*. Ordered dimensions use numeric ranges which support the concept of one unit being "nearby" another unit along that dimension. In the speech understanding domain, this allows a KS to extend a phrase by retrieving words that begin "close in time" to the phrase's end time.

GBB also supports *enumerated* dimensions. An enumerated dimension consists of a fixed set of labeled categories. For example, in the vehicle tracking domain a space might also have the enumerated dimension "classification" corresponding to a set of vehicle types.

Space dimensionality is a key means of abstracting the blackboard database. It provides information hiding by allowing the application code to create and retrieve units according to the dimensions of spaces, without regard to the underlying implementation of the blackboard structure. Dimensional references, however, contain enough information when combined with information about the structure of the blackboard to allow efficient retrieval code to be generated.

Here is an example of the space definitions from the DVMT application that specifies the *time*, *x-position*, *y-position*, dimensions discussed above (as well as a sensory event classification dimension):

```
(define-spaces (PT PL VT VL GT GL ST SL)
  :UNITS (hyp)
  :DIMENSIONS
    ((time        :ORDERED *bb-time-range*)
     (x           :ORDERED *bb-x-range*)
     (y           :ORDERED *bb-y-range*)
     (event-class :ORDERED *bb-event-class-range*))).
```

B. Abstracting Unit Insertion

When a unit is created in GBB, it is inserted on the blackboard based on the unit's attributes. There are two decisions to be made when inserting a unit on the blackboard. The first is what space or spaces to store the unit on and the second is the location of the unit within the n-dimensional volume of each space. The definition of each unit includes the information required to make these two decisions based on the values of the unit's attributes. This insulates the KS code from the details of the blackboard structure. For example, the KS code does not need to know which attributes and dimensions are actually used to create locators for the unit. Thus changes in the blackboard structure do not necessitate changing KS code.

Here is an example of the hypothesis unit class definition from the DVMT application:

```
(define-unit (HYP (:NAME-FUNCTION generate-hyp-name)
                  (:INCLUDE basic-hyp-unit))
  :SLOTS
  ((belief              0   :TYPE belief)
   (event-class         0   :TYPE event-class)
   (level               nil :TYPE symbol)
   (node                0   :TYPE node-index)
   (time-location-list ()   :TYPE time-location-list))
  :LINKS
  ((supported-hyps (hyp supporting-hyps)
      :UPDATE-EVENTS (supported-hyp-event))
   (supporting-hyps (hyp supported-hyps)
      :UPDATE-EVENTS (supporting-hyp-event)))
  :DIMENSIONAL-INDEXES
  ((time        time-location-list)
   (x           time-location-list)
   (y           time-location-list)
   (event-class event-class))
  :PATH-INDEXES
  ((node  node  :TYPE :label)
   (level level :TYPE :label))
  :PATHS
  ((t ('node-blackboards node 'hyp level)))).
```

The *dimensional indexes* define how attributes semantically specify the positioning of hypothesis units onto the dimensionality of a space. (The details of which attributes are actually used in locator construction are specified in the unit-space mapping discussed in Section E.) These specifications include the information required for destructuring when highly structured attribute values are used for unit positioning. Path indexes specify the space(s) on which created units are to reside. A unit is simply created by supplying its attributes:

```
(make-hyp :NODE                *current-node-number*
          :LEVEL               bb-level
          :TIME-LOCATION-LIST  time-location-list
          :EVENT-CLASS         event-class
          :BELIEF              computed-belief).
```

C. Abstracting Unit Retrieval

GBB's basic unit retrieval function, `find-units`, permits a complex retrieval to be specified in its pattern language. This declarative pattern language provides an abstraction over the blackboard database. A `find-units` pattern consists of an n-dimensional retrieval specification for particular classes of units on a blackboard space. This means that the KS code need only specify the desired classes of units, the spaces on which to look, and the values for the dimensions.

We will present an example of unit retrieval shortly.

