

# Embedable Problem-Solving Architectures: A Study of Integrating OPS5 with UMass GBB

Daniel D. Corkill, *Member, IEEE*

**Abstract**—Typically, AI shells have a Ptolemaic view of their universe. Although some shells provide advanced interfacing capabilities and others can be embedded within a conventional application, most cannot be easily integrated as closely-coupled components of a larger problem-solving system.

This paper discusses the requirements of a problem-solving architecture that can:

- be tightly embedded within other architectures and
- coexist with multiple instances of itself and of other problem-solvers.

The additional effort needed to produce an embedable problem-solving architecture is minor, compared to the substantial increase in applicability of the architecture.

A specific need for embedable problem-solvers arose with the UMass Generic Blackboard Framework (UMass GBB). UMass GBB is based on the blackboard paradigm, which naturally integrates heterogeneous problem-solving representations as individual knowledge sources (KS's). In principle, in UMass GBB, a KS can be written using any embedded AI shell. Thus, this need was pursued by developing general specifications for embedable problem-solving architectures, and then the specifications were used to modify the public-domain version of OPS5 in order to embed it as an integral KS language within UMass GBB.

**Index Terms**—AI languages, blackboard systems, Common Lisp, interfaces, OPSS, rule-based systems, system embedding, UMass GBB.

## I. INTRODUCTION

A SINGLE knowledge representation and reasoning system is seldom ideal for all aspects of an ambitious AI application. In such an application, some portions of the application might be well suited to one approach while others might be suited to a very different approach. A problem-solving environment that provides an integrated collection of representation languages and reasoning techniques addresses the need for problem-solving heterogeneity. To function as elements of an integrated collection, the individual systems must be amenable to operating with other problem-solving systems in the context of a larger problem-solver.

Integrating multiple expert systems and problem-solving representations becomes increasingly important as the scope of AI applications grows beyond restricted domains. General approaches to integration include:

- *Emulation*: Select one problem-solving framework as the *base language* and use it to implement the other frameworks. This approach requires considerable effort in order to implement all the desired frameworks in a single representation, and the emulation effort must be repeated whenever a different base language is selected. Performance can also

Manuscript received March 2, 1990; revised August 20, 1990. This work was supported in part by gifts from Texas Instruments, Inc., by the National Science Foundation under CER Grant DCR-8500332, and by the Office of Naval Research under a University Research Initiative Grant (Contract N00014-86-K-0764).

The author is with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.  
IEEE Log Number 9042065.

suffer as frameworks are recorded (with varying quality and optimization) in the base language rather than their original implementation language.

- *Distribution*: Place each problem-solving framework in a different process or machine and exchange messages among them. This approach often involves distinct processes or even networked machines that interact using communication protocol suggestive of a blackboard system. Such a paste-up approach increases hardware costs; adds to system complexity; and adds communication overhead. For example, using distributed system techniques to implement a heterogeneous blackboard application ignores the important issue of opportunistic control of individual KS activations, and typically results in poor resource utilization and unfocused problem-solving activity.
- *Embedding*: Convert AI representation languages and shells to callable modules that support multiple independent knowledge bases and local working-memories. The embedding approach requires some additional effort in implementing a problem-solving architecture, or in modifying an existing one. However, once a problem-solver is embedable,<sup>1</sup> it can be easily integrated with other problem-solvers.

In general, embedding is the preferred solution to attaining heterogeneous language representations.

The form of embedding we consider here, in which a problem-solver is made coresident with other problem solvers in the same Common Lisp environment, differs from integrating an AI shell as the sole problem-solving component of an application. We are concerned with closely coupled problem-solvers that share a common (Common Lisp) address space, and can easily access and manipulate shared data. We are also concerned with issues of debugging and refining the individual problem-solvers in the context of the entire application. Thus, the issues we address involve more than constructing a problem-solving architecture that can be invoked, as a subroutine, with a few parameter values and that returns a few result values on completion.

