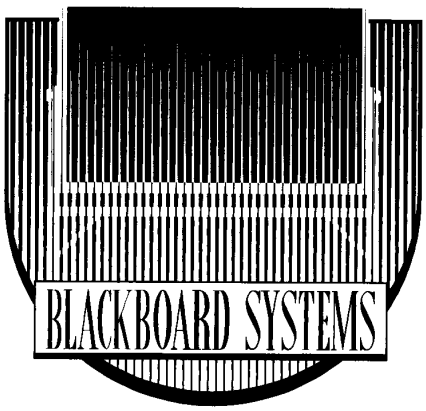

A Blackboard Approach for Diagnosis in Pilot's Associate

Bruce Pomeroy and Russell Irving
General Electric

Diagnosis is an essential activity for any system that controls autonomous equipment such as unmanned air vehicles or self-tending satellites; to be effective, the controls must detect and accommodate equipment faults. Control-advisory systems that interact with human operators have a similar requirement for diagnosis.

Pilot's Associate, a control-advisory system, is expected to fly in late-1990s tactical fighter aircraft. PA will improve pilot effectiveness in these complex aircraft by improving pilot awareness of not only external situations — threats, for example — but also the internal status of aircraft systems, and options for accommodating faulty equipment.

We have used a blackboard architecture in developing the internal-awareness function called System Status. Since this subsystem interacts not only with the pilot and the aircraft but also with other PA functions, let's digress for a moment to view this larger context before considering System Status details.

Purposes of Pilot's Associate

PA will provide several kinds of assistance to pilots. These functions include

(1) Assessing external threats and target environments;
(2) Assessing the internal status of aircraft systems;
(3) Planning the mission route, and replanning to respond to changes in threats or targets, or to accommodate equipment faults;
(4) Planning optimal tactics to perform mission goals within constraints of threat or target behavior and aircraft performance limits; and
(5) Planning emergency actions to correct faults or mitigate their effects.

PA's ultimate purpose is to display useful information to the pilot and to pass the pilot's action commands to the

0885/9000/90/0800-0039$1.00© 1990 IEEE

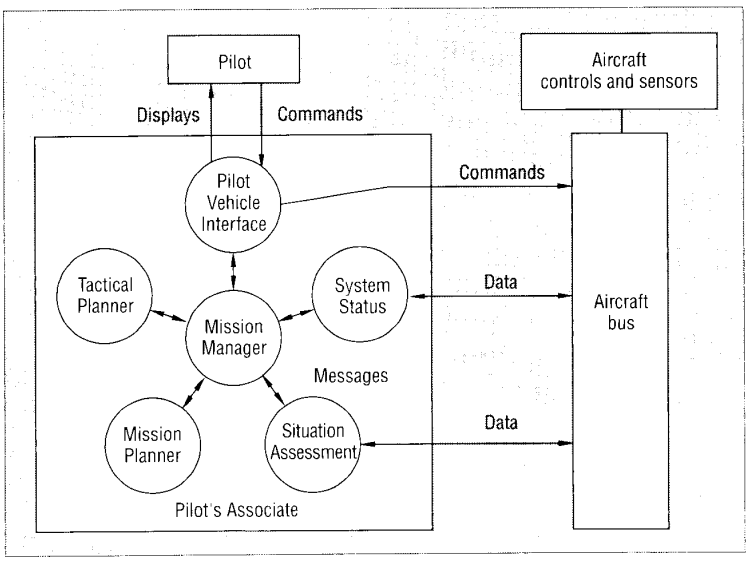

Figure 1. Pilot's Associate architecture and interfaces.

aircraft's actuators. PA monitors signals on the aircraft's digital control and communications bus to obtain data from battle sensors (for example, radar). It also monitors data on the functioning of aircraft systems, including temperatures, automatic controller commands, and error codes. Figure 1 illustrates the architecture of these functions and their interactions with the pilot and aircraft.

PA contains subsystems represented by circles in Figure 1. The Situation Assessment subsystem performs function **(1)** above, while the Mission and Tactical Planners are responsible for functions **(3)** and **(4)**, respectively. The System Status subsystem performs functions **(2)** and **(5)**, and supports functions **(3)** and **(4)** by providing data to planners describing aircraft operating limits.

