APPLYING BLACKBOARD TECHNIQUES TO REAL-TIME SIGNAL PROCESSING AND MULTIMEDIA NETWORK MANAGEMENT

Randall J. Calistri–Yeh
ORA
301 Dates Dr.
Ithaca, NY 14850-1326
Email: calistri@oracorp.com

ABSTRACT

Two recent projects at ORA have used a blackboard framework during system design.[1] DANA is a multimedia network monitoring system, while MARRS is a real-time passive radar system. Both projects have benefited from the AI principles captured in a blackboard design: hierarchically ordered data structures, independent knowledge sources, and intelligent opportunistic control. DANA has profited most by using the knowledge sources to implement a policy of guaranteed update consistency. MARRS has used the independence of knowledge sources and the flexibility of opportunistic control to support quick and easy experimentation with new system designs. This paper presents a new way of incrementally adding opportunistic control to an existing system, and describes several lessons that we learned about the benefits and difficulties of designing and implementing blackboard systems.

INTRODUCTION

A blackboard architecture [EM88, Nii86a, Nii86b, Nii89] consists of three main components:

1. A collection of independent modules called **Knowledge Sources** that contain specialized knowledge needed for solving one part of the domain problem. Each knowledge source can use any method to come up with its contribution: numerical analysis algorithms, rule-based expert systems, neural networks, etc.
2. A global memory area called the **Blackboard**. The blackboard is organized hierarchically, and knowledge sources typically read information at one level and write at one (or more) levels.
3. A separate opportunistic **Control** module. Rather than having a fixed sequence of steps that is followed blindly in a control loop, the control module considers all potential contributions from each knowledge source and selects the one that is most appropriate to apply at the current time. The control module selects only among knowledge sources that are known to have something to contribute; each individual knowledge source independently indicates whether or not it can currently contribute to the problem-solving process.

Blackboard architectures offer several advantages over traditional designs. First of all, since each knowledge source is treated as a black box by the rest of the blackboard, higher levels of modularity are automatically enforced. This modularity allows easier rapid prototyping, development, and maintenance. Secondly, each knowledge source is free to use whatever computational means is most appropriate to arrive at its contributions to the overall solution, regardless of the methods used by other knowledge sources. Independently selecting the computational methods allows the various knowledge sources to represent and reason about their data in completely different ways; the only constraint is that they must agree on the representation of data in shared areas of the blackboard. Finally, opportunistic control allows more flexibility to process the most appropriate information at the most appropriate time, rather than being locked into a more rigid control loop.

Two recent projects at ORA have used blackboard principles during system design. In the next section we describe DANA, a multimedia network monitoring system for Navy communications networks. We then present MARRS[2], a real-time passive radar system. After presenting our new method for real-time opportunistic control, we compare the two systems and conclude by describing the lessons that we learned during the two projects.

DANA

DANA (Display and Analysis of Network Activity) is a system that can monitor, record, report, and dis-

[1] This research was funded in part by the US Navy (NRaD) under contract N66001-91-C-7015.

[2] MARRS (Made-up Acronym for a Real-time Radar System) is the fictitious name of a real system; the name has been altered to protect the anonymity of the developers.

play various information about arbitrary communications networks. This information can include data such as hardware status, network traffic, and various statistical information. DANA was developed for the US Navy (NRaD, San Diego); to serve as a network monitoring system for the Communication Support System (CSS) [Nav90, Nav91a, Nav91b], a next-generation architecture that provides multimedia access, load sharing, and modular updates for Navy communications.

The CSS is defined by a layered network design called the connection plan (connplan). At the highest layer of a connplan are **users**. Users require certain communications **services** such as intelligence queries in order to carry out their tasks. Each service is mapped onto a set of **resources**, which describe distinct physical paths through **hardware** components such as modems and UHF satellites. Every n seconds, DANA polls each hardware component in the communications network to request basic status information. From this raw data, DANA calculates various statistical information. Using this raw and derived data, DANA calculates the status of all CSS resources and services, and it also calculates the effects of these status changes on each CSS user. Any significant changes are reported to the user using a customizable graphical viewer.