We began developing specifications for embedable problem-solving architectures with a simple goal: integrating a classic rule-based AI language (OPS5 [1]) into the UMass Generic Blackboard Framework (UMass GBB) [2], [3]. The UMass GBB system contains a high-performance blackboard database compiler and run-time system that can be extended into a complete blackboard-development framework by selecting a control shell and appropriate knowledge-source (KS) representation languages. In early versions of UMass GBB, application developers coded KS's directly in Common Lisp using functions provided by GBB's blackboard-database run-time system. Although Common

<sup>1</sup>Whether the term "embedable" is a suitable extension of English is controversial. We acquiesce to increasingly common practice by using embedable to succinctly describe an architecture that can be embedded within a larger system.

Lisp is an appropriate language for coding computational KS's, other KS's are expressed more naturally using rule- or frame-based AI languages. In addition to domain-independent KS languages, a particular application may require a domain-specific reasoning system as a KS language (for example, a model-based diagnosis module).

Some blackboard frameworks provide a single, specialized language for writing KS's [4], [5]. We envisioned a different approach with UMass GBB. Why not allow KS's to be coded in the most appropriate AI language? Thus, KS's could be written in native Common Lisp, OPS5, Prolog, and other popular AI shells and languages. In this way, the independence of KS's within the blackboard paradigm is extended from the knowledge itself to the language used to codify the knowledge.

Such independence among KS's is a restricted form of the multiarchitecture integration approach exemplified by the ABE project [6]. While ABE uses a recursively defined module hierarchy that contains primitive language-framework modules as its leaves, the KS language approach presented here integrates modules as a single-level structure within the blackboard KS framework.<sup>2</sup>

The cooperating KS model of the blackboard framework is well suited for integrating heterogeneous problem-solvers, and it reduces the complexity of embedding a language framework within a blackboard-based application. From the viewpoint of the blackboard's control components, a KS written in an embedded AI language is no different from a KS written in Common Lisp. Both are modules with specific calling conventions that perform blackboard read/write operations during execution and that return values to the control component upon completion. Integrating heterogeneous problem-solvers as individual KS modules simplifies the integration effort (as compared to ABE), but it limits, to individual KS's, the choice of representation. For blackboard architectures, coupling KS modularity with language integration is a reasonable strategy, and it is the approach we consider in this paper.<sup>3</sup>

With our vision of embeddable KS languages in place, we began modifying OPS5 to make it UMass GBB's first embedded KS language.<sup>4</sup> Although conceptually simple, achieving an integration of OPS5 with UMass GBB required dealing with a number of implementation incompatibilities. Use of OPS5 with UMass GBB also highlighted issues related to debugging and refining knowledge represented in OPS5 KS's, in the context of the encompassing application. Our efforts have resulted in both a successfully integrated OPS5 KS language and an improved understanding of the requirements for embedding other AI languages.

The OPS5 integration effort is described below, followed by a general specification for integrating embeddable AI shells and languages.

## II. OPS5 AND UMASS GBB

In a blackboard system (Fig. 1), individual KS's are triggered in response to a specified blackboard activity (such as the creation

<sup>2</sup>Our specifications for integrating problem-solving architectures (Section IV) are similarly related to ABE's black box framework.

<sup>3</sup>The interface specifications we will present are not limited to KS language modules and, in fact, apply to the use of AI shells as callable inference engines. For example, the Blondie-III blackboard system provides a callable rule-based engine that can be invoked from procedural code with a named rule base [7].

<sup>4</sup>OPS5 was selected as the initial KS language for use in UMass GBB, for reasons that included popularity and publicly-available Common Lisp source code. All remaining references to OPS5 pertain to this public version.

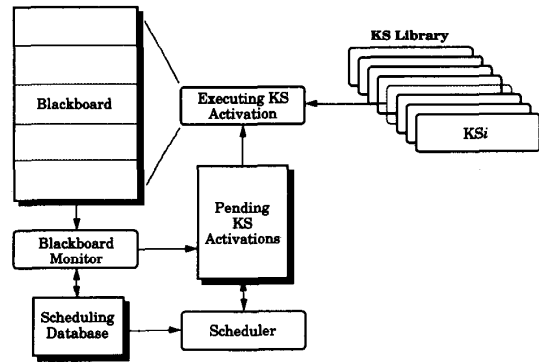


Fig. 1. Basic blackboard architecture.

or modification of a particular class of blackboard object). If conditions warrant execution of the KS, a KS activation is created and scheduled for execution. A KS activation includes references to the triggered KS, the triggering blackboard objects, and other information that specifies the execution context of the KS. Thus, a KS interested in a particular class of sensory data may be triggered by numerous sensory-input objects and may have numerous activations pending execution, each with a different execution context.

