TRANSACTIONAL BLACKBOARDS

J. Robert Ensor and John D. Gabbe
AT&T Bell Laboratories
Holmdel, NJ 07733

ABSTRACT

The blackboard architecture is a popular structuring framework for expert systems. With this structure, an expert system is built as a collection of knowledge sources which are scheduled by a controller and communicate through a shared data region, called a blackboard. The performance of such a system may be significantly enhanced by the concurrent execution of the knowledge sources. However, introduction of concurrent execution into blackboard systems requires extension of the architecture with new mechanisms for scheduling knowledge source activities, synchronizing knowledge source interactions, and accessing shared data. This paper describes our design for transaction-based facilities supporting parallel execution of knowledge sources in a blackboard system.

I. INTRODUCTION

The blackboard architecture is an important structural framework for expert systems. In this architecture, an expert system consists of a shared data region (called the blackboard), a set of knowledge sources, and a control mechanism. The blackboard is a data base which is shared by the knowledge sources as their communication medium. Containing rules and hypotheses which express the domain expertise of the system, the knowledge sources respond to each other through observed changes in the blackboard. The control mechanism schedules execution of the knowledge sources according to information from its goal queues and the blackboard.

Several expert systems have been built according to the blackboard architecture. Examples include a speech-understanding system (Erman et al, 1980), a sonar interpretation system (Nil and Feigenbaum, 1978), a vehicular tracking system (Lesser and Corkill, 1978), and a protein crystallography interpretation system (Terry, 1983). Although these systems are founded on the blackboard architecture, they vary significantly within the framework, demonstrating the utility and flexibility of the paradigm. Experience suggests that this architecture is particularly suitable for systems representing multiple areas of expertise and for systems solving problems with complex information interdependences.

Multiprocessor computing environments should be capable of increasing the scope and utility of expert systems and successfully addressing problems beyond the reach of most uniprocessors, such as real time speech recognition or robot control. The domain and control knowledge of an expert system may be distributed onto several processors. The interactions of modular knowledge sources may simulate their modeled events, with both communication paths and timing of interactions. Thus multiprocessor configurations have the potential to support the construction and execution of expert systems with new and useful properties.

Multiprocessor computers are often difficult to use. While the processors can execute in parallel, the exchange of data, code, and results among these processors can often make the overall system slow. Therefore, a balance must be reached among the costs of loading code, accessing data, and communicating requests and responses. Two extreme approaches have received most attention by researchers. At one extreme are systems in which processing nodes frequently exchange small sets of data and do small computations with each data set (e.g., Dennis, 1980). At the other extreme are systems that place a large, autonomous program on each processing node. In these systems, the nodes exchange data infrequently and spend most of their time performing "local" computations (e.g., Lesser, 1978). The work described in this paper focuses on supporting systems closer to the latter extreme. We present mechanisms for constructing expert systems as collections of knowledge sources communicating through a shared data medium. These are systems in which knowledge sources executing on different processors perform moderate to large computations between communications.

The integrity of data that is accessed asynchronously by several clients must be maintained. Providing transactional access to shared data bases is a common solution to this problem. A sequence of operations on one or more data elements, beginning with a start-tran3action request and ending with either a

commit- or an abort-transaction request, a *transaction* is a unit of activity with three properties: atomicity, consistency preservation, and permanence. Atomicity means that, in net effect and even when failures occur, either all operations in the unit happen (the transaction commits) or none of them happens (it aborts). Consistency preservation means that a transaction moves data from one consistent state to another. Permanence means that the effect of a committed transaction persists, surviving any noncat as trophic failures, until the next transaction involving that data is committed.

We extend the blackboard architecture to support systems executing in multiprocessor environments by providing transactional access to the blackboard. Our extensions are novel in their ease of use and in the richness of structure that they support. Two mechanisms are provided for safe access to the blackboard data. Knowledge sources can communicate by accessing shared data in separate transactions. Furthermore, several knowledge sources can participate in a common transaction if they need to see a common, consistent view of shared data.

II. SYSTEM STRUCTURE

Figure 1 illustrates a system that we designed to understand the use of the blackboard. We term the control and knowledge sources *agents* because they are both modular units of activity. The agents are distributed on various processors and may execute concurrently. Knowledge source activities on each node are controlled by the control sources on that node. (The collection of control sources is the controller mentioned in the blackboard architecture description.) In our present implementation, the distribution of agents is subject to restrictions. The initial distribution is specified by the system designer, and we provide no mechanism to support agent migration among processors. Although the blackboard resides on a single machine, it could be distributed without changing its interface.

A. The Blackboard

The blackboard is a repository of data; each datum holds an arbitrary Lisp s-expression. Because agents may share data and reference them in an interleaved fashion, some mechanism is needed to maintain consistency of the blackboard. We associate a transaction manager with this data base, and require that any reference to the blackboard be part of a transaction.