DANA must operate in real time, since it needs to process all of the status information from one polling interval before the next interval starts. However, since DANA has some control over the polling interval, the system can be considered "soft" real-time, as opposed to "hard" real-time systems that have no control over the incoming data rate, such as the MARRS system described in the next section.

We had three primary design criteria during the initial development of the DANA architecture. The most important was flexibility to adapt to an unknown underlying network design. At the time, the CSS architecture was in a constant state of flux, and the hardware that we would be monitoring had not even been completely determined. The second criterion was update consistency: we had no way of knowing what other software modules might have access to DANA's databases, and we still needed to guarantee that the appropriate changes would propagate to all affected users, services, resources, and hardware even if some external module made the changes. Finally, we had a requirement of viewer consistency: two people using DANA at the same time at different locations should see exactly the same thing.

These three criteria can be supported by a blackboard design. The viewer consistency argues for a single central data repository that can be accessed by all viewers. The flexibility argues for independent knowledge sources that can be plugged in and out without affecting other parts of the system. And the update consistency argues for a flexible control module that can enforce propagation of changes regardless of where the changes are originating.

The resulting blackboard design is shown in Figure 1. The global blackboard, containing hierarchical data levels such as `Hardware` and `Resources`, is stored as a database using the Interbase database management system [Int90]. Primary knowledge sources are implemented as separate C modules such as `Update_Hardware` and `Update_Resource`; they are alerted by changes at one level of the blackboard and propagate those changes to the next higher level. Control is maintained through *database triggers*, which are independent daemons that essentially act as miniature secondary knowledge sources. A trigger is fired whenever its preconditions are satisfied by a blackboard update. DANA uses triggers for three purposes:

- **Statistical triggers** update the statistical information whenever a status record changes. These triggers are shown as dashed arcs in Figure 1.
- **Propagation triggers** make simple updates to higher levels when appropriate status changes occur. For example, one trigger will change a resource status to "down" whenever any of the resource's hardware components are down. These triggers are shown as dashed lines from one data level to another.
- **Notification triggers** alert the primary knowledge sources to perform more complicated updates such as updating certain time-aggregate summaries. These triggers are shown as dashed lines from a data level to a knowledge source. The updates performed by the primary knowledge sources are shown as solid lines.

Each trigger has a priority associated with it, so control can be fine-tuned by adjusting the order in which simultaneous triggers fire[3]. Consistency is maintained because the triggers fire whenever the appropriate changes are made to the database, *regardless* of how those changes are made. This implementation turned out to be a very useful way of guaranteeing consistency, especially with cascading updates such as one hardware update setting off a long chain of modifications.

Unfortunately, this design does not lend itself to modular code and independent knowledge sources. Because of system requirements from Interbase, the code associated with one data type is scattered between database record definitions, database trigger definitions, and primary knowledge sources. We also use a special-purpose communication protocol for passing information between the blackboard and the graphical viewers, and this language is sensitive to the types of objects stored in the blackboard. Therefore, a modification such as changing the type of status information collected for a particular piece of hardware requires carefully coordinated changes to several separate source files. With more careful coding, we could have simplified some of the modification problems,

[3] These prioritized triggers worked reasonably well as a control structure, but they meet only the minimal definition of opportunistic control required for a true blackboard system.

