INVITED PAPER

Blackboard Systems at the Architecture Level

H. PENNY NII
Knowledge Systems Laboratory, Computer Science Department, Stanford University, Stanford, CA

Abstract—Since Hearsay-II, many application systems using the blackboard architecture have been built. Some of these applications have been built using expert system shells or tools. These tools contain the skeletal constructs of blackboard systems into which the user adds the necessary information and knowledge from the application domain to create an application system. Although the tools are useful in the implementation phase, they provide little help in the design of the system, which must necessarily precede the implementation phase. The determination of the blackboard levels, the knowledge sources, and the details of the control strategy is a task reserved for the knowledge engineer/system designer. In order to automate the design process, one needs to understand the relationships between the architectural constructs and domain information that provide the semantics of the architecture. In this paper the authors explore the components of blackboard systems from the perspective of the type of knowledge that goes into the makeup of blackboard systems at the architecture level. They describe a research system, KASE, that captures the various knowledge needed for design and applies the knowledge to aid knowledge engineers design blackboard systems.

1. INTRODUCTION

The terms "blackboard model," "blackboard architecture," and "blackboard system" are associated with a particular class of software programs. An attempt was made in Nii (1986) to distinguish the differences between a model, a framework (or, an architecture), and an application system. The model was viewed as a conceptual problem-solving entity consisting of two basic components, a global data base called the blackboard, and knowledge sources that created and modified the contents of the blackboard. The model includes a description of its behavior. An attempt at abstracting the software constructs of various application systems resulted in a blackboard architecture. In addition to the blackboard and the knowledge sources, a blackboard architecture includes a control component that enables the realization of the behavior in a serial computing environment (see Fig. 1). The application systems can be described as the instances and/or specializations of the architecture. The description of the application task and the associated task domain serves to establish a vocabulary to describe the specifics of the blackboard data structure, the knowledge source contents, and the problem-solving behavior of the different systems. That is, the application domain provides the semantics of the architecture.

There has been a steady increase in the number of blackboard systems [a recent example is Springer, Buta et al. (1991); older examples can be found in Engelmore & Morgan (1988); Corkill & Gallagher (1989); Jaganathan, Dodhiawala et al. (1989); Velthuijsen (1992)], but the designs of these systems continue to fit within the blackboard architecture described earlier. Although the implementation detail may differ greatly, the basic architecture of the blackboard systems has not changed over the years. This situation provides us with an opportunity to study the various properties of architecture, in particular, the blackboard architecture. What is an architecture? What is its form? What implicit information is embedded within an architecture? And, of particular interest to the knowledge system community, what kinds of knowledge are needed to design a system, and what roles do these different kinds play?

KASE (Knowledge Assisted Software Engineering) is a research project whose goal is to build a set of power tools for software analysts and designers at the architecture level. The tools should contain sufficient knowledge about the artifact and the design process to enable the automation of many of the activities in-

Requests for reprints should be sent to H. Penny Nii, Knowledge Systems Laboratory, Computer Science Department, Stanford University, Stanford, CA 94305, USA.
volved in design. Although its focus has been on answering the questions posed above for architectures in general, KASE has used a blackboard architecture as its first test bed. From the point of view of blackboard systems, the project represents an attempt to codify various kinds of knowledge about blackboard systems.

In this paper we describe the constituent components of an architecture in terms of the knowledge domains needed to construct an architecture. In particular, we begin with the contents of a generic blackboard architecture. We than describe a knowledge-based technique for specializing the generic architecture to create application-specific architectures using domain information.

2. THE ARCHITECTURE LEVEL

A software architecture is an abstraction of software systems. It comprises a topology of functional components, the interfaces between the components, and the semantic contents of these constituent parts. An architecture is usually described hierarchically to show different levels of detail, and at the most detailed level it must have sufficient detail to serve as a guide or specifications to the coding process.

Often, an architecture is treated as one of multiple views of a system. In LaSSIE (Devanbu, Brachman et al. 1991), for example, the system is partitioned into a domain model view, an architectural view, a feature view, and a code view. Others, for example Iscoe, Williams et al. (1991), focus on the domain model as a basis for software design. The difficulty with separating the domain model and the architecture is that the domain information forms the basis for the semantics of the architecture and must be expressible in terms of, as well as within, the computational constructs that make up the architecture.

A software architecture is a product of integrating knowledge from multiple knowledge domains. In her study of human software designers, Guindon (1990) found that designers drew on many kinds of knowledge, including knowledge about the domain, knowledge about problem solutions and their representation, and knowledge about design schemas and computational constructs.

In KASE, an attempt has been made to identify in various software architectures their constituent knowledge domains and to discover the relationships among them. The objective is to rationalize software designs in order to identify useful knowledge-based design tools to help in the process of design synthesis. KASE begins with a generic architecture. A generic architecture is an abstract architecture designed to solve a particular class of problems (Bhansali & Nii, 1992a). KASE has been working with a class of problems called tracking problems in which the task is to identify and track objects in space based on signal data.

A generic architecture represents design decisions and, by its very nature, is associated with certain architectural commitments. The design decisions are guided by knowledge in the following areas:

1. A problem, or a task, class. The selection of a problem class establishes an ontology of generic terms with which to describe the task, and the objects and the operations in the problem class. Diagnostics, tracking, payroll, and transaction processing are examples of problem classes.

2. A solution strategy. An architecture is a commitment to solve a problem using a particular problem-solving strategy. Thus, there may be more than one architecture associated with a problem class representing different ways to solve the problem. One way to solve a tracking problem, for example, is to cumulate incrementally and opportunistically evidence for the identity and the behavior of the objects being tracked.

3. Design schemas (architectural constructs). The solution strategy is realized using a variety of design schemas, such as pipelines and layered hierarchies. [See Shaw (1991) for a description of architectural constructs.] The incremental and opportunistic problem-solving strategy for tracking problems can be realized using a blackboard architecture, which in turn is made up of what Shaw calls "interpreter" constructs.

Figure 2 shows how the decisions within these knowledge domains determine the shape of a generic architecture. In short, a generic architecture represents commitments to (1) a particular class of problems, (2) a solution strategy for that class, and (3) the architectural constructs used to realize the strategy.

The process of designing a specific application is a process of mapping domain information onto the architecture. To automate this process, the generic architecture needs two types of associated knowledge.

1. Customization knowledge. Associated with a generic architecture is knowledge about how to customize
the architecture into application-specific architectures. This knowledge includes how to instantiate components of the architecture in terms of the specific functionalities of the application domain. Customization knowledge captures design knowledge that is specific to the architecture. For example, the customization knowledge for the problems using the blackboard approach may include a heuristic: “Each intermediate data class needed to solve a problem should be a level in the blackboard data structure.”

2. Meta-model. Also associated with a generic architecture is an ontology of generic terms established by the commitment to a problem class. Called a meta-model, it contains the necessary vocabulary for describing the task and the domain. A domain model for a specific application is built by instantiating this ontology. For the tracking problem class, the generic terms include “objects-to-be-tracked” and “signal-data,” as well as their generic properties, such as, “location” and “speed.”

An application-specific architecture can be defined as a generic architecture instantiated with information from the application domain (Figure 2).

3. BLACKBOARD SYSTEMS REVISITED

In this section we revisit blackboard systems in terms of the concepts just described. We describe the various commitments inherent in blackboard systems by reviewing the task characteristics that are suited to blackboard architecture. In Section 4 we discuss the differences in blackboard architectures that can be attributed to different solution strategies. In Section 5 we discuss some of the ways in which the components of blackboard systems can be designed semi-automatically. Finally, in Section 6 we describe, using two examples from a tracking problem class, automated tools from KASE for designing specific blackboard systems from a generic architecture.

3.1. Task Characteristics

The applications for which the blackboard approach has been useful include signal understanding, vision, and situation assessment (Engelmore & Morgan, 1988; Jagannathan, Dodhiwala et al. 1989). These applications all share common characteristics:

1. The problems are complex and ill-structured with large solution spaces. Systematic generation of solution candidates is neither possible nor feasible.
2. The solutions require situation-dependent, or opportunistic, invocation of diverse sources of knowledge. That is, control decisions are made while the problem is being solved instead of using a priori established control paths.
3. Often, the problems demand both synthetic and analytic processes. These often translate into bottom-up information fusion or composition, and top-down model-based reasoning.

3.2. Solution Strategies: Search vs. Recognition

One way to categorize a problem-solving approach is by determining whether it uses a search or a recognition method of problem solving. Recognition is a term coined by McDermott and Newell (1982)—“the complete recognition does not search, it knows.” At any particular computational state, instead of generating and evaluating the possible next states as in search, a recognition system simply knows what the next state should be. A classic example of a blackboard system that uses search as its solution strategy is HEARSAY-II (Erman, Hayes-Roth et al., 1980), and one using recognition strategy is HASP (Nii, Feigenbaum et al.,
4. ARCHITECTURAL IMPLICATIONS OF DIFFERENT SOLUTION STRATEGIES

4.1. Search

As stated earlier, search requires at least a generator of the solution space and an evaluator. The evaluation can be of two types: a local look-ahead that evaluates possible future states; and a global evaluator that selects the best operator based on the information provided by the look-ahead or the overall state of the solution. In the HEARSAY-II architecture, the action part of the knowledge source was used as a generator (see Fig. 3). Upon its execution, the action part of the knowledge source placed on the blackboard new hypotheses. The condition part served as a look-ahead for the action part. When executed, it determined the state changes that would be produced if its action part were to be executed. The scheduler decided on the next activity—either to look-ahead or to generate changes; that is, it determined the solution path by evaluating the current state of the solution and a set of possible future states.

In HEARSAY-II, the global evaluation criteria was hidden inside the scheduling module; that is, the scheduling module was a primitive module. Within the search paradigm the quality of the solution may depend on the amount of search conducted, which is also a function of the quality of the knowledge available for evaluation. The premise in HEARSAY-II is that a favorable trade-off can be made between the amount of control knowledge and knowledge source activations—that is, spending time on evaluation pays off in reducing search and in improving the quality of the solution.

4.2. Recognition

The recognition problem-solving paradigm is conceptually simpler than the search paradigm (see Fig. 4). Here, the knowledge base is scanned for a piece of knowledge that can be applied to a state. Of course,

H. P. Nii

1982). BB1 is an example of a shell that allows for the use of both strategies (Hayes-Roth, 1985).

In the search paradigm, each step in problem solving requires an evaluator to select from a pre-enumerated or enumerable next state. Each problem-solving step consists of a generation of operators or state, followed by an evaluation, followed by a selection and an application of the state transforming operator. Often, evaluation precedes generation so that generation can occur selectively. The generator can be algorithmic or heuristic; it can be a legal-move generator or a plausible-move generator. The evaluator can be knowledge-poor (general and weak), or it can be knowledge-rich (task-specific). To date most programs that use search use weak knowledge—the search paradigm is called a weak method.

In HEARSAY-II all the possible states were enumerable by its knowledge sources. For example, given a grammar, a table of all legal adjacent words for each word could be generated. The evaluator, or the scheduler, used weak criteria. For example, criteria from selecting from a number of possible word sequences might use the number of words in the sequence. The knowledge is weak, because it does not use task-specific criteria, or when used, the task-specific criteria are very simple. A hypothetical, knowledge-rich criterion might be: "select a word sequence beginning with a preposition following a noun clause."

In the recognition paradigm, the knowledge base must be scanned for a piece of knowledge that can be applied to the current situation. This is called the match step. Each problem-solving step consists of a match step followed by the application of a state transforming operator. If a piece of knowledge is encoded as a conditional if-then rule, then it contains information about what the next state would be as well as the state transformation under which the state transformation can occur. Most current expert systems are recognition systems. (It should be noted that if-then rules are not necessary for a recognition system, and conversely the mere use of if-then rules does not imply a recognition system. The problem-solving approach is independent of the knowledge representation.)