What does an activation of an OPS5 KS look like in UMass GBB? Each OPS5 KS has its own knowledge base (KB) that is shared by all activations of the KS. Each activation of an OPS5 KS has its own private working memory (WM). When a KS activation is invoked, its WM is initialized with stimulus data, and control is transferred to the OPS5 inference engine. At this point, one or more OPS5 rules have been triggered by the insertion of the stimulus data into the activation's WM. Some of these rule activations might perform actions that retrieve other objects from the blackboard and place them or some of their attributes into WM. During execution, the KS activation will likely execute rules that create or modify objects on the blackboard. Finally, the activation will return one or more values to GBB's control shell, thereby indicating the completion status of the KS.

The rules in an OPS5 KS activation are triggered only by activities in the activation's WM. There are important reasons for not allowing OPS5 rules to directly trigger on blackboard activities, such as:

- to avoid the overhead of matching individual rules in pending OPS5 KS activations that are unlikely to be executed as a result of other problem-solving activities
- to avoid the issue of matching OPS5 rules in KS's that have not been triggered and that, therefore, do not have pending KS activations and WM instances.

Therefore, the KB and WM of an OPS5 KS activation are conceptually and implementationally distinct from the blackboard as shown in Fig. 2. This separation preserves the modularity of the blackboard paradigm's cooperating, independent KS model.<sup>5</sup> The private computations held in WM are not seen by other activations of OPS5 KS's or by any other KS's in the blackboard application. Similarly, the OPS5 inference engine's conflict-resolution strategy and the blackboard's KS-scheduling cycle are completely independent.

<sup>5</sup>The MUSE embedded-system blackboard framework [8] enforces a similar memory-separation structure.

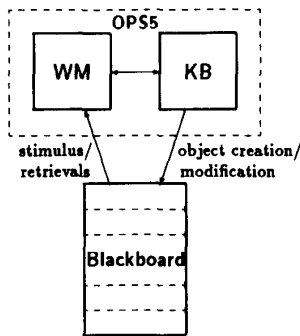


Fig. 2. UMass GBB's blackboard and OPS5's knowledge base (KB) and working memory (WM).

Depending on the UMass GBB control shell used, activating and invoking an OPS5 KS can require a number of KS activities. In this paper, we use UMass GBB's *simple shell*. The simple shell implements the precondition/action KS model first used in the Hearsay-II speech understanding system [9]. In this control model, a KS is composed of two distinct modules: a precondition procedure and an action procedure. The precondition procedure is invoked on a *stimulus* blackboard object and typically uses the stimulus object as a context for searching for other relevant blackboard objects. If sufficient data are found, the precondition procedure returns a local estimate of the importance of scheduling the action portion of the KS. It is useful to allow either the precondition procedure, the action procedure, or both, to be coded in OPS5. The two procedures are effectively distinct modules (with distinct KB's and WM's), coupled only by a data-passing convention in the *stimulus/response* frame of the KS activation.

Another way of looking at the relationship between UMass GBB and OPS5 is to consider OPS5's three major components: the OPS5 inference engine and support code, the KB, and the WM. The OPS5 inference engine must be loaded into Common Lisp.<sup>6</sup> A KB-library entry is defined and maintained for each OPS5 KS precondition or action procedure. Finally, every OPS5 KS precondition or action procedure that is active (that is, initiated but not completed) requires a distinct WM instance. These component relationships are illustrated in Fig. 3.

Some UMass GBB control shells buffer all *blackboard events* (the triggers for KS activation) until the end of the currently executing KS. In those shells, KS action procedures written in OPS5 are not interrupted by the execution of precondition procedures. This is not the case with all control shells.