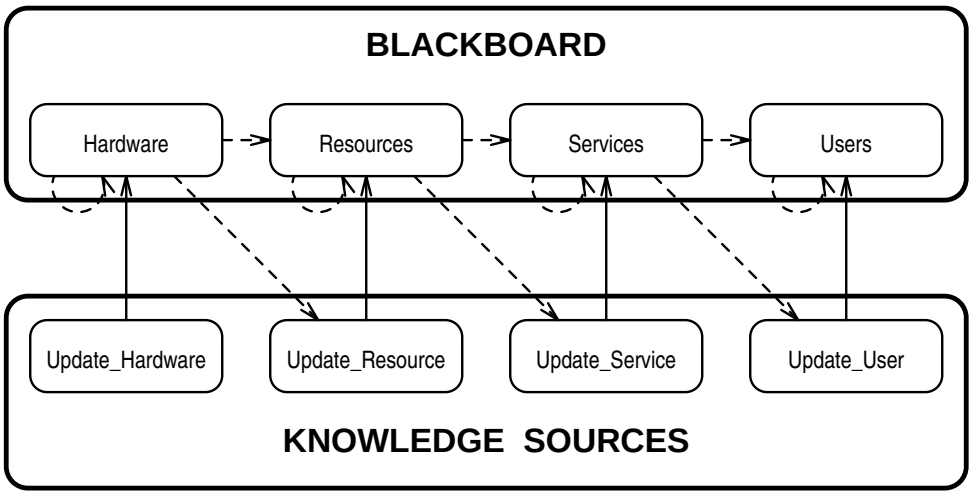

Figure 1: Blackboard design for the DANA system.

but it will never be possible to completely modularize DANA using the current architecture.

MARRS

A computer company (name deliberately withheld) is currently in the process of reimplementing a passive radar system that has been in development for a number of years. The company is redesigning and enhancing the current system to create a new system, referred to in this paper by the pseudonym "MARRS". ORA's role in this effort was to provide consultation on software techniques and overall system design to be used in its reimplementation of the MARRS system.

The MARRS system is part of a passive radar system that tracks multiple targets in real time. The purpose of the MARRS system is to take information from a digital signal processor, identify specific aircraft targets, and track those targets over time. While the details of the algorithm can not be described in this paper, the data processing is quite similar to that used by the AMEP[4] system [Bar89, BATW87]. In the AMEP system, a front-end processor receives electromagnetic **signals** from a transmitter at fixed time intervals and translates these signals into **pulse** descriptors. These pulses are then assigned to **track fragments** based on sharing common parameters. The fragments are analyzed and merged into a single **track** if it is determined that they belong to the same emitter (**target**).

Although an earlier working prototype of MARRS was successfully built, the implementors were not completely satisfied with the design. Their main concern was that it was a very rigid system. They had several interesting ideas that they wanted to try, but they decided that it was too difficult to modify the prototype to incorporate the new ideas. They also found it difficult to add new data features; for example, a new data level was only partially implemented because it was so difficult to incorporate new information into the existing data structures. Finally, they had reached the limit of computing power on their current hardware platform, and they were hoping that upgrading to more powerful hardware would keep the system running in real time for the short term.

These shortcomings led to a set of design criteria for the new MARRS system: system designers and future maintainers must be able to add new layers to the global data structure easily, they must be able to replace modules easily, they must be able to experiment with different system designs and ideas easily, and the final system must be able to meet hard real-time requirements with graceful degradation in the event of data overload.

We decided that this project was a perfect opportunity to apply blackboard design methods, since all three components of a blackboard architecture could benefit the MARRS design. The existing data types and processing functions decompose into blackboard structures quite naturally. By enforcing independence between the knowledge sources, it will be easier for future implementors to perform design experiments by plugging modules in and out (for example, using AMEP terminology, testing a new algorithm for merging track fragments will only involve replacing the Track Manager knowledge source). Also, by using an opportunistic control module, the new MARRS system will be able to run more gracefully under peak data periods. The proposed blackboard design for MARRS[5] is shown in Figure 2. Each knowledge source observes

[4] AMEP was developed by the Defence Research Establishment of Ottawa (DREO). We do not imply any connection between AMEP and MARRS, or between DREO and the developers of MARRS.

[5] The blackboard data type names and the knowledge source names are taken from the AMEP system, but the actual data organization in the MARRS system is very similar. No implications about the inner workings of the AMEP system are intended.

changes made at one data level and incorporates the new information into the next higher level, possibly revisiting earlier decisions that it had made at the higher level.

