FROM PROTOTYPE TO PRODUCT: EVOLUTIONARY DEVELOPMENT WITHIN THE BLACKBOARD PARADIGM

Daniel D. Corkill, Kevin Q. Gallagher
and Philip M. Johnson
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 86-46

November 26, 1986

From Prototype to Product: Evolutionary Development within the Blackboard Paradigm

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

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

COINS Technical Report 86-46

November 26, 1986

1 Introduction

Blackboard architectures have become a popular paradigm for developing knowledge-based systems since they were introduced in the Hearsay Speech Understanding System [1]. The functional independence of knowledge sources, flexibility in the choice of control strategy, and the structuring of blackboard information make blackboard architectures a powerful framework for understanding the computational requirements of a knowledge-based application. A generic blackboard implementation tool allowing rapid implementation of a prototype blackboard-based application would encourage more knowledge-based application developers to explore the characteristics of their application using the generality of the blackboard approach.

Presented at Workshop on High Level Tools for Knowledge Based Systems, Columbus, Ohio, October 7–8, 1986.

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

Although the blackboard paradigm is useful during an application's development, many question the efficiency of the blackboard approach. They feel that applications developed as blackboard systems must be rewritten to become a more efficient (but less flexible) production quality system. This transformation can complicate the understandability, maintenance, and further development of the application as well as increase the programming effort required to deliver the application.

We believe that, with proper software support, an application can be optimized *within* the blackboard paradigm, and that prototyping ease and high performance need not be contradictory goals. The notion that blackboard systems are inefficient has been aided by blackboard implementations that have been layered on top of other support systems that are ill-suited to blackboard operations. If the generic blackboard implementation tool can generate code that is tuned to the particulars of the application, the prototype system itself can be transformed into the production quality system.

Support for this transformation philosophy was a major design goal behind the development of the Generic Blackboard (GBB) development system [2]. The approach taken in GBB emphasizes flexibility, modularity and ease of implementation without sacrificing system efficiency or performance. In this paper we describe how GBB is structured to separate design decisions from efficiency decisions (i.e., the implementation machinery).

2 The Basic Philosophy

In GBB the blackboard and blackboard objects are specified using high-level declarations. Application design decisions can be changed by simply changing the declarations and efficiency decisions can be delayed by leaving some declarations underspecified. Delaying detailed efficiency decisions allows an application to be quickly prototyped while postponing efficiency issues.

To achieve optimum performance in blackboard systems the low-level blackboard database operations as well as the control shell must be made as efficient as possible. In GBB, after the design is prototyped the implementation can be tuned to enhance its performance by adding additional information to the declarations.

The emphasis on database efficiency separates GBB from the generic blackboard architectures of Hearsay-III [3] and BB1 [4]. Although both Hearsay-III and BB1 are domain independent blackboard architectures, their focus is on generalizing control capabilities. The major contribution of GBB is not in any extension of the technology of blackboard architectures, but in the unification of existing blackboard technologies into a development system for high-performance applications.

3 Specifying the Blackboard Structure

An important feature of blackboard systems is that the blackboard database is structured. In addition to being divided into multiple information levels, each level can be organized by *dimensions*. That is, the blackboard objects are placed onto appropriate areas within each level based on their attributes. For example, in the Distributed Vehicle Monitoring Testbed (DVMT) [5], each level has three dimensions: *time* and *x*, *y* position. A blackboard system can use this information to minimize search associated with insertion/retrieval of blackboard objects.

An important aspect of the level dimensionality in the DVMT is that each dimension is *ordered*. This means that there is a notion of objects being "nearby" other objects. In the DVMT, a vehicle classification is made from a component frequency track by looking on the blackboard for other component frequency tracks that are positioned close to the original track throughout its length. In addition to ordered dimensions, GBB supports *enumerated* dimensionality.

In GBB, the blackboard is a hierarchical structure composed of atomic blackboard pieces called *spaces*.[1] For example, blackboard levels such as signal location and signal track of the DVMT are implemented as spaces in GBB. In addition to being composed of spaces, a blackboard can also be composed of other blackboards (themselves eventually composed of spaces).

The definition of the logical structure of the blackboard is kept separate from the storage specification so that a problem may be expressed in the most natural fashion without sacrificing performance. This is an example of a space definition in GBB:

```
(define-spaces (vehicle-location vehicle-track)
   :DIMENSIONS
     ((time :RANGE (0 30))
      (x    :RANGE (-1000 1000))
      (y    :RANGE (-1000 1000))
      (classification
            :ENUMERATED (chevy porsche toyota vw-beetle unknown))))
```

The dimensionality of each space is an important part of system design. Although GBB provides flexibility in specifying space dimensionality, the application implementor must determine what is appropriate for the particular application. It should be stressed that the choice of dimensions is an issue of representational clarity and expressiveness — not of database efficiency. Efficiency decisions will be discussed in Section 6.