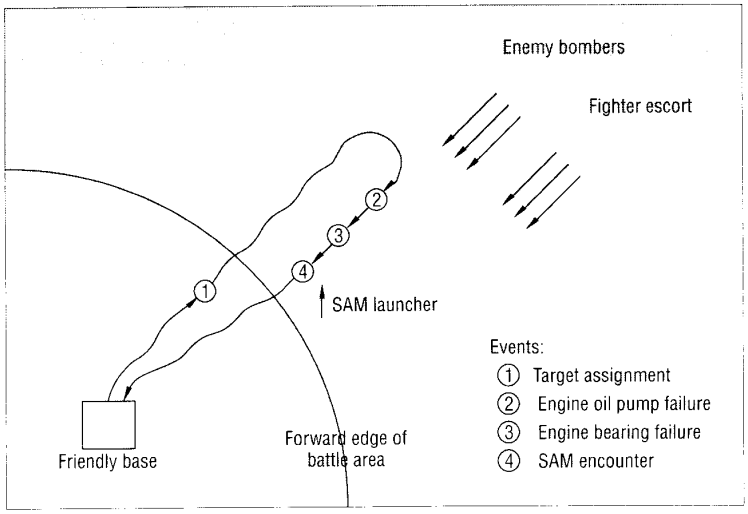

Figure 2. An example fighter mission.

System Status examples

To get a clearer view of System Status functionality, let's consider several examples showing how it aids pilots and supports other Pilot's Associate functions.

Consider the air-to-air mission described in Figure 2. A group of fighter aircraft is assigned an "on-call" mission: They fly to a point in friendly territory, and wait there until they receive a target assignment — enemy bombers, in this case. We will focus on four events in this mission.

When the target is assigned — Event 1 — the Mission Planner evaluates and selects ingress and egress routes through enemy defenses. SS supplies performance data to the Mission Planner during this process to assure that the route is within the aircraft's speed capabilities, and to establish a mission fuel budget.

After attacking the bombers, the lead fighter experiences an oil pump failure in one of its two engines — Event 2. SS detects this failure, alerts the pilot, and recommends that the engine be shut down because bearing failure is imminent. Anxious to leave enemy territory as soon as possible, the pilot rejects engine shutdown. SS proposes an alternate plan for running the faulty engine at 80-percent power to maximize bearing life. The pilot implements this plan.

Two minutes later, engine bearings fail — Event 3. SS alerts the pilot to the failure, and alerts the Mission Planner that the mission route's outbound leg is no longer feasible, because the fighter cannot achieve supersonic cruising speed as originally planned.

Shortly after engine failure, an enemy surface-to-air-missile launcher begins tracking the fighter — Event 4. The Tactical Planner evaluates the SAM threat, and begins planning evasion options. One option is to maneuver, using the fighter's superior turning ability to break the missile's radar lock. Another option is to deceive the radar by dispersing a decoy cloud of radar-reflective chaff. Normally, pilots would avoid the second option, because using a decoy is likely to make the fighter visible to other SAM radars that had not seen it previously.

However, estimates of single-engine performance provided by SS

force the Tactical Planner to select a decoy-turn maneuver, because the fighter cannot sustain a tight turn long enough for it to be effective against the SAM. During the decoy turn, SS provides turn-limits data to the pilot to optimize the fighter's remaining maneuverability and defeat the SAM.

These examples show that SS's role goes beyond detecting equipment failures; it is an active partner in helping the pilot and PA planners accommodate the aircraft's operational constraints. SS also provides options to the pilot for correcting faults or, at least, mitigating their effects.

SS function	Logic type	Input	Output
Diagnosis			
Fault detection	Kalman filters and models	Data*	Result**
Hypothesis generation	Constraint propagation and models	Result	
Test planning	Backward chaining	Result	Message***
Test evaluation	Forward chaining and models	Data	Result
Limits estimation			
Constraint generation	Forward chaining and models	Message	Message
Plan monitoring	Backward chaining and models	Result	Message
Corrective action			
Action planning	Forward chaining	Result	Message
Action monitoring	Forward chaining	Message	Result

* *Data = Input from bus*
** *Result = SS database assertion*
*** *Message = Input/Output with PA Mission Manager*

Table 1. System Status logic types and I/O.