In HASP the search paradigm was tried and failed, because there was no generator of the solution space. It was one of the first complete recognition systems. It consisted of 40–50 knowledge sources, each with an average of 10 rules. The active knowledge sources consisted of domain- and situation-specific pieces of knowledge that "knew" what to infer or what action to take for each intermediate solution state. In addition, it had a plan for the application of the knowledge in the form of control knowledge sources.

In the next section we elaborate on the basic differences in system architectures attributable to the differences between the search and recognition methods of problem solving.
there may be many pieces of knowledge applicable for a given state, and a decision must be made whether to apply all the pieces of knowledge or a subset of them. This control decision can be algorithmic (for example, first-in-first-out) or it can be knowledge-based. When the control decision is knowledge-based, there is a certain amount of uniformity between the way decisions are made about which knowledge to apply and about the manipulations of the solution space objects. In recognition-oriented blackboard systems, the condition part of the knowledge source specifies the situations under which it can contribute to the solution state. The action part of the knowledge source changes the solution state. Since the applicable knowledge source for any given situation is predetermined, in the sense that knowledge sources are always associated with situations, the scheduler focuses on the selection of the best region of the solution space (the specific situation) to process next.

In theory, no distinction is made between control and domain-specific knowledge in recognition systems. However, since a domain knowledge source looks at the blackboard data, and a control knowledge source looks at a description of the changes to the blackboard in addition to the specifics of the changes, they are in fact different. It might be said that the control knowledge source has wider scope. Because control knowledge is different from other knowledge sources, systems can be designed to exploit the differences. Informal plans either can be built into the control structure or dynamically generated. In BB1 (Hayes-Roth, 1985) control plans for the execution of the domain knowledge sources are dynamically created.

5. ARCHITECTURAL ALTERNATIVES

Some of the architectural differences between HEARSAY-like and HASP-like systems can be attributed to the different computational methods inherent in the use of search and recognition paradigms. There are also variations in the structure of the components of the architecture driven by the different needs within a problem class or the specific application itself. These different architectural schemas can be stored in a library. The designer can select appropriate design schema for the various component parts of the architecture. For example, as mentioned earlier, knowledge source selection may be algorithmic or knowledge-based, and the library may contain several different knowledge-selection modules. In this section we describe some of the alternative constructs for the major components of the blackboard architecture.

5.1. The Blackboard

In addition to serving as a medium of communication, and thus of cooperation, between the knowledge sources, the blackboard represents a design of the solution. Since it is a data structure, one could not quite call it a problem-solving plan, yet in the process of problem formulation the structure of the blackboard serves as the backbone of the system design. It is where the design process begins. The blackboard partitions the solution space, and these partitions dictate what knowledge sources will be needed to solve the problem. What constitutes a solution, what intermediate solutions are needed or available, what relationships exist among hypotheses, must all be expressed within the blackboard structure.

From the standpoint of structure there is very little variability in the blackboard. The blackboard is organized into levels corresponding to the solution space partitions. These form either abstraction or compositional hierarchies. The objects on a given level share the same properties, or vocabulary.

5.1.1. Levels. As a data structure, the level object holds information about the nodes on its level. For example, it must keep the names of the attributes of the nodes and maintain housekeeping information, such as the number of nodes on the level at any given time. For these reasons, it is often convenient to view the level object as a class, and the nodes on the level as the instances of the class. This has the further advantage that nodes can inherit default property values at node creation time. Also, operators (methods) associated with creation and destruction of nodes can be stored with the level object.

5.1.2. Nodes. Nodes are objects on a level; for example, a particular word in a sentence is a node on the word level in HEARSAY-II. Nodes are created dynamically as they are needed. The vocabulary relevant to the level of analysis is represented as attributes and values; each node on the level shares the same attributes. In an object-oriented programming language slots and slot-values can be used, and there is nothing to preclude the use of record structures available in other languages.
5.1.3. **Attribute Values.** The value of an attribute can have multiple fields; for example, credibility value and/or time stamps. Often it is useful to maintain value history. This is especially true in continuous-input signal understanding programs in which temporal trends are important. In object-oriented systems, a value can be represented as a method that computes it when the slot is accessed.

5.1.4. **Panels.** Panels are useful when more than one hierarchy is needed to represent the solution state data. The most common use of a second panel is to hold control-related information, as in BB1.

5.2. **The Knowledge Sources**

A knowledge source represents knowledge needed to solve a sub-problem. All knowledge sources are event-triggered agents—the activation of knowledge sources are solution-state-dependent, rather than being process-dependent. The implication is that the most important consideration in the design of the knowledge source is its ability to evaluate the current state of the blackboard.

In all blackboard systems knowledge sources are designed with condition and action parts (Fig. 5).

5.2.1. **Condition Part.** The condition part can: (1) evaluate possible future states, or (2) make context-independent or context-dependent evaluation of the situation on the blackboard. In HASP, possible changes to the blackboard were categorized into different "event" types. The name of the event type was used to record a blackboard change. The event name was also used as a triggering condition in the condition part of the knowledge sources. In this type of construct the condition part is context-independent—it does not take into consideration the specifics of the change. Alternatively, the condition part can be a procedure that evaluates the specifics of the change, making the triggering of knowledge sources context-dependent as in HEARSAY-II.

One can think of the condition part as a filter for the activation of the action part. In newer systems, the design of the condition part consists of multi-staged filters. The first stage, called the trigger, is usually a context-independent filter that is followed by a number of context-dependent filters. If the action part of the knowledge source is a collection of rules, then the condition part of the knowledge source can be viewed as a high-level filter for the rule set. Rules have their own condition parts that serve as more detailed filters for the rule actions.

5.2.2. **Action Part.** The action part of a knowledge source makes modifications to the current blackboard state. The action part can be a procedure or a collection of rules. The selection of the knowledge representation scheme is independent of the selection of solution strategies.

Each knowledge source can make one or more changes to the blackboard. In a rule-based knowledge source, multiple changes can come from multiple firing of rules, or from multiple actions on the right-hand-side of rules.

The changes to the blackboard are new solution candidates (and new data) or modifications to the existing solution. Changes also can be made to the control information, as in BB1. In addition to changes to intermediate solutions on the blackboard, state changes can occur from the posting of goals and sub-goals to be achieved (Lesser & Corkill, 1983) or posting of model-based expectations of changes to occur in the future, as in HASP.

5.3. **The Control**

Control is the most complex component of the blackboard architecture, and it can have many architectural variations. The basic function of the control component is to select and apply knowledge sources in the case of recognition-based systems, or the condition and action components of knowledge sources in search-based systems. Major design differences occur in the following areas: (1) schedulable entities—whether parts of the knowledge sources are separate schedulable entities or not; (2) scheduling—whether the focus of attention is based on events or on knowledge sources; (3) noticing—who notices the changes in the blackboard; and (4) control data—where and in what form the control data is stored.

1. **Schedulable Entities:** In a search-based architecture the condition and the action parts of the knowledge sources are scheduled separately. In recognition-based architecture, each knowledge source is scheduled as a single entity. If the condition part is satisfied, then the action part is immediately executed.

   One thing to note is that in the search-based architecture, the blackboard state may change between the time a condition part and its action part are executed. In such a situation the condition part must be re-evaluated.

2. **Scheduler:** There are two bases on which to approach the selection of the next thing to do:
A knowledge source, or knowledge sources, can be selected on the basis of the current state of the blackboard, or more specifically, on the latest changes to the blackboard, which are called events. Called event-oriented scheduling, the focus is on selecting the best solution island on which to base further computation. Most recognition systems take this approach. The basic function of the scheduler is to decide where to work next. Figure 6 shows a simple event-oriented control schema.

b. A knowledge source can be selected on the basis of what the knowledge source can contribute to the current state. This approach, called knowledge-oriented scheduling, is taken most often in search systems. The scheduler decides among the look-ahead evaluators and the generators of the solution space, and it applies the selected components to all the relevant data objects on the blackboard. The basic function of the scheduler is to decide what to do next. Figure 7 shows one possible way to design a knowledge-oriented control.

In reality, many problems can gain from a mixed strategy of search and recognition, and BB1, for example, allows for both event-oriented and knowledge-oriented scheduling.

5.3.1. Posting and Noticing Blackboard Changes. What changes are posted and who notices these changes differ from system to system. Often, in search-oriented systems, every change to the blackboard is a candidate for a solution, and there must be a knowledge source to evaluate each change. On the other hand, event-oriented recognition systems can process only those blackboard changes for which there is available knowledge. By making the event creation explicit, the rule writer can control what blackboard changes merit further action by the knowledge sources. For example, if a new piece of evidence is found to support a hypothesis but the only consequence is that the credibility rating is increased, this change may not need to be an event.
What blackboard changes should be events depend on the application.

5.3.2. Control Data. Control information needed to drive the system always resides outside the blackboard containing the solution state. In HASP different types of changes to the blackboard (modification to nodes, expectation posting, and so on) were stored in separate data structures. Each event was a recording of the type of change, the node that was modified, the new values, the rule that made the modification. In HEARSAY-II the control data was stored on a global list, the scheduling queue. The items on the queue were of two types: information generated by the condition part that pointed to its action part and all the places on the blackboard it could be applied to, and the names of the condition parts of knowledge sources. Since HASP-like systems focus on event scheduling, the control data are primarily about events; in HEARSAY-like systems that focus on knowledge source scheduling the control data are primarily about knowledge sources.

In BB1, control must determine what domain knowledge source to apply and to what object on the blackboard. The control problem is formulated as a planning sub-problem, and its solution state appears on a separate control panel. All data dealing with control are posted on the control panel and are shared by both the control-planning and the domain knowledge sources.

5.3.3. Domain-specific Knowledge, Meta-knowledge, and Control Knowledge. The distinction between domain-specific knowledge and meta-knowledge is not always clear cut. Meta-knowledge, knowledge about knowledge, is often thought to be weak, general, and domain-independent. But, in both search and recognition systems, meta-knowledge is often treated as control knowledge about what to do next, and this type of knowledge is often very domain-specific. In general, making distinctions between domain-specific and meta-knowledge does not contribute much to the design of the system; rather, it tends to distract. Instead it is preferable to organize knowledge sources hierarchically according to some control hierarchy. In most systems the control hierarchy partitions the knowledge sources into those that deal directly with the emerging solution states and those that deal with the process history or problem-solving states, called control states. The knowledge sources that operate on control states are called control knowledge sources.

6. DESIGNING APPLICATION SYSTEMS USING A GENERIC ARCHITECTURE

Up to this point we have discussed the three types of knowledge that are needed to create a generic blackboard architecture—knowledge about the characteristics of a class of problems, its solution strategies, and alternative design schemas to realize the solution strategy. As was shown in Figure 2, a generic architecture has explicitly associated with it (1) a meta-model consisting of a set of concepts and objects to describe the class of problems, and (2) knowledge about how to integrate domain information with the generic architecture to create application-specific architectures. In this section we describe these two items by means of examples from work done on the KASE project.

6.1. A Generic Architecture For Tracking Problems

6.1.1. The Tracking Problem. This class of problems can be described simply as follows: Identify, track, and sometimes, infer the intentions of objects in space using signal information about the objects (see Fig. 8). As examples, we use the HASP and the ELINT (Brown, Schoen et al., 1986) problems. In HASP the objects to be tracked are ships and submarines using passive sonar signals. In ELINT the objects to be tracked are airplanes using radar signals. In both problems, there are multiple sources of signal for each platform (engines and pro-

![Figure 8. The tracking problem.](image-url)
pellers on ships, for example), and the platforms may be clustered together obviating individual platform identification.

