GBB: A Generic Blackboard Development System

Daniel D. Corkill, Kevin Q. Gallagher, and Kelly E. Murray

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

Abstract

This paper describes a generic blackboard development system (GBB) that unifies many characteristics of the blackboard systems constructed to date. The goal of GBB is to provide flexibility, ease of implementation, and efficient execution of the resulting application system. Efficient insertion/retrieval of blackboard objects is achieved using a language for specifying the detailed structure of the blackboard as well as how that structure is to be implemented for a specific application. These specifications are used to generate a blackboard database kernel tailored to the application. GBB consists of two distinct subsystems: a blackboard database development subsystem and a control shell. This paper focuses on the database support and pattern matching capabilities of GBB, and presents the concepts and functionality used in providing an efficient blackboard database development subsystem.

I Introduction

Historically, blackboard-based AI systems have been implemented from scratch, often by layering a blackboard architecture on top of other support systems. This has fostered a notion that blackboard-based architectures are difficult to build and slow in execution. Despite this notion, AI system implementers are increasingly considering blackboard architectures for their applications. Unlike rule-based and frame-based AI architectures where a variety of commercial and academic system development shells are now available, an application developer considering an blackboard approach remains largely unassisted.

A microcosm of this situation existed at the University of Massachusetts. Several large blackboard-based AI systems had been implemented [1,2], and a number of additional blackboard-based applications were being considered. We decided to pool our experience in implementing blackboard systems into a common development system. We felt that by consolidating our implementation resources we could construct a generic system that would be more efficient than any of the individual systems, if they all were constructed from scratch. The goal for the blackboard development system was to reduce the time required to implement a specific application and to increase the execution efficiency of the resulting implementation.

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.

This paper describes the resulting generic blackboard development system, termed **GBB** (Generic Blackboard). The GBB approach is unique in several aspects:

1. A strong emphasis was made on efficient insertion and retrieval (pattern matching) of blackboard objects. GBB was designed to efficiently implement large blackboard systems containing thousands of blackboard objects.
2. A non-procedural specification of the blackboard and blackboard objects is kept separate from a non-procedural specification of the insertion/retrieval storage structure (Figure 1). This allows a "blackboard administrator" to easily redefine the blackboard database implementation without changing the basic blackboard/object specification or any application code. Such flexibility is not only important during the initial development of the application system, but also to maintain efficient database operation as the scale and characteristics of the application evolve during its use. Both specifications are used by the GBB database code generator to produce an efficient blackboard kernel tailored for the specific application.
3. We have defined a general *composite* blackboard object for representing objects composed of discrete elements (such as a phrase of words or a track of vehicle sightings).
4. We have defined a pattern language for retrieving simple and composite objects from the blackboard. The application programmer has the ability to insert additional procedural filtering functions into the basic retrieval process. This can be significantly more efficient than applying the filters to the results of the retrieval.
5. Ä clean separation was made between the database support subsystem of GBB and the control level.

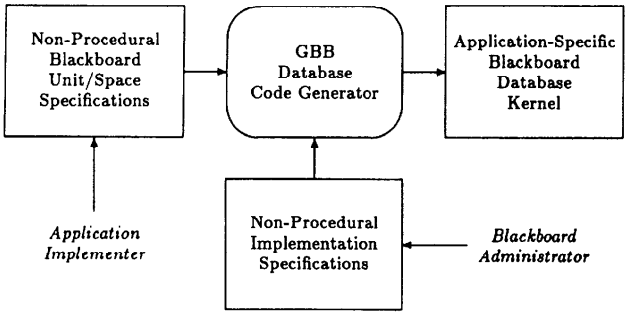

Figure 1: The GBB Database Subsystem

This allows different control shells to be implemented using the common database support subsystem. (We feel that it is premature to force a particular control architecture on all blackboard applications.) The interface between the two subsystems is a set of *blackboard events*, signals indicating the creation, modification, or deletion of blackboard objects. We are implementing several control shells as part of GBB, however an application implementer is free to develop a different control shell using the GBB database subsystem.