SS architecture

The system architecture reflects the above roles. SS comprises the following groups of functions (see Table 1):

- **Diagnosis** — Monitoring the aircraft data bus to detect equipment failures, and conducting tests to isolate failures in a component or subsystem;
- **Limits estimation** — Providing the pilot and PA planners with estimates of the aircraft's operational limits, monitoring maneuver plans before and during execution, and alerting planners if a new failure makes the plan infeasible;
- **Corrective action** — Generating corrective-action plans, and monitoring these plans during execution to determine whether they have been effective;
- **Input/Output** — Decoding and encoding data from external transmission protocols to internal database representations; and
- **Control and database management** — Prioritizing and dispatching SS functions, alerting these functions to database changes, and maintaining database consistency.

We implemented the decision logic of these functions in Lisp and the KEE frame-based tool,[1] and adapted existing Fortran code to provide equipment simulation models for the diagnostic and limits estimation functions. We organized these functions as a blackboard control architecture following the model described in Hayes-Roth.[2] The functions are subdivided into modules based upon the intermediate results that each module would produce. Each of these modules is a blackboard "knowledge source," and the database is structured to express expected module interactions. Our current prototype contains 18 KSs and 331 blackboard database objects.

Figure 3 gives an overall structural view — showing the division of KSs into groups, and SS's interactions with the outside world. The important aspects of this architecture from the viewpoint of software engineering are the modularity and information management that result from the architecture's control and communication strategy.

System Status control. KSs communicate through the database; they do not communicate in any private way. KS control is centralized, and control decisions are explicit.

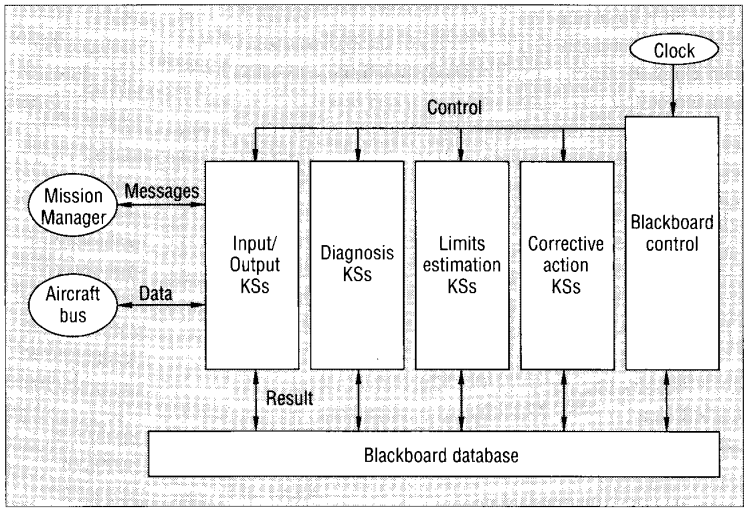

Figure 3. System Status architecture.

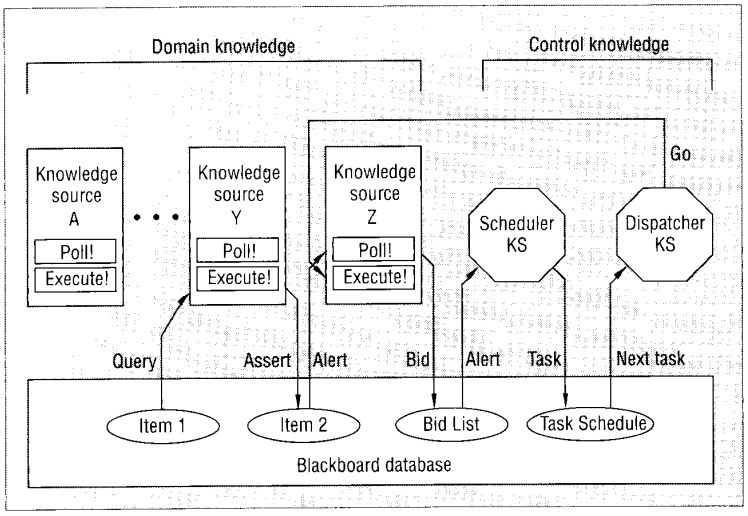

Figure 4. Blackboard control architecture.

We have implemented KSs as frames with data and method slots. Each KS frame can have several slots, but two of these slots must be methods entitled Poll! and Execute! with the following functions:

- **Poll!** determines when the state of the database is such that this KS can act, and bids an executable knowledge source activation record (KSAR) that will perform the action.
- **Execute!** evaluates a KSAR passed from the blackboard controller.