A generic architecture is most often created by abstracting common architectural constructs from several applications in a particular problem class. In the case of blackboard systems, the blackboard architecture often can be converted into a generic architecture for a problem class by selecting the appropriate constructs for the various components and renaming some of them to fit the application glass. Figure 9 shows a generic architecture for tracking problems created in such a manner. The solution strategy is to use the recognition method with a simple context-independent, event-oriented control as shown in Figure 6. There are two blackboard panels, one for the solution space and one for control information.

6.1.2. Representation of Generic Architectures. In KASE, the generic architecture is represented as a collection of hierarchically organized functional modules. Each module is an object describing services (operations and procedures) that it provides, that it has access to, and that it needs to provide the promised services (see Fig. 10). Other relationships between the modules such as input/output and control relationships, can be derived from the service information. For example, input to a module is the collection of data types of the arguments of the service functions that the module provides, plus the results of the services it uses.

In a generic architecture any attribute of the modules can be abstracted as parameters. These parameters need to be instantiated with domain information. For the tracking architecture the following operations and procedures are treated as parameters: (1) the blackboard levels and the properties of the objects on each level; (2) the condition and the action parts of the knowledge sources; and (3) some control functions, such creating event types and selecting an event for focus of attention.

In order to provide automated tools to help a designer instantiate the parameters, KASE needs information about the domain. As shown in Figure 2, the domain information is acquired separately using the information in the meta-model.

**FIGURE 9.** A generic architecture for tracking problems (based on Figs. 4 and 6).

**FIGURE 10.** Minimal internal representation of a module (Bhansali & Nii, 1992) (* derived properties).
6.2. Meta-Model and Domain Models

The names of the components that were shown in Figure 9 are parts of the meta-model associated with the problem class. For the class of tracking problems, some of the terms and concepts that form the meta-model are: for domain objects—objects-to-be-tracked, signal-collection-sites, and signal-data; for relations—line-of-bearing; and for events—object-to-be-track-updated, and so on. The properties of these objects are partitioned into attributes and operations. Some of the generic attributes and operations are: position, heading, and speed for the attributes; and create-object, compute-position, and compute-heading for the operations.

The domain models are created by instantiating the meta-model objects. Figure 11 shows instantiated domain-model objects for HASP and ELINT created from the meta-model associated with the generic tracking architecture. A radar-signal data in ELINT is called observation, and a sonar data reading over a given time period is called line-segment.

Figure 12 shows typical attributes and properties for the two applications. Typically, the attributes and operations are provided by the application designer. However, some generic operations, such as create-objects (instantiated as create-cluster and create-source) and compute-speed are provided as a part of the generic architecture. These operations can be instantiated for different applications as will be described in Section 6.3.

6.3. Customizing the Generic Architecture

The customization, or instantiation of the generic architecture, involves instantiating the parameters in the generic architecture using the information in the domain model. Because the customization often involves non-simple parameters, for example, determining knowledge sources, a variety of methods are needed. These methods may consist of selecting from a pre-computed list of alternatives, transforming an abstract schema using a set of transformational rules, or using heuristics design rules and/or algorithms. KASE stores all the necessary methods in its customization knowledge base (see Fig. 2). We illustrate the use of some of the design knowledge in the customization process through two examples.

6.3.1. Customizing Blackboard Levels. Given a domain model, determining the blackboard levels is quite straightforward. There are heuristic rules that suggest the levels. Some are generic to blackboard architectures

<table>
<thead>
<tr>
<th>Meta-model terms</th>
<th>ELINT</th>
<th>HASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aircraft cluster</td>
<td>fleet</td>
<td>ship</td>
</tr>
<tr>
<td>Radar emitter</td>
<td>source</td>
<td>harmonic</td>
</tr>
<tr>
<td>Ship</td>
<td>line</td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 11.** Some meta-model instantiation for HASP and ELINT.

<table>
<thead>
<tr>
<th>attributes</th>
<th>operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELINT Cluster</td>
<td>create-cluster</td>
</tr>
<tr>
<td></td>
<td>split-cluster</td>
</tr>
<tr>
<td></td>
<td>merge-cluster</td>
</tr>
<tr>
<td></td>
<td>delete-cluster</td>
</tr>
<tr>
<td></td>
<td>compute-position</td>
</tr>
<tr>
<td>HASP Source</td>
<td>create-source</td>
</tr>
<tr>
<td></td>
<td>suspend-source</td>
</tr>
<tr>
<td></td>
<td>refine-source-type</td>
</tr>
<tr>
<td></td>
<td>compute-position</td>
</tr>
<tr>
<td></td>
<td>compute-confidence</td>
</tr>
</tbody>
</table>

**FIGURE 12.** A typical attributes and operations of objects in HASP and ELINT (Boxed operations are generic operations that are part of the generic architecture).
in general: “If an object class is mentioned in the output report, then it should be a level,” and “If an object class contains properties needed to compute other properties, then it should be a level.” Some others are specific to the generic architecture for a problem class: “If x is an object-to-be-tracked, then x should be a blackboard level.”

Figure 13 shows object-to-be-tracked instantiated for HASP, and the objects in the HASP domain model made into levels. Using the above rules, other objects, such as collection-sites whose coordinates are used in line-of-bearing computation, will also become levels. The KASE-generated levels are complete in the sense that it will list all possible levels; they can be used as suggestions that can be pruned by the designer. For example, in most applications collection-site level is not needed.

6.3.2. Customizing Knowledge Sources. In KASE the knowledge source design consists of three parts. In addition to the knowledge source trigger (context-independent condition) and the action part described in Section 5.2., it has a part called “event posting,” which makes explicit the need to post events on the control panel when the situation board is modified.

All the operations in the domain model are searched to identify those operations that either use or modify the objects on the blackboard. As shown in Figure 14, each selected operation becomes an action part of a knowledge source. For example, because compute-position uses bearing data, it is selected and becomes a part of a knowledge source.

Once the action parts of the knowledge sources are determined, design rules are used to determine events and triggers. An example rule for event creation is: “If an object is represented on the blackboard level, then create events for attribute of the object that can be modified.” Thus, for the example above, new-position would be an event.

Rules that determine the knowledge source triggers are stated in terms of the operation that constitute the action part of the knowledge source. For example,

If an x updates the value of an attribute a, and its value depends on the value of some other attribute a, then any event that signals an update of the value of a must trigger the operation x;

and

If an operation x on an object takes as its argument an instance of another object, then any event that signals the creation of the new instance of the object should trigger the operation.

As shown in Figure 14, since compute-position operation uses an attribute new-bearing, any time a new bearing is computed the knowledge source containing compute-position operation must be triggered.
The creation of knowledge sources in the manner just described creates a complete set of knowledge sources needed to compute all the attributes on the blackboard based on the operations available in the domain model. However, the knowledge sources are small-grained and numerous, and there may be more events generated than necessary. In generating the knowledge sources for HASP, KASE generated 46 events and 26 knowledge sources, about twice as many as the actual system. The design can be optimized by the user. KASE provides some heuristic optimization facilities. For example, merging knowledge sources: if several knowledge sources are triggered by the same event, merge the operations into one knowledge source.

7. SUMMARY CONCLUSIONS

Blackboard systems are unique among AI systems in that they can be recognized by their architectural features. Yet, there has been very little formal discussion of systems in terms of their architectures, either in the AI or in the software engineering literature. In order to understand architectures better, we analyzed the blackboard architecture from the perspective of the kinds of knowledge it contained. An architecture at a level of detail sufficient to serve as specifications for programming is an integration of its problem statement, solution strategies for the problem, computational design schemas to realize the solution strategies, and specific information, operations, and constraints from the actual application domain. We enumerated on each of these areas.

In the KASE project, we introduced the notion of a generic architecture for a problem class and its solution strategies. Associated with a generic architecture is an ontology of terms about the problem class with which specific domain information could be acquired. The generic architecture also had associated with it design knowledge with which the generic architecture and domain information could be used to create application-specific blackboard architectures. We gave examples of design knowledge, accumulated in KASE, about blackboard systems that were used to automatically generate the designs of application systems.

Acknowledgement—This research was supported by DARPA/NASA Order 6822 and Sumex Core Research Grant, NIH LM05208.

The research on the customization process within the KASE project was conducted by Sanjay Bhansali (Bhansali & Nii, 1992a, 1992b). Parts of Sections 4 and 5 were taken from (Nii, 1989).

REFERENCES


INVITED PAPER

Blackboard Systems at the Architecture Level

H. PENNY NII
Knowledge Systems Laboratory, Computer Science Department, Stanford University, Stanford, CA

Abstract—Since Hearsay-II, many application systems using the blackboard architecture have been built. Some of these applications have been built using expert system shells or tools. These tools contain the skeletal constructs of blackboard systems into which the user adds the necessary information and knowledge from the application domain to create an application system. Although the tools are useful in the implementation phase, they provide little help in the design of the system, which must necessarily precede the implementation phase. The determination of the blackboard levels, the knowledge sources, and the details of the control strategy is a task reserved for the knowledge engineer/system designer. In order to automate the design process, one needs to understand the relationships between the architectural constructs and domain information that provide the semantics of the architecture. In this paper the authors explore the components of blackboard systems from the perspective of the type of knowledge that goes into the makeup of blackboard systems at the architecture level. They describe a research system, KASE, that captures the various knowledge needed for design and applies the knowledge to aid knowledge engineers design blackboard systems.

1. INTRODUCTION

The terms "blackboard model," "blackboard architecture," and "blackboard system" are associated with a particular class of software programs. An attempt was made in Nii (1986) to distinguish the differences between a model, a framework (or, an architecture), and an application system. The model was viewed as a conceptual problem-solving entity consisting of two basic components, a global data base called the blackboard, and knowledge sources that created and modified the contents of the blackboard. The model includes a description of its behavior. An attempt at abstracting the software constructs of various application systems resulted in a blackboard architecture. In addition to the blackboard and the knowledge sources, a blackboard architecture includes a control component that enables the realization of the behavior in a serial computing environment (see Fig. 1). The application systems can be described as the instances and/or specializations of the architecture. The description of the application task and the associated task domain serves to establish a vocabulary to describe the specifics of the blackboard data structure, the knowledge source contents, and the problem-solving behavior of the different systems. That is, the application domain provides the semantics of the architecture.

There has been a steady increase in the number of blackboard systems [a recent example is Springer, Buta et al. (1991); older examples can be found in Engelmore & Morgan (1988); Corkill & Gallagher (1989); Jagannathan, Dodhiawala et al. (1989); Velthuijsen (1992)], but the designs of these systems continue to fit within the blackboard architecture described earlier. Although the implementation detail may differ greatly, the basic architecture of the blackboard systems has not changed over the years. This situation provides us with an opportunity to study the various properties of architecture, in particular, the blackboard architecture. What is an architecture? What is its form? What implicit information is embedded within an architecture? And, of particular interest to the knowledge system community, what kinds of knowledge are needed to design a system, and what roles do these different kinds play?

KASE (Knowledge Assisted Software Engineering) is a research project whose goal is to build a set of power tools for software analysts and designers at the architecture level. The tools should contain sufficient knowledge about the artifact and the design process to enable the automation of many of the activities in-
volved in design. Although its focus has been on answering the questions posed above for architectures in general, KASE has used a blackboard architecture as its first test bed. From the point of view of blackboard systems, the project represents an attempt to codify various kinds of knowledge about blackboard systems.