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.

The remainder of the paper describes GBB in more detail, focusing on its database support subsystem and pattern matching capabilities. The implementation of control shells will not be described. GBB is implemented in Common Lisp and is now being tested in a Common Lisp reimplementation of the Distributed Vehicle Monitoring Testbed [1].

II Specifying the Blackboard Structure

An application implementer using the GBB system must specify the structure of the blackboard and the objects that will reside on it. In GBB, the blackboard is a hierarchical structure composed of atomic blackboard pieces called *spaces*.[1] For example, the blackboard abstraction levels (phrase, word, syllable, etc.) of the Hearsay-II speech understanding system would be 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). Finally, blackboards and spaces can be replicated at blackboard initialization time (discussed in Section V).

Spaces are defined first, using **define-spaces**:

define-spaces *spaces* [*documentation*] &KEY *dimensions* [*Macro*]

Spaces is a list of space names. *Dimensions* is a list of specifiers defining the dimensionality of *spaces*. The concept of *space dimensionality* is crucial to efficient insertion/retrieval of blackboard objects, and is best introduced by examples from existing systems.

In addition to having the blackboard subdivided into multiple information levels, the levels of the Hearsay-II speech understanding system (HS-II) [5] and the Distributed Vehicle Monitoring Testbed (DVMT) [1] are *structured*. That is, the blackboard objects are placed onto appropriate areas within each level based on their attributes. In HS-II, each level has one dimension, *time*. In the DVMT, each level has three dimensions: *time* and *x*, *y* position. In both systems, each level in the system has the same dimensionality. This may not be the case for other application areas, and GBB allows individual spaces to have different dimensionality. (Spaces with differing dimensionality are declared using multiple calls to **define-spaces**.)

An important aspect of the level dimensionality in HS-II and the DVMT is that each dimension is *ordered*. This means that there is a notion of objects being "nearby" other objects. In HS-II this idea of neighborhood allows retrieval of words to extend a phrase whose begin time is "close" to the phrase's end time. 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. An enumerated dimension contains a fixed set of labeled categories. For example, in the DVMT, a hypothesis classifying a vehicle could be placed on a space containing a "classification" dimension, where the dimension's label set consists of vehicle types. GBB allows a space to have both ordered and enumerated dimensions.

The dimensionality of each space is an important part of system design. Although GBB provides flexibility in specifying space dimensionality, the application implementer must determine what is appropriate for the particular application. It should be stressed that specifying the dimensionality of spaces is primarily an issue of representation—not of database efficiency. Efficiency decisions will be discussed in Section IV.

Returning to the dimension specification in **define-space**, each dimension is specified as a list where the first element is the name of the dimension and the remainder is a list of keyword/value pairs describing the dimension. The two defined keywords are :RANGE, corresponding to ordered dimensions, and :ENUMERATED, corresponding to dimensions of enumerated classes.

The argument to :RANGE is a list of (*lower-bound upper-bound*) or :INFINITE, indicating a range of $(-\infty, +\infty)$. The argument to :ENUMERATED is the label set for the enumerated dimension. For example:

```
(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))))
```

defines two spaces with identical dimensionality. It is an error to attempt placement of a blackboard object outside the range of an ordered dimension or outside the label set of an enumerated dimension. If *dimensions* is omitted or nil, the space has no dimensionality. Such spaces are *unstructured*.

Once an application's spaces have been defined, the blackboard hierarchy is defined using **define-blackboards**:

[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".

define-blackboards *blackboards components* [*documentation*] [*Macro*]

Blackboards is a list of blackboard names. *Components* is a list of symbols naming those spaces and/or blackboards that will be the children of *blackboards*. For example:

```
(define-blackboards (hyp-bb goal-bb)
   (signal-location vehicle-location vehicle-track))
```

defines two blackboards, each having three spaces.