To support multiple, interruptible procedures, additional embedding efforts are required. We define three embedding levels, which are based on KB-library entry and WM instance-execution relationships, and which are increasingly powerful:

- *Serially reusable*: A serially reusable embeddable-language architecture must be able to be invoked repeatedly on a library entry of a KB with an appropriately initialized WM.
- *Interruptable*: An interruptable embeddable-language architecture must be able to be suspended while another library entry of a KB and WM are processed.<sup>7</sup> The computation

<sup>6</sup>Since both UMass GBB and the public domain OPS5 system are written in Common Lisp, a tight coupling (desired for efficiency) requires that they reside in the same Common Lisp heap.

<sup>7</sup>This issue is local to the internals of the KS language architecture. Synchronization and blackboard context problems associated with suspending and resuming KS activations are handled at the blackboard control-shell level.

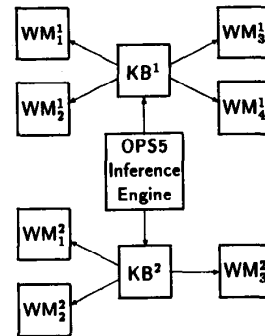


Fig. 3. Instances of the three major OPS5 components.

can be resumed when the second activation is completed or interrupted. An interruptible system requires the ability to quickly save and restore the execution context of the language architecture.

- *Interactive*: An interactive embeddable-language architecture is able to asynchronously receive and assimilate changes to its WM, whether the changes were created directly by another process or by external trigger events. The executions of interactive language architectures operating on a single processor are time-shared by a process scheduler. On a multiprocessor, multiple interactive architectures can be executing simultaneously. In either case, the inference engine in an interactive language architecture cannot assume it has sole control over WM contents.

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

### III. THE OPS5 MODIFICATIONS

Changes required in OPS5 to integrate it within UMass GBB included:

- support for a *library* of multiple, independent KB entries that can be individually invoked with appropriately initialized WM instances
- support for calling back GBB to perform blackboard read/write operations
- support for the return of values by OPS5 to its caller
- support for tracing, stepping, and debugging individual OPS5 invocations.

These changes are detailed below.

*Multiple KB's and WM's*: As with most AI shells and languages, the publicly available OPS5 implementation does not support multiple KB-library entries or WM instances. To achieve this capability, we collapsed all global information associated with an OPS5 KS into KB-library entry and WM context objects.<sup>8</sup> Therefore, running OPS5 on a particular KB library entry and WM instance simply required supplying the appropriate context objects to the OPS5 inference engine.

We also provided a means for defining, compiling, storing, invoking, and potentially redefining each KB-library entry. Each KB entry is named, and a particular KB entry is defined by enclosing the rules within a KB definition form, as follows:

```
(define-kb 'ops name (rule1 rule2 ... rulen)).
```

<sup>8</sup>The original implementation contained well over 200 global variables! Many were eliminated using Common Lisp lexical binding techniques. Property list information was also moved into the appropriate context objects.

```

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

```

Fig. 4. An OPS5 rule.

*Call Back and Lists:* When invoked, an OPS5 KS precondition or action procedure uses UMass GBB's run-time blackboard routines. Since both UMass GBB and OPS5 are coded in Common Lisp, interaction might appear to be a simple issue. In fact, even with OPS5's external routine capabilities, interfacing with UMass GBB required substantial modifications.

For example, UMass GBB's principal blackboard-retrieval operator is **find-units**. In **find-units**, a blackboard retrieval pattern is specified as a nested list of retrieval specifications and data values. However, since OPS5 does not support a list data type, how could an OPS5 KS directly represent a retrieval pattern? If lists were not added to OPS5, then a special interface between OPS5 and **find-units** would be needed.

Further, **find-units** returns a list of the retrieved blackboard objects. In this case, the need for a list data type could be circumvented by entering each of the returned objects as a separate WM element. However, this approach hides the common retrieval relationship among the returned objects. Another approach is to use an OPS5 vector-attribute to contain the returned items. The disadvantage with this approach is lack of control over the number of retrieved elements and, therefore, a potentially unbounded WM element size. Simply adding the returned items as individual WM elements is also unacceptable, because the relationship among the items is lost.