In this paper we describe the constituent components of an architecture in terms of the knowledge domains needed to construct an architecture. In particular, we begin with the contents of a generic blackboard architecture. We then describe a knowledge-based technique for specializing the generic architecture to create application-specific architectures using domain information.

2. THE ARCHITECTURE LEVEL

A software architecture is an abstraction of software systems. It comprises a topology of functional components, the interfaces between the components, and the semantic contents of these constituent parts. An architecture is usually described hierarchically to show different levels of detail, and at the most detailed level it must have sufficient detail to serve as a guide or specifications to the coding process.

Often, an architecture is treated as one of multiple views of a system. In LaSSIE (Devanbu, Brachman et al. 1991), for example, the system is partitioned into a domain model view, an architectural view, a feature view, and a code view. Others, for example, Iscoe, Williams et al. (1991), focus on the domain model as a basis for software design. The difficulty with separating the domain model and the architecture is that the domain information forms the basis for the semantics of the architecture and must be expressible in terms of, as well as within, the computational constructs that make up the architecture.

A software architecture is a product of integrating knowledge from multiple knowledge domains. In her study of human software designers, Guindon (1990) found that designers drew on many kinds of knowledge, including knowledge about the domain, knowledge about problem solutions and their representation, and knowledge about design schemas and computational constructs.

In KASE, an attempt has been made to identify in various software architectures their constituent knowledge domains and to discover the relationships among them. The objective is to rationalize software designs in order to identify useful knowledge-based design tools to help in the process of design synthesis. KASE begins with a generic architecture. A generic architecture is an abstract architecture designed to solve a particular class of problems (Bhansali & Nii, 1992a). KASE has been working with a class of problems called tracking problems in which the task is to identify and track objects in space based on signal data.

A generic architecture represents design decisions and, by its very nature, is associated with certain architectural commitments. The design decisions are guided by knowledge in the following areas:

1. A problem, or a task, class. The selection of a problem class establishes an ontology of generic terms with which to describe the task, and the objects and the operations in the problem class. Diagnostics, tracking, payroll, and transaction processing are examples of problem classes.

2. A solution strategy. An architecture is a commitment to solve a problem using a particular problem-solving strategy. Thus, there may be more than one architecture associated with a problem class representing different ways to solve the problem. One way to solve a tracking problem, for example, is to cumulate incrementally and opportunistically evidence for the identity and the behavior of the objects being tracked.

3. Design schemas (architectural constructs). The solution strategy is realized using a variety of design schemas, such as pipelines and layered hierarchies. [See Shaw (1991) for a description of architectural constructs.] The incremental and opportunistic problem-solving strategy for tracking problems can be realized using a blackboard architecture, which in turn is made up of what Shaw calls "interpreter" constructs.

Figure 2 shows how the decisions within these knowledge domains determine the shape of a generic architecture. In short, a generic architecture represents commitments to (1) a particular class of problems, (2) a solution strategy for that class, and (3) the architectural constructs used to realize the strategy.

The process of designing a specific application is a process of mapping domain information onto the architecture. To automate this process, the generic architecture needs two types of associated knowledge:

1. Customization knowledge. Associated with a generic architecture is knowledge about how to customize
The architecture into application-specific architectures. This knowledge includes how to instantiate components of the architecture in terms of the specific functionalities of the application domain. Customization knowledge captures design knowledge that is specific to the architecture. For example, the customization knowledge for the problems using the blackboard approach may include a heuristic: “Each intermediate data class needed to solve a problem should be a level in the blackboard data structure.”

2. **Meta-model.** Also associated with a generic architecture is an ontology of generic terms established by the commitment to a problem class. Called a meta-model, it contains the necessary vocabulary for describing the task and the domain. A *domain model* for a specific application is built by instantiating this ontology. For the tracking problem class, the generic terms include “objects-to-be-tracked” and “signal-data,” as well as their generic properties, such as, “location” and “speed.”

An *application-specific architecture* can be defined as a generic architecture instantiated with information from the application domain (Figure 2).

### 3. Blackboard Systems Revisited

In this section we revisit blackboard systems in terms of the concepts just described. We describe the various commitments inherent in blackboard systems by reviewing the task characteristics that are suited to blackboard architecture. In Section 4 we discuss the differences in blackboard architectures that can be attributed to different solution strategies. In Section 5 we discuss some of the ways in which the components of blackboard systems can be designed semi-automatically. Finally, in Section 6 we describe, using two examples from a tracking problem class, automated tools from KASE for designing specific blackboard systems from a generic architecture.

#### 3.1. Task Characteristics

The applications for which the blackboard approach has been useful include signal understanding, vision, and situation assessment (Engelmore & Morgan, 1988; Jagannathan, Dodhia, et al. 1989). These applications all share common characteristics:

1. The problems are complex and ill-structured with large solution spaces. Systematic generation of solution candidates is neither possible nor feasible.
2. The solutions require situation-dependent, or opportunistic, invocation of diverse sources of knowledge. That is, control decisions are made while the problem is being solved instead of using a priori established control paths.
3. Often, the problems demand both synthetic and analytic processes. These often translate into bottom-up information fusion or composition, and top-down model-based reasoning.

#### 3.2. Solution Strategies: Search vs. Recognition

One way to categorize a problem-solving approach is by determining whether it uses a search or a recognition method of problem solving. **Recognition** is a term coined by McDermott and Newell (1982)—“the complete recognition does not search, it knows.” At any particular computational state, instead of generating and evaluating the possible next states as in search, a recognition system simply knows what the next state should be. A classic example of a blackboard system that uses search as its solution strategy is HEARSAY-II (Erman, Hayes-Roth et al., 1980), and one using recognition strategy is HASP (Nii, Feigenbaum et al.,
1982). BB1 is an example of a shell that allows for the use of both strategies (Hayes-Roth, 1985).

In the search paradigm, each step in problem solving requires an evaluator to select from a pre-enumerated or enumerable next state. Each problem-solving step consists of a generation of operators or state, followed by an evaluation, followed by a selection and an application of the state transforming operator. Often, evaluation precedes generation so that generation can occur selectively. The generator can be algorithmic or heuristic; it can be a legal-move generator or a plausible-move generator. The evaluator can be knowledge-poor (general and weak), or it can be knowledge-rich (task-specific). To date most programs that use search use weak knowledge—the search paradigm is called a weak method.

In HEARSAY-II all the possible states were enumerable by its knowledge sources. For example, given a grammar, a table of all legal adjacent words for each word could be generated. The evaluator, or the scheduler, used weak criteria. For example, criteria from selecting from a number of possible word sequences might use the number of words in the sequence. The knowledge is weak, because it does not use task-specific criteria, or when used, the task-specific criteria are very simple. A hypothetical, knowledge-rich criterion might be: "select a word sequence beginning with a preposition following a noun clause."

In the recognition paradigm, the knowledge base must be scanned for a piece of knowledge that can be applied to the current situation. This is called the match step. Each problem-solving step consists of a match step followed by the application of a state transforming operator. If a piece of knowledge is encoded as a conditional if-then rule, then it contains information about what the next state would be as well as the situation under which the state transformation can occur. Most current expert systems are recognition systems. (It should be noted that if-then rules are not necessary for a recognition system, and conversely the mere use of if-then rules does not imply a recognition system. The problem-solving approach is independent of the knowledge representation.)

In HASP the search paradigm was tried and failed, because there was no generator of the solution space. It was one of the first complete recognition systems. It consisted of 40–50 knowledge sources, each with an average of 10 rules. The active knowledge sources consisted of domain- and situation-specific pieces of knowledge that "knew" what to infer or what action to take for each intermediate solution state. In addition, it had a plan for the application of the knowledge in the form of control knowledge sources.

In the next section we elaborate on the basic differences in system architectures attributable to the differences between the search and recognition methods of problem solving.

4. ARCHITECTURAL IMPLICATIONS OF DIFFERENT SOLUTION STRATEGIES

4.1. Search

As stated earlier, search requires at least a generator of the solution space and an evaluator. The evaluation can be of two types: a local look-ahead that evaluates possible future states; and a global evaluator that selects the best operator based on the information provided by the look-ahead or the overall state of the solution. In the HEARSAY-II architecture, the action part of the knowledge source was used as a generator (see Fig. 3). Upon its execution, the action part of the knowledge source placed on the blackboard new hypotheses. The condition part served as a look-ahead for the action part. When executed, it determined the state changes that would be produced if its action part were to be executed. The scheduler decided on the next activity—either to look-ahead or to generate changes; that is, it determined the solution path by evaluating the current state of the solution and a set of possible future states.

In HEARSAY-II, the global evaluation criteria was hidden inside the scheduling module; that is, the scheduling module was a primitive module.

Within the search paradigm the quality of the solution may depend on the amount of search conducted, which is also a function of the quality of the knowledge available for evaluation. The premise in HEARSAY-II is that a favorable trade-off can be made between the amount of control knowledge and knowledge source activations—that is, spending time on evaluation pays off in reducing search and in improving the quality of the solution.

4.2. Recognition

The recognition problem-solving paradigm is conceptually simpler than the search paradigm (see Fig. 4). Here, the knowledge base is scanned for a piece of knowledge that can be applied to a state. Of course,
there may be many pieces of knowledge applicable for a given state, and a decision must be made whether to apply all the pieces of knowledge or a subset of them. This control decision can be algorithmic (for example, first-in-first-out) or it can be knowledge-based. When the control decision is knowledge-based, there is a certain amount of uniformity between the way decisions are made about which knowledge to apply and about the manipulations of the solution space objects. In recognition-oriented blackboard systems, the condition part of the knowledge source specifies the situations under which it can contribute to the solution state. The action part of the knowledge source changes the solution state. Since the applicable knowledge source for any given situation is predetermined, in the sense that knowledge sources are always associated with situations, the scheduler focuses on the selection of the best region of the solution space (the specific situation) to process next.

In theory, no distinction is made between control and domain-specific knowledge in recognition systems. However, since a domain knowledge source looks at the blackboard data, a control knowledge source looks at a description of the changes to the blackboard in addition to the specifics of the changes, they are in fact different. It might be said that the control knowledge source has wider scope. Because control knowledge is different from other knowledge sources, systems can be designed to exploit the differences. Informal plans either can be built into the control structure or dynamically generated. In BB1 (Hayes-Roth, 1985) control plans for the execution of the domain knowledge sources are dynamically created.

5. ARCHITECTURAL ALTERNATIVES

Some of the architectural differences between HEAR-SAY-like and HASP-like systems can be attributed to the different computational methods inherent in the use of search and recognition paradigms. There are also variations in the structure of the components of the architecture driven by the different needs within a problem class or the specific application itself. These different architectural schemas can be stored in a library. The designer can select appropriate design schema for the various component parts of the architecture. For example, as mentioned earlier, knowledge source selection may be algorithmic or knowledge-based, and the library may contain several different knowledge-selection modules. In this section we describe some of the alternative constructs for the major components of the blackboard architecture.

5.1. The Blackboard

In addition to serving as a medium of communication, and thus of cooperation, between the knowledge sources, the blackboard represents a design of the solution. Since it is a data structure, one could not quite call it a problem-solving plan, yet in the process of problem formulation the structure of the blackboard serves as the backbone of the system design. It is where the design process begins. The blackboard partitions the solution space, and these partitions dictate what knowledge sources will be needed to solve the problem. What constitutes a solution, what intermediate solutions are needed or available, what relationships exist among hypotheses, must all be expressed within the blackboard structure.

From the standpoint of structure there is very little variability in the blackboard. The blackboard is organized into levels corresponding to the solution space partitions. These form either abstraction or compositional hierarchies. The objects on a given level share the same properties, or vocabulary.