D. Abstracting the Blackboard Path

Specifying a blackboard space in KS and control code is another area where data abstraction is important. In GBB, the *blackboard* is a hierarchical structure composed of atomic blackboard pieces called *spaces*. In addition to being composed of spaces, a blackboard can also be composed of other blackboards (themselves eventually composed of spaces). This hierarchy is a tree where the leaves are spaces and the interior and root nodes are blackboards. Units are always stored on spaces; GBB's blackboards simply allow the implementer to organize the set of spaces in the system. At a conceptual level, the space upon which to store the unit is specified by the sequence of nodes traversed from a root blackboard node through all intermediate blackboard nodes to the leaf space node. This sequence, which unambiguously specifies a space, is called the *blackboard/space path*. In addition, blackboards and spaces can be *replicated*, which creates multiple copies of blackboard subtrees. These copies of the blackboard structure are disambiguated by qualifying the replicated blackboard or space with a index.

In the original design of GBB, the blackboard path was directly specified in `find-units`. Even here, the lack of abstraction caused difficulty in modifying the blackboard structure without modifying the application code. For example, consider the DVMT application where the basic data blackboard consists of eight spaces (the abstraction levels SL, GL, VL, PL, ST, GT, VT, and PT). Using a very simple control shell for initial prototyping of the KSs, the blackboard structure might consist of a single blackboard containing the eight levels and another blackboard containing the scheduling queues. Later on, however, a more complicated control shell might be desired which contains a separate *goal blackboard* on which goal processing activities are performed. The goal blackboard mirrors the structure of the data blackboard, and contains eight corresponding spaces. Specifying complete blackboard/space paths makes such a transition cumbersome, because each call to `find-units` must be changed to reflect the new blackboard-space paths.

To eliminate this problem, GBB now provides an abstract path specification mechanism which allows blackboard/space paths to be specified relative to other paths, to another space instance, or to the spaces on which a unit instance resides. For example, the path to a stimulus hypothesis's space is coded as:

```
(make-paths :UNIT-INSTANCES stimulus-hyp).
```

The path to the ST level of a hyp in the DVMT application can be coded as: r

```
(change-paths
  (make-paths :UNIT-INSTANCES stimulus-hyp)
  '(:CHANGE-RELATIVE :UP st))
```

where `:UP` indicates to move up one level in the blackboard/space hierarchy and `st` indicates to move back down to the ST space.

The path to a corresponding goal space given a hypothesis unit in the DVMT application would be coded as:

```
(change-paths
  (make-paths :UNIT-INSTANCES stimulus-hyp)
  '(:CHANGE-SUBPATH hyp goal)).
```

The following call to `find-units` illustrates the use of abstraction in unit retrieval:

```
(find-units 'hyp
  ;; We look on the same space as the 'stimulus-hyp' ::
  (make-paths :UNIT-INSTANCES stimulus-hyp)
  '(:AND
     ;; Check for adjacent (in time) hypotheses within
     ;; the maximum velocity range of vehicle movement ::
     (:PATTERN-OBJECT
       (:INDEX-TYPE    time-location-list
        :INDEX-OBJECT  ,(hyp$time-location-list stimulus-hyp)
        :DISPLACE      ((time 1))
        :DELTA         ((x ,*max-velocity*)
                        (y ,*max-velocity*)))
       :ELEMENT-MATCH :within)
     ;; Check event class for frequency within
     ;; *max-frequency-shift* of stimulus-hyp ::
     (:PATTERN-OBJECT
       (:INDEX-TYPE    event-class
        :INDEX-OBJECT  ,(hyp$event-class stimulus-hyp)
        :DELTA         ((event-class ,*max-frequency-shift*))
        :ELEMENT-MATCH :within))))
```

E. Specifying the Implementation Machinery

Specifying how locators are to be constructed from unit attribute values is made by defining a mapping for each unit class onto each blackboard space. The mapping is specified in terms of the dimensionality of the space. For example, here is a simple implementation of the levels in the DVMT application where only the *time* dimension is used for locator construction (the other dimensions are checked during the filtering step of the retrieval process):

```
(define-unit-mapping (hyp) (pt pl vt vl gt gl st sl)
   :INDEXES (time)
   :INDEX-STRUCTURE
      ((time :SUBRANGES (:START :END (:WIDTH 1))))).
```