Figure 4 illustrates the function of these routines in the control scheme. The Dispatcher KS performs the lowest level control action of the blackboard when it selects the next task to be performed from the Task Schedule data structure, and activates the domain KS that performs that task. All higher level control actions trigger KS Poll! routines for generating task bids, and order tasks on the schedule.

For instance, consider what happens when knowledge source Y asserts new data to Item 2 of the database. This assertion causes the database to "alert" (that is, send a message to) the Poll! routines of all KSs. Each Poll! assesses the potential for action; in this example, only knowledge source Z finds that it can act. Z expresses this conclusion by posting a bid to the Bid List data structure. The bid is a frame with the following slots:

- **KS** — The name of the KS;
- **Focus** — The data item to be acted on — in this case, Item 2;
- **KSAR** — The KS activation record (this executable Lisp expression sends messages to the methods and rules of the bidding KS that perform the action);
- **Result** — A list of message types and database changes that will occur when a KS executes a KSAR;
- **Urgency** — An index expressing the relative urgency of this bid in comparison with others having the same expected result; and
- **Before** — A list of bids that must be performed before this one.

Posting a bid activates a special KS — the Scheduler (see Figure 4) — which embodies SS control strategy. The Scheduler KS prioritizes tasks, both new and old, on the Task Schedule: It sorts tasks with respect to their Result and Urgency rankings (stored in a static table), with adjustments for task precedence expressed in the Before slot. When this KS finishes scheduling, the Dispatcher KS resumes its cycle by selecting Z's task from the schedule, extracting the task's Focus and KSAR elements, and sending them to Z's Execute! method. For the tests reported below, KSAR queue sizes varied from a maximum of 20 during SS initialization to an average of four when SS was simply monitoring incoming bus data.

This control mechanism, implemented in the current prototype, has three major deficiencies from the viewpoint of SS interactions with the pilot and other PA subsystems. First, control decisions are not context dependent. We could make our current approach to prioritization situational by adding logic to use different tables in each mission context — for instance, to cause different SS behaviors in cruise mode versus engaged-offensive mode. However, the context in which the system operates constitutes a complex space of tactical and mission plans and goals[3,4] with potentially numerous modes. In this environment, assigning priorities through knowledge-based planning[2] may be a better approach. Metaplanning concepts developed in the Molgen[5] and PAM[6] projects point toward methods for merging the reactive behavior of KS Poll! routines with a strategic level of planning about longer range goals of the pilot and PA planners.

Second, control does not recognize task threads. SS ranks each task without considering that task's relationship to previously executed tasks or to tasks that may be bid in the future. Therefore, the controller cannot distinguish important task sequences, and SS actions may seem random to a human observer. Random behavior is most likely during overload situations where several threads of activity are intertwined. Metaplanning might solve this control problem by relating tasks to long-term goals.

Third, control actions are hard to explain. Insofar as SS control decisions seem random, they will be difficult to explain to the pilot. Building on previous work[6]

demonstrating that one can represent human language in terms of plans and goals, PA developers have adopted planning as the underlying logic[7] for understanding pilot action. If we were to use metaplanning as the control logic for SS, these concepts for understanding could be turned around to provide explanations that relate SS temporal activity to the system's near-term and long-term goals.

System Status blackboard database. The SS blackboard database maintains a record of KS intermediate results in the following data structures:

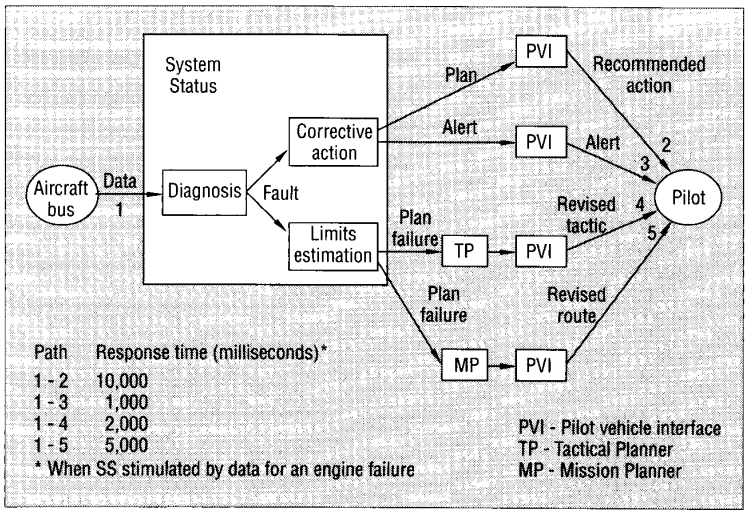

Figure 5. PA end-to-end response time requirements.

