GBB Reference Manual
GBB Version 1.2

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

June 1988
COINS Technical Report 87-120

Abstract

This manual describes Version 1.2 of the Generic Blackboard Development System (GBB), a high-level implementation tool designed to provide an application builder both speed and flexibility in implementing a blackboard-based application as well as efficient execution of the resulting application. GBB views the blackboard as a highly structured, n-dimensional volume, with blackboard objects occupying dimensional extent within the blackboard. Efficient insertion/retrieval of blackboard objects is achieved using a language specifying the detailed structure of the blackboard and how that structure is to be implemented. These specifications are used to generate a highly-tuned blackboard database kernel tailored to the application.

GBB contains two distinct subsystems: a high-level blackboard database compiler and a set of generic control shells. This modularization allows application-independent implementation of control strategies, as well as allowing changes to the blackboard database without affecting the control shell and vice-versa. An application implementer can select an appropriate control shell and use it to prototype the system. As experience with the application is gained, the generic shell is tailored by adding refined, application-specific knowledge to the shell's decisionmaking machinery.

Copyright © 1986, 1987, 1988
Department of Computer and Information Science
University of Massachusetts at Amherst

All rights reserved.

Support for GBB has been provided by:

- the National Science Foundation under CER Grant DCR-8500332;
- donations from Texas Instruments, Incorporated;
- the Defense Advanced Research Projects Agency under ONR University Research Inititive contract N00014-86-K-0764;
- the Defense Advanced Research Projects Agency under ONR Cooperative Distributed Problem Solving contract N00014-79-C-0439;
- the National Science Foundation under Support and Maintenance grant DCR-8318776;

Contents

1 Concepts and Terminology 1

1.1 Introduction to GBB . . . 1
1.2 The Blackboard Structure . . . 2
1.2.1 Blackboards and Spaces . . . 2
1.2.2 Space Dimensionality . . . 4
1.3 Blackboard Objects . . . 5
1.3.1 Unit Features . . . 5
1.3.2 Unit Inclusion and Inheritance . . . 7
1.3.3 Unit Type Specifiers . . . 7
1.4 Indexes . . . 8
1.5 Events . . . 8
1.6 Specifying the Blackboard Implementation . . . 9
1.7 Operations on the Blackboard . . . 10
1.7.1 Creating the Blackboard Database . . . 10
1.7.2 Unit Creation and Deletion . . . 10
1.7.3 Unit Retrieval . . . 10

2 Application Implementer Functions 12

2.1 Defining Spaces and Space Dimensionality . . . 12
2.1.1 **Define-spaces** . . . 12
2.1.2 Changing Space Dimensions . . . 13
2.2 **Define-blackboards** . . . 14
2.3 **Define-unit** . . . 14
2.3.1 **Define-unit** Options . . . 16
2.3.2 Slot Options . . . 20
2.3.3 Links . . . 23
2.3.4 Indexes . . . 25
2.3.5 Paths . . . 25
2.3.6 Unit Type Specifiers . . . 26
2.4 Events . . . 27
2.4.1 Event Handler Functions . . . 27
2.4.2 Event Inheritance Keywords . . . 28
2.5 **Define-index-structure** . . . 29
2.5.1 Overview of Index Structures . . . 29
2.5.2 Syntax of **Define-index-structure** . . . 29

2.5.3 Examples of **Define-index-structure** . . . 30
2.6 Instantiating the Blackboard Database . . . 35
2.7 Unit Creation and Deletion . . . 36
2.7.1 **Make-*unit-type*** . . . 36
2.7.2 **Delete-*unit-type*** . . . 37
2.7.3 Moving Units Among Spaces . . . 37
2.8 Path Structures . . . 37
2.9 The Linkf Macros . . . 38
2.10 **Find-units** . . . 39
2.10.1 Syntax of **Find-units** . . . 40
2.10.2 The Pattern . . . 40
2.10.3 Examples of **find-units** . . . 44
2.11 Other Retrieval Functions . . . 47
2.11.1 **Find-unit-by-name** . . . 47
2.11.2 **Find-*unit-type*** . . . 47
2.11.3 **Map-unit-types** . . . 47
2.11.4 **Map-space** . . . 48
2.12 Saving and Restoring Blackboards . . . 48
2.12.1 **Save-blackboard-database** . . . 48
2.12.2 Restore-blackboard-database . . . 49
2.12.3 Limitations of the Save/Restore Blackboard Functions . . . 49
2.13 Utilities . . . 50

3 Database Administrator Functions **53**
3.1 Mapping Units onto Spaces . . . 53
3.2 Performance Monitoring Aids . . . 54

4 GBB Control Shells **57**
4.1 The Simple-shell Control Shell . . . 57
4.1.1 Overview of the Simple-shell Control Shell . . . 57
4.1.2 Experimenting with the Simple-shell Control Shell . . . 58
4.1.3 A Trivial Example Using the Simple-shell Control Shell . . . 61
4.2 GBB1 . . . 62
4.2.1 Overview of the BB1 Control Model . . . 62
4.2.2 Overview of GBB1 . . . 64
4.2.3 User Guide to the Execution Shell . . . 65
4.2.4 User Guide to the KS Shell . . . 73

5 GBB Graphics **83**
5.1 The GBB Graphics Window . . . 83
5.2 The Mouse . . . 85
5.3 Operations on Panes . . . 86
5.3.1 Space Selection . . . 86
5.3.2 Dimension Selection . . . 86
5.3.3 Pane Options . . . 87
5.3.4 Display Operations . . . 88

5.3.5 Finding Units 88
5.3.6 Miscellaneous Operations 88
5.4 Operations on Units 88
5.5 Progam Interface 89

A A Simple GBB Database **91**

References **97**

Index **99**

Chapter 1

Concepts and Terminology

1.1 Introduction to GBB

Blackboard architectures have become a popular paradigm for developing knowledge-based systems since their introduction 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.

This manual describes the Generic Blackboard System (GBB) [2,3,4], a flexible, high-level tool for building efficient blackboard systems. This presentation of GBB is divided into four chapters. The first chapter introduces the concepts and terms used throughout the document and gives an overview of the operation of GBB. The second chapter describes the functions used to define, build and run an application. The third chapter describes functions used to tune the blackboard application to achieve better performance. The fourth chapter discusses control in GBB. In addition, an example blackboard database is provided in Appendix A.

GBB contains two distinct subsystems: a *blackboard database compiler* and a set of generic *control shells*. This modularization allows application-independent implementation of control strategies, as well as allowing changes to the blackboard database without affecting the control shell and vice-versa.

The blackboard database compiler is composed of two specification languages: one for defining blackboards and blackboard objects, and another for specifying the insertion/retrieval storage structure. This allows the construction of a blackboard-based application to be separated into two phases: implementing the application, performed by an *application implementer*, and selecting an appropriate blackboard storage/retrieval strategy, performed by a *blackboard administrator*. The application implementer defines the representational aspects of the application system in terms of blackboards and blackboard objects, and writes application-specific code. The blackboard administrator defines the specific implementation strategy to be used for blackboard storage and retrieval. Both the representation and

implementation specifications are used by the GBB database code generator to produce an efficient blackboard kernel tailored for the specific application.

The control shell defines knowledge sources, specifies the conditions for their activation, and how they are scheduled for execution. Control shells may also create other control objects, such as goals or plans. The interface between the control shell and code produced by the blackboard database compiler is a set of *blackboard events*, signals indicating the creation, modification, or deletion of blackboard objects. A set of generic control shells for GBB facilitates rapid prototyping. An application implementer can 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.

1.2 The Blackboard Structure

1.2.1 Blackboards and Spaces

An application implementer using the GBB system must specify both the structure of the blackboard and the objects that will reside on it. In GBB, the *blackboard* is a hierarchical tree structure composed of atomic blackboard pieces called *spaces*. In GBB, spaces can be viewed as the "containers" that hold blackboard objects (called *units*). Blackboards are the "containers" for holding spaces and other blackboards. This results in a blackboard/space tree (actually, it can be a forest) where the leaves are spaces and the interior and root nodes are blackboards. Blackboard objects can only be stored on spaces. The non-leaf blackboard nodes in the blackboard/space tree simply allow the developer to organize the set of spaces in the system.[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. These spaces might themselves be part of a "solution" blackboard.

The space on which a unit is to be stored can be 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*. For example, if the blackboard `BB1` had the blackboards `BB2` and `BB3` as components, and `BB2` and `BB3` had the spaces `SP1` and `SP2` as components, the two paths `(BB1 BB2 SP1)` and `(BB1 BB3 SP1)` specify different instances of the space `SP1`. Figure 1.2.1 illustrates this blackboard structure.

In addition, blackboards and spaces can be *replicated* to create multiple copies of blackboard subtrees. These copies of the blackboard structure are disambiguated by qualifying the replicated blackboard or space with a (zero-based) index. For example, if `BB1` had two

[1] Sections 2.3, 2.1.1, and 2.2 describe how to define units, spaces and blackboards.

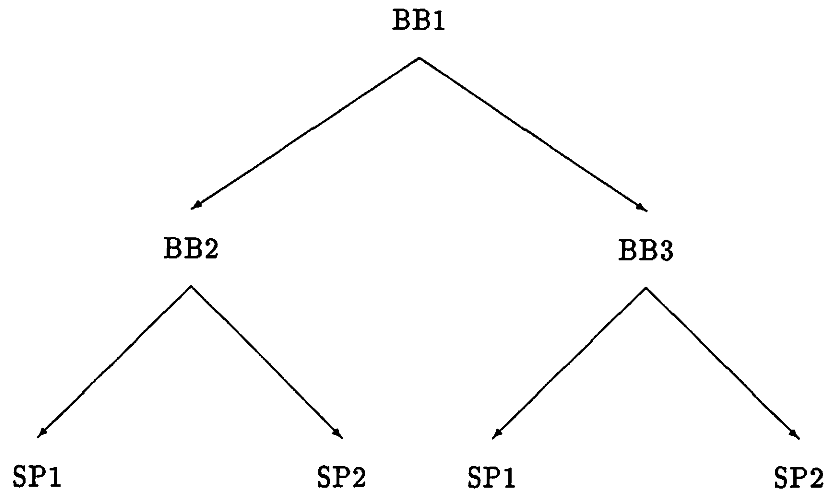


Figure 1.1: An example GBB Blackboard Structure

replications, then (BB1 0 BB2 SPACE1) and (BB1 1 BB2 SPACE1) would specify two of the four instances of sp1 created. Figure 1.2.1 illustrates this situation.[2]

GBB provides several mechanisms for specification of the blackboard/space path, depending upon the application context:

- When creating the blackboard structure, the blackboard/space path is directly specified as a list, similar to the above examples (Section 2.6).
- When making an instance of a unit, any or all of the elements of the blackboard/space path(s) can be specified implicitly through the values given to the unit's slots. This allows information about the location of the unit to be obtained through the normal slot value access mechanisms. In addition, units can be declared :DYNAMIC, allowing them to move within a space by simply modifying these slot values (Section 2.3).
- It is often useful to specify a space by its relationship to another space when using the GBB facilities for finding units. In addition, it is useful to specify subset(s) of the blackboard database when saving the blackboard structure. To meet these needs, GBB provides an abstract blackboard space/path definition facility called *path structures*. Path structures can be defined by providing either a blackboard/space path or a unit from which the path can be derived. New path structures can be created from old ones by relative specification, such as specifying the sibling of a space. By grouping path structures into a list, any subset of the blackboard can be specified (Section 2.8).

[2]More information about replication is presented in Section 2.6.

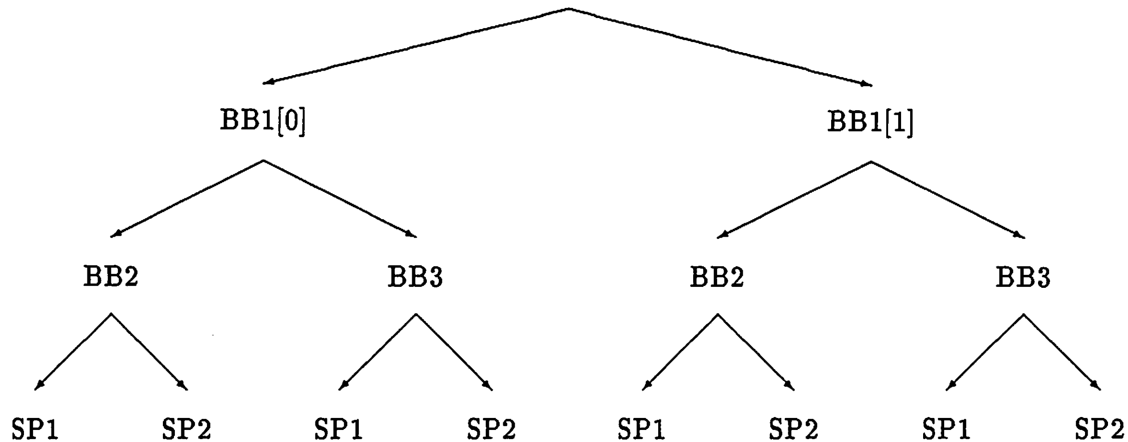


Figure 1.2: An example of a replicated GBB Blackboard Structure

1.2.2 Space Dimensionality

In GBB, a space is structured storage, organized by *dimensions*. Dimensions represent the location of objects on the space in terms natural to the application domain. For example, in speech understanding systems, a space might have the single dimension *time*. In the domain of vehicle monitoring, a space might have the dimensions *time*, *x-position*, and *y-position*.

GBB supports two types of dimensions: *ordered* and *enumerated*. Ordered dimensions use numeric ranges which support the concept of one object being "nearby" another object. In speech understanding systems, for instance, this allows the system to extend a phrase by retrieving words which begin "close" to the phrase's end. In contrast, enumerated dimensions consist of a fixed set of labeled categories. For example, in vehicle monitoring a space might have the dimension "classification," corresponding to a set of vehicle types.

The dimensionality of each space is an important part of the design of an application. The choice of what dimensions to use for a space is primarily an issue of representation. Use the dimensions that are best suited to the objects to be stored on the space. Combining an appropriate set of dimensions with a good blackboard database implementation will provide good performance. Blackboard implementation isssues are discussed in Sections 1.6 and 3.1.

1.3 Blackboard Objects

Once the blackboards and spaces have been specified, the blackboard objects are defined. In GBB, all blackboard objects are termed *units*[3], and are defined using the macro **define-unit** (Section 2.3). 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. Like `defstruct`, **define-unit** defines a new data type as well as functions to create and delete instances of the unit, and accessor functions for its slots and links. Although units are similar to `defstruct` structures, only units can be placed onto blackboard spaces.

Units are either *named* or *anonymous*. Named units have an implicitly defined slot called name, which contains a unique identifier for all the unit instances of a particular unit type. This identifier will be generated by default, or the application developer can supply a function to generate them. These identifiers are used to retrieve units by name and to determine which unit instances should be linked together when saving and restoring blackboards. The application developer can also specify that no name slot should be generated, in which case the unit type is said to be anonymous. Anonymous units cannot be saved and restored nor retrieved by name or unit-type. Because of these restrictions, we recommend using anonymous units only in extraordinary situations.

1.3.1 Unit Features

Units have five major features which are specified with **Define-unit**: *slots*, *links*, *dimensional-indexes*, *path-indexes*, and *paths*. We highlight the purpose of each feature in the following sections.

1.3.1.1 Slots

Slots are analogous to `defstruct` slots. GBB defines an accessor function for each slot and the value of the slot can be modified by using `setf`. Several slot options are available, such as `:TYPE` and `:READ-ONLY`. In addition, the user can specify *events* to be run when the slots are accessed or updated (Section 1.5).

1.3.1.2 Links

Links are special purpose slots that contain link pointers between units. Links are bidirectional; if the definition of *unit-type1* declares a link to *unit-type2*, then *unit-type2* must also declare a corresponding link to *unit-type1*. Links can enforce one-to-one, many-to-one, and many-to-many mappings between units. Instead of `setf`, special purpose macros, such as **linkf**, are provided to update links slots (see Section 2.9). These macros maintain consistent

[3]The term *unit type* will be used to emphasize when we are referring to the class of objects of a particular type. The term *unit instance* will be used to emphasize when we are referring to instances of a particular class. The term *unit* should be read as meaning either unit type or unit instance depending on the context.

link structure among units. As with slots, the user can specify events to be run when the links are accessed or updated.[4]

1.3.1.3 Dimensional Indexes

Dimensional indexes associate slot values (or portions of slot values) in the unit with dimensions of the space(s) it will be stored on. This is necessary because GBB must determine values for at least some of the space dimensions in order to place the unit on the space(s). These values for the space dimensions in the unit are called *dimensional index values*. If the value of the slot can be used directly as the dimensional index value, then this association is all that is needed. However, in many cases the dimensional index value is a component of a structured value for the slot, and GBB must be told how to extract it from the structured slot value. GBB also needs more information in the case of *composite units*.

A composite unit is a unit that has multiple elements along zero or more of its dimensions.[5] An example of a composite unit is a track of vehicle sightings. A track unit does not occupy a single large volume of the blackboard, but rather a series of points connected along the time dimension. In order to handle the above situations, GBB provides the macro **define-index-structure** for specifying how to obtain index values from structured slot values or composite units (Section 2.5).

1.3.1.4 Path Indexes

Path indexes are similar to dimensional indexes, except that they associate slots with elements of the blackboard/space path used to determine the space(s) to store the unit on. These values for the blackboard/space path are called *path index values*. If the value of the slot can be used directly as the path index value, then this association is all that is needed. However, in many cases the path index value is a component of a structured slot value, and GBB must be told how to extract it from the slot value. Again, **define-index-structure** is used for this purpose (Section 2.5).

1.3.1.5 Paths

While the path indexes tell GBB where to obtain blackboard/space path elements, the user must still specify in what order they should be combined to create the path. In addition, the user may want to store different instances of the unit on different spaces, or store a single unit instance on several spaces. The *paths* argument provides these capabilities. This argument is a list of *path clauses*, each consisting of a keyword indentifying the type of the path clause followed by arguments for that type of clause. The arguments are evaluated in the context of the slot values of the new unit instance, allowing for dynamic determination of the blackboard/space path based on slot values. For more information, see Section 2.3.5.

[4]See Section 1.5.

[5]The current implementation of GBB does not support composites composed of multiple dimensions. Only zero and one dimensional composites are allowed.

1.3.2 Unit Inclusion and Inheritance

The `:INCLUDE` option to **define-unit** provides a mechanism for building a new unit type as an extension of an old unit type. This allows the user to construct hierarchies of unit types which share common functionality. Like `defstruct`, the accessor functions constructed for a unit type will also work on any other unit type which includes this type. Unit inclusion has the following major properties:

- The slots, links, dimensional and path indexes, and paths of the included unit will be incorporated into the unit being defined. All of the properties of these objects are also inherited. (For example, events associated with an individual slot are inherited.)
- Events specified for the unit in its *name-and-options* part are also inherited. This inheritance is controllable through the use of *event-inheritance-keywords* (Sections 1.5 and 2.4.2).
- Using the accessor functions of an included unit type with an instance of the new unit will only trigger the events associated with the *included* unit, not the events associated with the new unit. For example, if `unit2` includes `unit1` which has a slot `slot1`, then `(unit1$slot1 *unit2-instance*)` and `(unit2$slot1 *unit2-instance*)` both access the same value. However, the first accessor will run events associated with `unit1`, while the second accessor will run events associated with `unit2`. These may or may not be the same set of events.

1.3.3 Unit Type Specifiers

If one unit type includes another unit type then the first is a subtype of the second. For example, suppose you have two types of knowledge source, `control-kss` and `domain-kss` and both include a basic knowledge source type, `basic-ks`. Then `control-ks` is a subtype of `basic-ks` and all instances of `control-ks` are also instances of `basic-ks`.

Several GBB functions take an argument which specifies what types of units to operate on. In most cases you want to operate on all subtypes of a particular type. However in some cases you only only want to operate on instances of that exact type and exclude instances of more specific types. *Extended unit types* provide this capability.

Normally a unit type is simply a symbol (this will be called a *simple unit type*). An *extended unit type* is either a simple unit type or a list of the form

(*simple-unit-type subtypes-keyword*)

Subtypes-keyword indicates whether or not subtypes of *simple-unit-type* are included or not. *Subtypes-keyword* may be either `:NO-SUBTYPES` or `:PLUS-SUBTYPES`. A simple unit type is interpreted as including the subtypes of that unit type. That is, the following two forms are equivalent:

```
(find-units 'basic-ks paths pattern ...)
(find-units '(basic-ks :plus-subtypes) paths pattern ...)
```

Note that there is no ambiguity between a single extended unit type and a list of two simple unit types because a unit type can not be a keyword.

The functions that accept extended unit types are **define-space**, **define-spaces**, **find-units**, **map-space** and **map-unit-types**.

1.4 Indexes

Indexes, in general, associate labels (called *indexes*) with values in a user data structure. This provides a uniform way to refer to data that is represented differently in different contexts. Indexes provide a layer of abstraction between the application builder's data structures and GBB's use of the data. The value of an index (called the *index-value*) may be the data structure itself or a component of the data structure.

To illustrate the need for indexes, suppose a space has two dimensions, x and y, and you would like to store three different types of units on that space. In one unit, the values for the two dimensions are stored in two separate slots. In the second unit, a single slot contains a two element list of the values for x and y. In the third unit, a single slot contains a complicated data structure which, in addition to other information, has the values for x and y. Indexes allow you to map the single dimension x to the appropriate data structure or data structure component in each unit. GBB can then retrieve dimension values, in a sense, "by name."

Similarly, when retrieving units from the blackboard, it would be inefficient (and inconvenient) to have to build a retrieval specification from scratch. Typically, there is an intermediate data structure on hand (e.g., data from another unit) which is the basis of a retrieval specification. Using indexes, GBB can compare the value of x in the retrieval specification with values for x in any unit, regardless of the actual representation of the data in the unit or the retrieval specification.

Index values are restricted to a small number of types called *index-element-types*. The available index-element-types are:

`:POINT` A single numeric value.

`:RANGE` A pair of numeric values: one for the minimum and one for the maximum. This type can only be specified using an index structure. (See Section 2.5)

`:LABEL` A single label. Labels are usually symbols but can be any Common Lisp datatype.

1.5 Events

Controlling problem-solving activity is 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 [5,6]. We feel that there is insufficient experience to select a single control approach for 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 exists between code produced by the database compiler of GBB and the control level. This allows different control shells to be implemented using a common blackboard database support subsystem. The separation is achieved through the use of *events*, which are the sole means of communication between code generated by the blackboard database compiler and the control shell. Events occur whenever the state of the blackboard changes, whether by unit creation or deletion, slot initialization, access, or update, or link initialization, access, update, or deletion. Events trigger *event handlers*, functions which are called when particular events occur. The arguments to the event handler functions describe the change in state about to occur.

The application implementer can specify the binding of event handlers to events in two places. Use of the event specification facilities in the *name-and-options* part of **define-unit** allows the definition of events for unit creation and deletion, as well as operations upon any of the slots or links in the unit. The events specified will also be inherited to other units if the unit is included, though this inheritance can be controlled through *event-inheritance-keywords*. Event inheritance keywords specify whether sets of events should never be inherited, should always be inherited, or whether the default inheritance condition of inheriting all events should be disabled (Section 2.4.2).

For more fine-grained control, the developer can specify events for each slot or link (Section 2.3.2).

1.6 Specifying the Blackboard Implementation

The previous sections presented the GBB machinery for defining representational aspects of the application. This section describes how the database administrator specifies particular implementations of the blackboard database. It concentrates on ordered dimensions—enumerated dimensions are typically 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 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. These strategies can each be specified using calls to **define-unit-mapping** (Section 3.1). If no implementation is specified, GBB stores units residing on each space as a list.

1.7 Operations on the Blackboard

1.7.1 Creating the Blackboard Database

Once the structure of the blackboard and the implementation strategy have been specified, the blackboard database can be created through a call to **instantiate-blackboard-database**. This function 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 would like to instantiate a copy of the blackboard database for each processor. GBB provides *replication counts* as a way of specifying the number of copies of a component blackboard or space to be created by **instantiate-blackboard-database**. When blackboards are replicated, the user must specify *replication indexes* in the blackboard/space paths to disambiguate among the duplicated spaces.

Blackboard databases can also be created incrementally through multiple calls to **instantiate-blackboard-database**, where each call specifies additional blackboards and/or spaces to be appended onto the database. If a call to **instantiate-blackboard-database** attempts to append a blackboard database that has some portion already defined through a previous call to **instantiate-blackboard-database**, an error will be signaled.