[1] In designing GBB, we used names that did not evoke preconceived notions from previous blackboard systems. Hence the term "space" rather than "level" and the term "unit" rather than "hypothesis" or "object".

4 Specifying Blackboard Objects

In GBB, all blackboard objects are termed *units*. Hypotheses, goals, and knowledge source activation records, are typical examples of units. A unit is an aggregate data type similar to those created using the Common Lisp **defstruct**.

In order to place blackboard objects onto spaces a blackboard system must be able to extract from the object the values associated with the space dimensions. For example, to place a signal on a space that has the dimensions *time*, *x*, and *y* the blackboard system would have to get values for *time*, *x*, and *y* from the object. We call values for space dimensions extracted from objects like this *indexes*.

An application may represent this indexing information in many ways. An index may be simply the value of a slot in an object. One or more indexes may also be computed from a single structured slot value, in which case the blackboard system must be able to extract the index information from the overall structure.

There may be very strong reasons for grouping information in a particular way and the blackboard system shouldn't force the application programmer to break it up in an unnatural way. In the DVMT hypotheses may be associated with a point in three-space (*time*, *x*, *y*), or a two-dimensional *x*, *y* region at a particular *time*. The indexing information for these two types of location is represented differently but for the purpose of placing and retrieving objects on the blackboard the indexes should be treated similarly.

A much more complex situation stems from the need to support composite-units. A *composite unit* is a unit that has multiple *elements* along one or more of its dimensions. An example of a composite unit is a track of vehicle sightings. Each sighting is an *x*,*y* point at a particular moment in *time*. One way to represent such a track is a *time-location-list*:

```
((time1 (x1 y1))
 (time2 (x2 y2))
    ...
 (timeN (xN yN))).
```

Such a unit does not occupy a single large volume of the blackboard, but rather a series of points connected along the time dimension. To indicate this, **time-location-list** must be declared as a composite index-structure.

In GBB, the information needed to decode a structured slot value into its dimensional indexes is specified using **define-index-structure**. This example specifies how to get the indexes *time*, *x*, and *y* from a point in three-space:

```
(define-index-structure TIME-LOCATION
    :TYPE time-and-location
    :INDEXES ((time :POINT time)
              (x :POINT location (location x))
```

```
(y :POINT location (location y))))).
```

Here, `time-and-location` is a defstruct with two slots: `time` and `location`. The data in the `location` slot contains another a defstruct of type `location` which has two slots `x` and `y`. GBB defines internal access functions to extract each index. In the example, GBB would know how to access the `x` index from the datatype time-location as:[2]

```
(location$x (time-and-location$location time-location)).
```

5 Retrieval Issues

Obviously, the underlying implementation of the blackboard has a great impact on the performance of the whole system. A strategy used by some blackboard systems is to implement each blackboard level as a list. New objects are simply pushed onto the list and objects are retrieved by checking each object in the list to see if the match conditions were statisfied. But the simple operation of mapping across all the elements in the list becomes an expensive operation when the blackboard contains thousands of objects. This cost is incurred on each retrieval and is multiplied by the cost of the match computation. This is not a viable strategy for production quality systems.

A better strategy is to select a small set of likely candidates and then apply the match criteria to the elements of that set. The space dimensionality is used to partition the storage for each space based on the values of some or all of the dimensions. Each partition contains a list of objects which fall within a narrow range of values along the selected dimensions. The retrieval then proceeds as follows. The primary retrieval step examines the match criteria and identifies which partitions contain the desired objects. It passes this subset on to the secondary retrieval step which filters out the objects that don't satisfy the match criteria. This eliminates the exhaustive search of the space and replaces it with an efficient focused search.

6 Implementing the Database

The previous sections presented the blackboard and object specifications that must be specified by the application implementer. To this point, the specifications defined representational aspects of the application. This section describes how particular implementations of the blackboard database are specified. We concentrate on ordered dimensions—enumerated dimensions are implemented as sets.

Simple hashing techniques do not work for ordered dimensions due to the neighborhood relationship among units. The storage structure must be able to quickly

[2] In this paper, all accessor functions are defined using :CONC-NAMEs ending with "$".

locate units within any specified range of a dimension. A standard solution is to divide the range of the dimension into a series of *buckets*. Each bucket contains those units falling within the bounds of the bucket. The number of buckets and their sizes provide a time/space tradeoff for unit insertion/retrieval. The bucket approach requires that a pattern range be converted into bucket indexes and that units retrieved from the first and last bucket be checked to insure that they indeed are within the pattern range.

In a three dimensional blackboard (*x*, *y*, and *time*) the bucket approach becomes more complicated. One approach would be to define a three dimensional array of buckets. A second approach would be to define three one dimensional bucket vectors and have the retrieval process intersect the result of retrieving in each dimension.

In GBB, the application developer can specify what dimensions should be used to organize the storage. This defines what dimensions will be used in the primary retrieval, which need not include all the dimensions defined for the space. The programmer can also specify how to partition the dimensions into buckets. GBB can not infer the distribution of objects across a dimension, so this provides the developer a way to optimize the storage/retrieval of objects.