Since lists were needed to represent UMass GBB patterns, we extended OPS5 to include a list data type that contained a special header that makes each list appear atomic to OPS5. We added the following pseudo-list operators to OPS5: **\$cons**, **\$first**, **\$list**, **\$quote**,<sup>9</sup> and **\$rest**.

The external routine interface in OPS5 is cumbersome to use from a Common Lisp environment. In particular, a subroutine must explicitly manage the passing of values through OPS5's *result element*. This would require GBB's run-time routines to know when they are invoked by a call back from OPS5. Instead, we added a new OPS5 operator, **cl-call**, that automatically manages result-element values, allowing any number of evaluated arguments to be passed to an external Common Lisp function. Multiple values returned by the Common Lisp function are automatically placed into the result element for extraction within OPS5. **Cl-call** also transparently supports the pseudo-list data-type extensions.

Fig. 4 contains an example of an OPS5 rule that illustrates the use of the **cl-call** and pseudo-list operators.

*Result Values:* A new operator, **return-values**, was added. This operator terminates the OPS5 recognize-act cycle and returns

multiple values to the calling routine. It returns nil if the last recognize-act cycle evaluates an OPS5 **halt** operator or if no further rules remain to be fired.

*Tracing, Stepping, and Debugging:* Tracing, stepping, and debugging an OPS5 KS execution is also an issue. OPS5 provides its own interactive command loop, watch, and trace facilities. In the modified OPS5, these facilities are disabled by default, and they are selectively enabled on specified OPS5 KS activations. Our first approach was to extend the OPS5 **initialization** command to specify whether an invocation of the KB should enter OPS5's interactive command loop for debugging purposes.<sup>10</sup> The **initialization** command is also used to control the tracing (watch) level and the enabling of history recording (for backing up).

There are two disadvantages to this approach. First, a single KB activation can be invoked many times during a blackboard application (once per KS activation). Only a particular invocation may be of interest for debugging. This means that the KB-specific, command-loop entry predicate, specified in the OPS5 **initialization** command, must be conditional on the initial contents of WM. The second disadvantage is that the debugging information is kept within the KB-library entry itself. To enable or disable debugging, the desired KB(s) must be individually edited or redefined. However, a global specification of debugging needs is more appropriate. By having each language architecture provide a facility for specifying and modifying the debugging and tracing specifications for its KB-library entries, language-specific details are retained within the language module, but general debugging and tracing specifications can be provided by the parent system (see next section).

#### IV. AN INTERFACE FOR EMBEDDABLE ARCHITECTURES

What have we learned from our modifications of OPS5? What generalizations can we make for repeating the procedure with other AI shells and languages?

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

- The ability to define and maintain a library of multiple, independent KB instances.
- The ability to be called as a subroutine with a particular library entry of its KB on an appropriately initialized WM.

<sup>10</sup>If the OPS5 debugging command loop is enabled, the OPS5 invocation does not immediately return when a **return-values** or **halt** command is evaluated or when there are no further rules to execute. Instead, an explicit command-loop exit command must be entered. This requirement is especially useful when investigating rule-firing stagnation.

<sup>9</sup>OPS5's // operator also functions as **\$quote** for pseudo-lists.

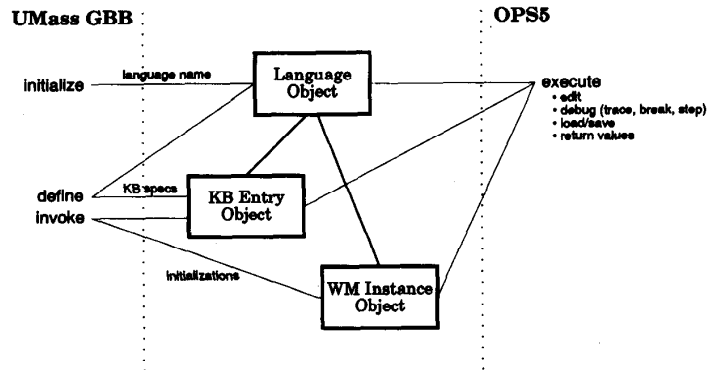


Fig. 5. Interface relationship between UMass GBB and OPS5.