- **Messages** — Messages from the external environment (for example, queries from the pilot or PA), and messages pointing to results achieved by SS processing (for example, answers to queries awaiting transmission);
- **Bus data** — Time-sequenced data packets from the aircraft control bus; and
- **Control** — The Bid List and Task Schedule.

Our system implements structures as class frames,[1] and instances as member frames of these classes. SS reuses instances to avoid the processing cost of creating and destroying frames. The system maintains a record of the aircraft's physical and functional health in the Status database, whose frames represent aircraft components and their functions. Dependency links connect components to functions. Physical links describe system-subsystem relations — for example, the ailerons "contribute-to" the flight control system. Functional links describe how aircraft functions depend upon these systems — for instance, the flight control system "contributes-to" turning.

As a causal-association network,[8] the Status database provides the major link between SS diagnosis and limits estimation functions: The network propagates effects of physical failures to assess their impact on functional capability. Each frame points to its parents and children, using slots, which form contributes-to and reciprocal depends-on links in the network. Four database routines implement the active part of the network. Three of these routines are generic, and are inherited by all network frames through class-member links. They are as follows:

- **Assert!** changes data in the slots of a network frame, and triggers a message to that frame's Alert!;
- **Alert!** marks the data in a frame as "stale," and sends a message to the Alert! of each parent in the frame's contributes-to list. It also sends messages to trigger the Poll! routines of all KSs; and
- **Query!** retrieves data from the slots of a network frame, and sends a message to that frame's Update! if the data in that frame is "stale."

In addition, each frame has an Update! routine that queries the frame's depends-on children, computes fresh values for the frame's own data slots, and marks the frame's data as "fresh." This routine is specific to the domain characteristics of each frame.

These four routines not only trigger KSs but also implement a selective database update strategy. Assert! and Alert! work together in a forward-chaining manner to trigger a KS's Poll! routines when the network changes state; they also leave a trail of stale data markers to indicate which parts of the network require updating. Updates are expensive — SS performs them only on demand and only for focused portions of the network. Controlling this update behavior, Query! and Update! produce backward chaining through dependent portions of the network when data is requested from a stale source.

Performance testing and evaluation

PA designers defined performance metrics early in the development cycle,[9] specifying response times required to support pilot tasks. Figure 5 shows an end-to-end analysis of PA dataflows, providing top-level specification of SS processing-response times as follows:

- **Diagnosis** — From appearance of fault data on the bus to isolation of faults (700 milliseconds);
- **Limits estimation** — From isolation of faults to transmission of plan failure messages (300 milliseconds);

Table 2. System Status response time requirements.

SS functions	Required response (milliseconds)	Actual response (millisecond range)
Control and database		
Schedule bid	15	5 - 20
Dispatch bid	10	7 - 38
Maintain DB consistency	15	
Alert!		247 - 2960
Assert!		380 - 3118
Query!		1 - 37065
Posting message	10	365 - 1793
Posting bid	10	30 - 48
Input/output		
Bus monitor	20	1654 - 3265
PA monitor	10	984 - 1989
Report generator	20	24 - 5090

• **Corrective action** — From isolation of faults to transmission of alert messages (250 milliseconds), to transmission of corrective-action-plan messages (1000 milliseconds).

Table 2 shows how we allocated these time budgets to SS functions.[10] We had two goals in mind: The system had to meet requirements set by the end-to-end analysis above, as well as perform useful work while maintaining a maximum I/O lag of one second. With this latter goal in mind, we assigned control and I/O budgets so that these functions would take no more than 110 milliseconds in any cycle for reading input, scheduling, and reporting results.

The actual-response column lists initial results from real-time testing. While bid, schedule, and dispatch activities were fast enough for real time, database management activities were slow. This sluggishness is due in part to the KEE environment, which imposes an overhead on frame access functions as the price for its flexibility. We have relied heavily on KEE search routines to find frames by name within knowledge bases; access would have been faster had we maintained full path descriptions. Another factor is the combined size of SS knowledge bases and the KEE environment, which causes excessive disk-to-RAM swapping; the variability in Table 2's actual times is partly due to garbage collection, but is primarily attributable to this swapping.