The designers of MARRS expressed three primary concerns with adopting a blackboard architecture: they perceived it as too slow, they suspected that it would be too hard to debug based on their experiences with similar architectures, and they did not see any need for opportunistic control in the current system.

Upon further analysis, we concluded that all three concerns related to the opportunistic control module. The main source of performance degradation over traditional architectures is the overhead associated with the more advanced control module. Because of their independence, the knowledge sources should actually be *easier* to debug than traditional modules; however, opportunistic control does make it more difficult to trace the execution path between modules. The final criticism, that the MARRS system did not need opportunistic control, is actually true for the current situation. This is because there is a single source of information (incoming signals from one transmitter), there is a single direction of information flow (from signals to targets), and there is no resource exhaustion (all incoming information can be processed completely during each cycle without falling behind). However, all three of these properties may very well change in future versions: multiple sources of information may contribute information at a variety of levels and may overload the system, forcing it to choose which information to process immediately and which to postpone.

Based upon our analysis, we proposed that the new MARRS system use a full blackboard architecture, including opportunistic control. However, in order to provide a minimum transition time between functional implementations, we realized that the implementors would need to first lay the basic blackboard framework but continue to have the MARRS system perform exactly as the current version does. They would then be able to add more useful features over time, gradually transitioning the system to take full advantage of the blackboard design. As part of this transition process, we proposed the opportunistic control method described in the next section.

TRANSITIONING TO REAL-TIME OPPORTUNISTIC CONTROL

Based on our experience in the DANA and MARRS systems, we have developed a constant-overhead opportunistic control design that is suitable for real-time systems. This design provides the ability to transition gradually from traditional fixed control setups to full opportunistic configurations.

Associated with each knowledge source k in the blackboard system is a companion **utility function** u_k that tells the control module how useful or important is it for this knowledge source to execute at the current time. This utility function returns a real number between 0 (k should not execute at all now) and 1 (k must execute immediately). This value reflects both the importance of the information that the knowledge source could potentially contribute to the solution and the timeliness of that information. In order to provide real-time response, the utility functions must be constrained to return in constant time, using simple and efficient tests as a heuristic for how useful the (still uncalculated) information *might* be. The utility function is myopic and egotistical, in the sense that it views its own knowledge source as the only one that can contribute to the solution. The control module applies a multiplicative weight w_k to the utility of each knowledge source based on that knowledge source's relative importance compared to the rest of the system.

During an initialization phase, possibly involving timing tests, each knowledge source k sends the control module an estimated upper bound of α_k, the amount of time to perform one atomic action[6]. The control module uses this information to allocate the available time in the most efficient manner. After initialization, the control module runs in a fixed loop cycling every s seconds (s may be less than 1). Each time through the loop, the control module polls each knowledge source k for its utility u_k, weights the utilities using w_k, and ranks the weighted utilities. It then allocates the available time for this cycle to the n top knowledge sources, using the estimates of α_k.

This style of opportunistic control leaves the decision-making responsibilities exactly where they should be. The question of whether a particular knowledge source is ready to execute can be decided only by that knowledge source itself. But since knowledge sources are independent, no knowledge source can decide by itself how important it is compared to another one. Only the top-level control module can make that determination.

Since this control methodology is very different from the traditional fixed control loop that most developers are used to, and since it may be difficult to debug the entire system when each knowledge source essentially executes nondeterministically, it is useful to provide a way to mimic a fixed control strategy. It is easy to implement a traditional round-robin nonpreemptive control loop: the utility function for each knowledge source simply returns a constant (1), the control module uses fixed weights to ensure that the knowledge sources always execute in the same order, and each knowledge source is allocated an unbounded amount of time.

This general control algorithm is a powerful mechanism that supports quick and easy experimenting with new designs. Here are two examples:

- To simulate a parallel distributed environment, simply allocate the minimum amount of time to each knowledge source (based on α_k) instead of an unbounded amount of time. The knowledge sources

[6] An atomic action is defined as the smallest task that the knowledge source can completely execute and leave the blackboard in a consistent state.