- The ability to call back the calling routine. UMass GBB required that the call back capability support an extensible foreign data-structure capability (including lists). The results returned from a call back must be available for use by the inference engine.<sup>11</sup>
- The ability to return values to its caller.
- The ability to debug, step, and trace individual KB invocations.

These capabilities are represented in the following interface specification for serially reusable, embeddable-language architectures. By extending each language architecture to meet these interface specifications, the details of managing KB's and WM's are hidden within the embedded architecture, yet they remain accessible from the encompassing system (Fig. 5). Such modularity is important, especially if proprietary KB representation mechanisms are to be used within a larger system.

*The Interface Specifications:* The following interface specifications are presented from a Common Lisp perspective and use Common Lisp Object System (CLOS) capabilities:

**initialize-language** *language* [Function]  
Performs all initializations for the specified KS representation language, *language*, except those that are specific to a particular KB-library entry of the language. This function is called once, regardless of the number of KB-library entries that are to be defined or invoked for the language. Furthermore, this function must be called before any of the related generic functions. The function returns a CLOS object representing *language* (used for method dispatching).

**define-kb** *language-instance kb-name* & optional *kb-declarations compile-p* [Generic Function]  
Creates and initializes a new KB-library entry named *kb-name* in the KS representation language specified by *language-instance*. If *kb-name* already exists, the function redefines it on the basis of the KB initialization data specified in *kb-declarations*, if any. If *compile-p* is true and the specified KS-representation language supports KB compilation, the KB-library entry is compiled. The function returns a KB-library entry object.

**delete-kb** *language-instance kb* [Generic Function]  
Deletes the *kb* entry from the KS representation language specified by *language-instance*. The resulting state of the language is as if the deleted KB-library entry had never been

defined (that is, all signs of its existence are erased).

**edit-kb** *language-instance kb* & optional *compile-p*

[Generic Function]

Instructs the representation language specified by *language-instance* to provide interactive editing support for its *kb* entry, if the language supports interactive KB editing. If *compile-p* is true and the specified KS representation language supports KB compilation, the KB-library entry is compiled when editing is completed.

**invoke-kb** *language-instance kb initialization-forms*

[Generic Function]

Instructs the KS representation language specified by *language-instance* to begin executing using KB *kb* and to initialize a new WM instance on the basis of *initialization-forms*. Note that the value of *initialization-forms* is dependent on the language. The function returns, as multiple values, *result* values specified in the **exit-kb** generic function.

**exit-kb** {*result*}\*

[Generic Function]

Performs cleanup activities for the invocation of the KB-library entry that is currently executing.<sup>12</sup> The *result* values are returned, as multiple values, by the **invoke-kb** generic function.

**debug-kb** *language-instance kb* & optional [Generic Function] *{debug-specification}*+

Causes the KS representation language specified by *language-instance* to prompt the user for information on how invocations of the *kb* entry are to be debugged.

Optionally, by using the *debug-specification* argument, information about tracing, stepping, and breaking techniques can be specified. These techniques are especially important when multiple KB-library entries of multiple KS representation languages are used within the same application. The form of *debug-specification* follows:

[trace *level pred*]  
[break *level pred*]  
[step *level pred*]

where values for *level* are as follows:

- For trace:
  - :OFF—Do not trace.
  - :MINIMAL—Do minimal tracing.

<sup>11</sup> OPS5 does not allow external calls to be placed in the left-hand side of rules, an inconvenience that was not remedied.

<sup>12</sup> The OPS5 **return-values** operator is implemented using **exit-kb**.

- :AVERAGE—Do average tracing.
- :MAXIMAL—Do maximum tracing.
- For break:
  - :OFF—Do not break.
  - :ENTRY—Break at entry.
  - :EXIT—Break at exit.
  - :BOTH—Break at entry and exit.
- For step:
  - :OFF—Do not step.
  - :STEP—Step.

An arbitrary predicate can be specified using the optional *pred* argument, which supplies conditional information for the debugging specifications that are specific to the KS representation language.

**load-kb** *language-instance kb-name file-name*

[Generic Function]

Instructs the KS representation language specified by *language-instance* to load KB data from the *file-name* file into the *kb* entry. The function returns the KB-library entry object.