5.1.1. Levels. As a data structure, the level object holds information about the nodes on its level. For example, it must keep the names of the attributes of the nodes and maintain housekeeping information, such as the number of nodes on the level at any given time. For these reasons, it is often convenient to view the level object as a class, and the nodes on the level as the instances of the class. This has the further advantage that nodes can inherit default property values at node creation time. Also, operators (methods) associated with creation and destruction of nodes can be stored with the level object.

5.1.2. Nodes. Nodes are objects on a level; for example, a particular word in a sentence is a node on the word level in HEARSAY-II. Nodes are created dynamically as they are needed. The vocabulary relevant to the level of analysis is represented as attributes and values; each node on the level shares the same attributes. In an object-oriented programming language slots and slot-values can be used, and there is nothing to preclude the use of record structures available in other languages.
5.1.3. *Attribute Values.* The value of an attribute can have multiple fields; for example, credibility value and/or time stamps. Often it is useful to maintain value history. This is especially true in continuous-input signal understanding programs in which temporal trends are important. In object-oriented systems, a value can be represented as a method that computes it when the slot is accessed.

5.1.4. *Panels.* Panels are useful when more than one hierarchy is needed to represent the solution state data. The most common use of a second panel is to hold control-related information, as in BB1.

5.2. The Knowledge Sources

A knowledge source represents knowledge needed to solve a sub-problem. All knowledge sources are event-triggered agents—the activation of knowledge sources are solution-state-dependent, rather than being process-dependent. The implication is that the most important consideration in the design of the knowledge source is its ability to evaluate the current state of the blackboard.

In all blackboard systems knowledge sources are designed with condition and action parts (Fig. 5).

5.2.1. *Condition Part.* The condition part can: (1) evaluate possible future states, or (2) make context-independent or context-dependent evaluation of the situation on the blackboard. In HASP, possible changes to the blackboard were categorized into different “event” types. The name of the event type was used to record a blackboard change. The event name was also used as a triggering condition in the condition part of the knowledge sources. In this type of construct the condition part is context-independent—it does not take into consideration the specifics of the change. Alternatively, the condition part can be a procedure that evaluates the specifics of the change, making the triggering of knowledge sources context-dependent as in HEARSAY-II.

One can think of the condition part as a filter for the activation of the action part. In newer systems, the design of the condition part consists of multi-staged filters. The first stage, called the trigger, is usually a context-independent filter that is followed by a number of context-dependent filters. If the action part of the knowledge source is a collection of rules, then the condition part of the knowledge source can be viewed as a high-level filter for the rule set. Rules have their own condition parts that serve as more detailed filters for the rule actions.

5.2.2. *Action Part.* The action part of a knowledge source makes modifications to the current blackboard state. The action part can be a procedure or a collection of rules. The selection of the knowledge representation scheme is independent of the selection of solution strategies.

Each knowledge source can make one or more changes to the blackboard. In a rule-based knowledge source, multiple changes can come from multiple firing of rules, or from multiple actions on the right-hand-side of rules.

The changes to the blackboard are new solution candidates (and new data) or modifications to the existing solution. Changes also can be made to the control information, as in BB1. In addition to changes to intermediate solutions on the blackboard, state changes can occur from the posting of goals and sub-goals to be achieved (Lesser & Corkill, 1983) or posting of model-based expectations of changes to occur in the future, as in HASP.

5.3. The Control

Control is the most complex component of the blackboard architecture, and it can have many architectural variations. The basic function of the control component is to select and apply knowledge sources in the case of recognition-based systems, or the condition and action components of knowledge sources in search-based systems. Major design differences occur in the following areas: (1) schedulable entities—whether parts of the knowledge sources are separate schedulable entities or not; (2) scheduling—whether the focus of attention is based on events or on knowledge sources; (3) noticing—who notices the changes in the blackboard; and (4) control data—where and in what form the control data is stored.

1. *Schedulable Entities:* In a search-based architecture the condition and the action parts of the knowledge sources are scheduled separately. In recognition-based architecture, each knowledge source is scheduled as a single entity. If the condition part is satisfied, then the action part is immediately executed.

One thing to note is that in the search-based architecture, the blackboard state may change between the time a condition part and its action part are executed. In such a situation the condition part must be re-evaluated.

2. *Scheduler:* There are two bases on which to approach the selection of the next thing to do:

![FIGURE 5. Use of filters in knowledge sources.](image-url)
A. A knowledge source, or knowledge sources, can be selected on the basis of the current state of the blackboard, or more specifically, on the latest changes to the blackboard, which are called events. Called event-oriented scheduling, the focus is on selecting the best solution island on which to base further computation. Most recognition systems take this approach. The basic function of the scheduler is to decide where to work next. Figure 6 shows a simple event-oriented control schema.

b. A knowledge source can be selected on the basis of what the knowledge source can contribute to the current state. This approach, called knowledge-oriented scheduling, is taken most often in search systems. The scheduler decides among the look-ahead evaluators and the generators of the solution space, and it applies the selected components to all the relevant data objects on the blackboard. The basic function of the scheduler is to decide what to do next. Figure 7 shows one possible way to design a knowledge-oriented control.

In reality, many problems can gain from a mixed strategy of search and recognition, and BB1, for example, allows for both event-oriented and knowledge-oriented scheduling.

5.3.1. Posting and Noticing Blackboard Changes
What changes are posted and who notices these changes differ from system to system. Often, in search-oriented systems, every change to the blackboard is a candidate for a solution, and there must be a knowledge source to evaluate each change. On the other hand, event-oriented recognition systems can process only those blackboard changes for which there is available knowledge. By making the event creation explicit, the rule writer can control what blackboard changes merit further action by the knowledge sources. For example, if a new piece of evidence is found to support a hypothesis but the only consequence is that the credibility rating is increased, this change may not need to be an event.
What blackboard changes should be events depend on the application.

5.3.2. Control Data. Control information needed to drive the system always resides outside the blackboard containing the solution state. In HASP different types of changes to the blackboard (modification to nodes, expectation posting, and so on) were stored in separate data structures. Each event was a recording of the type of change, the node that was modified, the new values, the rule that made the modification. In HEARSAY-II the control data was stored on a global list, the scheduling queue. The items on the queue were of two types: information generated by the condition part that pointed to its action part and all the places on the blackboard it could be applied to, and the names of the condition parts of knowledge sources. Since HASP-like systems focus on event scheduling, the control data are primarily about events; in HEARSAY-like systems that focus on knowledge source scheduling the control data are primarily about knowledge sources.

In BB1, control must determine what domain knowledge source to apply and to what object on the blackboard. The control problem is formulated as a planning sub-problem, and its solution state appears on a separate control panel. All data dealing with control are posted on the control panel and are shared by both the control-planning and the domain knowledge sources.

5.3.3. Domain-specific Knowledge, Meta-knowledge, and Control Knowledge. The distinction between domain-specific knowledge and meta-knowledge is not always clear cut. Meta-knowledge, knowledge about knowledge, is often thought to be weak, general, and domain-independent. But, in both search and recognition systems, meta-knowledge is often treated as control knowledge about what to do next, and this type of knowledge is often very domain-specific. In general, making distinctions between domain-specific and meta-knowledge does not contribute much to the design of the system; rather, it tends to distract. Instead it is preferable to organize knowledge sources hierarchically according to some control hierarchy. In most systems the control hierarchy partitions the knowledge sources into those that deal directly with the emerging solution states and those that deal with the process history or problem-solving states, called control states. The knowledge sources that operate on control states are called control knowledge sources.

6. DESIGNING APPLICATION SYSTEMS USING A GENERIC ARCHITECTURE

Up to this point we have discussed the three types of knowledge that are needed to create a generic blackboard architecture—knowledge about the characteristics of a class of problems, its solution strategies, and alternative design schemas to realize the solution strategy. As was shown in Figure 2, a generic architecture has explicitly associated with it (1) a meta-model consisting of a set of concepts and objects to describe the class of problems, and (2) knowledge about how to integrate domain information with the generic architecture to create application-specific architectures. In this section we describe these two items by means of examples from work done on the KASE project.

6.1. A Generic Architecture For Tracking Problems

6.1.1. The Tracking Problem. This class of problems can be described simply as follows: Identify, track, and sometimes, infer the intentions of objects in space using signal information about the objects (see Fig. 8). As examples, we use the HASP and the ELINT (Brown, Schoen et al., 1986) problems. In HASP the objects to be tracked are ships and submarines using passive sonar signals. In ELINT the objects to be tracked are airplanes using radar signals. In both problems, there are multiple sources of signal for each platform (engines and pro-
pellers on ships, for example), and the platforms may be clustered together obviating individual platform identification.

A generic architecture is most often created by abstracting common architectural constructs from several applications in a particular problem class. In the case of blackboard systems, the blackboard architecture often can be converted into a generic architecture for a problem class by selecting the appropriate constructs for the various components and renaming some of them to fit the application glass. Figure 9 shows a generic architecture for tracking problems created in such a manner. The solution strategy is to use the recognition method with a simple context-independent, event-oriented control as shown in Figure 6. There are two blackboard panels, one for the solution space and one for control information.

6.1.2. Representation of Generic Architectures. In KASE, the generic architecture is represented as a collection of hierarchically organized functional modules. Each module is an object describing services (operations and procedures) that it provides, that it has access to, and that it needs to provide the promised services (see Fig. 10). Other relationships between the modules such as input/output and control relationships, can be derived from the service information. For example, input to a module is the collection of data types of the arguments of the service functions that the module provides, plus the results of the services it uses.

In a generic architecture any attribute of the modules can be abstracted as parameters. These parameters need to be instantiated with domain information. For the tracking architecture the following operations and procedures are treated as parameters: (1) the blackboard levels and the properties of the objects on each level; (2) the condition and the action parts of the knowledge sources; and (3) some control functions, such creating event types and selecting an event for focus of attention.

In order to provide automated tools to help a designer instantiate the parameters, KASE needs information about the domain. As shown in Figure 2, the domain information is acquired separately using the information in the meta-model.

---

**FIGURE 9.** A generic architecture for tracking problems (based on Figs. 4 and 6).

---

**FIGURE 10.** Minimal internal representation of a module (Bhansali & Nii, 1992) (* derived properties).
### 6.2. Meta-Model and Domain Models

The names of the components that were shown in Figure 9 are parts of the meta-model associated with the problem class. For the class of tracking problems, some of the terms and concepts that form the meta-model are: for domain objects—objects-to-be-tracked, signal-collection-sites, and signal-data; for relations—line-of-bearing; and for events—object-to-be-track-updated, and so on. The properties of these objects are partitioned into attributes and operations. Some of the generic attributes and operations are: position, heading, and speed for the attributes; and create-object, compute-position, and compute-heading for the operations.

The domain models are created by instantiating the meta-model objects. Figure 11 shows instantiated domain-model objects for HASP and ELINT created from the meta-model associated with the generic tracking architecture. A radar-signal data in ELINT is called observation, and a sonar data reading over a given time period is called line-segment.

Figure 12 shows typical attributes and properties for the two applications. Typically, the attributes and operations are provided by the application designer. However, some generic operations, such as create-objects (instantiated as create-cluster and create-source) and compute-speed are provided as a part of the generic architecture. These operations can be instantiated for different applications as will be described in Section 6.3.

### 6.3. Customizing the Generic Architecture

The customization, or instantiation of the generic architecture, involves instantiating the parameters in the generic architecture using the information in the domain model. Because the customization often involves non-simple parameters, for example, determining knowledge sources, a variety of methods are needed. These methods may consist of selecting from a pre-computed list of alternatives, transforming an abstract schema using a set of transformational rules, or using heuristics design rules and/or algorithms. KASE stores all the necessary methods in its customization knowledge base (see Fig. 2). We illustrate the use of some of the design knowledge in the customization process through two examples.