1.7.2 Unit Creation and Deletion

Constructor and destructor functions for each unit are defined by **define-unit**. When the creation function is called any creation events for that unit type are run and the newly created unit instance is placed on the appropriate spaces. The deletion function unlinks a unit from all other units (triggering any unlink events), runs any deletion events associated with units of that type, and removes the unit from any spaces it is stored on.

1.7.3 Unit Retrieval

GBB provides four functions for retrieving unit instances from the blackboard database: **find-units**, **find-unit-by-name**, **map-unit-types**, and **map-space**. Since blackboard systems spend a significant amount of time searching the database, GBB provides efficient functions for the most common modes of retrieval.

Find-units is the general function for retrieving unit instances based upon their attributes and a particular space to be searched. **Find-units** is made as efficient as possible by arguments which allow elimination of candidate units early in the retrieval process. This

is done in two ways. First, the pattern language for unit attributes is rich enough to allow the application programmer to specify complex retrieval patterns that can be analyzed and optimized by GBB. Second, 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. 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.

Find-unit-by-name is useful when the application implementer desires to retrieve a particular unit instance by specifying its name and type, regardless of the space it is stored on. This mode of retrieval is typically used when intentionally inspecting units (such as during debugging of the application.) **Find-unit-by-name** cannot be used with anonymous units since they don't have a name. Shorthand functions for **find-unit-by-name**, **find-*unit-type***, are created by **define-unit**.

Map-unit-types is a mapping function that allows the implementer to apply a function to each of the unit instances in a set of unit types.

Finally, the application implementer may wish to retrieve all the units from a space. In this case, **map-space** provides the application implementer the ability to apply a function to each unit on a particular blackboard/space path.

Chapter 2

Application Implementer Functions

This chapter describes the GBB functions needed by the application implementer. These functions are used for

- defining and creating the blackboard database,
- defining and creating blackboard objects,
- storing objects on the blackboard and retrieving objects from the blackboard,
- saving and restoring the state of the blackboard.

2.1 Defining Spaces and Space Dimensionality

GBB represents the blackboard database as a hierarchical tree (actually, it can be a forest) of primitive blackboard chunks called *spaces*. Spaces and space dimensionality play a major role in GBB. Spaces and their dimensions are specified using **define-space** and **define-spaces**.

2.1.1 Define-spaces

Spaces must be defined before all other blackboard structures by calls to **define-space** or **define-spaces**:

define-space *space* [*documentation*] &KEY *units dimensions* [*Macro*]

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

Space is a symbol specifying a space name. *Spaces* is a list of symbols specifying space names.

Documentation is an optional documentation string.

Units is a list of all unit types that may stored on *space(s)*. Each unit type in *units* is either a simiple unit type (a symbol naming a unit type) or an extended unit type (Section 2.3.6, page 26).

Dimensions is a list of specifiers defining the dimensionality of *space(s)*. Each dimension is specified by a list of the form:

(*name dimension-type dimension-values* {*option value*}*) .

GBB supports two *dimension-types*: `:ORDERED`, corresponding to ordered dimensions, and `:ENUMERATED`, corresponding to dimensions of enumerated classes.

The *dimension-values* for the ordered dimension type is a list of (*lower-bound upper-bound*), where *lower-bound* and *upper-bound* are numbers or the keyword `:INFINITY`.[1] No *options* are available for ordered dimensions.

The *dimension-values* for the enumerated dimension type is the label set for that dimension. The `:TEST` option specifies how labels will be compared. The possible values are `eq`, `eql`, and `equal`. The default is `eq`. Here is an example using both types of dimensions:

```
(define-spaces (vehicle-location vehicle-track)
   "Spaces for vehicle locations and tracks."
   :UNITS (hyp goal)
   :DIMENSIONS
     ((time :ORDERED (0 30))
      (x    :ORDERED (-1000 1000))
      (y    :ORDERED (-1000 1000))
      (classification
            :ENUMERATED (chevy porsche toyota vw-beetle unknown)
            :TEST equal)))
```

This example defines two spaces, *vehicle-location* and *vehicle-track*, which may store units of type *hyp* and *goal*; each space has four dimensions:

- *time* (from 0 to 30 inclusive);
- *x* and *y* (from -1000 to 1000 inclusive);
- *classification* (containing 5 labels).

It is an error to attempt to place a unit outside the range of an ordered dimension of a space or outside the label set of an enumerated dimension of a space. If *dimensions* is omitted or `nil`, the space has no dimensionality. Such spaces are *unstructured*.

Spaces with differing dimensionality must be declared using multiple calls to **define-space** or **define-spaces**.

2.1.2 Changing Space Dimensions

The range of values that is acceptable for a space dimension of an uninstantiated space can be changed using the function **update-space-dimension**:

update-space-dimension *space dimension new-dimension-value* [*Function*]

[1] `:INFINITE` values are not supported in the current version of GBB.

Update-space-dimension changes the range of values that is acceptable for a space dimension (e.g., change the label set for an enumerated dimension or expand the range of an ordered dimension).

Space is the name of a space that has been defined with **define-space** or **define-spaces**. *Space* must not be instantiated in the current blackboard database.

Dimension is the name of a dimension defined for *space*.

New-dimension-value is the new value for the specified dimension. It should be in the same form used with **define-space**. Thus, if *dimension* is an ordered dimension, *new-dimension-value* should be a list of (*lower-bound upper-bound*); if *dimension* is an enumerated dimension, *new-dimension-value* should be a list of labels.

2.2 Define-blackboards

Once an application's spaces have been defined, the blackboard hierarchy is defined using **define-blackboards**. Blackboards serve to group together spaces and other blackboards — units are not actually stored on blackboards.

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

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

Blackboards is a symbol naming the blackboard. *Blackboards* is a list of symbols specifying blackboard names.

Components is a list of symbols naming those space(s) and/or blackboard(s) that are the components of *blackboard* or *blackboards* .

Documentation is an optional documentation string.

Here is a simple example defining a hierarchical blackboard structure:

```
(define-blackboards (hyp-bb goal-bb)
   (vehicle-location vehicle-track)
   "Hypothesis and goal blackboards")

(define-blackboards (bb1 bb2)
   (hyp-bb goal-bb)
   "Blackboards for nodes 1 and 2")
```

This example defines a blackboard database with three levels. The top level nodes in the database are *bb1* and *bb2*, each of which have two components, *hyp-bb* and *goal-bb*. *Hyp-bb* and *goal-bb* in turn have 2 components: the spaces *vehicle-location* and *vehicle-track*. Because blackboards and spaces must be defined before they can be referenced, the definition of *hyp-bb* and *goal-bb* must precede their use in the definition of *bb1* and *bb2*.

2.3 Define-unit

In GBB, all blackboard objects are termed *units*. Units are defined using the macro **define-unit**. A unit is an aggregate data type similar to those created using the Common Lisp

`defstruct` macro. Like `defstruct`, **define-unit** defines a new data type as well as functions to create and delete instances of the unit, and accessor functions for its slots and links.

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

The *name-and-options* argument is either a non-keyword symbol naming the unit type or a list of the form: (*unit-type-name* {*option*}*). This is the same format as the *name-and-options* argument to `defstruct`. As with `defstruct`, *name* is established as a new type. The options available and their syntax are described in Section 2.3.1 (page 16).

The *slots* argument contains a list of slot-descriptions similar to `defstruct` slot-descriptions. Each slot-description is either a symbol or a list of the form:

> (*slot-name* [*default-initial-value*
> {*slot-option-name slot-option-value*}*]).

The slot options are described in Section 2.3.2 (page 20).

The *links* argument is a list of link-descriptions which define additional slots holding interunit links. See Section 2.3.3 (page 23) for a detailed description of the links. An access function is automatically defined for each link in the same way that access functions are defined for the slots. However, instead of using `setf` to update a link, use the macros `linkf`, etc. These macros, which maintain consistent link structure among units, are described in Section 2.9 (page 38).

The *dimensional-indexes* associate slots in the unit with dimensions from the space it will be stored on. There must be a space dimension corresponding to each unit index, though the space might define additional dimensions not used by the unit. The argument to *dimensional-indexes* is a list of *index-descriptions* which are described in Section 2.3.4 (page 25).

The *path-indexes* associate slots in the unit with elements of the blackboard/space path used to determine the space to store the unit on. The argument to *path-indexes* is a list of *index-descriptions* with the same format as for *dimensional-indexes*. See Section 2.3.4 (page 25) for a complete description of an index-description.

The *paths* argument specifies what space or spaces instances of this unit should be stored on. See Section 2.3.5 (page 25) for a complete description. This argument is required unless no constructor function is defined for the unit by specifying `(:CONSTRUCTOR nil)` in the **define-unit** options list (see page 16).

Here is an example of a simple unit definition:

```
(define-unit (HYP)
  :SLOTS    ((belief nil)
             (node 0)
             (location nil :TYPE time-location))
  :DIMENSIONAL-INDEXES
            ((x location)
             (y location))
  :LINKS    ((supporting-hyps (hyp supported-hyp :SINGULAR))
             (supported-hyp :SINGULAR (hyp supporting-hyps)))
  :PATH-INDEXES
            ((node-count node :TYPE :point))
  :PATHS    ((:PATH '(root 0 blackboards ,node-count
                      vehicle-track 0))))
```

In addition to the slot and link accessor functions, **define-unit** defines several other functions: **make-***unit-type*, **delete-***unit-type*, and **find-***unit-type*. The **make-***unit-type* function creates an instance of the unit (see Section 2.7.1, page 36). The **delete-***unit-type* function deletes a unit instance (see Section 2.7.2). The **find-***unit-type* function provides a shorthand for **find-unit-by-name**, saving the need to supply the unit type as a second argument (see Sections 2.11.1, page 47, and 2.11.2, page 47).

2.3.1 Define-unit Options

This section describes each of the options that can be given to **define-unit**. An option is either a keyword or a list of a keyword and arguments for that keyword. The arguments are not evaluated.

The `defstruct` options `:COPIER`, `:TYPE`, `:NAMED`, and `:INITIAL-OFFSET` are not supported for units. The options available are described below.

`:CONSTRUCTOR`

This is a limited version of the `defstruct :CONSTRUCTOR` option. By default, the name of the construction function is `make-`*name* (for example, `make-hyp`). This default name can be overridden by supplying an argument which is used as the name of the constructor function. If a unit is not intended to be instantiated by itself, but only included in other units, then specifying `nil` will not only prevent definition of the unit construction function, but eliminate other overhead associated with unit creation, storage, and deletion.

`:CONC-NAME`

This is the same as the `defstruct :CONC-NAME` option. This option allows the user to specify what prefix to use for the slot and link accessor functions. Note that, in contrast to `defstruct`, the default prefix for units is the name of the unit with a dollar sign appended (for example, `hyp$belief`).

`:PRINT-FUNCTION`

This option is similar to the `defstruct :PRINT-FUNCTION` option. It allows the user to customize the printed representation of unit instances. The argument to the `:PRINT-FUNCTION`

option should be a function of three arguments: the unit instance, the stream to print to, and the current print depth. If no print function is provided then a default print function is used. However, if this unit includes another unit and no print function is provided then the included unit's print function is used.

`:PREDICATE`

This option is the same as the `defstruct` `:PREDICATE` option.

`:INCLUDE`

This option is similar to the `defstruct` `:INCLUDE` option. It is used for building a new unit type as an extension of an existing unit type. The argument must be the name of a previously defined unit type. The slots, indexes, links, and paths of the included unit type are incorporated into the unit type being defined. In addition, the `:NAME-FUNCTION` and `:PRINT-FUNCTION` are inherited, if they were defined. The accessor functions of the included unit will also work with the unit being defined. Events from the included unit are incorporated according to the inheritance rules specified in Section 2.4.2 (page 28).

Sometimes when one unit includes another, the default initial values for the slots that are inherited from the included unit are not appropriate. To specify a different initial value for an included slot simply add a slot-description with the same slot-name to the current unit definition. For example, to define a unit type, `global-hyp`, which is identical to `hyp` except that the default value for the belief slot is 500 we could do the following.

```
(define-unit (GLOBAL-HYP (:INCLUDE hyp))
  :SLOTS ((belief 500)))
```

No other slot options may be given when an included slot definition is being modified in this way. Note that this syntax is different than that used in `defstruct`.

`:EXPORT`

This causes the symbols generated by **define-unit** to be exported from the current package. This includes the name of the unit as well as the names of all the functions (accessor functions, creation and deletion functions, predicate function, etc.). However, the names of the slots themselves are not exported.

`:NAME-FUNCTION`

This specifies the name of a function to be used to generate a string to uniquely identify each unit instance. This string is stored in a special read-only slot, **name**, that is normally defined implicitly. The function is called with one argument, the newly created unit instance, after all slots in the unit have been initialized, but before the unit is placed onto the blackboard. If no argument is supplied to `:NAME-FUNCTION` or the option is omitted, then a default function is used that simply appends a number to the end of the unit type (for example, "`GOAL-319`"). However, if this unit includes another unit and no name function is provided then the included unit's name function is used.

`:UNNAMED`

This option allows one to disable the default naming of units mentioned above. Instances of this unit type will be *anonymous* unit instances. This means that name slot will not be added to the unit instance and the functions **find-unit-by-name** and **map-unit-types** will not retrieve units of this type. In addition, anonymous unit instances will not be saved by **save-blackboard-database**. If this option is used then `:NAME-FUNCTION` may not be used. Because of these restrictions, it is recommended that anonymous units be avoided except under unusual circumstances.

`:CREATION-EVENTS`

This option specifies events that signal to the control shell when an instance of this unit is created. The option takes arguments in the form:

{*event-name* | *event-inheritance-keyword*}*

Each event function is called with one argument: the unit instance. *Event-inheritance-keywords* allow the developer control over the inheritance of events from included units. For their description, see Section 2.4.2 (page 28).

`:DELETION-EVENTS`

This option specifies events that signal to the control shell when an instance of this unit is deleted. The option takes arguments in the form:

{*event-name* | *event-inheritance-keyword*}*

Each event function is called with one argument: the unit instance. *Event-inheritance-keywords* allow the developer control over the inheritance of events from included units. For their description, see Section 2.4.2 (page 28).

`:SLOT-INITIALIZATION-EVENTS`

This option specifies events that signal to the control shell when *any* of the slots in a unit instance are initialized (explicitly given an initial value in the call to **make-***unit-type*). (To signal an event for an individual slot, see Section 2.3.2, page 20.) Each option takes arguments in the form:

{*event-name* | *event-inheritance-keyword*}*

The event functions are run after the initialization occurs and after the unit instance has been stored in the blackboard database. The arguments to the initialization event functions are: the unit instance, the name of the slot that is being initialized (this is a symbol, such as `belief`), and the newly-initialized value of the slot. *Event-inheritance-keywords* allow the developer control over the inheritance of events from included units. For their description, see Section 2.4.2 (page 28).

`:LINK-INITIALIZATION-EVENTS`

This option specifies events that signal to the control shell when *any* of the links in a unit instance are initialized (explicitly given an initial value in the call to make-*unit-type*). (To signal an event for an individual link, see Section 2.3.2, page 20.) Each option takes arguments in the form:

{*event-name* | *event-inheritance-keyword*}*

The event functions are run after the initialization occurs and after the unit instance has been stored in the blackboard database. The arguments to the initialization event functions are: the unit instance, the name of the link that is being initialized (this is a symbol, such as `supporting-hyp`), and the newly-initialized value of the link. *Event-inheritance-keywords* allow the developer control over the inheritance of events from included units. For their description, see Section 2.4.2 (page 28).

`:SLOT-ACCESS-EVENTS`

This option specifies events that signal to the control shell when *any* of the slots in a unit instance are accessed. (To signal an event for an individual slot, see Section 2.3.2, page 20.) Each option takes arguments in the form:

{*event-name* | *event-inheritance-keyword*}*

The event functions are run before the access occurs and are called with three arguments: the unit instance, the name of the slot that is being accessed (this is a symbol, such as `belief`), and the current value of the link. *Event-inheritance-keywords* allow the developer control over the inheritance of events from included units. For their description, see Section 2.4.2 (page 28).

`:LINK-ACCESS-EVENTS`

This option specifies events that signal to the control shell when *any* of the links in a unit instance are accessed. (To signal an event for an individual link, see Section 2.3.2, page 20.) Each option takes arguments in the form:

{*event-name* | *event-inheritance-keyword*}*

The event functions are run before the access occurs and are called with three arguments: the unit instance, the name of the link that is being accessed (this is a symbol, such as `supporting-hyp`), and the current value of the link. *Event-inheritance-keywords* allow the developer control over the inheritance of events from included units. For their description, see Section 2.4.2 (page 28).

`:SLOT-UPDATE-EVENTS`

This option specifies events that signal to the control shell when *any* of the (non-link) slots in a unit instance are updated. (To signal an event for an individual slot, see Section 2.3.2, page 20.) Each option takes arguments in the form:

{*event-name* | *event-inheritance-keyword*}*

The event functions are run after the update occurs and are called with four arguments: the unit instance, the name of the slot that is being updated (this is a symbol, such as `belief`), the current value of the slot, and the previous value of the slot. *Event-inheritance-keywords* allow the developer control over the inheritance of events from included units. For their description, see Section 2.4.2 (page 28).

`:LINK-UPDATE-EVENTS`

This option specifies events that signal to the control shell when *any* of the links in a unit instance are updated. (To signal an event for an individual link, see Section 2.3.2, page 20.) Each option takes arguments in the form:

{*event-name* | *event-inheritance-keyword*}*

The event functions are run after the update occurs and are called with four arguments: the unit instance, the name of the link that is being updated (this is a symbol, such as `supporting-hyps`), the newly updated set of links, and the links that were added. *Event-inheritance-keywords* allow the developer control over the inheritance of events from included units. For their description, see Section 2.4.2 (page 28).

`:UNLINK-EVENTS`

This option specifies events that signal to the control shell when *any* of the links in a unit instance are unlinked. (To signal an unlink event for an individual link, see Section 2.3.2, page 20.) `:UNLINK-EVENTS` takes arguments in the form:

{*event-name* | *event-inheritance-keyword*}*

The event functions are run after the unlinking occurs and are called with four arguments: the unit instance, the name of the link that is being modified (this is a symbol, such as `belief`), the current value of the link, and the links that were removed. *Event-inheritance-keywords* allow the developer control over the inheritance of events from included units. For their description, see Section 2.4.2 (page 28).

2.3.2 Slot Options

Each slot-description in the *slots* argument to **define-unit** may include one or more slot-options. A slot option is a keyword-value pair. The slot-option value is not evaluated. The slot-options available are:

`:TYPE`

This option is exactly the same as the `defstruct` `:TYPE` slot option. In some cases a type may need to be specified for a slot. In particular, if an index refers to a slot and that slot will contain an index structure then the slot must specify the index structure as its type.

`:READ-ONLY`

This option is exactly the same as the `defstruct` `:READ-ONLY` slot option. So, for example, `setf` will not work with the slot accessor form. This option is mutually exclusive with `:STATIC` and `:DYNAMIC`.

`:STATIC`

This option only has an effect when the slot is used as the source of a dimensional index. It specifies that the slot may be altered but that any indexes which depend on the slot will not be changed. On every slot update, GBB checks to insure that this is the case, and an error is signaled if the changed slot value changes a dimensional index value. If the slot is not the source of a dimensional index then this option simply declares that the slot is *not* read only. By default, any slots containing a dimensional index are declared `:STATIC`. This option is mutually exclusive with `:READ-ONLY` and `:DYNAMIC`.

`:DYNAMIC`

This option only has an effect when the slot is used as the source of a dimensional index. It specifies that the slot may be altered and that the dimensional indexes which depend on the slot are recomputed on every update. If the slot is not the source of a dimensional index then this option simply declares that the slot is *not* read only. If any of the indexes have changed the unit instance will be repositioned on all the spaces that it is stored on. This option is mutually exclusive with `:READ-ONLY` and `:STATIC`.

`:INITIALIZATION-EVENTS`

The `:INITIALIZATION-EVENTS` option specifies a list of events that are signaled when the slot is initialized. The argument is a list of function names. The functions are run after the initialization occurs and are called with three arguments: the unit instance, the name of the slot that is being accessed (this is a symbol like `belief`), and the newly-initialized value of the slot.

`:ACCESS-EVENTS`

The `:ACCESS-EVENTS` option specifies a list of events that are signaled when the slot is accessed. The argument is a list of function names. The functions are run before the access occurs and are called with three arguments: the unit instance, the name of the slot that is being accessed (this is a symbol like `belief`), and the current value of the slot.

An important use of access events is computing the value of a slot when it is first referenced. The access function can check for a special "uninitialized" value, compute the actual value,

and store it in the slot (triggering any update events for the slot). Since the access event functions are run *before* the slot value is actually retrieved, the accessor function will return the newly-computed value.

`:UPDATE-EVENTS`

The `:UPDATE-EVENTS` slot option specifies a list of events that are signaled when the slot is accessed. The argument is a list of function names. The functions are run after the update occurs and are called with four arguments: the unit instance, the name of the slot that is being updated (this is a symbol like `belief`), the newly-changed value of the slot, and the value of the slot before the update occurred.

`:PRIVATE`

This option provides a limited form of information hiding. If a unit declares a slot private then no accessor functions will be generated for that slot for units that include the first unit. The slot will exist in the including units, but it can only be accessed by using the accessor functions of the included unit.

For example, consider the following two unit definitions.

```
(define-unit (BASIC-KS)
  :SLOTS  ((precondition-function nil :PRIVATE t)
           (action-function nil :PRIVATE t)
           (input-levels nil)
           (output-levels nil))
  :LINKS  ((ksis (basic-ksi ks :SINGULAR) :PRIVATE t)))

(define-unit (CONTROL-KS (:INCLUDE basic-ks))
  :SLOTS ((control-type nil)))
```

Normally the `control-ks` definition would create an accessor function, `control-ks$action-function`. That accessor is not generated because `basic-ks` declares the `action-function` slot private. Instances of `control-ks` will still have an `action-function` slot, but to access or modify the slot the function `basic-ks$action-function` must be used.

`:SAVE-PRINT-FUNCTION`

The `:SAVE-PRINT-FUNCTION` slot option takes the name of a function which is called with two arguments, the slot value and an output stream. This function should print out the contents of the slot to the stream in a Common Lisp readable format (i.e. a format acceptable to `read`). This option is supplied for use when saving and restoring blackboards. It is necessary because the slot value may not print in a readable form by default, or the value may have circularities or shared structure which will not be captured by the default printed representation, or a shorthand output notation for the slot value is desired.

2.3.3 Links

Each *link-description* in **define-unit** defines additional slots that hold interunit links. Each link description is a list with the form

(*link-name* [:SINGULAR]
    {:REFLEXIVE | (*other-unit-type other-link* [:SINGULAR])}
    {*link-option-name link-option-value*}*).

The name of the link is used as the slot name. The slot-names defined by *links* are implicitly defined as slots and should not be included in **define-unit**'s *slots* argument. GBB forces all links to be bidirectional; each outgoing unit link must be defined with an accompanying inverse incoming link. To check that the links in your unit definitions are consistent use the function **check-unit-links**.

Accessor functions are defined for links just as they are for slots. For example, the form:

```
(hyp$supporting-hyps current-hyp)
```

will return the value of the `supporting-hyps` link for the unit instance `current-hyp`.

The keyword `:SINGULAR` is used to implement one-to-one, one-to-many, and many-to-one links (the default is many-to-many). Singular links are stored as atoms (thus, the value of the slot will be either a unit instance or `nil`). Links that are not singular are stored as lists. This is an example of a one-to-many link between hypothesis units:

```
(define-unit HYP
  ...
  :LINKS ((supporting-hyps (hyp supported-hyp :SINGULAR))
          (supported-hyp :SINGULAR (hyp supporting-hyps)))
  ...)
```

This implements a tree of support relationships between hypotheses units. One hypothesis may be supported by many other hypotheses but may itself only support one other hypothesis.

An example of a one-to-one link would be implementing a linked list of agenda items where each agenda item has a previous neighbor and a following neighbor.

```
(define-unit AGENDA-ITEM
  ...
  :LINKS ((next-neighbor :SINGULAR
             (agenda-item previous-neighbor :SINGULAR))
          (previous-neighbor :SINGULAR
             (agenda-item next-neighbor :SINGULAR)))
  ...)
```

The keyword `:REFLEXIVE` is simply a shorthand for:

(*link-name* (*this-unit-type link-name*)).