To indicate that several dimensions should be stored together in one array, they are grouped together with an extra level of parentheses. For example, `((time x y))` would specify a three-dimensional array, and `(time (x y))` would specify a vector for *time* and a two dimensional array for (*x*, *y*).

In this example, the storage for the `vehicle-track` space for units of type `hyp` and `goal` will be three one-dimensional arrays, one array for each dimension *time*, *x*, and *y*.

```
(define-unit-mapping (hyp goal) (vehicle-track)
  :INDEXES (time x y)
  :INDEX-STRUCTURE
  ((time :SUBRANGES
         (:START 5)
         (5 15 (:WIDTH 5))
         (15 25 (:WIDTH 2))
         (25 :END))
   (x :SUBRANGES (:START :END (:WIDTH 5)))
   (y :SUBRANGES (:START :END (:WIDTH 2)))))
```

If no *unit-mapping* is specified then the storage is implemented as a list. So, in the prototyping stage the programmer need not define a unit-mapping; later, as the application develops, a unit-mapping can be defined and then tuned to achieve better performance.

7 Object Retrieval (Pattern Matching)

Blackboard systems spend a significant amount of time searching the database. Because retrieval is so important we have given the application programmer the means to make it as efficient as possible by eliminating candidate units early in the retrieval process. This is done in two ways. First, the user can specify specialized filter functions that are applied between the primary retrieval of units (such as from a set of buckets) and the subsequent checking of pattern inclusion. Second, the pattern language is rich enough to allow the application programmer to specify complex retrieval patterns that can be analyzed and optimized by GBB. The result is a reduction in retrieval time and, equally important, a reduction in the amount of temporary storage and consing required for unit retrieval.

The retrieval *pattern* describes the criteria that must be met by the units retrieved. The simplest pattern is the keyword `:ALL`; that matches all of the specified units on the specified space. A pattern can also be quite complex, represented by a list of pattern specifiers. Much of the richness in the pattern specifier language supports the retrieval of composite-units.

A non-trivial pattern may be either an index element, a composite structure, or a concatenation of index elements or composite structures. Index structures that are concatenated together need not all be the same nor does the index structure of the pattern need to be the same as the index structure of the unit. GBB is able to efficiently map from one index structure representation to another. When a pattern needs to be constructed by splicing together components of different index structures GBB decomposes all patterns/objects into sequences of simple dimensional ranges to avoid expensive type conversions.

The pattern is expressed in terms of the unit's indexes and it is independent from the implementation of the blackboard database. Changes to the implementation of the database don't affect retrieval behavior but do affect the performance. Initial prototypes may not even define a unit-mapping, in which case all the units on a space will be checked for pattern inclusion. By tuning the unit-mapping the programmer can make the primary retrieval more effective at reducing search during retrieval.

8 Blackboard Control Shells

Controlling problem-solving activity can be crucial to the performance of blackboard-based AI applications. The blackboard paradigm's control flexibility encourages an opportunistic approach to problem solving, where problem solving activities can be quickly refocused as new information is uncovered. A number of control approaches have been used to date, ranging from the utility-based, priority-ordered agenda of the Hearsay-II speech understanding system [1] to the layered control approaches of Hearsay-III and BB1 [3,4]. We feel that there is insufficient experience to select a

particular control approach on all blackboard applications. In fact, the diversity of blackboard-based applications may require a repertoire of control approaches.

To support a range of control architectures, a clean separation was made between the database support subsystem of GBB and the control level. This allows different control *shells* to be implemented using the common database support subsystem. A control shell has its own language for defining knowledge sources, specifying the conditions for their activation, and how they are to be scheduled for execution. Control shells may also create other control units, such as goals or plans. The interface between the two subsystems is a set of *blackboard events*, signals indicating the creation, modification, or deletion of blackboard objects.

A GBB control shell often requires the ability to quickly retrieve control information. For example, identifying which pending knowledge source activations potentially generate results in a particular blackboard region can be naturally expressed in terms of blackboard operations. By implementing the control shell *itself* as a blackboard-based problem solving activity, the full power and efficiency of GBB's blackboard database support subsystem can be utilized. In particular, efficient storage and retrieval of blackboard control objects is as important to the performance of the control layer as it is in the problem solving domain.

The control shell approach facilitates rapid prototyping, by providing a set of generic control shells. An application implementer would select an appropriate control shell and use it during the early prototyping stages. As experience with the application is gained, the generic shell is tailored by adding refined, application-specific knowledge to the shell's decisionmaking machinery. Such tailoring increases the efficiency of the application system by reducing inappropriate problem solving activities. In extreme cases where a tailored shell cannot provide sufficient control capabilities, a custom-built control layer can be written, using GBB's database machinery.

References

[1] 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.

[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.

[3] 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.

[4] Barbara Hayes-Roth.
A blackboard architecture for control.
Artificial Intelligence, 26(2):251–321, March 1985.

[5] 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.