#### 6.3.1. Customizing Blackboard Levels

Given a domain model, determining the blackboard levels is quite straightforward. There are heuristic rules that suggest the levels. Some are generic to blackboard architectures

<table>
<thead>
<tr>
<th>attributes</th>
<th>operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELINT Cluster</td>
<td>create-cluster</td>
</tr>
<tr>
<td></td>
<td>split-cluster</td>
</tr>
<tr>
<td></td>
<td>merge-cluster</td>
</tr>
<tr>
<td></td>
<td>delete-cluster</td>
</tr>
<tr>
<td></td>
<td>compute-position</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>HASP Source</td>
<td>create-source</td>
</tr>
<tr>
<td></td>
<td>suspend-source</td>
</tr>
<tr>
<td></td>
<td>refine-source-type</td>
</tr>
<tr>
<td></td>
<td>compute-position</td>
</tr>
<tr>
<td></td>
<td>compute-confidence</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

**FIGURE 12.** A typical attributes and operations of objects in HASP and ELINT (Boxed operations are generic operations that are part of the generic architecture).
in general: "If an object class is mentioned in the output report, then it should be a level," and "If an object class contains properties needed to compute any other properties, then it should be a level." Some others are specific to the generic architecture for a problem class: "If x is an object-to-be-tracked, then x should be a blackboard level."

Figure 13 shows object-to-be-tracked instantiated for HASP, and the objects in the HASP domain model made into levels. Using the above rules, other objects, such as collection-sites whose coordinates are used in line-of-bearing computation, will also become levels. The KASE-generated levels are complete in the sense that it will list all possible levels; they can be used as suggestions that can be pruned by the designer. For example, in most applications collection-site level is not needed.

6.3.2. Customizing Knowledge Sources. In KASE the knowledge source design consists of three parts. In addition to the knowledge source trigger (context-independent condition) and the action part described in Section 5.2., it has a part called "event posting," which makes explicit the need to post events on the control panel when the situation board is modified.

All the operations in the domain model are searched to identify those operations that either use or modify the objects on the blackboard. As shown in Figure 14, each selected operation becomes an action part of a knowledge source. For example, because compute-position uses bearing data, it is selected and becomes a part of a knowledge source.

Once the action parts of the knowledge sources are determined, design rules are used to determine events and triggers. An example rule for event creation is: "If an object is represented on the blackboard level, then create events for attribute of the object that can be modified." Thus, for the example above, new-position would be an event.

Rules that determine the knowledge source triggers are stated in terms of the operation that constitute the action part of the knowledge source. For example,

If an x updates the value of an attribute a₁, and its value depends on the value of some other attribute a₂, then any event that signals an update of the value of a₂ must trigger the operation x;

If an operation x on an object takes as its argument an instance of another object, then any event that signals the creation of the new instance of the object should trigger the operation.

As shown in Figure 14, since compute-position operation uses an attribute new-bearing, any time a new bearing is computed the knowledge source containing compute-position operation must be triggered.
The creation of knowledge sources in the manner just described creates a complete set of knowledge sources needed to compute all the attributes on the blackboard based on the operations available in the domain model. However, the knowledge sources are small-grained and numerous, and there may be more events generated than necessary. In generating the knowledge sources for HASP, KASE generated 46 events and 26 knowledge sources, about twice as many as the actual system. The design can be optimized by the user. KASE provides some heuristic optimization facilities. For example, merging knowledge sources: if several knowledge sources are triggered by the same event, merge the operations into one knowledge source.

7. SUMMARY CONCLUSIONS
Blackboard systems are unique among AI systems in that they can be recognized by their architectural features. Yet, there has been very little formal discussion of systems in terms of their architectures, either in the AI or in the software engineering literature. In order to understand architectures better, we analyzed the blackboard architecture from the perspective of the kinds of knowledge it contained. An architecture at a level of detail sufficient to serve as specifications for programming is an integration of its problem statement, solution strategies for the problem, computational design schemas to realize the solution strategies, and specific information, operations, and constraints from the actual application domain. We enumerated on each of these areas.

In the KASE project, we introduced the notion of a generic architecture for a problem class and its solution strategies. Associated with a generic architecture is an ontology of terms about the problem class with which specific domain information could be acquired. The generic architecture also had associated with it design knowledge with which the generic architecture and domain information could be used to create application-specific blackboard architectures. We gave examples of design knowledge, accumulated in KASE, about blackboard systems that were used to automatically generate the designs of application systems.

Acknowledgement—This research was supported by DARPA/NASA Order 6822 and Sumex Core Research Grant, NIH LM05208.

The research on the customization process within the KASE project was conducted by Sanjay Bhansali (Bhansali & Nii, 1992a, 1992b). Parts of Sections 4 and 5 were taken from (Nii, 1989).

REFERENCES
INVITED PAPER

Blackboard Systems at the Architecture Level

H. PENNY NII

Knowledge Systems Laboratory, Computer Science Department, Stanford University, Stanford, CA

Abstract—Since Hearsay-II, many application systems using the blackboard architecture have been built. Some of these applications have been built using expert system shells or tools. These tools contain the skeletal constructs of blackboard systems into which the user adds the necessary information and knowledge from the application domain to create an application system. Although the tools are useful in the implementation phase, they provide little help in the design of the system, which must necessarily precede the implementation phase. The determination of the blackboard levels, the knowledge sources, and the details of the control strategy is a task reserved for the knowledge engineer/system designer. In order to automate the design process, one needs to understand the relationships between the architectural constructs and domain information that provide the semantics of the architecture. In this paper the authors explore the components of blackboard systems from the perspective of the type of knowledge that goes into the makeup of blackboard systems at the architecture level. They describe a research system, KASE, that captures the various knowledge needed for design and applies the knowledge to aid knowledge engineers design blackboard systems.

1. INTRODUCTION

The terms “blackboard model,” “blackboard architecture,” and “blackboard system” are associated with a particular class of software programs. An attempt was made in Nii (1986) to distinguish the differences between a model, a framework (or, an architecture), and an application system. The model was viewed as a conceptual problem-solving entity consisting of two basic components, a global data base called the blackboard, and knowledge sources that created and modified the contents of the blackboard. The model includes a description of its behavior. An attempt at abstracting the software constructs of various application systems resulted in a blackboard architecture. In addition to the blackboard and the knowledge sources, a blackboard architecture includes a control component that enables the realization of the behavior in a serial computing environment (see Fig. 1). The application systems can be described as the instances and/or specializations of the architecture. The description of the application task and the associated task domain serves to establish a vocabulary to describe the specifics of the blackboard data structure, the knowledge source contents, and the problem-solving behavior of the different systems. That is, the application domain provides the semantics of the architecture.

There has been a steady increase in the number of blackboard systems [a recent example is Springer, Buta et al. (1991); older examples can be found in Englemore & Morgan (1988); Corkill & Gallagher (1989); Jaganathan, Dodhiawala et al. (1989); Velthuijsen (1992)], but the designs of these systems continue to fit within the blackboard architecture described earlier. Although the implementation detail may differ greatly, the basic architecture of the blackboard systems has not changed over the years. This situation provides us with an opportunity to study the various properties of architecture, in particular, the blackboard architecture. What is an architecture? What is its form? What implicit information is embedded within an architecture? And, of particular interest to the knowledge system community, what kinds of knowledge are needed to design a system, and what roles do these different kinds play?

KASE (Knowledge Assisted Software Engineering) is a research project whose goal is to build a set of power tools for software analysts and designers at the architecture level. The tools should contain sufficient knowledge about the artifact and the design process to enable the automation of many of the activities in-
In KASE, an attempt has been made to identify in various software architectures their constituent knowledge domains and to discover the relationships among them. The objective is to rationalize software designs in order to identify useful knowledge-based design tools to help in the process of design synthesis. KASE begins with a generic architecture. A generic architecture is an abstract architecture designed to solve a particular class of problems (Bhansali & Nii, 1992a). KASE has been working with a class of problems called tracking problems in which the task is to identify and track objects in space based on signal data.

A generic architecture represents design decisions and, by its very nature, is associated with certain architectural commitments. The design decisions are guided by knowledge in the following areas:

1. A problem, or a task, class. The selection of a problem class establishes an ontology of generic terms with which to describe the task, and the objects and the operations in the problem class. Diagnostics, tracking, payroll, and transaction processing are examples of problem classes.

2. A solution strategy. An architecture is a commitment to solve a problem using a particular problem-solving strategy. Thus, there may be more than one architecture associated with a problem class representing different ways to solve the problem. One way to solve a tracking problem, for example, is to cumulate incrementally and opportunistically evidence for the identity and the behavior of the objects being tracked.

3. Design schemas (architectural constructs). The solution strategy is realized using a variety of design schemas, such as pipelines and layered hierarchies. [See Shaw (1991) for a description of architectural constructs.] The incremental and opportunistic problem-solving strategy for tracking problems can be realized using a blackboard architecture, which in turn is made up of what Shaw calls "interpreter" constructs.

Figure 2 shows how the decisions within these knowledge domains determine the shape of a generic architecture. In short, a generic architecture represents commitments to (1) a particular class of problems, (2) a solution strategy for that class, and (3) the architectural constructs used to realize the strategy.

The process of designing a specific application is a process of mapping domain information onto the architecture. To automate this process, the generic architecture needs two types of associated knowledge.

1. Customization knowledge. Associated with a generic architecture is knowledge about how to customize
the architecture into application-specific architectures. This knowledge includes how to instantiate components of the architecture in terms of the specific functionalities of the application domain. Customization knowledge captures design knowledge that is specific to the architecture. For example, the customization knowledge for the problems using the blackboard approach may include a heuristic: "Each intermediate data class needed to solve a problem should be a level in the blackboard data structure."

2. Meta-model. Also associated with a generic architecture is an ontology of generic terms established by the commitment to a problem class. Called a meta-model, it contains the necessary vocabulary for describing the task and the domain. A domain model for a specific application is built by instantiating this ontology. For the tracking problem class, the generic terms include "objects-to-be-tracked" and "signal-data," as well as their generic properties, such as, "location" and "speed."

An application-specific architecture can be defined as a generic architecture instantiated with information from the application domain (Figure 2).

3. BLACKBOARD SYSTEMS REVISITED

In this section we revisit blackboard systems in terms of the concepts just described. We describe the various commitments inherent in blackboard systems by reviewing the task characteristics that are suited to blackboard architecture. In Section 4 we discuss the differences in blackboard architectures that can be attributed to different solution strategies. In Section 5 we discuss some of the ways in which the components of blackboard systems can be designed semi-automatically. Finally, in Section 6 we describe, using two examples from a tracking problem class, automated tools from KASE for designing specific blackboard systems from a generic architecture.

3.1. Task Characteristics

The applications for which the blackboard approach has been useful include signal understanding, vision, and situation assessment (Engelmore & Morgan, 1988; Jagannathan, Dodhiawala et al. 1989). These applications all share common characteristics:

1. The problems are complex and ill-structured with large solution spaces. Systematic generation of solution candidates is neither possible nor feasible.

2. The solutions require situation-dependent, or opportunistic, invocation of diverse sources of knowledge. That is, control decisions are made while the problem is being solved instead of using a priori established control paths.

3. Often, the problems demand both synthetic and analytic processes. These often translate into bottom-up information fusion or composition, and top-down model-based reasoning.