2.3.3.1 Link Options

Each link-description in the *links* argument to **define-unit** may include one or more link-options. A link option is a keyword-value pair. The link-option value is not evaluated. Several of the link-options are similar to the slot options described above. The link-options available are:

`:INITIALIZATION-EVENTS`

This link option is the same as the `:INITIALIZATION-EVENTS` slot option discussed above.

`:ACCESS-EVENTS`

This link option is similar to the `:ACCESS-EVENTS` slot option discussed above. It specifies a list of event functions that are run when the link is accessed. The functions are run before the access occurs and are called with three arguments: the unit instance, the name of the link that is being accessed (this is a symbol like `supporting-hyps`), and the current value of the link (this will be an atom if the link is singular, otherwise it will be a list).

`:UPDATE-EVENTS`

This link option is similar to the `:UPDATE-EVENTS` slot option. It specifies a list of event functions that are run when the link is accessed. The functions are run after the update occurs and are called with four arguments: the unit instance, the name of the link that is being updated, the new value of the link (this will be an atom if the link is singular, otherwise it will bea list), and the unit instance or instances that were added to the link. Note that it is not possible to completely determine the prior set of links because the previous link set may have included links in the added link set.

`:UNLINK-EVENTS`

The `:UNLINK-EVENTS` link option specifies a list of events functions that are run when a unit is *unlinked* (removed from a link). The functions are run after the unlinking occurs and are called with four arguments: the unit instance, the name of the link that is being changed (this is a symbol like `supporting-hyps`), the new link set, and the links that were removed.

`:PRIVATE`

This option provides a limited form of information hiding. If a unit declares a link private then no accessor functions will be generated for that link for units that include the first unit. The link will exist in the including units, but it can only be accessed by using the accessor functions of the included unit.

For example, consider the following two unit definitions.

```
(define-unit (BASIC-KS)
  :SLOTS  ((precondition-function nil :PRIVATE t)
           (action-function nil :PRIVATE t)
           (input-levels nil)
           (output-levels nil))
  :LINKS  ((ksis (basic-ksi ks :SINGULAR) :PRIVATE t)))

(define-unit (CONTROL-KS (:INCLUDE basic-ks))
  :SLOTS ((control-type nil)))
```

Normally the `control-ks` definition would create an accessor function, `control-ks$ksis`. That accessor is not generated because `basic-ks` declares the `ksis` slot private. Instances of `control-ks` will still have an `ksis` slot, but to access or modify the slot the function `basic-ks$ksi` must be used.

2.3.4 Indexes

Indexes, in general, associate labels (called *indexes*) with values in the unit. The value of an index (called the *index-value*) may be the value of a slot or it may be computed from the value of a slot. Index values must be one of the following *index-element-types*.

`:POINT` A single numeric value.

`:RANGE` A pair of numeric values: one for the minimum and one for the maximum. This type can only be specified using an index structure.

`:LABEL` A single label. Labels are usually symbols but can be any Common Lisp datatype.

Indexes are used for two purposes. *Dimensional-indexes* are used to place a unit on a space. At least one of the space dimensions must be a dimensional index. The second type of indexes are *path-indexes*. These are used to construct the blackboard/space path which determines which space to store the unit on.

The *dimensional-indexes* and *path-indexes* arguments are both a list of *index descriptions*. The format of an index description is:

(*index-name slot-name* [`:TYPE` *index-element-type*]).

The *slot-name* is the name of a slot (from the `:SLOTS` argument). This indicates which slot is used to compute the index. If the type of the slot is an index-structure then *index-name* must be an index in that index-structure and the index-value is computed from the slot as specified by the index structure. If the slot value is itself the index-value then an *index-element-type* must be specified in the index description.

2.3.5 Paths

The argument to *paths* is a list of *path clauses*. Each path clause is a list whose first element is a keyword which identifies the type of the clause and the remaining elements are arguments for that type of clause. The arguments are evaluated at the time that the unit is instantiated. During the evaluation of the arguments the *path-indexes* defined for the unit are bound to the appropriate values (which are derived from the slots of the new unit instance) so that they can be used in constructing the path or path structures.

There are two types of path clauses:

(`:PATH` *bb/space-path*)
: The argument, *bb/space-path*, must evaluate to a blackboard/space path (in other words, a list specifying a valid blackboard/space path).

(:PATH-STRUCTURE *path-structure* {*change-spec*}*)
Path-structure must evaluate to a list of path structures. If any *change-specs* are provided they are interpreted in exactly the same way they would be by **change-paths** (Section 2.8, page 38).

To illustrate the path mechanism, here is a somewhat contrived example:

```
(define-unit (HYP)
  :SLOTS         ((node  0)
                  (level 'signal))
  :PATH-INDEXES  ((node-count node  :TYPE :point)
                  (level      level :TYPE :label))
  :PATHS         ((:PATH `(blackboards ,node-count ,level 0))
                  (:PATH `(global ,level))
                  (:PATH-STRUCTURE
                     (compute-more-paths node level)
                     '(:CHANGE-SUBPATH (root) (secondary)))))
```

Now, suppose HYP-17 is created by the call (make-hyp :NODE 2 :LEVEL 'track). HYP-17 will be stored on the two spaces (blackboards 2 track 0) and (global 2) as well as the spaces computed by:

```
(change-paths (compute-more-paths 2 'track)
              '(:CHANGE-SUBPATH (root) (secondary)))
```

2.3.6 Unit Type Specifiers

If one unit type includes another unit type then the first is a subtype of the second. For example, suppose you have two types of knowledge source, control-kss and domain-kss and both include a basic knowledge source type, basic-ks. Then control-ks is a subtype of basic-ks and all instances of control-ks are also instances of basic-ks. Also, domain-ks is a subtype of basic-ks and all instances of domain-ks are also instances of basic-ks.

Several GBB functions take an argument which specifies what types of units to operate on. In most cases you want to operate on all subtypes of a particular type. However in some cases you only only want to operate on instances of that exact type and exclude instances of more specific types. *Extended unit types* provide this capability.

Normally a unit type is simply a symbol (this is called a *simple unit type*). An *extended unit type* is either a simple unit type or a list of the form

(*simple-unit-type subtypes-keyword*)

Subtypes-keyword indicates whether or not subtypes of *simple-unit-type* are included or not. *Subtypes-keyword* may be either :NO-SUBTYPES or :PLUS-SUBTYPES. A simple unit type is interpreted as including the subtypes of that unit type. That is, the following two forms are equivalent:

(find-units 'basic-ks *paths pattern* ...)
(find-units '(basic-ks :plus-subtypes) *paths pattern* ...)

Note that there is no ambiguity between a single extended unit type and a list of two simple unit types because a unit type can not be a keyword.

2.4 Events

2.4.1 Event Handler Functions

Any change to a blackboard object or to the state of the blackboard database is called an *event* in GBB. Examples of events are the creation or deletion of a unit, the access or modification of a slot, or the access or modification of a link. Events trigger *event handlers*, which are functions that are called when a particular event occurs.

Each unit type has its own set of event handlers. Event handlers are specified directly in the definition of a unit type as well as being inherited from included units. The inheritance of events handlers can be controlled by using *event-inheritance-keywords*.

Some event functions are called before the event and some are called after the event. The order is summarized below.

Before	After
Deletion	Initialization
Access	Creation
	Update
	Unlink

The argument order for all the event functions is summarized in the following table. Events can be divided into three groups: events that affect the entire unit, events that affect slots, and events that affect links.

Unit Events				
Event Type	*Arguments*			
Creation	unit			
Deletion	unit			

Slot Events				
Event Type	*Arguments*			
Initialization	unit	slot	slot-value	
Access	unit	slot	slot-value	
Update	unit	slot	new-value	old-value

Link Events				
Event Type	*Arguments*			
Initialization	unit	link	link-value	
Access	unit	link	link-value	
Update	unit	link	new-link-set	added-links
Unlink	unit	link	new-link-set	deleted-links

The arguments are as follows:

unit The unit instance that is being operated on.

slot The name of the slot. This will be a symbol

link	The name of the link. This will be a symbol.
slot-value	The current value of the slot.
link-value	The current value of the link.
new-value	The new value of the slot.
old-value	The old value of the slot.
new-link-set	The new value of this link. If the link is singular this will be a single unit, otherwise it will be a list of units.
added-links	The link or links that were added. If this event was signalled by a call to **linkf** then this will be a single unit. If the event was signalled by **linkf-list** then this will be a list of units.
deleted-links	The link or links that were removed. If this event was signalled by a call to **unlinkf** then this will be a single unit. If the event was signalled by **unlinkf-list** then this will be a list of units.

2.4.2 Event Inheritance Keywords

By default, event functions are inherited from an included unit just as slots, links, and indexes are. However, *event-inheritance-keywords* allow the developer some control over the inheritance process. All *event-inheritance-keywords* have the form:

```
{:NO-DEFAULT-INHERITANCE |
 (event-name :NEVER-INHERIT) |
 (event-name :ALWAYS-INHERIT)}*
```

Specifying `:NO-DEFAULT-INHERITANCE` disables the normal inheritance of events from the included unit. (This disabling can be overridden, however, by the included unit specifying `:ALWAYS-INHERIT`.) Specifying (*event-name* `:ALWAYS-INHERIT`) indicates that the specified event should always be inherited to any units that include this unit, even if those units specify `:NO-DEFAULT-INHERITANCE`. (If those units specified (*event-name* `:NEVER-INHERIT`), however, an error would be signaled.)

Specifying (*event-name* `:NEVER-INHERIT`) indicates that the specified event should never be inherited to this unit from its included unit. (If the included unit specified (*event-name* `:ALWAYS-INHERIT`), however, an error would be signaled.)

Both the event name and its inheritance specification are inherited. Thus if `unit2` includes `unit1`, and `unit1` specifies `(event1 :ALWAYS-INHERIT)`, then `unit2` will inherit the entire specification `(event1 :ALWAYS-INHERIT)`, not just `event1`. The sole exception is the specification `:NO-DEFAULT-INHERITANCE`, which is never inherited to included units.

2.5 Define-index-structure

2.5.1 Overview of Index Structures

An *index structure* defines a mapping from *indexes* to components of a user data structure. An index is simply a name (usually the name of a space dimension). This mechanism allows

data that is represented differently to be referenced uniformly. Index structures are used in two situations. The first is mapping from indexes to slots (and components of slots) in **define-unit**. The second is in specifying the pattern for **find-units**.

Indexes can be either *scalar* or *composite*. A scalar index is simply a single index value. A composite index is a sequence of index values. An example of the use of a composite index is a sequence of vehicle sightings over time.

Composite indexes can be further divided into *series* and *sets*. The distinction here is whether or not the order of the elements in the sequence is important. For series indexes the order is important. The value of one index is used to order one series index relative to another.[2] For set indexes the order of the elements in the sequence is *not* important.

2.5.2 Syntax of Define-index-structure

Index structures are defined by the function **define-index-structure**.

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

The *name* argument is a symbol that is defined as a new Common Lisp datatype.

Documentation is an optional documentation string.

Type is used when the index structure being defined is a simple (non-composite) index structure. The name of the index structure, *name*, is defined as a synonym for the existing datatype, *type*.

For a composite index structure, *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.[3]

Composite-index specifies the dimension connecting the composite elements (for example, *time*). *Composite-index* may also be the symbol :NONE, in which case the composite is unordered. Unordered composites are referred to as *sets*.

Element-type specifies the datatype of the elements of the composite.

The *indexes* argument must be supplied whether the index structure is simple or composite. It defines the mapping from indexes to index values. The *indexes* argument is a list of *index specifiers*. The format of an index specifier is:

```
(index-name { :POINT {selectors}* |
              :LABEL {selectors}* |
              :RANGE (:MAX {selectors}*)
                     (:MIN {selectors}*)})
```

[2]Series index structures must always be "in order" and must not contain any holes.

[3]Currently, GBB only supports composites that are lists so the *composite-type* argument must be the symbol `list`.

Index-name is the name of the index (a symbol). The second element (`:point`, `:label`, or `:range`) specifies the index element type for the index. If no *selectors* are given then the value of the index structure is itself the value of the index. Otherwise the selectors specify a path to the value of the index. The selectors determine in which component of the index structure the index value may be found. Each selector is a list element accessor (`first`, `second`, ...) or a structure slot accessor. The selector functions are applied right to left, so the path to `D` in the list `(A B (C (D)) E)` is `(first second third)`. One way to remember this format is to think of it as a lisp code fragment with the parentheses removed.

2.5.3 Examples of Define-index-structure

A few examples should make this clearer. Suppose we are tracking cars through an intersection. In the process of tracking the cars, we create hypotheses which represent guesses about the cars going by. In the examples that follow we use the following space definition:

```
(define-spaces (VEHICLE-LOCATION VEHICLE-TRACK)
   :UNITS (hyp)
   :DIMENSIONS
     ((time :ORDERED (0 100))
      (x    :ORDERED (-1000 1000))
      (y    :ORDERED (-1000 1000))
      (classification
            :ENUMERATED (chevy porsche toyota vw-beetle unknown))))
```

Each hypothesis represents information about a single event in the intersection. Among other things, each hyp stores the location of the event in (x, y) space, the time of the event, and the make of the car (the *classification*).

A Simple Example

The simplest case is when each index value is simply the value of a slot in the unit. If hyps have the slots, `time`, `x-loc`, `y-loc`, and `make`, then informing GBB of the relationship between the slots and space dimensions is achieved in the following manner (the `:PATHS`, `:PATH-INDEXES`, and `:LINKS` arguments have been omitted for brevity):

```
(define-unit HYP
  :SLOTS ((time  0)
          (x-loc 0)
          (y-loc 0)
          (make  'unknown))
  :DIMENSIONAL-INDEXES
         ((time           time  :TYPE :point)
          (x              x-loc :TYPE :point)
          (y              y-loc :TYPE :point)
          (classification make  :TYPE :label)))
```

In this example, because the index values are just the slot values, no index structure definition is needed. However, because index structures weren't used, a type declaration must be provided for each of the indexes (in the `:DIMENSIONAL-INDEXES` argument). See Section 2.3.4 (page 25).

A Simple Example—Again

This example could be coded using index structures as follows. First, define three index structures, `time-type`, `x-location-type`, and `y-location-type` which are implemented as integers. Each define a single index (*time*, *x*, and *y*, respectively). The value of the index structure (an integer) is itself the value of the index because there are no selectors provided:

```
(define-index-structure TIME-TYPE
  :TYPE    integer
  :INDEXES ((time :point)))

(define-index-structure X-LOCATION-TYPE
  :TYPE    integer
  :INDEXES ((x :point)))

(define-index-structure Y-LOCATION-TYPE
  :TYPE    integer
  :INDEXES ((y :point)))
```

Second, define an index structure for the hyp's classification. It is implemented as a symbol. The value of the index structure (a symbol) is itself the value of the index, *classification*, because there are no selectors provided:

```
(define-index-structure AUTO-MAKE
  :TYPE    symbol
  :INDEXES ((classification :label)))
```

Finally, the revised definition of hyp, using the new index structures:

```
(define-unit HYP
  :SLOTS ((time  0        :TYPE time-type)
          (x-loc 0        :TYPE x-location-type)
          (y-loc 0        :TYPE y-location-type)
          (make  'unknown :TYPE auto-make))
  :DIMENSIONAL-INDEXES
         ((time            time)
          (x               x-loc)
          (y               y-loc)
          (classification  make)))
```

In this example, the index values are just the slot values. The slots are declared to be index structures so no type declarations need to be provided for the indexes themselves (in the `:DIMENSIONAL-INDEXES` argument). GBB can obtain the index element type for each index from the index structure definition. See Section 2.3.4 (page 25). The index structure definition also gives GBB the information necessary to get the index value for each of the indexes from the corresponding slot.

A More Complicated Example

Let's alter this example slightly. Instead of having separate slots for *time*, *x*, and *y*, a single slot `time-location` is used. The value of the slot is a list of the form `(time (x y))`. First, we define an index structure, `time-location-type`, that is implemented as a list. It defines three indexes, *time*, *x*, and *y*.

```
(define-index-structure TIME-LOCATION-TYPE
  :TYPE     list
  :INDEXES  ((time :point first)
             (x    :point first second)
             (y    :point second second)))

(define-unit hyp
  :SLOTS ((time-location '(0 (0 0)) :TYPE time-location-type)
          (make          'unknown   :TYPE auto-make))
  :DIMENSIONAL-INDEXES
         ((time           time-location)
          (x              time-location)
          (y              time-location)
          (classification make)))
```

Note that hyp still has four indexes. Three of the indexes are derived from the single slot `time-location`.

A Composite Example

Now suppose that instead of a single hypothesis being associated with a single location at a single time, we want to be able to create hyps that span a number of locations. This would represent hypotheses about the progress of a car through the intersection. Instead of single values for the indexes *time*, x, and y we want to view the indexes as a list of triples, where each triple represents a single location at a particular time.

This requires the use of a composite index structure. (Without using a composite index structure, only a single, enclosing *time*, x, y cube object can be created.) To define a unit consisting of a temporally connected sequence of x and y points, we define a composite index structure, `time-location-list`, as a list of elements. Each element is a list of the form `(time (x y))`. This is an example of an instance of a `time-location-list`, representing the progress of a car from (4, 4) at time 1, to (4, 6) at time 2, to (5, 8) at time 3:

```
((1 (4 4))
 (2 (4 6))
 (3 (5 8)))
```

This is the definition of `time-location-list`:

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

Note that the *element-type* argument is `list` because each element of the composite is a list, `(time (x y))`. In general, this can be any Common Lisp datatype. The selectors for each index are applied to each element of the composite and not the composite as a whole.

The revised hyp unit definition changes the name of the slot `time-location` to `time-locations`. (This is not strictly necessary, since the choice of slot names is up to

the user. However, it makes it clearer that slot now contains a sequence of ojects rather than a single object.) The type of the slot is changed to `time-location-list` and the initial value is changed to conform to the new type specification:

```
(define-unit HYP
  :SLOTS ((time-locations '((0 (0 0))) :TYPE time-location-list)
          (make           'unknown     :TYPE auto-make))
  :DIMENSIONAL-INDEXES
         ((time           time-locations)
          (x              time-locations)
          (y              time-locations)
          (classification make)))
```

An Example of Sets

A slightly more complicated scheme involving composite types is needed if we decide to allow a hyp to have more than one *classification*, perhaps because we cannot always positively identify the make of the car. We will change the slot make to allow a *set* of values. A set is an unordered list of data elements.

First we define a new index structure, `auto-makes`, which is a list of elements. Each element is a symbol. `auto-makes` is a set because the composite index is `:NONE`:

```
(define-index-structure AUTO-MAKES
  :COMPOSITE-TYPE  list
  :COMPOSITE-INDEX :none
  :ELEMENT-TYPE    symbol
  :INDEXES         ((classification :label)))
```

Here is the revised hyp definition:

```
(define-unit HYP
  :SLOTS ((time-locations '((0 (0 0))) :TYPE time-location-list)
          (makes          '(unknown)   :TYPE auto-makes))
  :DIMENSIONAL-INDEXES
         ((time           time-locations)
          (x              time-locations)
          (y              time-locations)
          (classification makes)))
```

Index Structures and `Defstruct`-defined Structures

Up to this point, index structures with any components have been based on lists. An index structure can also be based on a `defstruct`-defined structure. Now, suppose we decided to change the implementation of `time-location-type` from the "More Complicated Example" above. Instead of representing the indexes *time*, x, and y as a list of the form `(time (x y))`, we want to keep them in `defstruct` structures. We define a defstruct, `time-location-struct`, with two slots, `time-value` and `location`. The `time-value` slot will contain an integer and the `location` slot will contain an instance of a `location-struct` structure. `Location-struct` will have two slots, `x-value` and `y-value`, which will contain the values for x and y:

```
(defstruct TIME-LOCATION-STRUCT
  time-value
  location)

(defstruct LOCATION-STRUCT
  x-value
  y-value)
```

This is the revised definition of the index structure `time-location-type`:

```
(define-index-structure TIME-LOCATION-TYPE
  :TYPE     time-location-struct
  :INDEXES  ((time :point time-location-struct-time-value)
             (x    :point location-struct-x-value
                          time-location-struct-location)
             (y    :point location-struct-y-value
                          time-location-struct-location)))
```

The only change that is required to hyp's definition is to change the initial value of the slot. `*initial-time-location*` is a dummy instance of a `time-location-struct` with zeros in all the fields:

```
(defconstant *initial-time-location*
             #S(TIME-LOCATION-STRUCT
                    :time-value  0
                    :location    #S(LOCATION-STRUCT
                                        :x-value 0
                                        :y-value 0)))

(define-unit HYP
  :SLOTS ((time-location *initial-time-location*
                         :TYPE time-location-type)
          (make          'unknown
                         :TYPE auto-make))
  :DIMENSIONAL-INDEXES
         ((time           time-location)
          (x              time-location)
          (y              time-location)
          (classification make)))
```

2.6 Instantiating the Blackboard Database

The function **instantiate-blackboard-database** creates all the internal structures needed by GBB to actually store unit instances. The blackboard must be instantiated before any units can be created.

instantiate-blackboard-database {*replication-descriptions*}* &KEY *include-all-components mode* [*Function*]

Each *replication-description* describes a blackboard hierarchy to be created. In the simplest case, it is a symbol that names the root of the tree to be instantiated.[4] 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 a *replication-description* is:

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

where *name* is the name of a blackboard or a space; *replication-count* specifies how many copies of the subtree to create; and *description* is a replication-description for one of the components of the blackboard (or space) given by *name*. *Replication-count* can be either list of (*lower-bound upper-bound*) or an integer. For example:

```
(instantiate-blackboard-database
  '(bb1 3 (goal-bb (vehicle-location 2))
          (hyp-bb (vehicle-track 3))))
```

This would create three copies of the blackboard database rooted at the blackboard `bb1`. Each copy of `bb1` would have one copy of `goal-bb` and `hyp-bb` as well as one copy of any other components defined in `bb1`'s blackboard definition. There would also be two copies of `vehicle-location` and three copies of `vehicle-track` for each copy of `bb1`.

The default inclusion of all the components defined for a blackboard can be overridden by the `:INCLUDE-ALL-COMPONENTS` keyword, which accepts three values: `t`, `nil`, or `:ask`. The default value is `t`, which forces all components to be included. The value `nil` disables this implicit inclusion, and `:ask` allows the user to be prompted for a decision when **instantiate-blackboard-database** is evaluated.

`:MODE` takes three values, `:overwrite`, `:append`, or `:ask`. (The default value is `:ask`.) This parameter indicates whether the blackboard database being defined should replace an existing database or be appended to it. The default prompts the user to decide between the two when the call is executed. When appending a new database onto another one, an error results if any of the new blackboard/space paths specified are identical to any of the previously created paths.

2.7 Unit Creation and Deletion

The functions for creating and deleting units are automatically generated by **define-unit**.

2.7.1 Make-*unit-type*

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

make-*unit-type* {*slot-keyword slot-value*}* [*Function*]

[4] 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.

Slot and link values for the newly created unit can be specified by *slot-keyword slot-value* pairs. GBB insures that any necessary 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 the unit is a named unit), and runs any creation and initialization events specified for units of that type.

To summarize the order of events when a unit instance is created:

1. The new unit instance is created. The slots are initialized with the initial value from the **make-*unit-type*** call or the default slot initialization form, as appropriate.
2. Initial link values given in the call to **make-*unit-type*** are set. This will cause any link update events for the existing unit instances to run. No events for the new unit instance are run yet.
3. BB/space paths or path structures are determined from the `:paths` argument to **define-unit** and the new instance is stored on the appropriate spaces.
4. Slot and link initialization events are run.
5. Unit creation events are run.

2.7.2 Delete-*unit-type*

A unit is deleted from all spaces and from the system using the function **delete-*unit-type***:

delete-*unit-type* *unit-instance* [*Function*]

Unit-instance must be a unit instance. It is deleted from the spaces it resides upon, unlinked from any units it was linked to, and removed from the unit hash table. Any deletion events specified for units of that type are run before the unit instance is unlinked.

2.7.3 Moving Units Among Spaces

When a unit is created, it is placed on spaces based on the `:PATHS` argument given to **define-unit**. An existing unit can be explicitly added to one or more spaces using the function **add-unit-to-space**:

add-unit-to-space *unit-instance path-structure* [*Function*]

Unit-instance is the unit instance to be added.

Path-structure is a single path structure or a list of path structures indicating the additional space(s) on which the unit instance is to be located. A correctable error is signaled if *unit-instance* is already stored on the specified space(s). Note that the specified space(s) must have been defined to accept units of the type of *unit-instance*.