The blackboard transaction manager controls asynchronous references to shared data via locks. There are two types of locks: write and read. The holder of a write-lock has exclusive access to the locked datum and may modify the datum. Holders of read-locks may read the datum concurrently. No writer may access a datum while a read-lock for that datum is held. When a client first references a datum, the transaction manager attempts to obtain the appropriate lock. All locks are held to the end of the transaction in which they were obtained. Thus the transaction manager preserves data consistency by preserving serializability (Eswaran, 1976).

When trying to obtain a lock, the transaction manager might find that it is not available. The transaction that needs the unavailable lock is suspended until the lock can be obtained. Sometimes more than one transaction may be waiting to obtain a lock, and this introduces the potential for deadlock among the waiting transactions. For example, transaction A might wait for a lock held by transaction B, while transaction B waits for some other lock held by transaction A. The transaction manager detects deadlocks and resolves them by aborting a suspended transaction. This abortion is simply reported to the agents participating in the transaction; these agents must then decide what action is appropriate.

Data consistency among agents interacting within a transaction is maintained by time stamps. If serializability among agents within a transaction is violated,

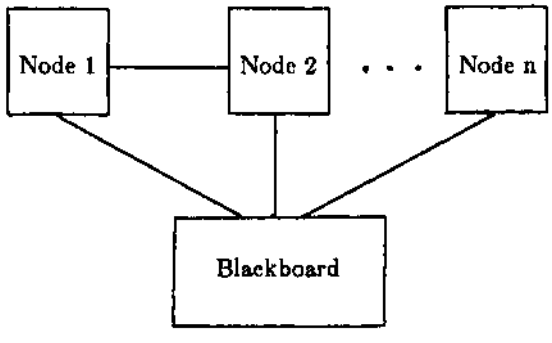

Figure Ia
Network of Processing Nodes

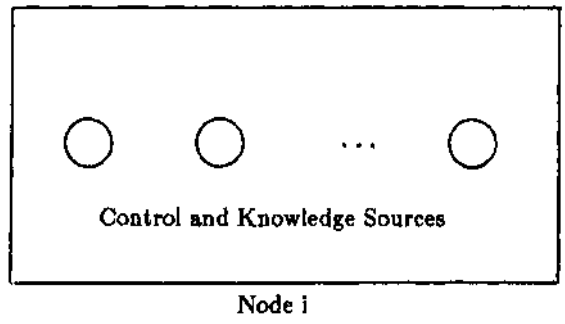

Figure Ib
Agents Within a Node

the blackboard transaction manager aborts the transaction. This abortion is reported to the agents participating in the transaction, as with deadlock detection.

Computation based on the transactional blackboard is not data driven; that is, accessing values in the blackboard does not automatically trigger agent activity. This seems appropriate in a distributed environment because the blackboard might not be able to schedule activities on remote sites. This is in contrast to the centralized case supported in previous proposals (e.g., the Hearsay III approach of Balzer et al, 1980).

B. The Agents

Each knowledge source contains some of the system's domain specific knowledge. This knowledge is expressed in terms of the data visible to the agent - that portion of the blackboard accessible to the agent plus those data sent as message parameters by other agents. As a knowledge source executes, it examines the visible system state. If the system state matches a condition known to the knowledge source, the agent takes specified actions. These actions include requesting that the controller schedule a knowledge source activity by placing an entry on the controller goal queue, performing some operations on the blackboard, and/or sending a message to another knowledge source.

C. Inter-agent Communication

The multiprocessor environment fosters a richness of system structure. Each processor can support a community of agents - complexes of control sources and groups of knowledge sources working closely together - and these communities can interact with communities on other processors. Agents executing on the same machine can communicate with efficiency and facility, for they may directly access common data and may include arbitrary references as parameters in the messages that they send to each other. Since the cost of message transmission between machines is higher than a few memory references on a single one, agents executing on separate machines cannot communicate so cheaply. Further, these agents may include only values in their message parameters, and the conversion of local data to transmittable data values may be expensive. Each node in our system then contains procedures to convert the value of arbitrary s-expressions to transmittable data values. In addition, each communicating agent needs access to procedures to reference these transmitted data values once they have been received.

Agents may also communicate through the blackboard, and two mechanisms are provided for this interaction. Agents can interact by accessing shared data in separate transactions, or several agents can participate in a common transaction. This latter mechanism is often useful; for example, the controller might start a transaction to check the precondition of a goal-queue entry. The knowledge source that the controller then activates might need to access the data mentioned in the goal-queue entry. Because the knowledge source should see these data in the same state as the controller, it continues the same transaction. To include a second agent in a transaction, the first agent merely passes its transaction identifier and status to the second. The transaction status indicates whether the transaction is to be committed, aborted, or continued.

D. Scheduling and Transaction Protocols

The controller maintains one or more goal queues, each comprised of entries generated by knowledge sources. A goal-queue entry has three parts: an expression (called the precondition), an action to be taken if that precondition is true, and a status indicator which may contain a transaction identifier if the action is to continue an on-going transaction.