3.2. Solution Strategies: Search vs. Recognition

One way to categorize a problem-solving approach is by determining whether it uses a search or a recognition method of problem solving. Recognition is a term coined by McDermott and Newell (1982)—"the complete recognition does not search, it knows." At any particular computational state, instead of generating and evaluating the possible next states as in search, a recognition system simply knows what the next state should be. A classic example of a blackboard system that uses search as its solution strategy is HEARSAY-II (Erman, Hayes-Roth et al., 1980), and one using recognition strategy is HASP (Nii, Feigenbaum et al.,
1982). BB1 is an example of a shell that allows for the use of both strategies (Hayes-Roth, 1985).

In the search paradigm, each step in problem solving requires an evaluator to select from a pre-enumerated or enumerable next state. Each problem-solving step consists of a generation of operators or state, followed by an evaluation, followed by a selection and an application of the state transforming operator. Often, evaluation precedes generation so that generation can occur selectively. The generator can be algorithmic or heuristic; it can be a legal-move generator or a plausible-move generator. The evaluator can be knowledge-poor (general and weak), or it can be knowledge-rich (task-specific). To date most programs that use search use weak knowledge—the search paradigm is called a weak method.

In HEARSAY-II all the possible states were enumerable by its knowledge sources. For example, given a grammar, a table of all legal adjacent words for each word could be generated. The evaluator, or the scheduler, used weak criteria. For example, criteria from selecting from a number of possible word sequences might use the number of words in the sequence. The knowledge is weak, because it does not use task-specific criteria, or when used, the task-specific criteria are very simple. A hypothetical, knowledge-rich criterion might be: “select a word sequence beginning with a preposition following a noun clause.”

In the recognition paradigm, the knowledge base must be scanned for a piece of knowledge that can be applied to the current situation. This is called the match step. Each problem-solving step consists of a match step followed by the application of a state transforming operator. If a piece of knowledge is encoded as a conditional if-then rule, then it contains information about what the next state would be as well as the situation under which the state transformation can occur. Most current expert systems are recognition systems. (It should be noted that if-then rules are not necessary for a recognition system, and conversely the mere use of if-then rules does not imply a recognition system. The problem-solving approach is independent of the knowledge representation.)

In HASP the search paradigm was tried and failed, because there was no generator of the solution space. It was one of the first complete recognition systems. It consisted of 40–50 knowledge sources, each with an average of 10 rules. The active knowledge sources consisted of domain- and situation-specific pieces of knowledge that “knew” what to infer or what action to take for each intermediate solution state. In addition, it had a plan for the application of the knowledge in the form of control knowledge sources.

In the next section we elaborate on the basic differences in system architectures attributable to the differences between the search and recognition methods of problem solving.

4. ARCHITECTURAL IMPLICATIONS OF DIFFERENT SOLUTION STRATEGIES

4.1. Search

As stated earlier, search requires at least a generator of the solution space and an evaluator. The evaluation can be of two types: a local look-ahead that evaluates possible future states; and a global evaluator that selects the best operator based on the information provided by the look-ahead or the overall state of the solution. In the HEARSAY-II architecture, the action part of the knowledge source was used as a generator (see Fig. 3). Upon its execution, the action part of the knowledge source placed on the blackboard new hypotheses. The condition part served as a look-ahead for the action part. When executed, it determined the state changes that would be produced if its action part were to be executed. The scheduler decided on the next activity—either to look-ahead or to generate changes; that is, it determined the solution path by evaluating the current state of the solution and a set of possible future states.

In HEARSAY-II, the global evaluation criteria was hidden inside the scheduling module; that is, the scheduling module was a primitive module.

Within the search paradigm the quality of the solution may depend on the amount of search conducted, which is also a function of the quality of the knowledge available for evaluation. The premise in HEARSAY-II is that a favorable trade-off can be made between the amount of control knowledge and knowledge source activations—that is, spending time on evaluation pays off in reducing search and in improving the quality of the solution.

4.2. Recognition

The recognition problem-solving paradigm is conceptually simpler than the search paradigm (see Fig. 4). Here, the knowledge base is scanned for a piece of knowledge that can be applied to a state. Of course,
there may be many pieces of knowledge applicable for a given state, and a decision must be made whether to apply all the pieces of knowledge or a subset of them. This control decision can be algorithmic (for example, first-in-first-out) or it can be knowledge-based. When the control decision is knowledge-based, there is a certain amount of uniformity between the way decisions are made about which knowledge to apply and about the manipulations of the solution space objects. In recognition-oriented blackboard systems, the condition part of the knowledge source specifies the situations under which it can contribute to the solution state. The action part of the knowledge source changes the solution state. Since the applicable knowledge source for any given situation is predetermined, in the sense that knowledge sources are always associated with situations, the scheduler focuses on the selection of the best region of the solution space (the specific situation) to process next.

In theory, no distinction is made between control and domain-specific knowledge in recognition systems. However, since a domain knowledge source looks at the blackboard data, and a control knowledge source looks at a description of the changes to the blackboard in addition to the specifics of the changes, they are in fact different. It might be said that the control knowledge source has wider scope. Because control knowledge is different from other knowledge sources, systems can be designed to exploit the differences. Informal plans either can be built into the control structure or dynamically generated. In BB1 (Hayes-Roth, 1985) control plans for the execution of the domain knowledge sources are dynamically created.

5. ARCHITECTURAL ALTERNATIVES

Some of the architectural differences between HEARSAY-like and HASP-like systems can be attributed to the different computational methods inherent in the use of search and recognition paradigms. There are also variations in the structure of the components of the architecture driven by the different needs within a problem class or the specific application itself. These different architectural schemas can be stored in a library. The designer can select appropriate design schema for the various component parts of the architecture. For example, as mentioned earlier, knowledge source selection may be algorithmic or knowledge-based, and the library may contain several different knowledge-selection modules. In this section we describe some of the alternative constructs for the major components of the blackboard architecture.

5.1. The Blackboard

In addition to serving as a medium of communication, and thus of cooperation, between the knowledge sources, the blackboard represents a design of the solution. Since it is a data structure, one could not quite call it a problem-solving plan, yet in the process of problem formulation the structure of the blackboard serves as the backbone of the system design. It is where the design process begins. The blackboard partitions the solution space, and these partitions dictate what knowledge sources will be needed to solve the problem. What constitutes a solution, what intermediate solutions are needed or available, what relationships exist among hypotheses, must all be expressed within the blackboard structure.

From the standpoint of structure there is very little variability in the blackboard. The blackboard is organized into levels corresponding to the solution space partitions. These form either abstraction or compositional hierarchies. The objects on a given level share the same properties, or vocabulary.

5.1.1. Levels. As a data structure, the level object holds information about the nodes on its level. For example, it must keep the names of the attributes of the nodes and maintain housekeeping information, such as the number of nodes on the level at any given time. For these reasons, it is often convenient to view the level object as a class, and the nodes on the level as the instances of the class. This has the further advantage that nodes can inherit default property values at node creation time. Also, operators (methods) associated with creation and destruction of nodes can be stored with the level object.

5.1.2. Nodes. Nodes are objects on a level; for example, a particular word in a sentence is a node on the word level in HEARSAY-II. Nodes are created dynamically as they are needed. The vocabulary relevant to the level of analysis is represented as attributes and values; each node on the level shares the same attributes. In an object-oriented programming language slots and slot-values can be used, and there is nothing to preclude the use of record structures available in other languages.
5.1.3. Attribute Values. The value of an attribute can have multiple fields; for example, credibility value and/or time stamps. Often it is useful to maintain value history. This is especially true in continuous-input signal understanding programs in which temporal trends are important. In object-oriented systems, a value can be represented as a method that computes it when the slot is accessed.

5.1.4. Panels. Panels are useful when more than one hierarchy is needed to represent the solution state data. The most common use of a second panel is to hold control-related information, as in BB1.

5.2. The Knowledge Sources

A knowledge source represents knowledge needed to solve a sub-problem. All knowledge sources are event-triggered agents—the activation of knowledge sources are solution-state-dependent, rather than being process-dependent. The implication is that the most important consideration in the design of the knowledge source is its ability to evaluate the current state of the blackboard.

In all blackboard systems knowledge sources are designed with condition and action parts (Fig. 5).

5.2.1. Condition Part. The condition part can: (1) evaluate possible future states, or (2) make context-independent or context-dependent evaluation of the situation on the blackboard. In HASP, possible changes to the blackboard were categorized into different “event” types. The name of the event type was used to record a blackboard change. The event name was also used as a triggering condition in the condition part of the knowledge sources. In this type of construct the condition part is context-independent—it does not take into consideration the specifics of the change. Alternatively, the condition part can be a procedure that evaluates the specifics of the change, making the triggering of knowledge sources context-dependent as in HEARSAY-II.

One can think of the condition part as a filter for the activation of the action part. In newer systems, the design of the condition part consists of multi-staged filters. The first stage, called the trigger, is usually a context-independent filter that is followed by a number of context-dependent filters. If the action part of the knowledge source is a collection of rules, then the condition part of the knowledge source can be viewed as a high-level filter for the rule set. Rules have their own condition parts that serve as more detailed filters for the rule actions.

5.2.2. Action Part. The action part of a knowledge source makes modifications to the current blackboard state. The action part can be a procedure or a collection of rules. The selection of the knowledge representation scheme is independent of the selection of solution strategies.

Each knowledge source can make one or more changes to the blackboard. In a rule-based knowledge source, multiple changes can come from multiple firing of rules, or from multiple actions on the right-hand-side of rules.

The changes to the blackboard are new solution candidates (and new data) or modifications to the existing solution. Changes also can be made to the control information, as in BB1. In addition to changes to intermediate solutions on the blackboard, state changes can occur from the posting of goals and sub-goals to be achieved (Lesser & Corkill, 1983) or posting of model-based expectations of changes to occur in the future, as in HASP.

5.3. The Control

Control is the most complex component of the blackboard architecture, and it can have many architectural variations. The basic function of the control component is to select and apply knowledge sources in the case of recognition-based systems, or the condition and action components of knowledge sources in search-based systems. Major design differences occur in the following areas: (1) schedulable entities—whether parts of the knowledge sources are separate schedulable entities or not; (2) scheduling—whether the focus of attention is based on events or on knowledge sources; (3) noticing—who notices the changes in the blackboard; and (4) control data—where and in what form the control data is stored.

1. Schedulable Entities: In a search-based architecture the condition and the action parts of the knowledge sources are scheduled separately. In recognition-based architecture, each knowledge source is scheduled as a single entity. If the condition part is satisfied, then the action part is immediately executed.

One thing to note is that in the search-based architecture, the blackboard state may change between the time a condition part and its action part are executed. In such a situation the condition part must be re-evaluated.

2. Scheduler: There are two bases on which to approach the selection of the next thing to do:
a. A knowledge source, or knowledge sources, can be selected on the basis of the current state of the blackboard, or more specifically, on the latest changes to the blackboard, which are called events. Called event-oriented scheduling, the focus is on selecting the best solution island on which to base further computation. Most recognition systems take this approach. The basic function of the scheduler is to decide where to work next. Figure 6 shows a simple event-oriented control schema.

b. A knowledge source can be selected on the basis of what the knowledge source can contribute to the current state. This approach, called knowledge-oriented scheduling, is taken most often in search systems. The scheduler decides among the look-ahead evaluators and the generators of the solution space, and it applies the selected components to all the relevant data objects on the blackboard. The basic function of the scheduler is to decide what to do next. Figure 7 shows one possible way to design a knowledge-oriented control.

In reality, many problems can gain from a mixed strategy of search and recognition, and BB1, for example, allows for both event-oriented and knowledge-oriented scheduling.