A unit can be explicitly deleted from one or more spaces using the function **delete-unit-from-space**:

delete-unit-from-space *unit-instance path-structure* [*Function*]

Unit-instance is the unit instance to be removed.

Path-structure is a single path structure or a list of path structures which indicate what spaces this unit is to be deleted from. A correctable error is signaled if *unit-instance* is not stored on a specified space. Note that deleting a named unit instance from all spaces on which it resides does not delete the unit instance. It can still be accessed via links from other unit instances or by name (if it is not an anonymous unit). Use `delete-`*unit-type* to completely remove a unit instance.

2.8 Path Structures

Path Structures are high level facility for defining and modifying blackboard/space paths. They are defined using **make-paths** and modified by **change-paths**.

make-paths `&KEY` *paths unit-instances* [*Function*]

The argument to *paths* is a single or list of valid full or partial blackboard/space paths.

Unit-instances is a single or list of unit instances.

Make-paths returns a list of *path structures*, one for each space instance specified by the *path*, *unit-instance*, and/or *space-instance* arguments. Any combination of the arguments may be supplied. Note that while several arguments may specify the same path structure, only one instance of the path structure is returned. If a unit instance resides on multiple spaces, path structures for each of these spaces is returned.

change-paths *path-structures* `&REST` *change-specs* [*Function*]

Change-paths takes a single or a list of path structures, and returns a list of new path structures, one for each path passed in, modified according to the supplied *change-specs*.

A change-spec has the form:

```
{(:change-subpath old-value new-value) |
 (:change-index old-value new-value) |
 (:change-relative
    ({:up}* {path-element}*))}
```

When a change-spec specifies `:change-subpath`, the *old-value* within the path-structure is replaced by *new-value*. *Old-value* and *new-value* can be either single elements of each path structure (such as `BB1`) or lists composed of subsequences of each path structure (such as `(BB1 0 BB3 1)`). In addition, *new-value* may be `nil`, in which case the *old-value* is simply deleted from the path. It is an error for *old-value* to be `nil` or to not be present within each of the argument *path-structures*.

When a change-spec specifies `:change-index`, the element following *old-value* is replaced by *new-value*. *Old-value* may be either a single element of each path structure or a subsequence, as described above. *New-value* should be a numeric index value.

When a change-spec specifies `:change-relative`, the path is traced up the tree from the space instance as many times as `:up` is specified, then the specified *path-elements* are appended to form the new path.

2.9 The Linkf Macros

GBB generates special modification macros that maintain consistent link bidirectionality. These are similar to the Common Lisp `setf` macro but only work for GBB unit links. The link manipulation macros are:

linkf (*link-accessor this-unit-instance*) *that-unit-instance* [*Macro*]

linkf! (*link-accessor this-unit-instance*) *that-unit-instance* [*Macro*]

linkf-list (*link-accessor this-unit-instance*) *list-of-unit-instances* [*Macro*]

Link-accessor is the name of a link accessor function constructed by **define-unit**.

This-unit-instance and *that-unit-instance* are unit instances. *List-of-unit-instancess* is a list of unit instances.

Linkf and **linkf!** both add *that-unit-instance* to the set of links for *this-unit-instance* and add *this-unit-instance* to the set of links for *that-unit-instance*. Which link is affected is determined by *link-accessor*. Their behavior differs only when either end of the link is declared singular in **define-unit**, and a link already exists. In this case, bf linkf will signal an error indicating that the user must unlink the existing singular link before attempting to use **linkf** with a singular link. **Linkf!** will automatically perform the unlink operation before adding the new link to the unit. For example:

```
(linkf (hyp$supporting-hyps this-hyp) supporting-hyp)
```

adds unit `supporting-hyp` to the current set of *supporting-hyps* of unit `this-hyp` and adds unit `this-hyp` to the current set of *supported-hyps* of unit `supporting-hyp`.

Linkf-list is a shorthand for:

```
(dolist (unit-instance list-of-unit-instances)
  (linkf (link-accessor this-unit-instance) unit-instance))
```

except that **linkf-list** generates only a single update event for the entire list of added unit links where the above code would generate a separate update event for each added unit link.

unlinkf (*link-accessor this-unit-instance*) *that-unit-instance* [*Macro*]

unlinkf-list (*link-accessor this-unit-instance*) *list-of-unit-instances* [*Macro*]

unlinkf-all (*link-accessor this-unit-instance*) [*Macro*]

Link-accessor is the name of a link accessor constructed by **define-unit**.

This-unit-instance and *that-unit-instance* are unit instances. *List-of-unit-instancess* is a list of unit instances.

Unlinkf removes *that-unit-instance* from the set of links for *this-unit-instance* and vice-versa. Event functions specified in **define-unit** for unlink operations are then run. **Unlinkf-list** is a shorthand for:

```
(dolist (unit-instance list-of-unit-instances)
  (unlinkf (link-accessor this-unit-instance) unit-instance))
```

except that **unlinkf-list** generates only a single update event for the entire list of deleted unit links where the above code would generate a separate update event for each deleted unit link. **Unlinkf-all** will unlink *this-unit-instance* from all unit instances that it is linked to. Which link is affected is determined by *link-accessor*. **Unlinkf-all** generates only a single update event for the entire list of deleted unit links.

2.10 Find-units

GBB provides four functions for retrieving units from the blackboard database: **find-units**, **find-unit-by-name**, **map-unit-types**, and **map-space**. This section describes the function **find-units**. See Section 2.11 for descriptions of the remaining retrieval functions.

find-units retrieves units from a space or spaces based on a retrieval *pattern*. The pattern specifies an n-dimensional volume of the blackboard in which the desired units will be found.

2.10.1 Syntax of Find-units

The primitive function for retrieving units from spaces is **find-units**:

find-units *unit-types path-structures pattern* &KEY *filter-before filter-after* [*Function*]

The *unit-types* argument identifies which types of unit instances are to be retrieved. This may be a simple unit type, a list of simple unit types, an extended unit type, or a list of extended unit types.

The *unit-types* argument may also be `t`, in which case all instances of all unit types on the spaces are considered. Unlike **find-unit-by-name** and **map-unit-types**, **find-unit** retrieves anonymous unit instances (unit types defined with the `:UNNAMED` option).

The *path-structures* argument is a list of path structures which indicates the spaces to be searched.

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 filters are applied in order as if by and — as soon as one predicate fails that unit is eliminated from consideration.

The *pattern* argument describes the criteria that must be met by the units retrieved. It is described in the next section.

2.10.2 The Pattern

The simplest pattern is the keyword `:all` which matches all of the specified units on the specified space. A pattern can also be quite complex, represented by a list of pattern

specifiers (a pattern specifier is a keyword/value pair). 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.[5]

A non-trivial pattern consists of a *pattern-object*, which specifies an n-dimensional region of the blackboard, and other modifiers which determine how unit will be matched with the pattern.

2.10.2.1 Pattern Specifier Keywords

The pattern specifier keywords are listed below.

`:PATTERN-OBJECT`

The *pattern-object* specifies a region of the blackboard in which to look for the units. The pattern-object is either an index element, a composite structure, or a concatenation of index elements or composite structures. The index structure of the pattern need to be the same as the index structure of the unit. The value is a list of keyword/value pairs. The keywords used to specify the *pattern-object* are described below.

`:INDEX-TYPE` This is the type of the *index-object* (which is described below). In most cases it is the name of an index structure. It can also be a list of the form:

```
(:DIMENSION dimension-name
 :TYPE index-element-type)
```

For example:

```
(:DIMENSION time :TYPE :point)
```

`:INDEX-TYPE` is a required keyword.

`:INDEX-OBJECT` This is the data upon which the pattern is based. It must be consistent with the *index-type*. In most cases it is an instance of an index structure (for example, a time-location-list). `:INDEX-OBJECT` is a required keyword.

`:SELECT` This allows extraction of a subsequence of a composite structure based on the *value* of the composite index (for example, *time*). The value is a list of (*lower-bound upper-bound*) which are both *inclusive* bounds. For example, if we have a composite index structure with values for *time* = (1, 2, 3, 4, 5, 6) then `:select (2 4)` will select the values for *time* = (2, 3, 4). This only makes sense when the *index-object* is a composite object.

`:SUBSEQ` This allows extraction of a subsequence of a composite structure based on the *position* in the sequence (this is the same as selection of a subsequence of a vector). The value is a list of (*lower-bound upper-bound*) which are both *exclusive* bounds. For example, if we have a composite index structure with values for *time* = (1, 2, 3, 4, 5, 6) then `:subseq (2 4)` will select the values for *time* = (3, 4). This only makes sense when the *index-object* is a composite object.

[5] The options that are only appropriate for only composite objects are `:match`, `:mismatch`, `:skipped`, `:contiguous`, `:before-extras`, `:after-extras`, `:select`, and `:subseq`.

`:DELTA` This expands or contracts a range or expands a point into a range. The value is a list of the form

((*index-name d-value*) ...)

index-name is the name of an index and *d-value* is a signed integer. Excessive contraction of a range or a point always results in a point.

`:DISPLACE` This allows the *index-object* to be translated along one or more of its dimensions. The value is a list of the form

((*index-name t-value*) ...)

index-name is the name of an index and *t-value* is a signed integer.

`: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.

The distinctions between the different match criteria are only meaningful when one or both of the index elements from the pattern and the unit are ranges. If both the pattern's and the unit's index elements are points or labels then these match criteria are all equivalent. In this situation, `:EXACT` is stylistically preferable. The default value for this option is `:EXACT`.

`:BEFORE-EXTRAS`

The argument is a range that specifies the minimum and maximum number of composite index elements that the unit may have before the index elements mentioned in the *pattern-object*. The argument can also be `:DONT-CARE`, which means that the unit may have any number of preceeding index elements. The default for this option is `(0 0)`, meaning that no before extras are allowed.

`:AFTER-EXTRAS`

The argument is a range that specifies the minimum and maximum number of composite index elements that the unit may have after the index elements mentioned in the *pattern-object*. The argument can also be `:DONT-CARE`, which means that the unit may have any number of following index elements. The default for this option is `(0 0)`, meaning that no after extras are allowed.

`:MATCH`

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

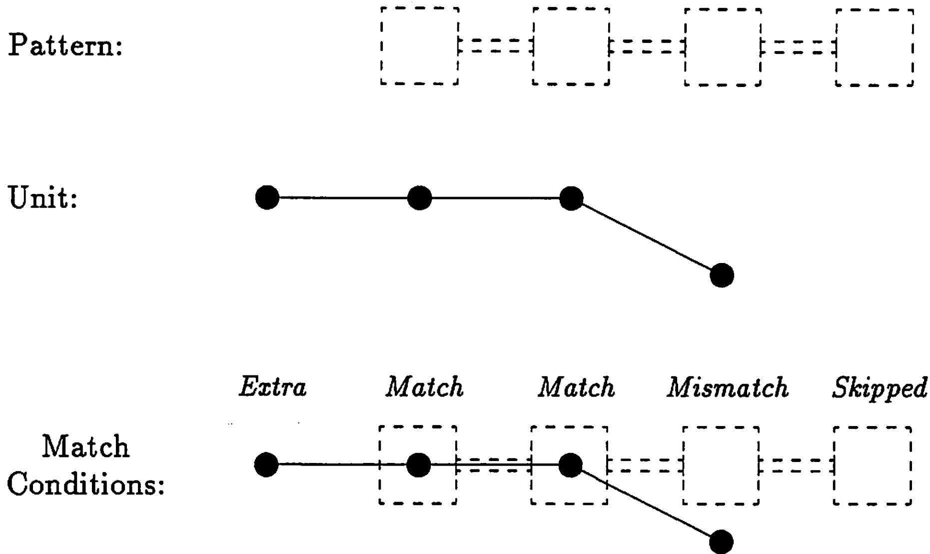


Figure 2.1: Composite Unit Matching Conditions

the length of the *pattern-object* by saying `(:ALL-BUT n)`. The default is `(:PERCENTAGE 100)`, meaning that all the elements of a composite are required to match.

`:MISMATCH`

This is an inclusive upper bound on the number of composite 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 2.1). The value for this option is expressed in the same format as `:MATCH`. By default no mismatching composite elements are allowed.

`:SKIPPED`

This is an inclusive upper bound on the number of composite index elements that may be skipped. "Skipped" means that the unit doesn't have an index element where the *pattern-object* does (see Figure 2.1). By default all the composite elements may be skipped. Note that because the `:MATCH` count is a lower bound, the default values require all the elements of a composite to match. The `:SKIPPED` option only comes into play if the `:MATCH` count is less than the length of the *pattern-object*.

`:CONTIGUOUS`

If this is true, then the composite index elements that match must be contiguous along the composite index dimension. The default for this option is true, specifying that the composite elements must be contiguous.

2.10.2.2 Pattern and Pattern Object Combination

Several patterns can be conjoined by specifying a pattern as follows.

(`:AND` *simple-pattern-1 simple-pattern-2* ...)

The *and* specifier allows several patterns to be combined into a single pattern. Each *simple-pattern-i* is a pattern as described above. The effect of this is to retrieve only those units which satisfy all the *simple-patterns.*

Several composite pattern objects can be concatenated by specifying a pattern object as follows.

(`:CONCATENATE` *simple-pattttern-object-1 simple-pattern-object-2* ...)

Each *simple-pattern-object-i* is a pattern object as described above. Each of the *simple-pattern-objects* must have the same indexes, but index structures that are concatenated together need not all be the same. GBB is able to efficiently map from one index structure representation to another. GBB decomposes all patterns/objects into sequences of simple dimensional ranges to avoid expensive type conversions.

2.10.2.3 Pattern Defaults

These are the default values for the pattern specifiers.

```
:pattern-object     unspecified
:element-match      :exact
:match              '(:percentage 100)
:mismatch           '(:count 0)
:skipped            '(:percentage 100)
:before-extras      '(0 0)
:after-extras       '(0 0)
:contiguous         t
```

Note that the `:MATCH` count is a lower bound while the `:MISMATCH` and `:SKIPPED` counts are upper bounds. Because the default values require all the elements of a composite to match the `:MISMATCH` and `:SKIPPED` counts are not used. The `:MISMATCH` and `:SKIPPED` options only comes into play if the `:MATCH` count is less than the length of the *pattern-object.*

2.10.3 Examples of find-units

For these examples we will continue using the example of tracking cars through an intersection presented in Section 2.5.3 (Page 30).

A Scalar Example

Suppose we have represented hypotheses as follows.

```
(define-unit HYP
  :SLOTS ((time  0)
          (x-loc 0)
          (y-loc 0)
          (make  'unknown))
  :DIMENSIONAL-INDEXES
         ((time           time  :TYPE :point)
          (x              x-loc :TYPE :point)
          (y              y-loc :TYPE :point)
          (classification make  :TYPE :label)))
```

We want to retrieve all the hyps between $x = 6$ and $x = 10$. To do this we need an index structure in which x is a `:RANGE`. One way to represent a range is as a single cons cell, `(6 . 10)`, where the car is the lower bound and the cdr is the upper bound. `x-range`, below, is an index structure based on such a representation.

```
(define-index-structure X-RANGE
  :TYPE    cons
  :INDEXES ((x :range (:min first) (:max rest))))
```

This is the call to **find-units**.

```
(find-units
  'hyp                   ; The type of units to find.
  (make-paths ...)       ; The path(s) to the space(s)

  ;; The Pattern:
  '(:element-match  :within
    :pattern-object (:index-type   x-range
                     :index-object (6 . 10))))
```

Typically, the range we are interested in searching is not going to be constant. As a slight modification of the above example, suppose the variables `lower-bound` and `upper-bound` contain the lower and upper bounds of the desired range. The call to **find-units** would then be:

```
(find-units
  'hyp                   ; The type of units to find.
  (make-paths ...)       ; The path(s) to the space(s)

  ;; The Pattern:
  `(:element-match  :within
    :pattern-object (:index-type   x-range
                     :index-object (,lower-bound . ,upper-bound))))
```

Note that these patterns will work regardless of how the index x is represented in the `hyp` unit. What is important is that x is a scalar index in the unit and a scalar index in the pattern.

A Series Example

Now suppose that a hypothesis represents the progress of a car through the intersection rather than a single observation. The following definitions will be used.

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

(define-unit HYP
  :SLOTS ((time-locations '((0 (0 0))) :TYPE time-location-list)
          (make           'unknown     :TYPE auto-make))
  :DIMENSIONAL-INDEXES
         ((time           time-locations)
          (x              time-locations)
          (y              time-locations)
          (classification  make)))
```

We want to retrieve all the hyps that were at the point (6, 15) at *time* = 12 and at the point (8, 15) at *time* = 13. We don't care how many observations there have been before *time* = 12 or after *time* = 13. The following call will achieve that.

```
(find-units
  'hyp                 ; The type of units to find.
  (make-paths ...)     ; The path(s) to the space(s)

  ;; The Pattern:
  '(:element-match  :exact
    :pattern-object (:index-type   time-location-list
                     :index-object ((12 (6 15))
                                    (13 (8 15))))
    :before-extras  :dont-care
    :after-extras   :dont-care))
```

A Set Example

Now, suppose we want to find all the cars that have passed through the point (28, 12) at *any* time. To do this we need a set index structure. Each element of the set is a list of the form (x y). The index structure definition and **find-units** call is shown below.

```
(define-index-structure LOCATION-SET
  :COMPOSITE-TYPE  list
  :COMPOSITE-INDEX :none
  :ELEMENT-TYPE    cons
  :INDEXES ((x :point first)
           (y :point second)))
```

```
(find-units
  'hyp                  ; The type of units to find.
  (make-paths ...)      ; The path(s) to the space(s)

  ;; The Pattern:
  '(:element-match  :exact
    :pattern-object (:index-type   location-set
                     :index-object ((28 12)))))
```

Finally, suppose we wanted to find all the cars that had passed through any one of several points. Assume the variable, `*interesting-points*` is an instance of a `location-set`.

```
(find-units
  'hyp                  ; The type of units to find.
  (make-paths ...)      ; The path(s) to the space(s)

  ;; The Pattern:
  `(:element-match  :exact
    :pattern-object (:index-type   location-set
                     :index-object ,*interesting-points*)
    :match          (:count 1)
    :mismatch       (:all-but 1)))
```

We must adjust the match and mismatch counts because we want to retrieve all units that match at least one of the elements in the set. The default match and mismatch values would require that all the elements of the set match the unit.

2.11 Other Retrieval Functions

In addition to **find-units**, GBB provides three other functions for retrieving units from the blackboard database: **find-unit-by-name**, **map-unit-types**, and **map-space**. In addition, **define-unit** defines a find-*unit-type* function for each defined unit type.

2.11.1 Find-unit-by-name

Units can be retrieved based on their name by using **find-unit-by-name.**

find-unit-by-name *name unit-type* [*Function*]

Find-unit-by-name will retrieve the unit of type *unit-type* named *name* from the blackboard database, if such a unit exists. If units of that type are *anonymous* (defined with the `:UNNAMED` option) then an error is signaled.

2.11.2 Find-*unit-type*

define-unit generates a function called **find-*unit-type*** (for example, `find-hyp`) for each unit-type.

find-*unit-type* *name* [*Function*]

This is a shorthand for **find-unit-by-name** where the second argument to **find-unit-by-name** (the unit type) is filled in with *unit-type*. That is:

```
(find-hyp "hyp-013") ≡ (find-unit-by-name "hyp-013" 'hyp)
```

No **find-***unit-type* function is created for *anonymous* unit types (defined with the `:UNNAMED` option).

2.11.3 Map-unit-types

Map-unit-types applies a function once to each named instance of a given unit type. (Note that **map-unit-types** will *never* retrieve an anonymous unit instance.)

map-unit-types *function unit-types* [*Function*]

Unit-types is a simple unit type, a list of simple unit types, an extended unit type, or a list of extended unit types. *Unit-types* may also be `t`, in which case *function* is applied to all named unit instances in the database.

Function is a one-argument function that is applied once to each named unit instance.

Map-unit-types returns `nil`.

2.11.4 Map-space

Another useful GBB mapping function is **map-space**:

map-space *function units paths* [*Function*]

Map-space provides for iteration over units that reside on the space(s) specified by *paths*, which is a path structure or a list of path structures.

Function is a function of one argument that is applied to each unit instance.

Units is a simple unit type, a list of simple unit types, an extended unit type, or a list of extended unit types. This causes *function* to be applied only to unit instances of the specified types. *Units* can also be the symbol `t` in which case *function* is applied to all unit instances on the space. **Map-space** ensures that *function* is not applied more than once to any unit instance. Unlike **find-unit-by-name** and **map-unit-types**, anonymous units are visible to **map-space**.

2.12 Saving and Restoring Blackboards

2.12.1 Save-blackboard-database

The function **save-blackboard-database** allows the developer to write out a file containing the current blackboard database (or a selected portion of it) in a portable format. The saved blackboard file can later be loaded using **restore-blackboard-database**. This save/restore system allows the saved blackboard to be restored either into a fresh environment, or else merged with another GBB blackboard database.

save-blackboard-database *file-or-stream path-structures* &KEY *comment instantiate-mode* [*Function*]

File-or-stream can be either a stream, a string indicating a file name, or a Common Lisp pathname.

Path-structures is either a single or a list of blackboard path structures (See Section 2.8).

Comment is an string to be included in *file-or-stream* as a comment.

Instantiate-mode allows the user to specify a mode other than the default of :ask to be included in the call to **instantiate-blackboard-database**.

The file created by **save-blackboard** contains calls to **define-spaces**, **define-blackboards**, **define-index-structure**, **define-unit-mapping**, **define-unit**, **instantiate-blackboard-database**, **make-*unit-type***, and **linkf**—in short, all the standard GBB functions needed to define and create a blackboard database. However, in order to allow saved blackboard databases to be merged gracefully with another blackboard, these calls will only be executed when the result of the call will not overwrite GBB object already defined. For example, the calls to **define-spaces**, **define-blackboards**, **define-index-structure**, **define-unit-mapping**, and **define-unit** will not be made if definitions for these spaces, blackboards, index-structures, unit-mappings, or units already exist. The saved unit instances will not be restored if an instance with the same name and type already exists at the time of the restore. Finally, instantiation of the blackboard database can be controlled in two ways. The *instantiate-options* keyword allows specification of the :MODE and :INCLUDE-ALL-COMPONENTS parameters to **instantiate-blackboard-database** when the blackboard is saved. In addition, the call to **instantiate-blackboard-database** can be omitted entirely at restore time by supplying an argument to **restore-blackboard**.

Relinking the saved unit instances poses special problems, since it is possible to save units which are linked to other units that are not being saved. When these unit instances are restored, they will be relinked to the other unit instances if they already exist in the new database (i.e., if there is a unit instance of the same name and type). Otherwise, GBB generates a *dummy-unit-instance* and links it to the restored unit instance. This dummy instance acts like a placeholder, in that if the real unit instance is later created, the link to the dummy unit instance will automatically be replaced by a link to the proper unit instance. If the development process includes the potential for creation of these dummy unit instances, application code should be sure to use **dummy-unit-instance-p** when following unit instance links to ensure that a real unit instance is being manipulated.

Another problem arises when the application developer supplies the :PRINT-FUNCTION option to defstruct, which redefines the way instances of the structure are printed. If the new printed representation of the structure is not readable by the lisp reader, then any instances of those structures saved by **save-blackboard** will cause an error when reloaded. The solution is to supply the :SAVE-PRINT-FUNCTION slot option in **define-unit**, giving it a function which will print the contents of the slot in a lisp readable format.

2.12.2 Restore-blackboard-database

Restore-blackboard-database loads files saved by **save-blackboard** back into the GBB environment, allowing control over running of events and instantiation.

restore-blackboard *file-name* &KEY *run-events omit-instantiation* [*Function*]

Restore-blackboard takes a string representing a file name. If :RUN-EVENTS is supplied and is non-nil, then the default disabling of events will be overridden. If :OMIT-INSTANTIATION is supplied with a non-nil value, then the call to **instantiate-blackboard-database** within the save file will not be executed. This allows the restored blackboard to be merged with an existing blackboard instantiation.

2.12.3 Limitations of the Save/Restore Blackboard Functions

Anonymous units (units defined as :UNNAMED) are not saved by **save-blackboard-database**.

Save-blackboard-database will not save top-level structures (such as symbols which point to a list of units) or retain shared structure within slot fillers.

Definitions for application-specific objects (such as the defstruct defined structures contained in a unit's slots) are not saved, so these definitions must be loaded before loading the file created by **save-blackboard-database**. In addition, since **save-blackboard-database** writes out embedded structures in the standard #S(...) format, application-specific structures should always have the default constructor function defined. (The user is free to define additional constructor functions.)

Units with pointers to other GBB units as *slot*-fillers (rather than as *links*) will not be restored correctly by the default slot printer/reader functions. In general, connections between GBB units should always be maintained through links.

2.13 Utilities

This section describes miscellaneous utility functions provided by GBB.

check-unit-links [*unit-types*] [*Function*]

The function **check-unit-links** checks that the links defined between unit-types are consistent (incoming links match outgoing links): The optional argument, *unit-types*, must be a list of unit-types whose links are to be checked for consistency (including those to other unit-types). If omitted, the links between all unit-types are checked for consistency.

reset-gbb [*Function*]

This function tries to destroy as much of the saved state of GBB as it can. This is intended to be used before loading a new GBB application into a Common Lisp image that already has a GBB application loaded.

clear-blackboard-database [*delete-p*] [*Function*]

Clear-blackboard-database removes all the unit instances from the blackboard database and from the system. This includes named units not residing on any space. If the optional argument *delete-p* is true then the database itself is also uninstantiated, and a new blackboard database must be instantiated before any more blackboard operations can take place. If *delete-p* is `nil` (the default) then the database itself is retained.

clear-space *path-structure* [*Function*]

This function clears one or more spaces of their unit instances. The unit instances themselves are not deleted. The argument, *path-structure*, a single path structure or a list of path structures, indicates which space(s) are to be cleared.

describe-blackboard-database `&optional` (*verbose* `t`) [*Function*]

This function lists the blackboards and spaces in the current blackboard database. The output is directed to `*standard-output*`. If *verbose* is true (the default) then this listing will also include the number of units stored on each space instance. If *verbose* is `nil` then a brief description of the structure of the blackboard database is printed. This format is the same as that of the replication descriptions given to **instantiate-blackboard-database**. The following example shows the output when *verbose* is true.