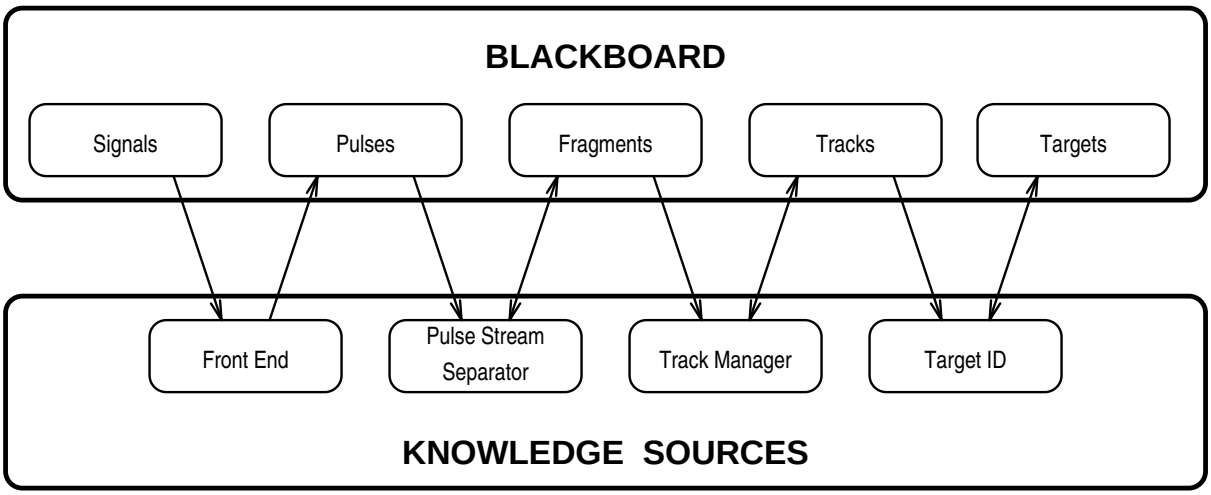

Figure 2: Blackboard design for the MARRS system.

and the opportunistic control algorithm itself do not need to change at all.

- To add a new knowledge source, just include its utility function in the control module's decision process and it will be automatically integrated with the other knowledge sources. To test it as a separate multiplexed process, the control module can artificially assign a maximum weight to the new knowledge source and always make sure that the new knowledge source gets exactly half of the available time.

Using constant-time utility functions for speedup is quite different from RETE-like pattern-matching networks suggested by others [For82, HH93]. These networks are capable of significant speedup in general blackboard systems, but we feel that it is overkill in many cases. For example, the MARRS system has a small number of knowledge sources, and testing whether one of the knowledge sources can be activated is trivial (simply test to see whether new data has arrived at the lower level). The overhead of maintaining the RETE network would be more expensive than just flagging changes at each data level. Deciding on the most advantageous order for activated knowledge sources is a little trickier, but this case is precisely where the constant-time utility functions are useful because RETE networks do not address ordering at all.

COMPARISON

DANA and MARRS have several similarities that make them excellent candidates for blackboard designs. Probably the most important feature that they share is that each system's data decomposes naturally into hierarchically layered structures that can be manipulated by software modules (knowledge sources) operating at one layer. Without this feature, a blackboard implementation would be very difficult.

The two systems also share a common need to anticipate and provide for unknown future features. In the case of DANA, we do not know what the final communication network will look like, we do not know what the hardware components will be, and we do not know what information will be contained in the status messages that we are monitoring. In the case of MARRS, we do not know what new features may be added to the system in the future. A blackboard design supports this need by encouraging modularity in the development of independent knowledge sources; if knowledge sources are truly independent, they can be added, modified, or removed without affecting the rest of the system.

A third common feature between DANA and MARRS is the need for real-time performance. Blackboards are often criticized for their high overhead cost and their inability to run in real time. However, the method we described for implementing opportunistic control in MARRS only adds a small constant overhead to the processing time, and the added benefit of more robust performance during short-term overloads more than makes up for that overhead.