5.3.1. Posting and Noticing Blackboard Changes. What changes are posted and who notices these changes differ from system to system. Often, in search-oriented systems, every change to the blackboard is a candidate for a solution, and there must be a knowledge source to evaluate each change. On the other hand, event-oriented recognition systems can process only those blackboard changes for which there is available knowledge. By making the event creation explicit, the rule writer can control what blackboard changes merit further action by the knowledge sources. For example, if a new piece of evidence is found to support a hypothesis but the only consequence is that the credibility rating is increased, this change may not need to be an event.
What blackboard changes should be events depend on the application.

5.3.2. Control Data. Control information needed to drive the system always resides outside the blackboard containing the solution state. In HASP different types of changes to the blackboard (modification to nodes, expectation posting, and so on) were stored in separate data structures. Each event was a recording of the type of change, the node that was modified, the new values, the rule that made the modification. In HEARSAY-II the control data was stored on a global list, the scheduling queue. The items on the queue were of two types: information generated by the condition part that pointed to its action part and all the places on the blackboard it could be applied to, and the names of the condition parts of knowledge sources. Since HASP-like systems focus on event scheduling, the control data are primarily about events; in HEARSAY-like systems that focus on knowledge source scheduling the control data are primarily about knowledge sources.

In BB1, control must determine what domain knowledge source to apply and to what object on the blackboard. The control problem is formulated as a planning sub-problem, and its solution state appears on a separate control panel. All data dealing with control are posted on the control panel and are shared by both the control-planning and the domain knowledge sources.

5.3.3. Domain-specific Knowledge, Meta-knowledge, and Control Knowledge. The distinction between domain-specific knowledge and meta-knowledge is not always clear cut. Meta-knowledge, knowledge about knowledge, is often thought to be weak, general, and domain-independent. But, in both search and recognition systems, meta-knowledge is often treated as control knowledge about what to do next, and this type of knowledge is often very domain-specific. In general, making distinctions between domain-specific and meta-knowledge does not contribute much to the design of the system; rather, it tends to distract. Instead it is preferable to organize knowledge sources hierarchically according to some control hierarchy. In most systems the control hierarchy partitions the knowledge sources into those that deal directly with the emerging solution states and those that deal with the process history or problem-solving states, called control states. The knowledge sources that operate on control states are called control knowledge sources.

6. DESIGNING APPLICATION SYSTEMS USING A GENERIC ARCHITECTURE

Up to this point we have discussed the three types of knowledge that are needed to create a generic blackboard architecture—knowledge about the characteristics of a class of problems, its solution strategies, and alternative design schemas to realize the solution strategy. As was shown in Figure 2, a generic architecture has explicitly associated with it (1) a meta-model consisting of a set of concepts and objects to describe the class of problems, and (2) knowledge about how to integrate domain information with the generic architecture to create application-specific architectures. In this section we describe these two items by means of examples from work done on the KASE project.

6.1. A Generic Architecture For Tracking Problems

6.1.1. The Tracking Problem. This class of problems can be described simply as follows: Identify, track, and sometimes, infer the intentions of objects in space using signal information about the objects (see Fig. 8). As examples, we use the HASP and the ELINT (Brown, Schoen et al., 1986) problems. In HASP the objects to be tracked are ships and submarines using passive sonar signals. In ELINT the objects to be tracked are airplanes using radar signals. In both problems, there are multiple sources of signal for each platform (engines and pro-

![Diagram](image-url)
Blackboard Systems at the Architecture Level

FIGURE 9. A generic architecture for tracking problems (based on Figs. 4 and 6).

pellers on ships, for example), and the platforms may be clustered together obviating individual platform identification.

A generic architecture is most often created by abstracting common architectural constructs from several applications in a particular problem class. In the case of blackboard systems, the blackboard architecture often can be converted into a generic architecture for a problem class by selecting the appropriate constructs for the various components and renaming some of them to fit the application glass. Figure 9 shows a generic architecture for tracking problems created in such a manner. The solution strategy is to use the recognition method with a simple context-independent, event-oriented control as shown in Figure 6. There are two blackboard panels, one for the solution space and one for control information.

6.1.2. Representation of Generic Architectures. In KASE, the generic architecture is represented as a collection of hierarchically organized functional modules. Each module is an object describing services (operations and procedures) that it provides, that it has access to, and that it needs to provide the promised services (see Fig. 10). Other relationships between the modules such as input/output and control relationships, can be derived from the service information. For example, input to a module is the collection of data types of the arguments of the service functions that the module provides, plus the results of the services it uses.

In a generic architecture any attribute of the modules can be abstracted as parameters. These parameters need to be instantiated with domain information. For the tracking architecture the following operations and procedures are treated as parameters: (1) the blackboard levels and the properties of the objects on each level; (2) the condition and the action parts of the knowledge sources; and (3) some control functions, such creating event types and selecting an event for focus of attention.

In order to provide automated tools to help a designer instantiate the parameters, KASE needs information about the domain. As shown in Figure 2, the domain information is acquired separately using the information in the meta-model.

| supermodule | a module that contains this module as a part |
| submodule | set of modules that are parts of this module |
| provides | set of services provided by this module to external modules |
| requires | set of services required by this module from other modules |
| has-locally | set of local data and procedures (services) within this module |
| has-access-to | set of modules that can provide services to this module |
| inputs* | data flow into this module |
| outputs* | data flow out of this module |
| calls* | modules called by procedures within this module |
| called-by* | modules that call procedures provided by this module |

FIGURE 10. Minimal internal representation of a module (Bhansali & Nii, 1992) (* derived properties).
6.2. Meta-Model and Domain Models

The names of the components that were shown in Figure 9 are parts of the meta-model associated with the problem class. For the class of tracking problems, some of the terms and concepts that form the meta-model are: for domain objects—objects-to-be-tracked, signal-collection-sites, and signal-data; for relations—line-of-bearing; and for events—object-to-be-track-updated, and so on. The properties of these objects are partitioned into attributes and operations. Some of the generic attributes and operations are: position, heading, and speed for the attributes; and create-object, compute-position, and compute-heading for the operations.

The domain models are created by instantiating the meta-model objects. Figure 11 shows instantiated domain-model objects for HASP and ELINT created from the meta-model associated with the generic tracking architecture. A radar-signal data in ELINT is called observation, and a sonar data reading over a given time period is called line-segment.

Figure 12 shows typical attributes and properties for the two applications. Typically, the attributes and operations are provided by the application designer. However, some generic operations, such as create-objects (instantiated as create-cluster and create-source) and compute-speed are provided as a part of the generic architecture. These operations can be instantiated for different applications as will be described in Section 6.3.

6.3. Customizing the Generic Architecture

The customization, or instantiation of the generic architecture, involves instantiating the parameters in the generic architecture using the information in the domain model. Because the customization often involves non-simple parameters, for example, determining knowledge sources, a variety of methods are needed. These methods may consist of selecting from a pre-computed list of alternatives, transforming an abstract schema using a set of transformational rules, or using heuristics design rules and/or algorithms. KASE stores all the necessary methods in its customization knowledge base (see Fig. 2). We illustrate the use of some of the design knowledge in the customization process through two examples.

6.3.1. Customizing Blackboard Levels. Given a domain model, determining the blackboard levels is quite straightforward. There are heuristic rules that suggest the levels. Some are generic to blackboard architectures

<table>
<thead>
<tr>
<th>Meta-model terms</th>
<th>ELINT</th>
<th>HASP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object-to-be-tracked</td>
<td>aircraft\ncluster\nradar emitter</td>
<td>fleet\nship\nsource\nharmonic\nline</td>
</tr>
<tr>
<td>signals</td>
<td>observation\nfeedback</td>
<td>line-segment\nintelligence-report</td>
</tr>
</tbody>
</table>

FIGURE 11. Some meta-model instantiation for HASP and ELINT.

<table>
<thead>
<tr>
<th>attributes</th>
<th>operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELINT Cluster</td>
<td>id\nposition\nheading\nactivity\nspeed\nthreat-potential</td>
</tr>
<tr>
<td>HASP Source</td>
<td>type\npoposition\nconfidence\nevolution</td>
</tr>
</tbody>
</table>

FIGURE 12. A typical attributes and operations of objects in HASP and ELINT (Boxed operations are generic operations that are part of the generic architecture).
in general: "If an object class is mentioned in the output report, then it should be a level," and "If an object class contains properties needed to compute any other properties, then it should be a level." Some others are specific to the generic architecture for a problem class: "If x is an object-to-be-tracked, then x should be a blackboard level."

Figure 13 shows object-to-be-tracked instantiated for HASP, and the objects in the HASP domain model made into levels. Using the above rules, other objects, such as collection-sites whose coordinates are used in line-of-bearing computation, will also become levels. The KASE-generated levels are complete in the sense that it will list all possible levels; they can be used as suggestions that can be pruned by the designer. For example, in most applications collection-site level is not needed.

6.3.2. Customizing Knowledge Sources. In KASE the knowledge source design consists of three parts. In addition to the knowledge source trigger (context-independent condition) and the action part described in Section 5.2., it has a part called "event posting," which makes explicit the need to post events on the control panel when the situation board is modified.

All the operations in the domain model are searched to identify those operations that either use or modify the objects on the blackboard. As shown in Figure 14, each selected operation becomes an action part of a knowledge source. For example, because compute-position uses bearing data, it is selected and becomes a part of a knowledge source.

Once the action parts of the knowledge sources are determined, design rules are used to determine events and triggers. An example rule for event creation is: "If an object is represented on the blackboard level, then create events for attribute of the object that can be modified." Thus, for the example above, new-position would be an event.

Rules that determine the knowledge source triggers are stated in terms of the operation that constitute the action part of the knowledge source. For example,

If an x updates the value of an attribute a1, and its value depends on the value of some other attribute a2, then any event that signals an update of the value of a2 must trigger the operation x;

and

If an operation x on an object takes as its argument an instance of another object, then any event that signals the creation of the new instance of the object should trigger the operation.

As shown in Figure 14, since compute-position operation uses an attribute new-bearing, any time a new bearing is computed the knowledge source containing compute-position operation must be triggered.
The creation of knowledge sources in the manner just described creates a complete set of knowledge sources needed to compute all the attributes on the blackboard based on the operations available in the domain model. However, the knowledge sources are small-grained and numerous, and there may be more events generated than necessary. In generating the knowledge sources for HASP, KASE generated 46 events and 26 knowledge sources, about twice as many as the actual system. The design can be optimized by the user. KASE provides some heuristic optimization facilities. For example, merging knowledge sources: if several knowledge sources are triggered by the same event, merge the operations into one knowledge source.

7. SUMMARY CONCLUSIONS

Blackboard systems are unique among AI systems in that they can be recognized by their architectural features. Yet, there has been very little formal discussion of systems in terms of their architectures, either in the AI or in the software engineering literature. In order to understand architectures better, we analyzed the blackboard architecture from the perspective of the kinds of knowledge it contained. An architecture at a level of detail sufficient to serve as specifications for programming is an integration of its problem statement, solution strategies for the problem, computational design schemas to realize the solution strategies, and specific information, operations, and constraints from the actual application domain. We enumerated on each of these areas.

In the KASE project, we introduced the notion of a generic architecture for a problem class and its solution strategies. Associated with a generic architecture is an ontology of terms about the problem class with which specific domain information could be acquired. The generic architecture also had associated with it design knowledge with which the generic architecture and domain information could be used to create application-specific blackboard architectures. We gave examples of design knowledge, accumulated in KASE, about blackboard systems that were used to automatically generate the designs of application systems.

The research on the customization process within the KASE project was conducted by Sanjay Bhansali (Bhansali & Nii, 1992a, 1992b). Parts of Sections 4 and 5 were taken from (Nii, 1989).

REFERENCES