To add in other dimensions into the locator structure, only the mapping declaration need be changed. Here is the same definition implementing a locator strategy for *time* and *x-y-position*:

```
(define-unit-mapping (hyp) (pt pl vt vl gt gl st sl)
   :INDEXES (time (x y))
   :INDEX-STRUCTURE
      ((time :SUBRANGES (:START :END (:WIDTH 1)))
       (x    :SUBRANGES (:START :END (:WIDTH 10)))
       (y    :SUBRANGES (:START :END (:WIDTH 16))))).
```

The parentheses in the :INDEXES value in the above example indicates that the locators for the *time* dimension are to be implemented as a single vector and the locators for the *x* and *y* dimensions are to be grouped into a two-dimensional array. Without the extra level of parentheses, three vectors of locator structures would be implemented.

F. Abstracting the Control Interface

In GBB, the control interface is separated from the blackboard database implementation by viewing changes to the blackboard as a series of *blackboard events*. Control components are then defined to be triggered on particular events.

An important capability for constructing generic control shells is the definition of basic units (such as basic-hyp) that can be included in the definition of application units. GBB's unit inclusion mechanism (see the definition of the HYP unit in Section B) allows event handling to be appropriately inherited to the including unit's definition. The application implementer does not need to know the details of the event handling machinery in specifying blackboard units, and different control shells can be substituted without changing the unit definitions.

V. Summary

Blackboard database abstraction is an appropriate implementation goal for all the reasons typically associated with data abstraction. In this paper, we have described how information hiding abstractions can be combined to permit a blackboard implementation system to simultaneously provide flexibility, efficiency, and generality. These abstractions are:

1. Viewing blackboard levels (*spaces*) as *structured* n-dimensional volumes, blackboard objects (*units*) as occupying some extent within a space's n dimensions, and retrieval patterns as constrained volumes within a space's dimensions.
2. Extracting the information determining a unit's dimensional extent and the space(s) on which the unit is to be placed (the *blackboard path*) directly from the values of the unit's attributes and from the general (class) definition of the unit.
3. Specifying the constraints of a retrieval pattern relative to the attribute values of another (*stimulus*) unit.
4. Specifying the blackboard path for unit retrieval relative to the path of another (stimulus) unit or relative to a particular space instance.
5. Separating control machinery from the blackboard database implementation via the use of *blackboard events* to trigger control activities.
6. Separating the three phases of blackboard system development (blackboard and unit definition, application and control coding, and blackboard implementation specification), but combining the product of each phase in a code generation facility to produce an efficient, customized implementation.

These abstractions are implemented in the current release of GBB, and our initial experience using these information hiding abstractions indicate that they work well at providing flexibility, efficiency, and generality in the development of blackboard-based AI applications.

References

[Corkill *et al.*, 1986] 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 to appear in *Blackboard Systems*, Robert S. Engelmore and Anthony Morgan, editors, Addison-Wesley, in press, 1987).

[Erman *et al.*, 1981] Lee D. Erman, Philip E. London, and Stephen F. Fickas. The design and an example use of Hearsay-III. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 409–415, Tokyo, Japan, August 1981.

[Erman *et al.*, 1980] 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.

[Hayes-Roth, 1985] Barbara Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, July 1985.

[Johnson *et al.*, 1987] Philip M. Johnson, Kevin Q. Gallagher, and Daniel D. Corkill. *GBB Reference Manual.* Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, GBB Version 1.00 edition, March 1987.

[Lesser and Corkill, 1983] Victor R. Lesser and Daniel D. Corkill. The Distributed Vehicle Monitoring Testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3):15–33, Fall 1983. (Also to appear in *Blackboard Systems*, Robert S. Engelmore and Anthony Morgan, editors, Addison-Wesley, in press, 1987 and in *Readings from AI Magazine 1980–1985*, in press, 1987).

[Nii, 1986] H. Penny Nii. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(2):38–53, Summer 1986.