One difference between the two systems is that we were designing and implementing DANA from scratch, while we were proposing a redesign of the existing MARRS system. Another difference is that the MARRS system is a single executable, while the blackboard portion of DANA is split between the database management system and a separate program containing the primary knowledge sources. So blackboard control in DANA must necessarily be distributed.

LESSONS LEARNED

The lessons that we learned about blackboards while working on DANA and MARRS can be applied to a wide range of design and implementation situations.

1. Knowledge sources are never as clean as you want.

Observation: Books and articles about blackboard systems invariably contain very neat pictures of independent knowledge sources reading exactly one layer of data and writing exactly one layer of data (just like Figures 1 and 2 in this paper). But it does not work this way in real life.

On several occasions, we started writing a knowledge source that would operate at a particular level and we realized that it needed one small piece of information from a completely different part of the hierarchy. In fact, this problem became so common in MARRS that at one point we proposed a special data manipulation language to help maintain the dependencies. The language would artificially promote copies of these stray bits of data to the appropriate level; the knowledge source would then appear to operate at a single level, but the physical data that it was actually reading and writing was still scattered through multiple layers.

The same problem exists between knowledge sources. Even though we were careful to try to write the knowledge sources as independent modular code, both DANA and MARRS contain places where the correct behavior of one knowledge source depends on another knowledge source doing some particular thing. For example, updating a hardware status record in DANA requires a primary knowledge source to fill in the fundamental changes and update the time-aggregate data; data triggers are used as secondary knowledge sources to fill in derived changes and statistical information. These operations need to be closely coordinated, since if the primary knowledge source overwrites some of the derived information before the data triggers get to it, the statistical information will not be calculated properly. Therefore, changing one of these knowledge sources will require making modifications to the others too.

Lesson learned: No matter how carefully a blackboard system is designed, there will always be problems with modularity of the knowledge sources. Either some knowledge sources will violate the modularity criteria, or the code will become unnecessarily complex simply to maintain modularity.

It is no accident that blackboard knowledge sources are defined to be black boxes. In theory, this works fine. But in real systems, implementors need to trade off the long-term desirability of modularity with the short-term inconvenience of enforcing it. Our experience has been that every time we violated modularity, it came back to haunt us later on in the implementation. Whenever modularity is violated, it must be carefully documented, and if possible an automated maintenance procedure should be implemented. For example, it may be possible to implement utility functions that automatically propagate changes from one knowledge source to other dependent knowledge sources. These functions can give the illusion of modularity, and can enforce consistency across software updates.

2. Opportunistic control is hard.

Observation: Of the three blackboard components, opportunistic control is by far the most difficult to implement. A global blackboard area is just a highly organized data structure, and knowledge sources are just an extension of traditional modular design. But opportunistic control is significantly different from the control modules that most systems use, and it requires a different way of thinking about the system.

Opportunistic control tends to make debugging more difficult. With a tight control loop, it is trivial to trace the execution path from one module to another. But when module selection is based on the relative weights of evaluation functions, it is much harder to predict which modules will execute at which times. It is also more difficult to force modules to synchronize at certain points; we had to make some careful adjustments of trigger priorities in DANA to ensure that the proper updates were done at the right time.

Additionally, it may be difficult to convince project managers that opportunistic control is worth the effort. The software complexity and performance cost of opportunistic control are readily apparent, but the benefits are not always as obvious. This was especially the case with MARRS: since the existing control loop worked fine, why change it to something that runs slower? In order to convince the developers that opportunistic control was worthwhile, we had to come up with several scenarios involving possible future extensions that the existing control could not handle. It was also important to show them that they could implement the new control module gradually and still have the option of running it exactly like the old one.