**save-kb** *language-instance kb file-name* [Generic Function]

Instructs the KS representation language specified by *language-instance* to save KB data from the *kb* entry into the *file-name* file.

*Additional Specifications:* In addition to the interface functions described above, the following capabilities are required within the embedded system.

- *The ability to call external routines.* In GBB's case, this includes the ability to construct and manipulate list data structures.
- *The ability to return values to the calling routine.* UMass GBB examples include returning a rating from an OPS5 precondition procedure and returning a termination indicator from an OPS5 KS action procedure.

The details of these two capabilities are specific to the particular AI shell or language.

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

```
(defun RUN-PRECONDITION (language-object ks-kb
                        stimulus)
  (invoke-kb language-object ks-kb
    '((make (stimulus ,stimulus))))).
```

## V. SUMMARY

A problem-solving architecture can be improved by making it embeddable. The embedding approach we presented uses the KS independence of the blackboard paradigm to integrate diverse problem-solving architectures. To demonstrate this approach, we embedded OPS5 within the UMass Generic Blackboard Framework (UMass GBB). Although the task of modifying the publicly available OPS5 implementation to make it embeddable required several programmer weeks, designing an AI shell or language with embedding requirements in mind will not significantly complicate the shell's implementation nor reduce its efficiency. The OPS5 modifications have resulted in an easily integrated GBB KS language (distributed with the UMass GBB system) that has been used in several GBB applications.

The OPS5 modifications highlighted some implementation guidelines for developing an embeddable system. Many of these

guidelines are simply good programming style, but we list them anyway:

- Avoid global variables. Where global values are required, implement them as a value in the appropriate KB or WM object.
- In Common Lisp, avoid the use of property lists. Again, use values in the appropriate KB or WM object. This includes language-parameterization information, switches, and so on, whose values may vary with different KB instances.
- In Common Lisp, allow KB and WM specifications to reside in user-defined packages. For example, do not require rule definitions to be made in a particular package.
- Support directly, or through user-level interface routines, all of the datatypes used in the underlying implementation language. (A Lisp-based language should support lists!)
- Be careful with input/output streams. Either use streams passed by the calling system or open fresh streams for each invocation. Do not assume a stream can be shared by all instances of a language.
- Assume that, if the system calls another system, it may be called back by that system.
- Assume that, if the system is to be called by another system, there may be need to call back that system.
- If possible, perform error handling within the embedded system, rather than expecting the calling system to deal with the error.

The interface specifications we presented were refined and tested by the OPS5 embedding experience. Although originally developed specifically for UMass GBB and OPS5, the issues of managing knowledge-base libraries, invoking and exiting a language, and debugging apply to the general problem of integrating heterogeneous problem-solvers into larger systems. These interface specifications are also being used to embed proprietary and third-party KS languages into the commercial GBB product.

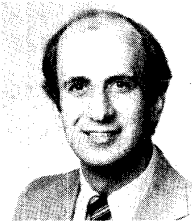
Finally, the lessons learned from embedding OPS5 are being applied to GBB itself. When completed, multiple independent blackboard systems, with separate blackboard databases, KS's, and control shells, will be able to be coresident. One known use for this capability is allowing a developer to multiplex programming effort among several applications by simply switching KB and WM context objects.<sup>13</sup>

## REFERENCES

- [1] C. L. Forgy, "OPS5 reference manual," Tech. Rep. CMU-CS-81-135, Comput. Sci. Dep., Carnegie-Mellon Univ., Pittsburgh, PA, 1981.
- [2] D. D. Corkill, K. Q. Gallagher, and K. E. Murray, "GBB: A generic blackboard development system," in *Proc. Nat. Conf. Artif. Intell.*, Philadelphia, PA, Aug. 1986, pp. 1008-1014. (Also in *Blackboard Systems*, R. S. Englemore and A. Morgan, Eds. Reading, MA: Addison-Wesley, 1988, pp. 503-518.)
- [3] K. Q. Gallagher, D. D. Corkill, and P. M. Johnson, "GBB reference manual," Dep. Comput. Inform. Sci., Univ. of Massachusetts, Amherst, MA 01003, GBB Version 1.2 ed., Sept. 1988. (Published as Tech. Rep. 88-66, Dep. Comput. Inform. Sci., Univ. of Massachusetts, Amherst, MA 01003, Sept. 1988.)
- [4] A. Garvey, M. Hewett, M. Vaughan Johnson, R. Schulman, and B. Hayes-Roth, "BB1 user manual," Knowledge Systems Lab., Dep. Medical and Comput. Sci., Stanford, CA 94305, Common Lisp ed., Oct. 1986. (Published as Working Paper KSL 86-61, Knowledge Systems Lab., Dep. of Medical and Comput. Sci., Stanford Univ., Stanford, CA 94305.)