```
There are 2 blackboard trees in the blackboard database:
 KS                    96 Units: (96 KS)
 BLACKBOARDS
    NODE-BLACKBOARDS [0]
       ...
    NODE-BLACKBOARDS [1]
       KSI            743 Units: (743 KSI)
       GOAL
          SL              Empty
          ST              Empty
          GL           96 Units: (96 DD-GOAL)
          GT              Empty
          ...
       HYP
          SL          192 Units: (192 HYP)
          ST              Empty
          GL          264 Units: (264 HYP)
          GT              Empty
          ...
```

pp-blackboard-database [*Function*]

This function lists the blackboards and spaces in the current blackboard database in a format that is the same as that of the replication descriptions given to **instantiate-blackboard-database**. The output is directed to `*standard-output*`.

describe-space *space* [*Function*]

This function prints information about *space*, which must be symbol which names a space. The output includes what unit types may be stored on the space and the dimensions of the space. Output is directed to `*standard-output*`. An example is shown below.

```
Space PL.
    Units:        (HYP GOAL DD-GOAL GD-GOAL R-GOAL)
    Dimensions:   TIME :ORDERED 0-50
                  X    :ORDERED -5-100
                  Y    :ORDERED -5-100
                  EVENT-CLASS :ORDERED 0-100
```

path-instantiated-p *paths* [*Function*]

This predicate determines if a path in the blackboard database has been instantiated. It returns true if each blackboard/space path in the argument, *paths*, is a path to an instantiated space. *Paths* is a list of blackboard/space paths or a single blackboard/space path. Note that these are blackboard/space paths—not path structures.

unit$name *unit-instance* [*Function*]

The generic slot accessor **unit$name** returns the name of *unit-instance*. If *unit-instance* is an anonymous unit then **unit$name** returns the symbol `:anonymous`.

unit-instance-p *object* [*Function*]

This function returns true if *object* is an instance of a unit; `nil` otherwise.

unit-type-p *symbol* [*Function*]

This function returns true if *symbol* is the name of a GBB unit type (not a unit *instance*); `nil` otherwise.

dummy-unit-instance-p *object* [*Function*]

This function returns `t` if *object* is an instance of a dummy unit; `nil` otherwise. See Section 2.12.1 (page 48) for information on using **dummy-unit-instance-p**.

Chapter 3

Database Administrator Functions

This chapter describes the GBB functions used by the database administrator to define the implementation strategy for the blackboard and blackboard objects. Currently, all implementation information is defined using the macro **define-unit-mapping**.

3.1 Mapping Units onto Spaces

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

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

Units is a list of unit names.

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. Also note that any unit-mapping can be redefined at any time *before* the specified spaces are instantiated. This means that the implementation strategy can be changed *without* having to recompile unit definitions or application code. The ease of changing the unit-mapping facilitates experimental tuning of the blackboard database implementation strategy.

Indexes is the list of dimensional indexes whose implementation machinery is being defined. 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))` specifies a three-dimensional array, and `(time (x y))` would specify a vector for *time* and a two dimensional array for (x, y).

Index-structure defines the implementation machinery. It is a list of mapping-descriptions each of which define how the storage will be arranged for one dimension. The form of a mapping-description is:

{(*dimension-name* :SUBRANGES {*ordered-bucket-description*}$^+$) |
(*dimension-name* :GROUPS {*enumerated-bucket-description*}$^+$)}.

Each bucket-description defines a range of values that will fall into one bucket. An ordered-bucket-description has the form: `(`*start end* `[(:WIDTH` *width*`)])`. *Start* is an inclusive lower bound and *end* is an exclusive upper bound on the value that will fall into this bucket. If a `:WIDTH` clause is not included, then a single bucket description specifies one bucket. If a width clause is included then *width* must evenly divide the interval from *start* to *end* and there will be $(end - start)/width$ buckets, each of which will be *width* wide.

An enumerated-bucket-description has the form: `(`{*label*}`*)`. It indicates which labels from the enumerated dimension should be grouped together in one bucket. Any labels not mentioned in the enumerated-bucket-description are each allocated one bucket by default.

Here is an example of a unit-mapping where the storage is organized into four one-dimensional vectors:

```
(define-unit-mapping (unit1 unit2) (space1)
  :INDEXES (time x y type)
  :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)))
   (vehicle-type :GROUPS (toyota subaru))))
```

If the space definition specified the ranges of the dimensions to be *time* 0–30, *x* 0–100, and *y* 0–100, then there would be 14 buckets for *time*, 20 buckets for *x*, and 50 buckets for *y*. For the vehicle-type dimension, the labels *toyota* and *subaru* would share one bucket, and each of the other labels in the enumerated type *vehicle-type* would have, by default, their own bucket.

3.2 Performance Monitoring Aids

GBB will provides analysis aids to help a database administrator to match the implementation strategy to the insertion/retrieval/deletion characteristics of a particular application.

metering-enabled *[Variable]*

This variable controls whether or not GBB records information about the number of blackboard operations performed and the time spent on blackboard operations. The default value of `*metering-enabled*` is `nil`, which means that no information will be recorded. If it is `t` then metering information will be recorded.

reset-meters *[Function]*

This function sets all the GBB meters to zero. The meters will only be reset by explicit action by the user. No GBB functions reset the meters implicitly.

report-meters `&optional` *minutes seconds* *[Function]*

This function prints the statistics collected by GBB. An example is shown below. The optional arguments, *minutes* and *seconds*, are the total runtime of the application. If they are provided then the ratio of time spent doing blackboard operations to time spent doing other operations is included in the output.

```
Total Run Time       = 22:34.00
Total Non-BB Time    =  8:04.47
Total BB Time        = 14:29.53
BB Read Time         = 14:14.02
BB Write Time        =  0:15.52
BB Misc Time         =  0

BB/Non-BB Ratio      = 1.79
Read/Write Ratio     = 55.04

*INSERT-UNIT-ON-SPACE-METER*           =  0:15.52
*MOVE-UNIT-ON-SPACE-METER*             =  0
*DELETE-UNIT-FROM-SPACE-METER*         =  0
*FIND-UNITS-ALL-METER*                 =  0
*MAP-SPACE-METER*                      =  0:00.07
*MAP-UNIT-TYPES-METER*                 =  0

*INSERT-UNIT-ON-SPACE-COUNT*           =  2972
*MOVE-UNIT-ON-SPACE-COUNT*             =     0
*DELETE-UNIT-FROM-SPACE-COUNT*         =     0
*FIND-UNITS-ALL-COUNT*                 =     0
*MAP-SPACE-COUNT*                      =     1
*MAP-UNIT-TYPES-COUNT*                 =     0

Counts and Runtimes of FINDS by space:
  (NODE-BLACKBOARDS 0 HYP 0 SL 0)   =    680  ( 3%)     0:04.33  ( 1%)
  (NODE-BLACKBOARDS 0 HYP 0 GL 0)   =   1184  ( 6%)     0:07.02  ( 1%)
  (NODE-BLACKBOARDS 0 HYP 0 VL 0)   =   1204  ( 6%)     0:06.22  ( 1%)
  (NODE-BLACKBOARDS 0 HYP 0 VT 0)   =    872  ( 4%)     0:22.50  ( 3%)
  (NODE-BLACKBOARDS 0 HYP 0 PT 0)   =   5343  (25%)     2:04.63  (15%)
  (NODE-BLACKBOARDS 1 GOAL 0 SL 0)  =    192  ( 1%)     0:00.97
  (NODE-BLACKBOARDS 1 GOAL 0 GL 0)  =    456  ( 2%)     0:05.80  ( 1%)
  (NODE-BLACKBOARDS 1 GOAL 0 VL 0)  =    348  ( 2%)     0:04.47  ( 1%)
  (NODE-BLACKBOARDS 1 GOAL 0 VT 0)  =    712  ( 3%)     1:09.87  ( 8%)
  (NODE-BLACKBOARDS 1 GOAL 0 PT 0)  =   2202  (10%)     5:50.63  (41%)
  ...
  (GHYP 0 GSL 0)                    =    556  ( 3%)     0:05.90  ( 1%)

  Total                             =  21228           14:13.95
```

describe-space-instance *path-or-paths* &key (*stream* *standard-output*) (*units* t) (*indexes* t) (*show-totals* t) (*show-empty-buckets* nil) *[Function]*

This function displays information about the space instance(s) specified by the argument *path-or-paths*, which is a path structure or list of path structures. In particular, this function shows the mappings that are defined for the space instance and the distribution of units within the buckets. Output will be directed to *stream*.

The keyword arguments *units* and *indexes* restrict the display to only those mappings which include the specified units and indexes. If these options are used the count of units in each bucket will still include all units in the bucket regardless of type.

If *show-totals* is true then the output will include the total number of units stored on the space instance. Note that the space total and the sum of the bucket counts may not be equal because units may be located in more than one bucket.

If *show-empty-buckets* is `nil` then the output will be abbreviated by simply showing the count of consecutive empty buckets instead of showing each empty bucket on a separate line.

In the example below there are two mappings of units onto the `pt` space. One mapping maps `hyp` units and the second maps `goal`, `gd-goal`, `dd-goal`, and `r-goal` units. The notation, -*n*-, means that *n* consecutive empty buckets were elided.

```
(BLACKBOARDS 0 NODE-BLACKBOARDS 1 HYP 0 PT 0) is an instance of PT.

Mappings:

 Units: (HYP)
 188 total units: 188 HYP

  Indexes: (TIME), 20 Buckets.
       -3-               0
      [1-2]             18
      [2-3]             34
      [3-4]             58
      [4-5]             76
      [5-6]             76
      [6-7]             70
       ...              ...

 Units: (GOAL GD-GOAL DD-GOAL R-GOAL)
 0 total units

  Indexes: (TIME), 21 Buckets.
       -21-              0
```

Chapter 4

GBB Control Shells

This chapter documents the GBB control shells. In GBB, a control shell is a set of functions which implement a *control model* that initiates and controls problem solving activity. Currently, GBB is distributed with one production-quality control shell called *GBB1* and a very simple control shell called *simple-shell*. The GBB1 control shell is based upon the blackboard model of control provided by BB1 [6]. It allows GBB users to use the BB1 control model for running their applications. The simple-shell control shell is provided to show the basic issues involved in developing a control shell. Although it can be used to initially test an application's knowledge sources, it is sufficiently deficient in its control capabilities, that it should not be used as part of a completed blackboard applications. Both of these control shells are described in this chapter, beginning with the simple-shell.

4.1 The Simple-shell Control Shell

The simple-shell GBB control shell is provided as a starting point for writing a customize control shell. It is a gross simplification of the control machinery used in the Hearsay-II speech understanding system [1], with some major needed functionality omitted. In particular, there is no machinery for updating the rating of activated, but unexecuted, knowledge sources awaiting execution on the scheduling queue. This omission makes the shell unsuitable for more than initially testing some knowledge sources early in the prototyping of an application. This shell should never be used in a completed application.

4.1.1 Overview of the Simple-shell Control Shell

The simple-shell control shell executes *knowledge sources* (KSs) in response to unit-creation events occurring on specially defined spaces called *levels*. When a KS is defined (using **define-ks**), a subset of the levels are specified as the *input levels* of the KS. Whenever a unit-creation event occurs on a level, all KSs that have been defined as having that level as one of their input levels are considered for activation or *KS instantiation* (KSI). Every KS has associated with it a *precondition* that is evaluated for every KS being considered for

instantiation. The purpose of the precondition is to further analyze the situation (beyond the mere occurrence of a unit-creation event on one of the KS's input levels) to see if there is sufficient information on the blackboard to make instantiating the KS worthwhile. If the precondition returns a positive rating for the KS (indicating that it may be worthwhile to instantiate the KS), the KS is instantiated and placed on the *scheduling queue*.

The scheduling queue is ordered by the numeric ratings of the instantiated KSs. At the start of each *KS cycle*, the highest rated KS is removed from the scheduling queue and executed. (To provide a trace of the activity of the system, the KS instance is placed on an *executed-KS-queue* that maintains the order of KS instantiation execution.) Problem solving activity is initially primed by executing a predefined *initial KS* that creates one or more hypotheses on the blackboard, triggering other KS instantiations in the process. The problem solving continues until a KS returns the value `:STOP` or until there are no more executable KS instantiations on the scheduling queue (a condition called *quiescence*).

4.1.2 Experimenting with the Simple-shell Control Shell

The simple-shell itself consists of one files: `simple-shell`, which contains the definitions of all variables, functions and units used by the simple shell. This file must be loaded before any application files are loaded.

The first step in building an application that uses the simple-shell is to define the blackboards, the levels, the KSs, and the hypothesis units for the application. Levels (special GBB spaces that trigger KS instantiations when hypothesis units are created on them) are created with special control shell macros **define-level** and **define-levels**. Blackboards are defined using the normal GBB functions **define-blackboard** and **define-blackboards**. KSs are defined using the control shell macro **define-ks**. Finally, hypothesis are defined using the normal GBB **define-unit** macro, but with the unit `basic-hyp` as an included unit. We describe these functions are in the next section.

4.1.2.1 Define-levels

The **define-level** and **define-levels** macros define blackboard *levels* to the simple-shell control shell:

define-level *level* [*documentation*] `&KEY` *units dimensions* [*Macro*]

define-levels *levels* [*documentation*] `&KEY` *units dimensions* [*Macro*]

Level is a GBB space name where hypothesis unit creation events cause the simple-shell to attempt to instantiate and schedule KSs. *Levels* is a list of level names.

Units specifies the types of units that will be stored on the *levels*. Of course, hypothesis unit types should be among these units.

Dimensions specifies the names and types of the *levels'* dimensionality.

These macros supersede **define-space** and **define-spaces** when defining triggering levels to the simple-shell. Additional, non-triggering spaces can be defined using the normal GBB macros **define-space** and **define-spaces**.

4.1.2.2 Define-ks

The **define-ks** macro notifies the simple-shell of the existence and requirements of a KS:

define-ks *name* `&KEY` *precondition-function ks-function input-levels output-levels* [*Macro*]

Name must be a symbol naming the KS.

Ks-function specifies the (top-level) function that implements the KS. It must be acceptable to `funcall`. When a KS instantiation is executed, this function is called with two arguments: the KS unit and the KSI unit for the instantiation. The result returned by this function is checked by the simple-shell. If it is the keyword `:STOP`, the control-shell terminates the KS execution cycle. Any other return value is ignored.

Precondition-function specifies the functions that implements the precondition for the KS. It must be acceptable to `funcall`, with one exception: a KS used only as the initial KS in an application can have a `nil` *precondition-function* value. When a KS is triggered by a hypothesis creation event on one of its *input-levels*, this function is called with two arguments: the KS unit and the triggering *stimulus hypothesis*. The precondition function should return two values, an integer between `most-negative-fixnum` and `most-positive-fixnum` indicating the *rating* to be associated with an instantiation of the KS and a *response-frame*, a private data structure for passing information between the *precondition-function* and the *ks-function*. (The *response-frame* data is stored in the `ksi$response-frame` slot of each KS instance unit.) A negative or zero rating indicates that a KS instantiation should not be created by the simple-shell. In this case, the response-frame value is ignored.

The *input-levels* argument specifies the levels on which the KS *precondition-function* is to be triggered by hypothesis creation events. This value can be a list of level names or a single level name.

The *output-levels* argument specifies all levels on which the KS might create a hypothesis. This value can be a list of level names or a single level name. This value is not directly used by the simple-shell, but is available to the application as part of the KS unit.

4.1.2.3 Defining Hypothesis, KS, and KS Instantiation Units

The simple-shell control shell requires that the application use hypothesis units for its triggering blackboard objects, KS units for representing the application's knowledge sources, and KS instantiation units to maintain the scheduling queue of pending KS executions. Certain slots and events are also required of these units. The simple-shell uses GBB's event and slot inheritance capabilities to provide these to an application developer with minimal effort.

A hypothesis unit is defined using the following idiom:

```
(define-unit (HYP (:INCLUDE basic-hyp))
  [ documentation ]
  :SLOTS                  [ application-slot-definitions ]
  :DIMENSIONAL-INDEXES    [ application-dimensional-indexes ]
  :LINKS                  [ application-link-definitions ]
  :PATHS                  [ application-path-definitions ]
  )
```

Inclusion of the `basic-hyp` unit provides slots named `level` and `belief`. The `level` slot must be initialized when a hypothesis unit is created. It specifies which level in the blackboard structure is to contain the hypothesis. The `belief` slot is used only by the default hypothesis unit creation print function. It should contain a value representing the belief or confidence in the hypothesis.

Inclusion of the `basic-hyp` unit provides links named `stimulated-ksis`, `supported-hyps`, `supporting-hyps`, `creating-ksis`, and `utilizing-ksis`. The `stimulated-ksis` links point to all KS instantiations triggered by the creation of the hypothesis. The `supported-hyps` links point to all hypotheses that are superior to the hypothesis (that use the hypothesis as support). The `supporting-hyps` links point to all hypotheses that are inferior to the hypothesis (that support the hypothesis).

The `basic-hyp` unit provides the path index `level` eliminating the need to specify a `:PATH-INDEXES` clause. The `level` path index should be used in an appropriate `:PATHS` clause for positioning the hypothesis.

Finally, the `basic-hyp` unit provides a default `:NAME-FUNCTION` and `:PRINT-FUNCTION` for the `hyp` unit.

A KS unit is defined using the following idiom:

```
(define-unit (KS (:INCLUDE basic-ks))
  [ documentation ]
  :SLOTS  [ application-slot-definitions ]
  :LINKS  [ application-link-definitions ]
  )
```

Inclusion of the `basic-ks` unit provides slots named `ks-function`, `precondition-function`, `input-levels`, and `output-levels`. These slots contain the values provided when the KS is defined using `define-ks`. The `basic-ks` unit also provides a link named `ksis` that points to all instantiations of the KS. Finally, the `basic-ks` unit provides a default `:NAME-FUNCTION` and `:PRINT-FUNCTION` for the `ks` unit.

A KS instance unit is defined using the following idiom:

```
(define-unit (KSI (:INCLUDE basic-ksi))
  [ documentation ]
  :SLOTS  [ application-slot-definitions ]
  :LINKS  [ application-link-definitions ]
  )
```

Inclusion of the `basic-ksi` unit provides slots named `rating` and `response-frame`. The `rating` slot contains the rating computed by the KS's precondition function. The `response-frame` contains the response frame value returned by the precondition function. The `basic-ksi` unit also provides a link named `ks` that points to the KS unit that is instantiated by this `KSI` unit. Finally, the `basic-ksi` unit provides a default `:NAME-FUNCTION` and `:PRINT-FUNCTION` for the `ksi` unit.

4.1.2.4 Instantiate-simple-control-shell

The blackboard database is instantiated using **instantiate-simple-control-shell**:

instantiate-simple-control-shell {*description*}* `&KEY` *mode* [*Function*]

This function builds the control data structures for the simple-shell. When using the simple-shell, it should be evaluated in place of **instantiate-blackboard-database** to create the internal structures and storage for the blackboard database. Its arguments are identical to those required by **instantiate-blackboard-database.** If the application requires additional portions of the blackboard database to be instantiated after the initial instantiation, they should be done using **instantiate-blackboard-database** rather than **instantiate-simple-control-shell.**

4.1.2.5 Simple-control-shell

simple-control-shell *initial-ks-name* [*Function*]

This function invokes the control shell beginning with an instance of the KS named by *initial-ks-name.* This initial KS is invoked with `nil` as its *stimulus-hyps* argument. Execution continues until either:

1. the scheduling-queue is empty;
2. no pending KSI on the scheduling queue is rated higher than `*minimum-ks-execution-rating*`;
3. a knowledge-source returns the value `:STOP`.

4.1.3 A Trivial Example Using the Simple-shell Control Shell

A trivial example application using the simple-shell control shell is included in the distribution of GBB. This example application consists of a blackboard with 4 spaces named space1...space4, all of which store blackboard objects called hypotheses. Execution of the system starts by placing a number of hypotheses on space1, each with a `value` slot containing a random integer from 1 to 50. Whenever a hypothesis has "neighbors" (when there exists two hypotheses on the same space with values within plus or minus two of its value) a new hypothesis is created with the same value on the next higher space. Thus three consecutively valued hypotheses on space1 will result in the creation of a new hypothesis on space2, three on Space2 will create one on space3, and so forth. These actions slowly

build a "pyramid" of supporting hypotheses until one is created on `space4`, at which point execution ceases.

The file `simple-shell-example` contains functions to compile and load this simple application. The application itself is contained in the file `simple-shell-application`.

4.2 GBB1

GBB1 is a GBB control shell based on the BB1 blackboard model of control [6]. It provides language constructs for knowledge source definition and an agenda-based execution cycle facility. GBB1 does not implement all of the facilities of BB1; in particular the specialized editing and graphic display environments are not provided by GBB1.

This section begins with overviews of the BB1 control model and its implementation in GBB1. Following these sections are "user guides" to the two levels of GBB1 support currently provided: the *GBB1 Execution Shell* and the *GBB1 KS Shell*. The GBB distribution tape includes two files, `gbb1-es-example` and `gbb1-ks-example`, which provide sample GBB1 applications at each of these support levels. In addition, the distribution tape contains a GBB1 version of the traveling salesman problem in file `tsp`.

4.2.1 Overview of the BB1 Control Model

The BB1 control model defines a special class of blackboard objects called *knowledge sources* (KSs), each of which is a specialist in one aspect of the problem domain. The *BB1 execution cycle* is an algorithm which decides the order in which these KSs are applied in problem solving. KSs and the BB1 execution cycle are the two fundamental parts of the BB1 control model.

4.2.1.1 BB1 knowledge sources

BB1 has three types of knowledge sources: *learning*, *domain*, and *control*. Learning KSs manipulate the information found on the *knowledge-base* blackboard, that contains the basic facts about the problem domain. This blackboard provides the "long term memory" of the system. For organizing these facts, BB1 provides specialized linking mechanisms to create a *generic-example-instantiation* hierarchy of knowledge. In this hierarchy, general knowledge about objects and their properties are stored at the *generic* level, examples of the object in a specific problem domain are stored at the *example* level, and instances of the object as used in a potential solution to a problem are stored at the *instantiation* level.

Domain KSs describe the actions to be taken in problem solving. These actions manipulate the objects on the *domain* blackboard. This blackboard thus provides the "short term memory" of the system, where hypotheses about solutions are posted and manipulated.

Control KSs direct problem solving activity using the *control-plan* blackboard. This objects on this blackboard are used by the BB1 scheduler to decide which KS to execute next. BB1 Control KSs come in three flavors:

- *Strategies* provide abstract, general descriptions of problem solving behavior to be performed during relatively long problem-solving time intervals;
- *Foci* are a part of one or more strategies, and prescribe more specific problem solving behavior as well as a goal that describes when the behavior should conclude;
- *Heuristics* implement a particular focus by defining the functions that describe the criteria to be used in evaluating how well the actions of a KS fit in to the goal of the focus.

4.2.1.2 The BB1 Execution Cycle

The basic problem solving cycle in BB1 begins with the *execution* of a KS's *actions* that manipulate blackboard objects and generate *BB1 events*. These BB1 events may cause other KSs to become *triggered* by satisfaction of their triggering conditions. A triggered KS creates *instantiations* of itself called *knowledge source activation records*, or KSARs.[1] Each KSAR waits for its entire set of *preconditions* to be satisfied, at which point it becomes *executable*. From the set of executable KSARs, the control shell selects one whose actions appear to best fit the current problem solving strategy and executes those actions. In addition, KSARs can be removed from the triggered and executable agendas by the satisfaction of their *obviation conditions*.

Here is a more detailed description of the BB1 execution cycle, which can be divided into three phases: interpretation, agenda maintenance, and execution;

1. **Interpretation.** The interpreter executes the actions of 1 KSAR. On the first cycle the user supplies the KS whose actions are to be executed, while on the remaining cycles the scheduler chooses the KSAR. Interpretation consists of the following steps:
   (a) confirm that the KSAR's preconditions are still satisfied;
   (b) confirm that the KSAR's obviation conditions are not satisfied;
   (c) evaluate and bind the internal variables of the KSAR;
   (d) execute the actions of the KSAR;
   (e) generate BB1 events caused by the KSAR actions.
2. **Agenda Maintenance.** The agenda maintainer updates the agendas of triggered, obviated, and executable KSARs. This is done in the following way:
   (a) if the current cycle is a *precondition recheck cycle*, move all KSARs on the executable agenda with unsatisfied preconditions back to the triggered agenda;
   (b) move all KSARs on the triggered agenda with satisfied preconditions to the executable agenda;
   (c) for each KS triggered by the BB1 events of the previous cycle, generate one or more KSARs depending upon the context variables, and make each KSAR triggered or executable depending upon whether its preconditions are satisfied;
   (d) if the current cycle is an *obviation check cycle*, move all triggered and executable KSARs with satisfied obviation conditions to the obviation agenda.

[1] This is equivalent terminology to the KS instantiations discussed in conjunction with the simple-shell control shell.

3. **Scheduling.** The scheduler chooses the next KSAR to be executed from all those on the executable agenda in the following way:
   (a) rate each KSAR against each heuristic currently active in problem solving by calling the heuristic's *action rating* function;
   (b) rate each KSAR against each focus currently active in problem solving by calling the focus' *integration rating* that combines the ratings of its implementing heuristics into a single value;
   (c) determine the priority of each KSAR using the *priority* function that combines the focus ratings of a KSAR into a single value;
   (d) choose the KSAR to execute using the *recommendation* function that selects a single KSAR to execute from among the KSARs with the highest priority the single one to execute. This allows for overriding the system's recommended KSAR with a user-selected KSAR.

The execution cycle repeats until either the *termination function* returns a non-nil value (indicating the problem has been solved) or until there are no longer any KSARs on the executable agenda.

For more information on the BB1 control model, see Hayes-Roth [6].

4.2.2 Overview of GBB1

GBB1 is not a monolithic control shell, but rather a layered set of three control shells which increasingly encompass the functionality of the BB1 control model. This provides two benefits to the application implementer. First, the additional overhead incurred by some parts of the BB1 control model can be eliminated if their functionality is not needed in the application. Second, this approach facilitates alteration of the control model if an alternative is more appropriate for the application. The three layers of GBB1 are the *Execution Shell*, the *KS Shell*, and the *KB Shell*.

The Execution Shell

The GBB1 Execution Shell is the core of the system, and provides the agenda-based execution cycle of BB1 as well as facilities for defining a single, "generic" knowledge source. This layer does not provide the specialized control and learning knowledge sources and blackboard structures of BB1. Since there is no control plan, KSARs are rated by the priority function alone.

The execution shell layer is appropriate for applications requiring a control model similar to the agenda-based facilities of the Hearsay-II Speech Understanding System, for example.

The KS Shell

The KS Shell incorporates the BB1 control knowledge sources and blackboard structures into the execution cycle facilities provided by the Execution Shell. Rather than a single generic KS, distinct control and domain KSs can be defined. Control KSs construct and

manipulate a control plan which is used to rate KSARs. This shell is useful to those who desire the BB1 model of control without (or with a different) learning/knowledge representation facility.

The KB Shell

The top layer of the GBB1 system is the KB Shell. This adds the learning KS and blackboard structures of BB1 to the KS Shell, thus providing the entire BB1 model of control. This layer is not provided in the current release of GBB.

4.2.2.1 GBB1 Application Structure

Defining a GBB1 application involves the following components: GBB definitions, KS definitions, control shell parameter setup, and application execution.

- **Application definitions** Domain-specific blackboards and objects are generally defined using standard GBB functions. For this reason, one component of a GBB1 application will be a set of calls to **define-unit**, **define-blackboard**, and **define-space** for these structures. The exception to this rule is when a domain-specific object needs to generate GBB1 events in order to communicate with the control shell. In this case, the function **define-gbb1-unit** should be used instead of **define-unit**.
- **KS definitions** The definition of control shell knowledge sources is a required part of all GBB1 applications. The functions used to define them, however, vary slightly depending upon which layer of GBB1 is being used. In the execution shell, the KS definition facilities are: **define-gbb1-ks** and **define-gbb1-ksar**. In the KS shell, however, **define-gbb1-ks** is replaced by the two functions **define-gbb1-domain-ks** and **define-gbb1-control-ks**.
- **Control shell parameter setup** A required part of GBB1 application definition is the defining of the priority, recommendation, and termination functions through a call to **define-gbb1-parameters**. An optional call is to the function **define-gbb1-output**, which provides facilities for tracing the progress of problem solving.
- **Application execution** The last component of a GBB1 application is a call to **run-gbb1** which initiates problem solving.

4.2.3 User Guide to the Execution Shell

The execution shell provides the simplest control model in the GBB1 system. The execution shell follows the BB1 control model in that events cause the triggering of KSs, which generate KSARs, which become executable upon satisfaction of their preconditions. The two models diverge in their method for selecting the single KSAR to execute from those executable. The BB1 control blackboard and knowledge sources for rating executable KSARs are omitted in the execution shell. Instead, the rating process is accomplished solely by the priority function. As in the BB1 model, ties between the set of KSARs receiving the highest priority are broken by calling the recommendation function.

The execution shell is appropriate for application domains where the rating of knowledge sources is relatively straightforward, and it is not necessary to construct and maintain an explicit control plan. It is also appropriate for applications which require control knowledge in a very different form from the BB1 model. In this case, the priority function would not compute the rating for a KSAR directly, but would instead pass the KSAR on to user-defined structures and objects to determine its rating. Once determined, the rating would be returned by the priority function and problem solving would proceed normally.

The following sections document the functions used to define a GBB1 execution shell application. Following this, the GBB1 KS shell is described. In contrast to the execution shell, the KS shell does provide the control blackboards and knowledge sources of the BB1 model.

4.2.3.1 Define-gbb1-unit

The application implementer is free to define blackboard objects and structures using the standard GBB constructs. However, these objects will not generate GBB1 events, and thus their creation, deletion, and manipulation cannot be used to trigger KSs. **Define-gbb1-unit** is a version of **define-unit** which can generate GBB1 events through its one additional keyword argument: `:EVENT-CLASSES`.

define-gbb1-unit *name-and-options* [*documentation*] `&KEY` *slots links* *dimensional-indexes path-indexes paths event-classes* [*Macro*]

The syntax and semantics of *name-and-options*, *documentation*, *slots*, *links*, *dimensional-indexes*, *path-indexes*, and *paths* is the same as for **define-unit**. (See Section 2.3 for details.)

Event-classes takes the keyword `:ALL` or a list of event class names. Event class names are one of: `:creation-events`, `:deletion-events`, `:slot-update-events`, `:slot-initialization-events`, `:link-initialization-events`,`:link-update-events`, `:slot-access-events`, `:link-access-events`, or `:unlink-events`. A GBB1 event (a blackboard event used to trigger KSs) is generated for each of the event types specified. The default value of *event-classes* is `nil`, so that no GBB1 events are generated.

4.2.3.2 Define-gbb1-ks

Define-gbb1-ks creates a new knowledge source.

define-gbb1-ks *name-and-options* [*documentation*] `&KEY` *slots links* *dimensional-indexes path-indexes paths event-classes* *trigger-conditions preconditions obviation-conditions* *context-slots ksar-unit-type action-function from-bb to-bb* *ks-phases author cost reliability* [*Macro*]

The syntax and semantics of *name-and-options*, *documentation*, *slots*, *links*, *dimensional-indexes*, *path-indexes*, and *paths* is the same as for **define-unit**.[2] (See Section 2.3 for details.) The *event-classes* keyword is described in Section 4.2.3.1.

[2]However, GBB1 KSs can only include other GBB1 KSs in their definition.

Trigger-conditions is a list of forms which are evaluated for each event generated on the spaces indicated in the *from-bb* slot. When all the trigger conditions return non-`nil`, a set of KSARs are generated from this KS. The number of KSARs and their component values depends upon the argument supplied to the *context-slots* slot. If no argument to *trigger-conditions* is supplied, then the KS will always trigger. Section 4.2.3.8 documents several functions for use in trigger conditions.

Preconditions is a list of forms which are evaluated to determine whether or not a KSAR generated from this KS should be placed (or kept) on the executable agenda. Each precondition must have one of the following forms:

(`:STABLE` *precondition*)
(`:DYNAMIC` *precondition*)
precondition

The first form indicates that once the precondition returns a non-`nil` value, it does not ever need to be evaluated again. The second and third forms are equivalent, and indicate that the precondition should be rechecked every precondition-recheck-interval. If no argument to *preconditions* is supplied, then the KSAR will always become executable.

Obviation-conditions is a list of forms which are evaluated to determine whether or not a KSAR generated from this KS should be obviated. Each obviation condition must have one of the following forms:

(`:STABLE` *obviation-condition*)
(`:DYNAMIC` *obviation-condition*)
obviation-condition

The first form indicates that once the obviation condition returns a non-`nil` value, it does not ever need to be evaluated again. The second and third forms are equivalent, and indicate that the obviation condition should be rechecked every obviation condition-recheck-interval. If no argument to *obviation-conditions* is supplied, then the KSAR will never be obviated.

Context-slots should be supplied with an argument to be processed after the trigger conditions of the KS has been satisfied. The argument should be of the form *(slot-name-list slot-values-list)*, where *slot-name-list* is a list of slot names, and *slot-values-list* is a form which evaluates to a list of lists of slot values. One KSAR for each set of slot values in the slot values list will be generated. Consider the following argument to *context-slots*:

```
((a b c) `((1 2 3) (,foo ,bar ,baz)))
```

In this example, two KSARs would be generated, one with the slots `a`, `b`, and `c` given the values 1, 2, and 3 respectively, and one with the same slots given the values of `foo`, `bar`, and `baz`, respectively. Note that when the GBB unit for a KSAR is defined explicitly through a call to **define-gbb1-ksar**, the context slot names must be included in slots argument. If no argument to *context-slots* is supplied, then a single KSAR without any context slots will be instantiated upon triggering of the KS.

Ksar-unit-type is used when a "custom" KSAR is provided for this KS through a call to **define-gbb1-ksar**. The argument to *ksar-unit-type* should be the name of the KSAR. If *ksar-unit-type* is not supplied, then a KSAR for this KS will be implicitly defined.

Action-function takes a function with one argument, the KSAR unit instance which is being executed.

From-bb takes a form which is evaluated when the KSs are instantiated at the beginning of problem solving. *From-bb* should evaluate to a list of path structures indicating the blackboard spaces where the events of interest to this KS occur. Only events occurring on these spaces are tested against the KSs trigger conditions. (In most cases, the argument consists of a call to **make-paths**.) If *from-bb* is not supplied, then the KS will be tested against all events.

To-bb takes a form which is evaluated when the KSs are instantiated at the beginning of problem solving. *To-bb* should evaluate to a list of path structures indicating the blackboard spaces where the actions of this KS will take place. At this time, this information is used for documentation purposes only.

Ks-phases takes a form which is evaluated when the KSs are instantiated at the beginning of problem solving. *Ks-phases* should evaluate to `:ALL` or a list of keywords indicating the phases during which this KS is active. When the current phases(s) of problem solving do not match one of the elements of *ks-phases*, the KS is not tested against any events and cannot be triggered. *Ks-phases* should be either supplied or absent in *all* KS definitions. If *ks-phases* is supplied in some but not all KS definitions, those KSs not supplying a value will never be triggered.

Author takes a form which is evaluated to indicate the author of the KS. *Author* defaults to `"Anonymous"`.

Cost takes a form which evaluates to a value indicating the cost of performing this KS's actions.

Reliability takes a form which evaluates to a value indicating the reliability of performing this KS's actions.

4.2.3.3 Define-gbb1-ksar

Normally, a call to **define-gbb1-ks** implicitly defines a unit type for the corresponding KSAR. Sometimes, however, an application may desire additional slots or links beyond those supplied automatically. **Define-gbb1-ksar** allows the application builder to freely specify the structure of the KSAR generated from a KS.

define-gbb1-ksar *name-and-options* [*documentation*] &KEY *slots links dimensional-indexes path-indexes paths event-classes ks-unit-type* [*Macro*]

The syntax and semantics of *name-and-options*, *documentation*, *slots*, *links*, *dimensional-indexes*, *path-indexes*, and *paths* is the same as for **define-unit**.[3] (See Section 2.3 for details.) The *event-classes* keyword is described in Section 4.2.3.1.

[3] However, GBB1 KSARs can only include other GBB1 KSARs in their definition.

Ks-unit-type is the name of the KS for which this KSAR is defined.

Note that when using **define-gbb1-ksar**, the context slot names defined in the *context-slots* argument to the KS corresponding to this KSAR must be explicitly defined as slots in the call to **define-gbb1-ksar**.

4.2.3.4 Define-gbb1-parameters

Define-gbb1-parameters specifies a number of important functions and values to the control shell, and must be called at some point before **run-gbb1**.

define-gbb1-parameters `&KEY` *max-execution-cycles priority-fn priority-fn-stability recommendation-fn termination-fn precondition-recheck-interval obviation-recheck-interval* [*Function*]

Max-execution-cycles should be supplied with a number indicating the maximum number of execution cycles expected during problem solving.

Priority-fn should be supplied a function of one argument. This function is called with every executable KSAR and should return a numeric rating indicating its priority.

Priority-fn-stability should be supplied `:STABLE` or `:DYNAMIC`. Stable priority functions do not change their rating of a KSAR, so they do not need to be rerated. Dynamic priority functions must rerated each KSAR every time.

Recommendation-fn should be supplied a function of one argument. This argument is a list of KSARs with the highest priority, and the recommendation function should return the one to execute. (This function can query the user if emulation of the BB1 execution mode is desired.)

Termination-fn should be supplied a function with no arguments. It is called once each execution cycle, after updating the agendas and before scheduling of a KSAR for execution. If *termination-fn* returns `nil`, execution of GBB1 ends immediately.

Precondition-recheck-interval is supplied a number indicating how many execution cycles to skip between rechecking preconditions. (Note that if a precondition is `:STABLE`, it will never be rechecked regardless of the value of *precondition-recheck-interval.*)

Obviation-recheck-interval is supplied a number indicating how many execution cycles to skip between rechecking obviation conditions. (Note that if an obviation condition is `:STABLE`, it will never be rechecked regardless of the value of *obviation condition-recheck-interval.*)

4.2.3.5 Define-gbb1-output

define-gbb1-output `&KEY` *trace-fn trace-print-points print-unit-width output-stream* [*Function*]

Define-gbb1-output provides information about what kind of trace information should be output during execution of GBB1.

Trace-fn can be supplied with a function of no arguments. This function is called once per execution cycle, right after interpretation of a KSAR. It is also called once after the execution cycle terminates.

Trace-print-points can be supplied with a list of keywords indicating the information about the execution cycle to print out. The current legal keywords are:

`:BEFORE-INTERPRETATION` Prints the current state of the agendas and events just before interpretation of a KSAR (i.e. just after a KSAR has been scheduled for execution).

`:BEFORE-AGENDA-UPDATE` Prints the current state of the agendas and events just before updating the agendas (i.e. just after interpretation of a KSAR).

`:BEFORE-SCHEDULING-KSAR` Prints the current state of the agendas and events just before scheduling a KSAR for execution (i.e. just after the agendas have been updated).

`:FINAL-STATE` Prints the current state of the agendas and events just before execution of the GBB1 system ends.

`:PRIORITY` Prints out the priority received by each KSAR.

`:FOCUS-RATINGS` Prints out each of the focus ratings received by the KSARs. (Note: this keyword is not valid in the execution shell.)

`:HEURISTIC-RATINGS` Prints out each of the heuristic ratings received by the KSARs. (Note: this keyword is not valid in the execution shell.)

`:ALL` This keyword turns on all the trace print points.

Trace-print-points defaults to `'(:BEFORE-AGENDA-UPDATE :FINAL-STATE)`.

Print-unit-width specifies the width required for printing KSARs. It defaults to 25.

Output-stream specifies where trace output should go. It defaults to `*standard-output*`.

4.2.3.6 Run-gbb1

run-gbb1 `&KEY` *initial-ks* *[Function]*

Run-gbb1 initiates execution of GBB1. It initializes the GBB1 agendas and events blackboards before executing the actions of *initial-ks*. It is the application implementer's responsibility to assure that user-defined blackboards and objects are reinitialized between calls to **run-gbb1**. To completely reinitialize the system in order to load a different application, **reset-gbb1** must be called.

Initial-ks is the name of the KS whose actions are to be interpreted.

4.2.3.7 Reset-gbb1

reset-gbb1 *[Function]*

Reset-gbb1 is used to completely reinitialize the state of GBB1. In contrast to **run-gbb1**, which reinitializes the system so that the loaded application can be run again, **reset-gbb1** clears all knowledge source definitions as well. **Reset-gbb1** is used when it is desired to load a different application into the GBB1 system.

4.2.3.8 Utility Functions for Trigger Conditions

The following functions have been found useful as components of trigger conditions.

trigger-unit *[Function]*

Trigger-unit returns the unit-instance which caused the event currently being tested against the KS.

trigger-event-level-p *path-structure* *[Function]*

Returns t if *path-structure* corresponds to the space on which the trigger event originated, nil otherwise.

trigger-event-class-p *event-class* *[Function]*

Returns t if *event-class* equals event class of the trigger event, nil otherwise.

trigger-event-slot-p *slot-or-link* [*new-value*] *[Function]*

Returns t if the trigger event involved a slot (or link) and its name equals *slot-or-link*, and if *new-value* was supplied, if the new value of the slot (or link) equals *value*. Returns nil otherwise.

4.2.3.9 KS and KSAR Phase Manipulation Functions

Phases of problem solving was introduced in the Boeing Blackboard System [7] as a way to increase the efficiency of the blackboard system. A common way to structure blackboard applications is into a set of phases, where only a subset of all the KSs defined are applicable to any one phase. In the BB1 control model, this effect can be achieved by including a trigger condition in each KS which tests to see some (application maintained) variable denoting the current phase of processing has an appropriate value. The problem with this method is that all of the KSs in the application must be retrieved from the blackboard and tested.

The GBB1 system provides phases of problem solving through the KS-phases keyword in the KS definition functions and the set of phase manipulation functions described below. The advantage of using this facility over a functionally equivalent use of trigger conditions is efficiency: only the KSs defined to be applicable to the current phase of problem solving are retrieved from the blackboard for testing of their trigger conditions.

The set of phases is implicitly defined as the union of all the KS phase names referenced in the *ks-phases* slot of **define-gbb1-ks**. The current phase(s) of problem solving is set by **set-ks-phase**, and can be determined by calling **current-ks-phase**.

This facility is strictly optional: by omitting the KS-phases keyword in KS definitions, all KSs will be retrieved and tested throughout problem solving.

set-ks-phase *phase* *[Function]*

Sets the current KS phase(s) to *phase*.

current-ks-phase *[Function]*

Returns current KS phase(s).

4.2.3.10 Global Variables

The following global variables contain useful information about the current state of GBB1 during problem solving. They are maintained by GBB1 and should not be altered by the user.

execution-cycle [*Variable*]

The execution cycle number.

trigger-event [*Variable*]

`*Trigger-event*`

evaluates to the current GBB1 event, and can be referenced within trigger conditions.

this-ks [*Variable*]

`*This-ks*`

evaluates to the current KS being tested, and can be referenced within trigger conditions.

this-ksar [*Variable*]

`*This-ksar*`

evaluates to the current KSAR being tested, and can be referenced within preconditions, obviation conditions, and during context slot value evaluation.

4.2.3.11 A Simple Example: Building a Hypothesis Pyramid

The file `gbb1-es-example` contains a simple example system illustrating the style of system definition and some of the facilities available in the execution shell. The system consists of a blackboard with 4 spaces named `space1`...`space4`, all of which store blackboard objects termed "hypotheses." Execution of the system starts by placing a number of hypotheses on `space1`, each with a `value` slot containing a random integer from 1 to 50. Whenever a hypothesis has "neighbors" (when there exists two hypotheses on the same space with values one greater and one less than its value) a new hypothesis is created with the same value on the next higher space. Thus three consecutively valued hypotheses on `space1` will result in the creation of a new hypothesis on `space2`, three on `space2` will create one on `space3`, and so forth. These actions slowly build a "pyramid" of supporting hypotheses until one is created on `space4`, at which point execution ceases.

Following the structure given in 4.2.2.1, the example system can be divided into the following components:

- **Application definitions** The blackboard, spaces, and the **hyp** object in this example are the domain-specific structures. The blackboard and spaces are defined through calls to **define-blackboard** and **define-space**. The **hyp** object, since it needs to generate GBB1 events whenever an instance of it is created, is defined using **define-gbb1-unit** rather than **define-unit**.

- **KS definitions** Two knowledge sources are defined in this application: **initial-ks** which puts the initial set of objects on space1, and **pyr-builder**, a KS which is triggered by the creation of a **hyp** object, and whose action is to add a new **hyp** to the next higher space when the hyp that triggered it has neighbors on either side.
- **Control shell parameter setup** Both **define-gbb1-output** and **define-gbb1-parameters** are used in this example system. **Define-gbb1-parameters** is used to define the termination, recommendation, and priority functions, and **define-gbb1-output** is used to specify trace information.
- **Application execution** This system defines the function **build-pyramid** which is called to initiate execution. This function instantiates the domain blackboard and then calls **run-gbb1** to begin problem solving.

4.2.4 User Guide to the KS Shell

The GBB1 KS Shell extends the functionality of the execution shell by introducing control KSs and a control plan blackboard. These control structures and objects augment the priority function in deciding which KSAR to execute during each execution cycle.

The *function* of the control KSs and blackboard is relatively straightforward in the BB1 control model. They provide a blackboard-based method for rating the executable KSARs which can dynamically change at run-time in response to the state of problem solving. This method defines three new types of KSs: strategy KSs, focus KSs, and heuristic KSs, collectively known as the control KSs. The primary function of these KSs is to create and manipulate strategy, focus or heuristic objects on the control blackboard. For more information on general aspects of control knowledge in the BB1 model, see [8].

The *implementation* of the control KSs and blackboard in GBB1 involves addressing the following issues:

- *Representation of control plan objects.* Each type of control plan object has a characteristic structure which is known to and manipulated by the internal rating facilities of the KS shell. To ensure that control plan objects have the correct structure, they are implicitly defined during the call to **define-gbb1-control-ks**. The instances of these objects are normally made during execution of the actions of its corresponding KS. For more information about the structure of control plan objects, see Section 4.2.4.5.
- *Implementation of the rating process.* Efficient implementation of the rating process involves determining which rating functions need to be applied or re-applied to the KSAR as the control plan evolves. The GBB1 implementation provides a stability attribute for each of the rating functions, as well as the two functions **map-focus-ratings** and **map-heuristic-ratings** to accomplish efficient KSAR rating. For more information on the rating process, see Section 4.2.4.6.
- *Initial setup and maintenance of the control plan.* Control KSs actually serve two purposes: they must encode the knowledge used to rate the KSARs, and they must encode knowledge about how to set up and modify the interconnected network of control plan objects collectively known as the control plan. In BB1, this latter task is the responsibility of a special set of KSs called the *prescription* KSs. General

information about the nature and function of the prescription KSs can be found in [8]. Information about the implementation of prescription KSs supplied with the GBB1 KS shell can be found in Section 4.2.4.9.

The following sections provide a complete list of the user interface to the KS shell. To minimize duplication, however, the reader is referred to earlier descriptions whenever possible.

4.2.4.1 Run-gbb1

See Section 4.2.3.6.

4.2.4.2 Reset-gbb1

See Section 4.2.3.7.

4.2.4.3 Define-gbb1-unit

See Section 4.2.3.1.

4.2.4.4 Define-gbb1-domain-ks

Define-gbb1-domain-ks defines domain knowledge sources. Its arguments are identical to **define-gbb1-ks**. See Section 4.2.3.2 for details.

4.2.4.5 Define-gbb1-control-ks

Define-gbb1-control-ks defines control knowledge sources. Control KSs are used to create and manipulate the objects on the control plan blackboard, which in turn are used to rate KSARs. The *control-type* argument is used to indicate the type of control knowledge source being defined, and can take one of four values: `:STRATEGY`, `:FOCUS`, `:HEURISTIC`, or `:META`. Normally,[4] the definition of a strategy, focus, or heuristic KS results in the definition of three GBB unit types: one for the KS, one for its KSARs, and one for the control plan objects it adds to the control plan during the execution of its actions. Sometimes, however, a control KS does not create new plan objects, but merely manipulates those currently instantiated. GBB1 provides the meta-KS control type for this situation, which does not implicitly define a control plan object unit type.

define-gbb1-control-ks *name-and-options* [*documentation*] &KEY *slots links dimensional-indexes path-indexes paths event-classes trigger-conditions preconditions obviation-conditions context-slots ksar-unit-type action-function from-bb to-bb ks-phases author cost reliability control-type* [*Macro*]

[4] "Normally" means when **define-gbb1-ksar** is not being employed to explicitly define the KSAR.

Control-type must be one of the following: `:STRATEGY`, `:FOCUS`, `:HEURISTIC`, or `:META`.

For documentation on the other arguments to **define-gbb1-control-ks**, see Section 4.2.3.2.

The actions of strategy, focus, or heuristic control KSs usually involve the creation of a control plan object. The control plan unit types implicitly defined by GBB1 have a specific set of slots for use by the control shell. The following paragraphs document the structure of these control plan unit types.

Strategy Plan Object Structure

Definition of a strategy KS implicitly defines a strategy control plan object unit type. The name of the control plan object unit type is the name of the KS with "`-PO`" appended. Its structure is as follows:

Control-type: Always equal to `:STRATEGY`. Set by GBB1.

Name: A string containing the name of the strategy plan object. GBB supplies a default value, which may be overridden by the user.

KS: A link to the KS which generated this plan object. Set by GBB1.

First-cycle: The execution cycle when this plan object was created. Set by GBB1.

Last-cycle: The execution cycle when this plan object was deleted. Set by GBB1.

Goal-function: A function returning non-`nil` when this strategy has been completed. Defined by the user. This slot is used by the prescription KSs described in Section 4.2.4.9.

Future-prescription: A list indicating the sequence of foci which remain in the implementation of this strategy. This slot must be initialized by the user when the prescription KSs (Section 4.2.4.9) are being used.

Current-prescription: A link to the focus plan objects currently implementing this strategy. This slot is set and maintained by the prescription KSs described in Section 4.2.4.9.

Past-prescription: A list of the foci which were used previously to implement this strategy. This slot is set and maintained by the prescription KSs described in Section 4.2.4.9.

Structure of Focus Objects

Definition of a focus KS implicitly defines a focus control plan object unit type. The name of the control plan object unit type is the name of the KS with "`-PO`" appended. Its structure is as follows:

Control-type: Always equal to `:FOCUS`. Set by GBB1.

Name: A string containing the name of the focus plan object. GBB supplies a default value, which may be overridden by the user.

KS: A link to the KS which generated this plan object. Set by GBB1.

Status: One of `:OPERATIVE` or `:INOPERATIVE`. An inoperative focus is not used to rate KSARs (and therefore, none of the heuristics which implement it will be used either.) The default value of *status* is `:OPERATIVE`.

Stability: One of `:STABLE` or `:DYNAMIC`. A stable focus stores the value returned by its integrated rating function for a KSAR, and returns that value as its rating of the KSAR during subsequent execution cycles until this slot is changed to `:DYNAMIC`, or until a change occurs to its set of implementing heuristics. A dynamic focus calls its integrated rating function every time a KSAR is rated. The default value for *stability* is `:DYNAMIC`.

First-cycle: The execution cycle when this plan object was created. Set by GBB1.

Last-cycle: The execution cycle when this plan object was deleted. Set by GBB1.

Goal-function: A function returning non-`nil` when this focus is completed. Defined by the user. This slot is used by the prescription KSs described in Section 4.2.4.9.

Integrated-rating-function: A function which takes two arguments: a KSAR unit instance and a focus plan object unit instance. It returns a value which represents the combined rating of all the heuristics implementing that focus. This value should be computed through a call to **map-heuristic-ratings**, which is described in Section 4.2.4.6. The default value of *integrated-rating-function* is **sum-of-weights-times-ratings**, which is described in Section 4.2.4.6.

Heuristics: A list of heuristic KS names which implement this focus. Used by the prescription KSs described in Section 4.2.4.9.

Weight: A value indicating the weight of this focus which is often used by the integrated-rating-function. The default value of *weight* is 1.

Structure of Heuristic Objects

Definition of a heuristic KS implicitly defines a heuristic control plan object unit type. The name of the control plan object unit type is the name of the KS with "`-PO`" appended. Its structure is as follows:

Control-type: Always equal to `:HEURISTIC`. Set by GBB1.

Name: A string containing the name of the heuristic plan object. GBB supplies a default value, which may be overridden by the user.

KS: A link to the KS which generated this plan object. Set by GBB1.

Status: One of `:OPERATIVE` or `:INOPERATIVE`. Only operative heuristics are used to establish the rating of a KSAR. The default value of *status* is `:OPERATIVE`.

Stability: One of `:STABLE` or `:DYNAMIC`. A stable heuristic stores the value returned by its rating function for a KSAR, and returns that value as its rating of the KSAR during subsequent execution cycles until this slot is changed to `:DYNAMIC`. A dynamic heuristic calls its rating function every time a KSAR is rated. The default value for *stability* is `:DYNAMIC`.

First-cycle: The execution cycle when this plan object was created. Set by GBB1.

Last-cycle: The execution cycle when this plan object was deleted. Set by GBB1.

Goal-function: A function returning `nonnil` when this heuristic has been completed. Set by the user. This slot is used by the prescription KSs described in Section 4.2.4.9.

Rating-function: A function which is passed one argument, a KSAR unit instance. It returns a value indicating this heuristic's rating of the KSAR.

4.2.4.6 KS Shell Rating Function Utilities

Conceptually, rating a KSAR is straightforward. The priority function uses the ratings given to the KSAR by the foci to produce a single overall rating for the KSAR. Each focus rating is computed by combining the ratings given to the KSAR by all the heuristics implementing that focus.

The implementation of rating is not straightforward because recomputing every priority, heuristic, and integrated rating for each executable KSAR on every execution cycle is very expensive computationally. For this reason, the priority, focus, and heuristic rating functions can be made stable, which indicates that their values, once obtained, do not need to be recomputed as long as the parts of the control plan which they depend upon do not change. In addition, any focus or heuristic can be made inoperative, which indicates that their ratings should not be used. Finally, the actions of KSs may add or delete foci or heuristics from the control plan blackboard.

Given that the stability, operativeness, and even existence of control plan objects can change from cycle to cycle, determining the priority of a KSAR efficiently becomes a somewhat more subtle problem. For example, consider the situation where the priority, the foci, and their implementing heuristics are all stable. If the control plan doesn't change, then the priority of a KSAR, once computed, remains constant and does not need to be recomputed during subsequent execution cycles. However, different changes to the control plan will force recomputation of different parts of this "stable" rating structure. For example, if a KS action makes of the foci inoperative, then just the priorities of the KSARs must be recomputed. If a new heuristic is added, however, its rating, as well as the integrated rating of its implementing focus and the priority of every KSAR which had been previously rated by this focus must be recomputed.

To insulate the application implementer from these details, the GBB1 KS shell provides two functions which automatically maintain the control plan ratings: **map-focus-ratings** and **map-heuristic-ratings**.

map-focus-ratings `&KEY` *function KSAR* [*Function*]

Map-focus-ratings should be called by the priority function to obtain the integrated ratings of each operative focus for *KSAR*.

Function is a function which accepts two arguments: an integrated rating for *KSAR* and the focus plan object unit instance responsible for it.

Ksar is a KSAR unit instance.

A simple use of map-focus-ratings is in the following example priority function, which computes the priority of a KSAR as the sum of the integrated ratings returned by all active foci:

```
(defun EXAMPLE-PRIORITY (KSAR)

  "EXAMPLE-PRIORITY nil

   Compute the priority as the sum of the integrated ratings."

  (let ((total 0))
    (map-focus-ratings #'(lambda (integrated-rating po)
   (declare (ignore po))
   (incf total integrated-rating))
        KSAR)

    total))
```

map-heuristic-ratings &KEY *function KSAR focus-instance* [*Function*]

Map-focus-ratings should be called in the integrated rating function of *focus-instance* to obtain the ratings of *KSAR* given by its implementing heuristics.

Function is a function which accepts two arguments: a heuristic rating for *KSAR* and the heuristic plan object unit instance responsible for it.

Ksar is a KSAR unit instance.

Focus-instance is the focus plan object unit instance whose heuristics are to be used.

For an example of the use of **map-heuristic-ratings**, see the documentation for **sum-of-weights-times-ratings** below.

4.2.4.7 Default Integrated Rating Function

sum-of-weights-times-ratings &KEY *KSAR focus-instance* [*Function*]

The GBB1 KS shell supplies a default integrated rating function called **sum-of-weights-times-ratings** which is identical to the default integrated rating function of BB1. **Sum-of-weights-times-ratings** calculates and returns the integrated rating for KSAR as the sum of the product of each heuristic rating of KSAR multiplied by its weight. To illustrate the use of **map-heuristic-ratings**, the function definition follows:

```
(defun SUM-OF-WEIGHTS-TIMES-RATINGS (KSAR focus-instance)

  "SUM-OF-WEIGHTS-TIMES-RATINGS KSAR focus-instance

   The default integration rating function.
   Calculates and returns the integrated rating for KSAR as the sum of
   the product of each heuristic rating of KSAR multiplied
   by its weight."

  (let ((total 0))
    (map-heuristic-ratings
      #'(lambda (rating heuristic-po)
          (incf total (* (basic-heuristic-po$weight heuristic-po)
                         rating)))
      KSAR
      focus-instance)
  total))
```

4.2.4.8 Other KS Shell Utilities

The KS and KSAR phase manipulation functions described in Section 4.2.3.9, the trigger utilities described in Section 4.2.3.8, and the variables described in Section 4.2.3.10 are all accessible to the KS shell.

4.2.4.9 The Prescription Control KSs

The BB1 control model specifies a control hierarchy consisting of strategies, foci, and heuristics. While the model specifies how this network is used to rate KSARs, it does not specify how it is set up or maintained. However, the BB1 system does include a set of KSs to implement one method of maintaining the control plan called the *prescription KSs*. The prescription method is implemented by three KSs: **initialize-prescription**, **update-prescription**, and **terminate-prescription**. These KSs are supplied in the file `prescription-ks`.

The prescription method is documented in [8]. The following discussion illustrates how this method has been implemented in GBB1. In general, the prescription method is activated by the addition of a strategy plan object to the control plan blackboard. (This is normally one of the results of execution of the initial KSs actions.) Strategy plan objects have a slot called *future-prescription*, which is initialized to a list of the sequence of foci which must be satisfied in order to complete this strategy. When a strategy plan object is added, the **initialize-prescription** KS is triggered and becomes executable. Its action is to set the *current-prescription* link of the triggering plan object to the first element of its *future-prescription* slot. Each focus KS is triggered when its name becomes one of the elements of the current-prescription slot of a strategy KS, and its action should be to instantiate a focus plan object linked to this strategy. Focus plan objects, in turn, initialize their *heuristics* slot to a list of the names of their implementing heuristics. Each heuristic KSs is triggered and becomes executable when its name is a member of a focus' *heuristics* slot.

When the current prescription of a strategy has been satisfied, the **update-prescription** KS is triggered to update the *current-prescription* link of this plan object with the next set of foci from its *future-prescription* slot.

When the goal of a focus or strategy succeeds, the **terminate-prescription** KS is triggered which sets the status slot of the strategy to inoperative.

4.2.4.10 An Example: The Traveling Salesman Problem

The traveling salesman problem is to find the shortest path between a set of cities. The file `gbb1-ks-example` contains a GBB1 KS shell application which finds a short path between a set of cities. To do this, it uses two heuristics in deciding how to construct a path. The first is to prefer short arcs (connect each city to its two closest neighbors), and the second is to prefer outer arcs (don't zigzag). This example system is closely modeled on the BB1 system TSP.

Following the structure given in 4.2.2.1, this example system can be divided into the following components:

- **Application definitions** This system has two domain-specific blackboards: **tour-info** and **solution**. The **tour-info** blackboard has a single space, **cities**, which stores a blackboard object containing information about each city to be visited. The **solution** blackboard contains a single space, **paths**, which stores blackboard objects containing information about the current status of the solution path. These blackboards and spaces are defined using the standard GBB functions **define-blackboard** and **define-space**.

  The domain blackboard objects are **city** and **path**. The **city** object contains information about each city to be visited. The **path** object contains information about the current state of the path being built by the system. Since the creation of these objects must signal events to the control shell, they are defined using **define-gbb1-unit** rather than **define-unit**.
- **KS definitions** The domain KSs are **start-tsp**, **add-arc**, and **add-final-arc**. **Start-tsp** is the initial KS, whose action is to initialize the domain blackboards. **Add-arc** creates one KSAR for every city to city combination, and selecting that KSAR for execution results in adding that city to city "link" to the solution path. **Add-final-arc** is similar to **add-arc**, except that it is triggered when there is only one more arc to be added to complete the solution path. These KSs are defined using **define-gbb1-domain-ks**.

  The domain KSs described above are used to construct a solution path. However, they do not provide any information to the system on which path is to be constructed. The control KSs provide the information required for deciding which arc to add to the solution path at any point in time. As stated above, the control plan uses two heuristic KSs in rating the arcs available: **prefer-short-arcs** and **prefer-outer-arcs**. These heuristics implement the control focus KS called **create-good-path**, which implements the strategy KS called **find-short-path**. These KSs are defined using **define-gbb1-control-ks**.

One last KS issue remains: the triggering of these control KSs so that they can construct the control plan to be used to rate the **add-arc** KSARs as described in the preceding paragraph. Though this functionality could have been explicitly written into the above control KSs, it can also be abstracted out and represented by a set of control KSs whose sole purpose is to construct and maintain control plans. GBB1 provides one set of control KSs (the prescription KSs) to accomplish this, and these KSs are loaded and used in this example system.

- **Control shell parameter setup** Both **define-gbb1-output** and **define-gbb1-parameters** are used in this example system. **Define-gbb1-parameters** is used to define the termination, recommendation, and priority functions, and **define-gbb1-output** is used to specify trace information.
- **Application execution** This system defines the function **run-tsp** which is called to initiate execution. This function instantiates the domain blackboard and then calls **run-gbb1** to begin problem solving.

4.2.4.11 Using Prescription KSs in the TSP system

The TSP system provides an example of the use of prescription KSs in problem solving. The following "play-by-play" provides an overview of their use. For more detail, the reader is encouraged to load and examine the running of the TSP system.

Running the Initial KS The initial KS in the TSP system is `start-tsp`, which sets up the solution path and tour-info blackboards. This triggers the strategy `find-short-path`, which executes and creates its strategy plan object.

Initializing the Prescription The creation of the `find-short-path` strategy plan object triggers the `initialize-prescription` KS. This KS executes and sets the current-prescription slot of the `find-short-path` plan object to `create-good-path`.

Creating the control plan The `create-good-path` focus is triggered by the presence of its name in the current-prescription slot of `find-short-path`. It executes and adds its focus plan object to the control plan. Its heuristics slot is a list containing prefer-short-arcs and prefer-outer-arcs. The presence of a plan object with a heuristics slot containing their names triggers the `prefer-short-arcs` and `prefer-outer-arcs` heuristic KSs, which execute and add their plan objects to the control plan.

Creating the salesman path Now that the control plan is set up, the `add-arc` KSARs have a chance to run. (Control KSARs are always run before domain KSARs with the BB1 default heuristic.) These KSARs are evaluated based upon the ratings given to them by the `default-heuristic`, `prefer-outer-arcs`, and `prefer-short-arcs` heuristic plan objects. (The two ratings given by `prefer-outer-arcs` and `prefer-short-arcs` are combined into a single integrated rating by the integration function in the `create-good-path` focus. The single rating of `default-heuristic` is used as the integrated rating returned by `default-focus`. These two integrated ratings are combined into a single priority rating by the `tsp-priority` function.) The add-arc KSAR with the highest priority is selected to run.

Dismantling the control plan The goal of TSP foci and strategy is the creation of a complete path. When this has been accomplished, the `terminate-prescription` KS is triggered and runs, which unlinks the foci from the strategy and sets the status slot of all these

plan objects to :inoperative. Once the `terminate-prescription` KSAR has run, there are no longer any KSARs left on the executable agenda and execution terminates.

Chapter 5

GBB Graphics

This chapter describes GBB's graphics facility. At the present time, GBB graphics are only available on Texas Instruments Explorer workstations.

The GBB graphics is a window oriented program for examining the blackboard database. It works by taking a one or two dimensional slice through a space and displaying the units stored on the space in terms of the those dimensions. The way that a unit is displayed depends on its indexes. For example, if a unit has two indexes which are ranges then the unit will appear as a rectangle.

5.1 The GBB Graphics Window

The GBB graphics are not normally loaded with GBB. To load the graphics enter the form