Lesson learned: Opportunistic control can be a very powerful tool; once it is available, it makes design and experimentation much easier and much more flexible. But this power comes with a price: the overall system becomes more difficult to understand and debug, there may be a noticeable runtime penalty in some circumstances, and it takes some time to get used to the new programming philosophy. Because of this, programmers and administrators that are not already familiar with blackboard technology may be resistant to using opportunistic control, especially if they already have an existing traditional control model.

The general advice about anticipating future design changes early in the development process is particularly important when one is considering using a blackboard architecture. As the number and complexity of potential system modifications increases, the flexibility and adaptability of an opportunistic control module becomes increasingly appropriate.

When converting existing programs to a blackboard architecture, as with MARRS, it is important to provide an implementation of the opportunistic control module that mimics the traditional control structure in order to maintain continuity. But even when the blackboard architecture is designed in from the beginning, it is still useful to have the ability to use fixed-

order control as one of the available configurations. Using fixed-order control can simplify debugging because it eliminates the nondeterministic program flow. It also makes it easier to test and integrate new knowledge sources. The method of incremental opportunistic control described above supports this.

3. Blackboards are a philosophy, not a mandate.

Observation: The most important lesson we learned is that blackboards should not be viewed as a set of architectural mandates that dictate how the system must be configured. Instead, they should be used as a general programming philosophy to guide the development of the system.

Opportunistic control should not be required just so that a system can be called a blackboard; it should be used only if opportunistic control contributes something useful to the system. Similarly, knowledge sources should not always be forced to operate at specific data levels or to be completely independent black boxes. These rules of blackboard design are useful and important, but they were created for idealized systems. It is acceptable to break those rules in the appropriate circumstances, but it is very important to understand the consequences of doing so. As an analogy, "goto" statements are generally considered harmful because they obscure the control flow of an algorithm and make the code difficult to follow. But writing 50 extra lines of code to avoid one simple "goto" statement adds more obscurity than it avoids, and confuses programming discipline with blind pedantry.

Lesson learned: The concepts embodied in a blackboard architecture form a useful programming methodology. System designers should study and understand these concepts and the dangers of violating them; then they should adopt the parts that are useful to a particular project and leave behind the ones that are not.

References

[Bar89] Brian M. Barry. Prototyping a real-time embedded system in smalltalk. In *OOPSLA-89*, pages 255–265, October 1989.

[BATW87] Brian M. Barry, John R. Altoft, D.A. Thomas, and Mike Wilson. Using objects to design and build radar ESM systems. In *OOPSLA-87*, pages 192–201, October 1987.

[EM88] Robert Engelmore and Tony Morgan, editors. *Blackboard Systems*. Addison-Wesley, 1988.

[For82] Charles L. Forgy. RETE: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

[HH93] Michael Hewett and Rattikorn Hewett. A language and architecture for efficient blackboard systems. In *Proceedings of the 9th Conference on Artificial Intelligence for Applications*, pages 34–40, Orlando, Florida, March 1993.

[Int90] Interbase Software Corporation. *Interbase Data Definition Guide*, 1990.

[Nav90] Naval Oceans Systems Center. Communication Support System (CSS) Overview. Technical Report CSS-100001-01, Naval Oceans Systems Center, San Diego, California, July 1990.

[Nav91a] Naval Oceans Systems Center. System specification for the CSS. Technical Report CSS-SSS-U-B1V1-R0C0(DRAFT), Naval Oceans Systems Center, San Diego, California, June 1991.

[Nav91b] Naval Oceans Systems Center. System specification for the CSS standard communications environment (SCE). Technical Report CSS-SSS-SCE-U-B1V1-R0C0(DRAFT), Naval Oceans Systems Center, San Diego, California, June 1991.

[Nii86a] H. Penny Nii. Blackboard application systems, blackboard systems from a knowledge engineering perspective. *AI Magazine*, pages 82–106, August 1986.

[Nii86b] H. Penny Nii. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, pages 38–53, Summer 1986.

[Nii89] H. Penny Nii. Blackboard systems. In *The Handbook of Artificial Intelligence, Volume IV*, chapter XVI, pages 1–82. Addison-Wesley, 1989.