III Specifying Blackboard Objects

Once the blackboards and spaces have been specified, the blackboard objects are defined. 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** macro, but only units can be placed onto blackboard spaces. Units are defined using **define-unit**:

define-unit *name-and-options* [*documentation*] &KEY *slots links indexes* [*Macro*]

The *name-and-options* argument is exactly the same as **defstruct** with several extensions. First, a function to generate a name for each unit instance can be specified (using :NAME-FUNCTION). This function is called with the newly created unit instance after all slots in the unit have been initialized, but before the unit is placed onto the blackboard. The function returns a string that is used in a special read-only slot, **name**, that is implicitly defined if the :NAME-FUNCTION option is used. Second, it is often useful when interacting with a blackboard system to retrieve a unit by name rather than through a pattern match on its attributes. GBB provides this capability through a separate hash table of units (indexed by name) that can be dynamically created/destroyed as needed. Code for performing these activities is generated for a unit if the :HASH-UNIT option value is non-nil. Finally, the signaling of blackboard events associated with creating and deleting the unit can be controlled using the :EVENTS option. The :EVENTS argument is a list of :CREATION and/or :DELETION, indicating which events are to be signaled to the control shell for instances of this unit. The :EVENTS argument can also be nil, indicating that no unit events are to be signaled.

The *slots* argument contains a list of slot-descriptions that are also identical to **defstruct** with one addition. Any slot can have a slot option :EVENTS that is a list of *event-name* and *event-predicate-function* pairs. Each event-predicated function is evaluated each time the value of the slot is modified. If the event-predicate-function returns true, the corresponding event-name is signaled.

The *links* argument defines additional slots that hold interunit links. The name of the link is used as the new slot name.[2] By default, GBB forces all links to be bidirectional; each outgoing unit link must be defined with an accompanying inverse incoming link. GBB generates special modification functions (linkf for adding a single link, linkf-list for adding a list of links, and unlinkf and unlinkf-list for deleting links) that maintain consistent link bidirectionality. For example:

```
(linkf (hyp$creating-ksi this-hyp) current-ksi)
```

adds current-ksi as a new *creating-ksi* of this-hyp.

Each link-description in *links* has the form:

```
(link-name [:SINGULAR]
           {:REFLEXIVE |
            (other-unit other-link [:SINGULAR])}
           [:EVENTS event-descriptions]).
```

For example, here is a bidirectional link between hypothesis units:

```
(supported-hyps (hyp supporting-hyps))
(supporting-hyps (hyp supported-hyps)).
```

The keyword :SINGULAR is used to implement one-to-one, one-to-many, and many-to-one links (the default is many-to-many). The keyword :REFLEXIVE is simply a shorthand for:

```
(link-name (this-unit-name link-name)).
```

The optional :EVENTS argument is identical to the event-predicate-function specification discussed for *slots*.

The *indexes* argument specifies how the unit is mapped onto spaces (termed *indexing*) There must be a space dimension corresponding to each unit index (additional space dimensions are acceptable). Slots containing unit indexing information must be described by an index-description of the form:

```
(index-name slot-name)
```

where *index-name* is an *indexing-structure specification* that describes how to extract the dimensional indexes from *slot-name*. Indexing-structures are defined using **define-index-structure** discussed below.

In its simplest form, an index is just the name of a **define-unit** slot defined in *slots*. For example, if a unit had a slot named *time* containing a numeric value, GBB would have no problem placing that object on the time dimension of a space. Handling a slot value containing a range (such as the time span of a phrasal hypothesis) is also straightforward. Unfortunately, things are not always simple. One problem is that the indexes may be only a portion of a structured slot value, and GBB must be told how to extract the index information from the overall structure. 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))).
```

[2] Note that the slot-names defined by *links* are implicitly defined as slots and are not included in **define-unit**'s *slots* argument.

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.

The information needed to decode a datatype into its dimensional indexes is specified using **define-index-structure**:

define-index-structure *name* [*Macro*]
[documentation]
&KEY *type*
composite-type
composite-index
element-type indexes

The *name* argument is a symbol that is defined as a new Lisp datatype. *Type* is used when the datatype to be decoded is a simple (non-composite) datatype and simply defines the new datatype *name* as a synonym for the existing datatype *type*. For a composite datatype, *composite-type*, *composite-index*, and *element-type* must be specified in place of *type*. The *composite-type* argument specifies the type of sequence that contains the individual index elements. *Composite-index* specifies the dimension connecting the composite elements (for example, *time*). *Element-type* specifies the datatype of the composed elements. Finally, *indexes* defines how to extract the dimensional indexes from each element. The format for each index-dimension specifier is:

(*dimension* {:POINT *field* {(*type field*)}* |
:RANGE (:MAX *field* {(*type field*)}*)
(:MIN *field* {(*type field*)}*)}).

For example:

```
(define-index-structure TIME-LOCATION-LIST
    :COMPOSITE-TYPE list
    :COMPOSITE-INDEX time
    :ELEMENT-TYPE time-location
    :INDEXES ((time :POINT time)
              (x :POINT location (location x))
              (y :POINT location (location y))))
```

In the above example, GBB would know how to access the **x** index from the first element of the composite datatype **time-location-list** as:

```
(location$x (time-location$location
             (first time-location-list))).
```

Note that all *types* and *fields* must be defined using **defstruct** or **define-units**.

Returning to the *indexes* argument of **define-unit**, *slot-name* is the name of a slot (from the :SLOTS argument). The slot must have a :TYPE slot-option whose value is the name of an index-structure. *Index-name* must be an index in that index-structure.

Here is a highly-abridged version of the hypothesis unit specification in the DVMT:

```
(define-unit (HYP (:CONC-NAME "HYP$")
                  (:NAME-FUNCTION generate-hyp-name)
                  (:HASH-UNIT nil))

  "HYP (Hypothesis)"
  :SLOTS
  ((belief 0 :TYPE belief)
   (classification)
   (sensor-id 0 :TYPE sensor-index)
   (time-location-list () :TYPE time-location-list))
  :LINKS
  ((consistency-hyp :SINGULAR (hyp consistent-hyps))
   (consistent-hyps (hyp consistency-hyp :SINGULAR))
   (supported-hyps  (hyp supporting-hyps))
   (supporting-hyps (hyp supported-hyps))
   (creating-ksis   (ksi created-hyps)))
  :INDEXES
  ((time time-location-list)
   (x time-location-list)
   (y time-location-list)
   (classification classification))
```

IV Implementing the Database

The previous sections presented the blackboard and unit 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 typically implemented as sets or hash tables.

The implementation machinery for storing units on spaces is specified using **define-unit-mapping**:

define-unit-mapping *units spaces* [*Macro*]
[documentation] &KEY
indexes index-structure

Units is a list of unit names, where each unit has identical index dimensions (as defined by the `define-unit` *indexes* argument). *Spaces* is the list of spaces whose implementation machinery is being defined. Note that the same unit type can be stored differently on different spaces, and that different unit types can be stored differently on the same space. *Indexes* is the list of indexes whose implementation machinery is being defined. *Index-structure* defines the implementation machinery.

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

Here is a three one-dimensional vector example:

```
(define-unit-mapping (unit1 unit2) (space1)
  :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))))).
```

V Instantiating the Blackboard

Once the structure of the blackboard database has been specified with the functions presented above, it may be instantiated. This creates all the internal structures needed by GBB to actually store unit instances. Sometimes it is useful to be able to create several copies of the entire blackboard database or copies of parts of it. For example, to simulate a multiprocessor blackboard system one could instantiate a copy of the blackboard database for each processor. Instantiation is done via **instantiate-bb-database**:

instantiate-bb-database *replication-desc* [*Function*]

Replication-desc describes the blackboard hierarchy to be created. In the simplest case, it is a symbol that names the root of the tree to be instantiated.[3] This would instantiate one copy of each of the nodes in the tree (all the leaves would be space instances and the interior nodes would be blackboard instances). The general form of *replication-desc* is:

{*name* | (*name* [*replication-count*] [*description* ...])}.

Name is the name of a blackboard or a space; *replication-count* is an integer specifying how many copies of the subtree to create; and *description* is a *replication-desc* for one of the components of the specified blackboard (or space). For example:

```
(instantiate-bb-database
  '(top-level 3 (goal-bb (level-one 2))
                (hyp-bb (level-three 3))))
```

would create three copies of the blackboard database rooted at the blackboard `top-level`. Each copy would have two copies of `level-one` and three copies of `level-three`. Any defined blackboards or spaces not mentioned in the *replication-desc* would have one copy created.

[3]Note that this need not be the root of the entire blackboard hierarchy but can be any node in the tree. This would allow, for example, different parts of the blackboard database to be distributed (and possibly replicated) across a network of processors.

[4]The unit creation function is automatically generated by `define-unit`.

VI Creating Units

A unit is created and placed onto a space using the function `make-`*unit-type*:[4]

make-*unit-type* {*blackboard-path-element*}+ {*slot-keyword slot-value*}* [*Function*]

The *blackboard-path-element* arguments uniquely name the space that is to be searched. The simplest blackboard path is a space name. If a space name is not unique, it must be qualified by its parent blackboards' names until it is unique. In addition, replicated blackboards and/or spaces must be appropriately indexed.

Values for the newly created unit can be specified by *slot-keyword slot-value* pairs. Link slots can also be specified for the newly created unit, and GBB insures that inverse links are also created.

In addition to creating the unit, `make-`*unit-type* constructs the indexing information needed to retrieve the unit from the blackboard, invokes the name generation function (if specified in `define-unit`), and inserts the unit into the unit hash table (if unit hashing is enabled).

VII Unit 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 initial 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 primitive function for retrieving units from spaces is **find-units**:

find-units *units* {*blackboard-path-element*}+ &KEY *pattern filter-before filter-after* [*Macro*]

The *units* argument identifies which unit types are to be retrieved. The *blackboard-path-element* arguments uniquely name the space that is to be searched. The simplest blackboard path is a space name. If a space name is not unique, it must be qualified by its parent blackboards' names until it is unique. In addition, replicated blackboards and/or spaces must be appropriately indexed.

The two keyword arguments *filter-before* and *filter-after* specify predicates to perform application specific filtering of the candidate units. The *filter-before* predicates are applied to the initially retrieved units before the pattern matching tests and are intended as a quick first test to

shrink the search space. The *filter-after* predicates are run after the pattern matching tests and can perform additional acceptance testing.

The other keyword argument is the retrieval *pattern* that 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, and many of the options are meaningless unless the pattern's index structure is a composite structure.

A non-trivial pattern is based on a *pattern-object* that 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-object* specifies a region of the blackboard in which to look for the units. It is either a list of options or, to concatenate several index structures, a list whose first element is the symbol :CONCATENATE and whose remaining elements are lists of options. The keywords used to specify the *pattern-object* are:

index-object: This is an index structure, for example, a time-location-list.

index-type: This is the type of the *index-object*. It is the name of an index structure.

select: This allows extraction of a subsequence of a composite structure based on the *value* of the composite index (for example, *time*).

subseq: This allows extraction of a subsequence of a composite structure based on the *position* in the sequence (the same as selection of a subsequence of a vector).

delta: This expands or contracts a range or expands a point into a range.

displace: This allows the *index-object* to be translated along one or more of its dimensions.

The other pattern specifier keywords are:

element-match: This specifies how each index element from the unit is compared with the index element from the *pattern-object*. It may be one of :EXACT, :OVERLAPS, :INCLUDES, or :WITHIN. :EXACT means that the unit's index element must exactly match the pattern's. :INCLUDES means that the unit's index element must include the pattern's. :WITHIN means that the unit's index element must be within the pattern's. :OVERLAPS means that the unit's index element must overlap with the pattern's.

before-extras and *after-extras*: The argument is a range that specifies the minimum and maximum number of index elements that the unit may have before (or after) the index elements mentioned in the *pattern-object*. The argument can also be :DONT-CARE that is short for the range (0 MOST-POSITIVE-FIXNUM).

match: This is an inclusive lower bound on the number of index elements that must match. This can either be expressed as a percentage of the length of the *pattern-object*, by saying (:PERCENTAGE 50) or an absolute count by saying (:COUNT 5), or as a difference from the length of the *pattern-object* by saying (:ALL-BUT 2).

mismatch: This is an inclusive upper bound on the number of index elements that are allowed to not match. "Not matching" means that the unit has an index element for that composite index (for example, time) that does not match (according to the :ELEMENT-MATCH criterion) with the index element in the *pattern-object*. This does not include index elements that appear in the *pattern-object* but do not have a corresponding index element in the unit (call these *skipped*). (See Figure 2.)

contiguous: If this is true, then the index elements that match must be contiguous along the composite index dimension.

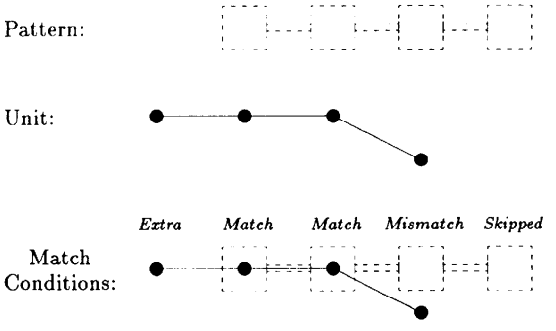

Figure 2: Composite Unit Matching Conditions

For example:

```
(find-units '(ghyp hyp) 'goal-bb 'vehicle-track
    :PATTERN
      (:PATTERN-OBJECT
          (:CONCATENATE
              (:INDEX-TYPE      time-region-list
               :INDEX-OBJECT
                 (#<TIME-REGION 3 (8 11) (4 6)>
                  #<TIME-REGION 4 (6 10) (6 8)>
                  #<TIME-REGION 5 (5 8) (8 9)>)
               :DISPLACE        ((x 4) (y 2)))
              (:INDEX-TYPE      time-location
               :INDEX-OBJECT
                 #<TIME-LOCATION 6 4 10>
               :DELTA           ((x 2) (y 2))))
       :ELEMENT-MATCH     :INCLUDES
       :MATCH             (:PERCENTAGE 75)
       :MISMATCH          2
       :BEFORE-EXTRAS     (0 5)
       :AFTER-EXTRAS      (0 0)
       :CONTIGUOUS        T)
   :FILTER-BEFORE '(sufficient-belief)).
```

Another useful form of unit retrieval is provided by **map-space**:

map-space *function units* [*Function*]
{*blackboard-path-element*}+

Function specifies a function that is to be applied to each type of unit (specified in *units*) that resides on the space specified by {*blackboard-path-elements*}+. **Map-space** insures that *function* is not applied more than once to any unit.

VIII Summary and Future Developments

High-performance blackboard-based AI systems demand much more than a multilevel shared database. An application's blackboard may have thousands of instances of a few classes of blackboard objects scattered within its database. GBB provides an efficient blackboard development system by exploiting the detailed *structure* of the blackboard in implementing primitives for inserting/retrieving blackboard objects. A *control shell* (implemented using GBB's blackboard database support) is used to generate a complete application system.

We have presented a brief description of the database subsystem of GBB. Length limitations have prevented a thorough discussion of all the details and rationale for particular decisions. We have tried to convey both the capabilities of GBB database support and some of the issues that must be faced in implementing a high-performance blackboard development system.

Although GBB has been implemented and is in use, its development continues. Much effort is being applied to performing compile time optimizations of insertion/retrieval operations. The next phase of GBB development will be to extend the space specification and initialization aspects of GBB to support a blackboard database that is distributed among a network of processing nodes.

References

[1] 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.

[2] Allen R. Hanson and Edward M. Riseman.
VISIONS: A computer system for interpreting scenes.
In Allen R. Hanson and Edward M. Riseman, editors, *Computer Vision Systems*, pages 303–333, Academic Press, 1978.

[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] 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.