```
(make-system 'gbb-graphics :noconfirm)
```

The graphics system is defined in the GBB system file. To bring up the GBB graphics type `SYSTEM Symbol-Shift-G`.

The GBB graphics window consists of a number of *graphics panes* that are grouped together in a *constraint frame*. Several arrangements of panes (called *configurations*) are available. Each pane can display the contents of one or more spaces. Figure 5.1 shows one configuration of the graphics frame.

This example shows four graphics panes on the left in a configuration with a lisp listener on the the right. The top two graphics panes are displaying the same space but are taking different two dimensional slices through the space. The upper left pane is displaying the dimensions x and y. There are two units on the space. The lines connecting the rectangles indicate that the units are composite units. The upper right pane is displaying the dimensions *time* and y. Notice that *time* is not a range, it is a point. When one index is a point and the other is a range the index elements are represented as a heavy line.

The lower left pane is displaying the x and y dimensions of another space. One composite unit is shown. Both the x and y indexes are points so the index elements are displayed as dots.

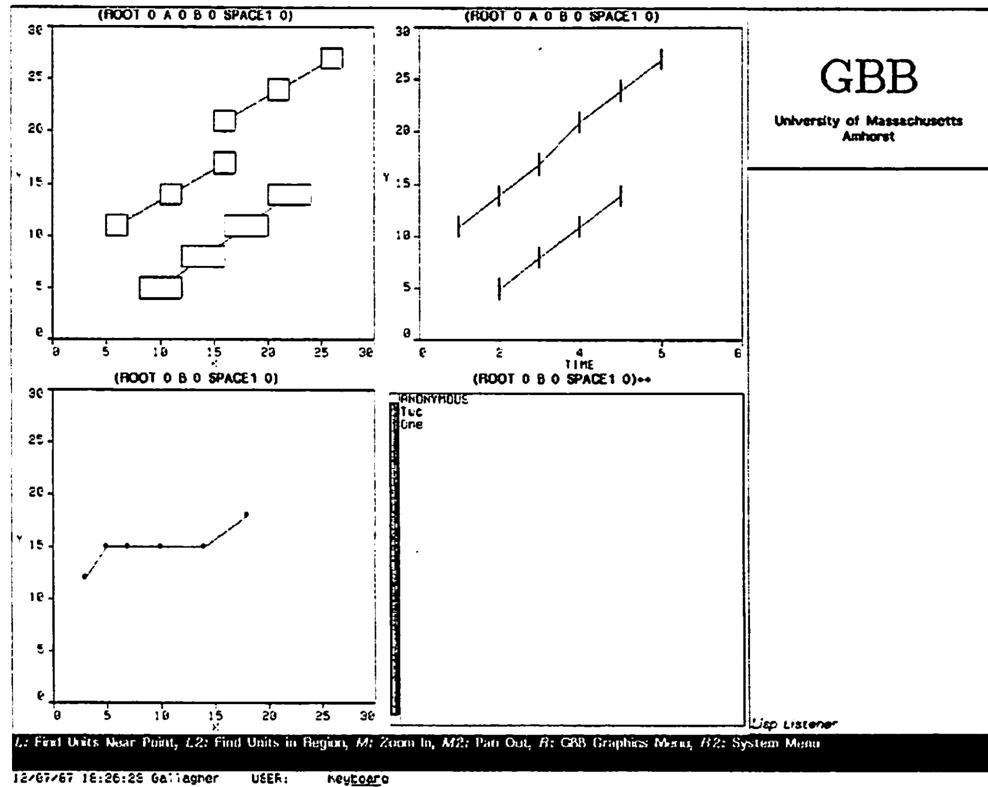


Figure 5.1: An example of the GBB Graphics

The lower right pane is not displaying any dimensions. In this case, rather than displaying each unit graphically, each unit on the space is listed by name in a scrollable display. The units are mouse sensitive.

5.2 The Mouse

Interactions with the graphics take place primarily through the mouse. The mouse buttons do different things depending on where the mouse is in the pane and what the pane is displaying. However the right button always does the same thing: clicking right once will bring up a menu of GBB graphics operations, and clicking right twice will bring up the system menu.

Generally, when you click in a pane the operation is specific to that pane. In particular, when you bring up the GBB graphics menu the operation you choose will affect the pane that you first clicked in regardless of where the mouse ends up. The functions of the mouse buttons are summarized below. A more complete description of the operations appears in Sections 5.3 and 5.4.

1. When the pane is displaying one or two dimensions and the mouse is in the margins of the graph (i.e., the region of the pane where the axis labels and are displayed) then the mouse buttons have the following meanings.

   [Left] Select which spaces this pane will display.

   [Left 2] Change the dimensions that this pane will display. This does not change which spaces are displayed.

   [Middle] Change the configuration of the entire frame.

   [Middle 2] Select which types of units are to be displayed.

2. When the pane is displaying one or two dimensions and the mouse is in the body of the graph then the mouse buttons have the following meanings.

   [Left] Find the units near the mouse position.

   [Left 2] Mark a region of the graph with the mouse and find the units that overlap with that region.

   [Middle] Zoom in on a portion of the space.

   [Middle 2] Display the entire space (unzoom).

3. When the pane is displaying a list of units and the mouse is over a unit so that the unit is highlighted then the mouse buttons have the following meanings.

   [Left] Describe the selected unit in a pop up window.

   [Middle] Set the value of the symbol `user::==` to the selected unit.

4. When the pane is displaying a list of units and the mouse is not over a unit (no unit is highlighted) then the mouse buttons have the same meanings as they do in 1 above.

5.3 Operations on Panes

This section describes the available operations. Most of these operations can be invoked through the GBB graphics menu. (Only [Find Units Near Point] and [Find Units in Region] are not in the menu.) In addition to the menu some of them can also be invoked directly by mouse clicks (see Section 5.2).

5.3.1 Space Selection

When a pane is associated with a space or spaces, changes to those spaces will be reflected immediately in the display. The first step, then, is to associate a pane with a set of spaces. This can be done in two ways.

[Select Space] This operation pops up a menu of the spaces instances in the blackboard database. Select the spaces you are interested in. You can select more than one space. When you are done click on [Do It]. You will then get another menu which allows you to choose which space dimesion to display on each axis.

[Copy Pane] This operation allows you to copy all the characteristics of one pane to another pane. The characteristics of the current pane (the one you clicked in to bring up the menu) are copied to another pane that you select with the mouse. The characteristics copied include the spaces and dimensions being displayed as well as all the other characteristics such as the step mode of the pane.

GBB will not automatically pause to let you set up your panes. That must be done in your application. However, if you instantiate a new blackboard database, the graphics will update the all the panes so that they are displaying the new space instances.

5.3.2 Dimension Selection

There are several operations to select which dimension to display on each axis and what portion of the dimension to display.

[Choose Dimensions] This operation pops up a menu which allows you to select which dimension to display on each axis of the pane. In addition to the actual space dimensions there is another choice for each axis, "No Dimension." Choosing "No Dimension" for both axes will cause the pane to display a scrollable list of mouse sensitive units. Choosing "No Dimension" for one axis will result in a one dimensional graphical display.

[Copy Dimensions] This operation will set the dimensions being displayed by another pane be the same as the dimensions of the current pane. Both the dimensions and the dimension bounds of the other pane are changed.

[Choose Dimension Bounds] This operation pops up a menu which allows you to set what portion of each dimension you want displayed.

[Zoom In] This operation lets you set what portion of each dimension you want displayed by using the mouse to mark a region. If either of the dimensions are ordered dimensions the the region is rounded out integer boundaries.

[Zoom Out] This operation sets the bounds of the dimensions to display the entire space. For ordered dimensions the bounds are set to display the entire range and for enumerated dimensions the bounds are set to display all the labels.

5.3.3 Pane Options

Each pane has several options which control what units are displayed, when they are displayed, and whether there should be a pause when a change to the display is made.

[Choose Unit Types] By default each pane will display all the unit types that are stored on the panes spaces. This operation allows you to limit the types of units that are displayed by the pane. It pops up a menu from which you can choose what unit types should be displayed.

[Choose Filter Function] Each pane can have a predicate function which determines which unit instances should be displayed. The function should take one argument, a unit instance. This operation pops up a window into which you enter the function or function name. To disable this feature set the filter function to `nil`.

[Choose Display Mode] Each pane can be in one of three *display modes*. This operation sets the display mode for a single pane. The display modes are:

All Units on Space In this mode all the units on the space (provided they satisfy the unit types and filter function restrictions) are displayed.

Last Unit Added In this mode only the unit that was most recently added to the space is displayed.

Recently Added Units In this mode all units that are added to the space after selecting *Recently Added Units* will be displayed.

The default value for display mode is *All Units on Space.*

[Choose Step Mode] If the blackboard is changing very quickly it can be difficult to see what is going on. Sometimes it is useful to be able to pause after a unit is displayed. Each pane has a *step mode* which indicates under what circumstances it should pause. The two situations are after a unit has been added to one of the pane's spaces and after a unit has been displayed due to one of the "display" operations below.

Pausing consists of waiting for the user to type a character. To indicate that the pane is waiting, the word "Step" in inverse video appears in the lower left corner of the pane.

[Choose Display Mode All] This operation sets the display mode for all the panes.

[Choose Step Mode All] This operation sets the step mode for all the panes.

5.3.4 Display Operations

Normally units will appear on the panes without any specific action by you. However, occasionally, the screen will be left in an inconsistant state. These operations force a the pane to be redrawn.

[Display Current Units] This operation displays the current units of the pane. The *current units* are determined by the display mode of the pane. If the display mode is *Last Unit Added* then the current unit is just the last unit added to the space(s). If the display mode is *Recently Added Units* then the current units are all units that have been added to the space since selecting *Recently Added Units*. Finally, if the display mode is *All Units* then this operation displays all the units on the space(s).

[Display Space Contents] This operation displays all the units on the space(s) regardless of the setting of the display mode.

[Display Current Units All] This operation displays the current units for all the panes.

[Display Space Contents All] This operation displays all the units on the space(s) of all the panes.

5.3.5 Finding Units

There are two operations that let you find units interactively by using the mouse. These operations are not available in the GBB graphics menu. They can only be invoked by clicking left or left twice in the graph portion of a pane (see Section 5.2).

[Find Units Near Point] This operation finds all the units near where the mouse is clicked. If any units are found they are presented in a menu. You can then select a unit and either describe it or set the value of the symbol `user::==`.

[Find Units in Region] This operation is similar to [Find Units Near Point]. It finds all the units that overlap with a region you define with the mouse.

5.3.6 Miscellaneous Operations

[Pane Status] This operation shows information about the pane.

[Clear Pane] Clear the pane. This resets the current units of the pane but doesn't affect any other characteristics of the pane such as what spaces it is displaying.

[Reset Pane] This disassociates this pane from all spaces and resets all the characteristics of the pane back to the default values.

[Change Configuration] Select a new frame configuration.

5.4 Operations on Units

The operations in this section are available when the mouse is over a particular unit instance such as when a unit is highlighted in the mouse sensitive scrolling pane display or when a group of units are being presented in a menu.

[Describe Unit] This describes the unit instance in a pop up window. The information includes the values of the unit's slots and links.

[Set `user::==`] This sets the value of the symbol `user::==` to the selected unit instance.

5.5 Progam Interface

This section describes the functions and variables provided to manipulate the graphics directly from your application.

gbb-graphics:setup-pane *pane path-structures x-dimension y-dimension* [*Function*]
`&KEY` *title frame step-modes filter-function*
(*unit-types* `t`) (*display-mode* `:all-units`)

This function sets up a pane to display one or more space instances. The arguments are as follows.

pane is a number from 1 to 10 which indicates which pane to set up.

path-structures is a list of path structures which indicate the spaces to display.

x-dimension is the dimension to display on the X axis. It should be a symbol. If *x-dimension* is `nil` then no dimension will be displayed on the X axis.

y-dimension is the dimension to display on the Y axis. It should be a symbol. If *y-dimension* is `nil` then no dimension will be displayed on the Y axis.

title is a string which will be used as the title for the pane. If it is omitted a title will be constructed from the path representing the space instance or instances being displayed.

frame is the frame from which to select the *pane* to initialize. This argument is required if more than one frame has been created.

step-modes indicates when the graphics should pause. The possible values are `nil` (never pause), `:UNIT-CREATION` (pause after a unit is created), and `:DISPLAY-ALL` (pause after any 'display' operation).

filter-function is a function of one argument, a unit instance, which should return true if the unit should be displayed and `nil` if not. If *filter-function* is `nil` then no filtering is done.

unit-types is a list of the types of units to display on this pane. It is either a single unit type, a list of unit types, or `t` which indicates that all unit types should be displayed.

display-mode this is one of `:ALL-UNITS`, `:RECENTLY-ADDED-UNITS`, or `:LAST-UNIT`.

gbb-graphics::*gbb-graphics-frames* [*Variable*]

This variable contains a list of the GBB Graphics frames that have been created. The user should not change this.

gbb-graphics::*show-unrelated-links* [*Variable*]

This variable determines whether lines should be drawn between the index elements of two unrelated dimensions. Two dimensions are unrelated if they are not part of the same composite index structure.

For example, suppose a unit has two dimensions, *time* and *type*, where *time* is a set of times (e.g., a list of contact times) and *type* is a label. Such a unit would be displayed as a group of dots, one dot for each *time*, *type* combination. In some situations you may want to connect the dots to indicate that they are all part of the same object; in other situations you may not.

The possible values for `*show-unrelated-links*` are as follows.

`nil`	No lines will be drawn between the index elements.
`:MINIMUM`	A single line will be drawn connecting all the index elements.
`:MAXIMUM`	Lines are drawn connecting the points resulting from all permutations of the X axis dimension and the Y axis dimension.

The default is `:MINIMUM`.

Appendix A

A Simple GBB Database

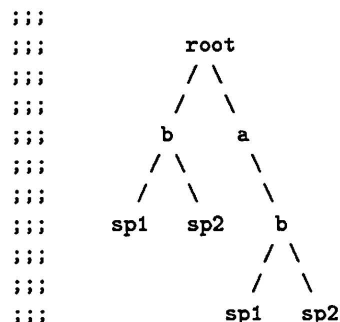
```
;;;; ==============================================================
;;;;
;;;;                ''SIMPLE'' GBB DATABASE EXAMPLE
;;;;
;;;; The following is a simple example of the set of calls needed
;;;; to define, create, and manipulate a GBB database.
;;;;
;;;; ==============================================================

;;; Make GBB and extended lisp symbols visible without prefixing.

(in-package 'user)

(use-package '(lisp umass-extended-lisp gbb))

;;; ----------------------------------------------------------------
;;; Define the Spaces and Blackboards
;;; ----------------------------------------------------------------
;;;  The following calls define a blackboard hierarchy that looks
;;;  like this.
;;;
```



```
;;;          root
;;;          / \
;;;         /   \
;;;        b     a
;;;       / \     \
;;;      /   \     \
;;;    sp1   sp2    b
;;;                / \
;;;               /   \
;;;             sp1   sp2


(define-spaces (SP1 SP2)
  "Two sample spaces."
  :UNITS      (unit1 unit2)
  :DIMENSIONS ((x    :ORDERED (0 30))
               (y    :ORDERED (0 30))
               (type :ENUMERATED (dog cat raccoon opossum)
```

```
                    :TEST equal)))

(define-blackboards (B) (sp1 sp2)
  "Basic blackboard structure.")

(define-blackboards (A) (B)
  "B is a component of A.")

(define-blackboards (ROOT) (a b)
  "The top level layer of BB.")

;;; -------------------------------------------------------------------
;;; Define an Index Structure
;;; -------------------------------------------------------------------
;;; Define the defstructs that define-index-structure refers to
;;; before the call to define-index-structure.

(defstruct (REGION (:CONSTRUCTOR boa-make-region
                                 (&OPTIONAL xmin xmax ymin ymax)))
  xmin
  xmax
  ymin
  ymax)


(define-index-structure X-Y-INDEX
  "A region of the x y plane."
  :TYPE region
  :INDEXES ((x :RANGE (:min xmin) (:max xmax))
            (y :RANGE (:min ymin) (:max ymax))))

;;; -------------------------------------------------------------------
;;; Define the Units
;;; -------------------------------------------------------------------

(define-unit (UNIT1 (:CREATION-EVENTS print-creation)
                    (:DELETION-EVENTS print-deletion))
  "Unit Number 1"

  :SLOTS
  ((area        nil :type x-y-index)
   (next-goal   nil :ACCESS-EVENTS (print-access)
                    :UPDATE-EVENTS (print-slot-update))
   (animal-kind 'dog)
   (which-space 'sp1)
   (which-bb    0))

  :DIMENSIONAL-INDEXES
  ((x    area :type x-y-index)
   (y    area :type x-y-index)
   (type animal-kind :type :label))

  :LINKS
  ((link1 (unit2 link1-inverse)
          :unlink-events (print-unlink))
   (link2 :reflexive)
```

```
   (link3 (unit2 link3-inverse :SINGULAR)))

  :PATH-INDEXES
  ((space-name   which-space :type :label)
   (bb-rep-count which-bb    :type :point))

  :PATHS
  ((:PATH (cond ((= bb-rep-count 0) `(root a 0 b ,space-name 0))
                (t `(root b ,bb-rep-count sp1)))))

  ) ;; End of Unit1

;;;-------------------------------------------------------------------

(define-unit (UNIT2)

  "Unit Number 2"

  :SLOTS
  ((ks       nil :TYPE t)
   (x        0   :TYPE fixnum)
   (y        0   :TYPE fixnum)
   (time     0   :TYPE fixnum))

  :DIMENSIONAL-INDEXES
  ((x x :TYPE :point)
   (y y :TYPE :point))

  :LINKS
  ((link1-inverse (unit1 link1))
   (link3-inverse :SINGULAR (unit1 link3)))

  :PATHS ((:path '(root B 0 sp1))))


;;; ------------------------------------------------------------------
;;; Define Event Functions
;;; ------------------------------------------------------------------
;;; These simple events just illustrate the arguments necessary
;;; for each event type.  Real events communicate with a control
;;; shell (not included in this simple example of GBB).

(defun PRINT-CREATION (unit)
  (format t "~%CREATION EVENT: ~s" unit))

(defun PRINT-DELETION (unit)
  (format t "~%DELETION EVENT: ~s" unit))

(defun PRINT-ACCESS (unit slot current-value)
  (format t "~%ACCESS EVENT:~%  UNIT=~s~%  SLOT=~s~%  CURRENT-VALUE=~s"
          unit slot current-value))

(defun PRINT-SLOT-UPDATE (unit slot new-value old-value)
  (format t "~%UPDATE EVENT:~
             ~%  UNIT=~s~%  SLOT=~s~%  NEW-VALUE=~s~%  OLD-VALUE=~s"
  unit slot new-value old-value))
```

```
(defun PRINT-LINK-UPDATE (unit link new-value link-added)
  (format t "~%UPDATE EVENT:~
            ~%  UNIT=~s~%  LINK=~s~%  NEW-VALUE=~s~%  LINK-ADDED=~s"
  unit link new-value link-added))

(defun PRINT-UNLINK (unit link new-value deleted-unit)
  (format t "~%UNLINK EVENT:~
            ~%  UNIT=~s~%  LINK=~s~%  NEW-VALUE=~s~%  DELETED-UNIT=~s"
  unit link new-value deleted-unit))


;;; ----------------------------------------------------------------------
;;; Define Unit Mappings
;;; ----------------------------------------------------------------------
;;; The call to unit mapping specifies how to store the units on
;;; the spaces. Since no unit mapping is specified for SP2,
;;; the storage will be implemented by default as a simple list
;;; of units.

(define-unit-mapping (UNIT1) (sp1)
  "Mapping UNIT1 instances onto SP1 as
   3 one-dimensional bucket structures."
  :INDEXES (x y type)
  :INDEX-STRUCTURE
  ((x    :SUBRANGES (:start :end (:width 5)))
   (y    :SUBRANGES (:start :end (:width 10)))
   (type :groups)))


(define-unit-mapping (UNIT2) (sp1)
  "Mapping UNIT2 instances onto SP1 as
   one two-dimensional bucket structure."
  :INDEXES ((x y))
  :INDEX-STRUCTURE
  ((x    :SUBRANGES (:start :end (:width 5)))
   (y    :SUBRANGES (:start :end (:width 5)))))


;;; ----------------------------------------------------------------------
;;; Create the Blackboard Database
;;; ----------------------------------------------------------------------


(instantiate-blackboard-database '(root (a 2 (b (sp1 2)))
                                        (b 2)))

;;;  The created database will look like:
;;;
;;;                          root
;;;                           |
;;;         -----------------------------------------------
;;;         |                 |            |            |
;;;       (a 0)             (a 1)        (b 0)        (b 1)
;;;         |                 |            |            |
;;;         b                 b         -------      --------
```

```
;;;             |                  |          |     |       |      |
;;;      -----------------         |         sp1   sp2     sp1    sp2
;;;      |        |      |         |
;;;   (sp1 0)  (sp1 1)  sp2        |
;;;                                |
;;;                       -----------------
;;;                       |        |       |
;;;                    (sp1 0)  (sp1 1)   sp2
;;;
;;;                       ^
;;;                       |
;;;  and the path to this unit, for example, is: (root a 1 b sp1 0).


;;; ----------------------------------------------------------------
;;; Make some units.
;;; ----------------------------------------------------------------

;;; Since which-bb is not zero, this unit will go on the space
;;; (root b 1 sp1).

(make-unit1
  :NAME "First Unit1"
  :AREA (boa-make-region 2 4 6 8)
  :WHICH-BB 1)

;;; Since which-bb is zero (the default), this unit will go on the
;;; space (root a 0 b sp1 0).

(make-unit1
  :NAME "Second Unit1"
  :AREA (boa-make-region 12 14 16 18))

;;; Unit2's always go on the space (root b 0 sp1).

(make-unit2
  :NAME "First Unit2"
  :X 12
  :Y 3)

(make-unit2
  :NAME "Second Unit2"
  :X 10
  :Y 24)


;;; ----------------------------------------------------------------
;;; Manipulate the BB a bit.
;;; ----------------------------------------------------------------

;;; Update one of the units, triggering an update event.

(setf (unit1$next-goal
        (find-unit-by-name "First Unit1" 'unit1))
        'new-goal)
```

```
;;; Trigger an access event.

(print (unit1$next-goal (find-unit-by-name "Second Unit1" 'unit1)))

;;; Make a few links. First we retrieve a few of the units.

(setf *unit1-1* (find-unit-by-name "First Unit1" 'unit1)
      *unit1-2* (find-units
                  'unit1
                  (make-paths :PATHS '(root a 0 b sp1 0))
                  '(:PATTERN-OBJECT
                     (:INDEX-TYPE x-y-index
                      :INDEX-OBJECT ,(boa-make-region 12 14 16 18))))
      *unit2-1* (find-unit-by-name "First Unit2" 'unit2))

;;; Link the first unit1 to the second unit1.

(linkf (unit1$link2 *unit1-1*) (first *unit1-2*))

;;; Link the second unit1 to itself.

(linkf (unit1$link2 (first *unit1-2*)) (first *unit1-2*))

;;; Link the first unit1 to the first unit2.

(linkf (unit1$link1 *unit1-1*) *unit2-1*)

;;; Unlink unit1 from unit2, triggering an unlink event.

(unlinkf (unit1$link1 *unit1-1*) *unit2-1*)

;;; Print the contents of a space.

(format t "~%;;; Units in (a b sp1):")
(map-space #'print
           t
           (make-paths :paths '(root a 0 b sp1 0)))

;;;; -----------------------------------------------------------------
;;;;
;;;; EOF
;;;;
;;;; -----------------------------------------------------------------
```

Bibliography

[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 Philip M. Johnson. *From Prototype to Product: Evolutionary Development within the Blackboard Paradigm.* Technical Report 86-46, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, October 1986. Presented at Workshop on High Level Tools for Knowledge Based Systems, Columbus, Ohio, October 7–8, 1986.

[3] 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. Also published as Technical Report 86-66, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, April 1986.).

[4] 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 as Technical Report 87-37, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003, April 1987.).

[5] 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.

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

[7] 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.

[8] 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.).

Index

metering-enabled, 54
:access-events, 21, 24
:after-extras, 42
:always-inherit, 29
:append, 36
:ask, 36
:before-extras, 42
:change-index, 38
:change-relative, 38
:change-subpath, 38
:conc-name, 16
:constructor, 16
:contiguous, 43
:creation-events, 18
:deletion-events, 18
:delta, 41
:displace, 41
:dynamic, 21
:element-match, 42
:exact, 42
:export, 17
:filter-after, 40
:filter-before, 40
:groups, 53
:include, 17
:include-all-components, 36
:includes, 42
:index-object, 41
:index-type, 41
:initialization-events, 21, 24
:label, 8, 25
:link-access-events, 19
:link-initialization-events, 19
:link-update-events, 20
:match, 42
:mismatch, 42
:mode, 36
:name-function, 17
:never-inherit, 29
:no-default-inheritance, 28
:overlaps, 42
:overwrite, 36
:pattern-object, 40
:point, 8, 25
:predicate, 17
:print-function, 16, 49
:private, 22, 24
:range, 8, 25
:read-only, 21
:reflexive, 23
:save-print-function, 22, 49
:select, 41
:singular, 23
:skipped, 43
:slot-access-events, 19
:slot-initialization-events, 18
:slot-update-events, 20
:static, 21
:subranges, 53
:subseq, 41
:type, 21
:unlink-events, 20, 24
:unnamed, 18
:update-events, 22, 24
:width, 53
:within, 42

add-unit-to-space, 37
application implementer, 1

blackboard, 2

creation, 10
replication, 10
blackboard administrator, 1
blackboard database compiler, 1
blackboard events, 2
blackboard/space path, 2
specification, 3
buckets, 9

change-paths, 38
check-unit-links, 50
clear-blackboard-database, 50
clear-space, 50
composite unit, 6
composite-index, 29
composite-type, 29
control shell, 2
control shells
GBB1, 62ff
simple-shell, 57ff

define-blackboard, 14
define-blackboards, 14
define-index-structure, 29
define-space, 12
define-spaces, 12
define-unit, 14
dimensional-indexes, 15
indexes, 25
link options
`:access-events`, 24
`:initialization-events`, 24
`:unlink-events`, 24
`:update-events`, 24
links, 15, 23
`:reflexive`, 23
`:singular`, 23
name-and-options, 15
options, 16
`:conc-name`, 16
`:constructor`, 16
`:creation-events`, 18
`:deletion-events`, 18
`:export`, 17
`:include`, 17
`:link-access-events`, 19
`:link-initialization-events`, 19
`:link-update-events`, 20
`:name-function`, 17
`:predicate`, 17
`:print-function`, 16
`:slot-access-events`, 19
`:slot-initialization-events`, 18
`:slot-update-events`, 20
`:unlink-events`, 20
`:unnamed`, 18
path-indexes, 15
paths, 15, 25
slot options, 20
`:access-events`, 21
`:dynamic`, 21
`:private`, 22, 24
`:read-only`, 21
`:save-print-function`, 22
`:static`, 21
`:type`, 21
`:update-events`, 22
slots, 15
define-unit-mapping, 53
dimensional-indexes, 53
index-structure, 53
definitions
blackboard, 2
blackboard/space path, 2
composite unit, 6
dimensional indexes, 6
events, 8, 27
extended unit type, 7, 26
index structure, 29
index-element-types, 8
index-value, 8
indexes, 8
links, 5
path clauses, 6
path indexes, 6
path structures, 37

paths, 6
sets, 30
simple unit type, 7, 26
slots, 5
unit, 5, 14
unit inclusion and inheritance, 7
unit types, 7, 26
unstructured space, 13
delete-*unit-type*, 16, 37
delete-unit-from-space, 37
describe-blackboard-database, 50
describe-space, 51
describe-space-instance, 55
dimensional index values, 6
dimensional indexes, 6
dimensional-indexes, 25, 53
dummy-unit-instance, 49
dummy-unit-instance-p, 52

element-type, 29
event-inheritance-keywords, 28
:always-inherit, 29
:never-inherit, 29
:no-default-inheritance, 28
events, 8, 27
event inheritance keywords, 9

find-*unit-type*, 16, 47
find-unit
:after-extras, 42
:before-extras, 42
:contiguous, 43
:delta, 41
:displace, 41
:element-match, 42
:exact, 42
:includes, 42
:overlaps, 42
:within, 42
:filter-after, 40
:filter-before, 40
:index-object, 41
:index-type, 41
:match, 42
:mismatch, 42
:pattern-object, 40
:select, 41
:skipped, 43
:subseq, 41
pattern, 40
find-unit-by-name, 11, 47
find-units, 10, 40

gbb-graphics::*gbb-graphics-frames*, 89
gbb-graphics::*show-unrelated-links*, 90
gbb-graphics:setup-pane, 89
GBB1 control shell, 62ff
execution-cycle, 72
this-ks, 72
this-ksar, 72
trigger-event, 72
current-ks-phase, 72
define-gbb1-control-ks, 75
define-gbb1-ks, 67
define-gbb1-ksar, 69
define-gbb1-output, 70
define-gbb1-parameters, 69
define-gbb1-unit, 66
map-focus-ratings, 77
map-heuristic-ratings, 78
reset-gbb1, 71
run-gbb1, 70
set-ks-phase, 72
sum-of-weights-times-ratings, 78
trigger-event-class-p, 71
trigger-event-level-p, 71
trigger-event-slot-p, 71
trigger-unit, 71

index-element-types, 8
:label, 8, 25
:point, 8, 25

:range, 8, 25
index-structure, 53
index-value, 8
indexes, 8, 25
inheritance
event, 28
instantiate-blackboard-database, 35

link macros, 38
linkf, 38
linkf!, 38
linkf-list, 38
links, 5

make-*unit-type*, 16, 36
make-paths, 37
map-space, 11, 48
map-unit-types, 11, 47

path clauses, 6
path structures, 37
path-indexes, 6, 25
path-instantiated-p, 51
paths, 6, 25
pattern, 40
pp-blackboard-database, 51

replication, 2
replication counts, 10
replication indexes, 10
replication-description, 35
report-meters, 55
reset-gbb, 50
reset-meters, 54
restore-blackboard-database, 49
retrieval, 10, 39, 47

save-blackboard-database, 48
sets, 30, 33
simple-shell control shell, 57ff
define-ks, 59
define-level, 58
define-levels, 58
instantiate-simple-control-shell, 61
simple-control-shell, 61
slots, 5
space, 2
dimensionality, 4
enumerated dimension, 4
ordered dimension, 4
specifying storage, 9

unit, 2, 5
anonymous, 5
composite, 6
creation, 10, 36
deletion, 10, 37
dimensional-indexes, 6
dummy-unit-instance, 49
inclusion, 7
inheritance, 7
links, 5
moving, 37
named, 5
path clauses, 6
path indexes, 6
paths, 6
retrieval, 10, 39, 47
slots, 5
type specifiers, 7, 26
unit types, 7, 26
extended, 7, 26
simple, 7, 26
unit$name, 51
unit-instance-p, 51
unit-type-p, 52
unlinkf, 39
unlinkf-all, 39
unlinkf-list, 39
unstructured space, 13
update-space-dimension, 13