The current responsiveness problem is actually worse than these single-frame access times imply, because SS often accesses many frames in each cycle. The contributes-to network can generate enormous Alert! cascades from a single Assert! action. Similarly, a single Query! can cause a large cascade of Update! activity. Once triggered, these cascades cannot be interrupted in the current prototype, and they happen frequently due to the dynamic character of the Status database.

Finally, regarding the issue of real-time responsiveness, SS will never be fast enough for all situations — not because it is inherently slow, but because processing power may not be available due to competition from other subsystems. SS will probably be overloaded at exactly the moment it is most needed — during combat after severe aircraft damage. Clearly, this subsystem must be faster, but the performance standard given in Table 2 is too rigid for software of this type. This is the kind of specification appropriate for "hard" real-time systems like aircraft flight controls; however, SS has a much wider range of activity than flight controls, and the importance of its outputs varies with time. It is probably more appropriate to judge its performance against a standard of "responsiveness" that specifies situation-dependent processing times and a required ordering of responses for several task overload scenarios.

Conclusions

First, let's summarize positive elements of our experience with the blackboard architecture. The blackboard emphasizes explicit control and indirect communication; when combined with frame-based code structuring, these features facilitated a modular software design. Modularity and a combination of information hiding inside frames with information sharing on the blackboard have helped us develop SS in the following ways:

• **Providing a uniform control mechanism** — We applied the Poll!/Execute! cycle to rule-based and procedural KSs easily. KEE provided convenient structures for integrating these paradigms within a single process, but the control scheme could just as well have been implemented across multiple processes or processors.

• **Clarifying the rationale for subdividing the system** — KS functionality is described in terms of intermediate results; this caused the design team to focus early on the cooperative nature of KS activity.

• **Providing a robust environment for module integration and testing** — Since KSs do not communicate directly, it is possible to add new KSs or modify existing ones without endangering vital but hidden communication links. It was also easy to run with partial software configurations, because control logic is localized in the Scheduler KS and easily modified.

• **Permitting incremental development of control knowledge** — The present prototype functions reasonably well with task scheduling based on a static priority table, yet the architecture permits expansion to more responsive methods.

We have already discussed problems we found in the current blackboard architecture. We are investigating solutions to these problems, as follows:

• **Frame access is slow** — Insofar as this problem is due to KEE, we can solve it by changing to more compact frame systems, including Flavors or CLOS, or to non-Lisp object-oriented representations like C++ or Objective-C. We can mitigate the other source of slow access — searching for frames — by using efficient memory organization and frame access by path name, as practiced in the University of Massachusetts' Generic Blackboard.[11]

• **RAM-to-disk swapping slows processing** — The easiest and cheapest near-term solution is simply to buy more memory. However, as SS moves toward a deployable real-time form, it must shed its KEE shell to become smaller, and we may need to implement explicit memory management to achieve task-coordinated swapping.

• **Assert!/Alert! cascades reduce responsiveness** — While it is possible to speed up individual frame access operations, we have reason to believe that these cascades will always be a problem. We may have to limit the scope of the network to only those relations required for fault propagation at a high level of abstraction. This is a design trade-off between explanatory detail and responsiveness.

• **Query!/Update! cascades reduce responsiveness** — Limiting the scope of the contributes-to network would also reduce this processing load. SS could control cascades if the database were reconfigured so that update tasks were scheduled just like KS tasks. This would enable the originating KS to decide whether it has time for this Query! in its current task; if not, the KS could bid to complete the task later or bid a similar, short-cut task to get the same answer.

System Status development has required the integration of logic expressed in several different programming paradigms — for instance, rules, frames, and procedural code in Lisp and Fortran. When we began designing the system in early 1986, blackboard architectures — pioneered in the Hearsay project[12] and developed more recently into a generalized concept for software control[2,13] — seemed to offer a solution to this integration problem. Since that time, several SS prototypes have proved the blackboard to be a robust and useful tool for the entire software engineering cycle — from design through testing. Furthermore, when combined with frame structuring of knowledge bases, the blackboard scheme helped solve several difficult software-project-management problems.