<sup>13</sup> For GBB, the blackboard database (including scheduling data structures) is the WM, and the KS's and control-shell components form the KB.

- [5] L. Baum, R. Dodhiawala, and V. Jagannathan, "Boeing blackboard system, version 1.0," Tech. Rep. BCS-G2010-31, Boeing Computer Services, P.O. Box 24346, Seattle, WA 98124, July 1986.
- [6] F. Hayes-Roth, L. D. Erman, S. Fouse, J. S. Lark, and J. Davidson, "ABE: A cooperative operation system and development environment," in *AI Tools and Techniques*, M. Richer, Ed. Norwood, NJ: Ablex, 1988. Also in *Readings in Distributed Artificial Intelligence*, A. H. Bond and L. Gasser, Eds. Los Altos, CA: Morgan Kaufmann, 1988, pp. 457-489.
- [7] G. van Liempd, H. Velthuisen, and A. Florescu, "Blondie-III," *IEEE Expert*, vol. 5, no. 4, pp. 48-55, Aug. 1990.
- [8] D. Reynolds, "MUSE: A toolkit for embedded, real-time AI," in *Blackboard Systems*, R. S. Engelmore and A. Morgan, Eds. Reading, MA: Addison-Wesley, 1988, pp. 519-532.
- [9] I. D. Erman, F. Hayes-Roth, V. R. Lesser, and D. Raj Reddy, "The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty," *Comput. Surveys*, vol. 12, no. 2, pp. 213-253, June 1980.



**Daniel D. Corkill** (M'87) received the B.S. and M.S. degrees in computer science from the University of Nebraska, Lincoln, in 1975 and 1976, respectively, and the Ph.D. degree in computer and information science from the University of Massachusetts, Amherst, in February 1983. His Ph.D. research used organizational structuring as a framework for coordinating the activities in a distributed network of problem-solving systems. While completing his Ph.D., he performed major design, implementation, and experimentation on

the Distributed Vehicle Monitoring Testbed (DVMT), a distributed interpretation testbed based on a network of communicating blackboard architectures.

Since 1983, he has been a Research Computer Scientist and a Senior Research Computer Scientist in the Department of Computer and Information Science at the University of Massachusetts. In these positions, he has headed two major AI software development projects: UMass Parallel Common Lisp and UMass GBB. UMass Parallel Common Lisp is a high-performance, shared-memory, multiprocessing Common Lisp system for Sequential Symmetry multiprocessors. Begun in January of 1987, this work was part of NSF/CER-supported research in cooperative distributed and parallel processing, for which he headed the Lisp/AI Development Group. The system was completed in June 1988, and it has been licensed for commercial development and support to Top Level, Inc., where it has been ported to other shared-memory multiprocessors. The UMass GBB (Generic Blackboard) system is a high-level implementation tool that provides both speed and flexibility in building blackboard-based AI applications, as well as efficient execution of the resulting applications. The UMass GBB effort was begun late in 1985, and versions have been provided to over 300 sites worldwide. UMass GBB has also been extended to distributed and parallel blackboard frameworks. He is a founder of Blackboard Technology Group, Inc., a full-service vendor of blackboard technology and support services. Blackboard Technology Group markets GBB Version 2.0, an enhanced version of the UMass GBB system. His research interests include blackboard architectures; parallel Common Lisp; coordination in distributed problem-solving networks; planning and control in large AI systems; design and programming methodologies for constructing large AI systems; and hardware and software support for AI systems.

Dr. Corkill is a member of AAAI and ACM.