However, the SS team is now focusing on real-time operation, and concern has arisen that the blackboard's overhead may compromise responsiveness. Initial testing against real-time metrics found the system slower than required. While the blackboard contributed to this sluggishness, this architecture is still desirable because of its software engineering benefits. We are investigating changes in the SS blackboard — including a more compact frame representation, faster database access and management techniques, and a better focus-of-attention mechanism — to provide the required responsiveness.

Acknowledgments

As part of a Lockheed team, GE developed the System Status function of Pilot's Associate. The US Defense Advanced Research Projects Agency sponsored this work under contract F33615-85-C-3804 in cooperation with the US Air Force. The insights summarized in this article resulted from a fusion of many contributions; the authors are especially grateful for the opportunity to share ideas with David Smith (Lockheed), Norman Geddes (Search Technology), Gary Edwards (ISX Corporation), Stanley Larimer (formerly of GE), H. Austin Spang III (GE), and David Tong (GE).

References

1. *KEE Software Development System — Core Reference Manual*, Intellicorp, Mountain View, Calif., 1986.
2. B. Hayes-Roth, "A Blackboard Architecture for Control," *Artificial Intelligence*, July 1985, pp. 251-321.
3. C.A. Leavitt and D.M. Smith, "Integrated Dynamic Planning in the Pilot's Associate," *Proc. AIAA Guidance Navigation and Control Conf.*, American Institute of Aeronautics and Astronautics, Washington, D.C., 1989, pp. 327-331.
4. "Phase 1 Interim Report of the Pilot's Associate Program," Tech. Report PA-INT/RPT, Lockheed Aeronautical Systems Company, Marietta, Ga., 1988.
5. M. Stefik, "Planning and Meta-Planning (Molgen: Part 2)," *Artificial Intelligence*, May 1981, pp. 141-169.
6. R. Wilensky, *Planning and Understanding*, Addison-Wesley, Reading, Mass., 1983.
7. W.B. Rouse, N.D. Geddes, and J.M. Hammer, "Computer-Aided Fighter Pilots," *IEEE Spectrum*, Mar. 1990, pp. 38-41.
8. S.M. Weiss and C.A. Kulikowski, *A Practical Guide to Designing Expert Systems*, Rowman and Allanheld, Totowa, N.J., 1984.
9. "System Specification for the Pilot's Associate Program," Tech. Report PA-SS, Lockheed Aeronautical Systems Company, Marietta, Ga., 1986.
10. "Subsystem Descriptions for the Pilot's Associate Program — System Status Manager Subsystem," Tech. Report PA-SD, Lockheed Aeronautical Systems Company, Marietta, Ga., 1987.

11. K.Q. Gallagher, D.D. Corkill, and P.M. Johnson, "Generic Blackboard Development System — Reference Manual," Tech. Report 88-66, Computer and Information Science Dept., University of Massachusetts, Amherst, Mass., 1988.

12. F. Hayes-Roth and V.R. Lesser, "Focus of Attention in the Hearsay II Speech Understanding System," *Proc. Fifth IJCAI*, AAAI, Menlo Park, Calif., Aug. 1977, pp. 27-35.

13. H.P. Nii, "Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures," *AI Magazine*, Summer 1986, pp. 38-53.

Bruce D. Pomeroy is a mechanical engineer at the GE Corporate Research and Development Center, having joined the staff in 1973. He directed the team responsible for the System Status Manager, one of six expert systems in Pilot's Associate. Prior to the PA project, he designed a smaller, free-standing expert system for equipment diagnosis and maintenance of electric utility gas turbine control faults. He also contributed to Jet-X, a maintenance-aiding system for military jet engines. Before becoming interested in AI, he worked on complex physical systems and energy-related projects. He holds a PhD in mechanical engineering from the University of Michigan, and is a member of the IEEE Computer Society, AAAI, and ASME.

Russell R. Irving is a computer scientist at the GE Corporate Research and Development Center. His work centers on AI applications involving advanced diagnostics and intelligent interfaces as they relate to the aerospace industry. Previously, he worked at GE's Advanced Technology Laboratory, where he was involved in real-time AI systems development and contributed to the Pilot's Associate System Status project.

A graduate of GE's Software Technology Program, he received his BS in computer science from Siena College and his MS in computer science from Rensselear Polytechnic Institute.

The authors can be reached at the GE Corporate Research and Development Center, PO Box 8, Schenectady, NY 12301.