Scheduling activity by selecting entries from its goal queue, the controller proceeds down the goal queue evaluating entry preconditions. If the precondition is false, its action can not be selected, and the goal is deferred. If the precondition holds, the corresponding action is executed. In evaluating a precondition, the controller might need to access data in the blackboard. If there is no transaction identifier associated with the queue entry, the controller begins a new transaction. If an identifier is already associated with the queue entry, the controller continues this transaction. At the end of the decision process, the controller aborts the newly-started transactions associated with deferred goals. If an agent is activated as a result of the queue entry selection, that agent is given any associated transaction identifier.

In addition to scheduling, the control agents are responsible for the initiation and termination of transactions. These activities are based on the contents of the goal queues. As mentioned above, the controller begins a transaction or continues an existing one when it schedules a knowledge source for activity. When an agent finishes executing, it notifies the controller. The controller now checks the goal queue for entries with the transaction identifier of the knowledge source just completed. If the queue has no entries with this transaction identifier, the controller terminates the transaction according to the transaction status. If other entries contain this identifier, the associated actions will presumably continue this transaction.

III. IMPLEMENTATION

Our initial implementation of an expert system using the transactional blackboard is designed to execute on a network of three Symbolics Lisp Machines connected via an Ethernet. Zetalisp flavors (Weinreb, 1981), the blackboard and knowledge and control sources communicate with each other via messages, the parameterized invocations of flavor methods. The

message parameter restrictions discussed above are enforced at run-time by the execution environment. The blackboard transaction manager, which reads or writes objects on behalf of agent requests, and its associated data base reside on a single machine.

A. Data

Lisp s-expressions are the unit of storage and retrieval in the blackboard, and a blackboard datum is indexed by a Lisp name. Because the blackboard has no knowledge of the internal structure of the data it stores, the storage and retrieval support functions available to an agent are responsible for constructing transmittable data for storage and reconstructing the representations on retrieval. Generally, these functions need access to the definitions of the data in order to reconstruct their representations.

B. Transactions

Access to the blackboard data is allowed only within transactions. The transaction manager associated with the blackboard receives requests from agents, executes on their behalf, and packages responses. With each request to the blackboard, an agent presents a unique agent identifier and a transaction identifier. The transaction identifier is returned to an agent when it begins a transaction as a return value of the *start-transaction* command.

The blackboard transaction manager supports *five* transaction states and state-changing messages. Figure 2 illustrates these states and the messages that cause state transitions. Starting in the ground state, a transaction moves with the *start-transaction* message into the active state where read and write messages are handled. *Commit-* and *abort-transaction* messages terminate a transaction by moving it to the committed and ground states respectively. The straddle and precommitted states provide for the implementation of two- and three-phase commit protocols (Skeen, 1981), which are a means of coordinating transactions involving more than one transactional server.

1. Atomicity. Atomicity ensures that at the end of a transaction all the write actions associated with the transaction have taken place (the transaction committed), or all the data referenced by the transaction are restored to the state that existed when the transaction began (the transaction aborted). In each transaction, an existing datum is copied before it is first written. (If there is no existing datum, the "copy" so indicates.) This copy then serves to save the state of the datum that existed before the transaction began. If the transaction commits, the copy is discarded; if the transaction aborts, the copy replaces the current version of the datum.

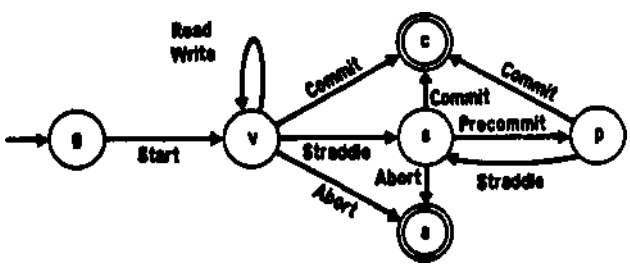

Figure 2
Transaction States

2. Consistency. Our transactional blackboard has two broad consistency tasks: first, to maintain data consistency among several transactions (inter-transactional consistency), and second, to preserve a consistent view of data for those agents within an individual transaction (intra-transactional consistency).

Inter-transactional consistency of blackboard data is maintained with locks. The write locks are exclusive and guarantee that no activity can interfere with the writer while it is modifying data. Read locks are shared, and many agents may concurrently read a datum. Since the datum is not modified during this time, consistency is maintained. All locks are held to the end of the transaction, and requests made to locked data are queued until release of locks.

The transaction manager checks for deadlock whenever it queues a read or write request. In the present implementation, the blackboard retries queued requests referencing a particular datum in the order in which they are received. If a deadlock exists, the manager calls a deadlock handler to abort one of the transactions. Although the transaction manager contains a default handler, a preferred deadlock handler may be specified by an agent when it initiates a transaction to allow deadlock resolution to be based on domain knowledge.

Intra-transactional consistency is maintained through the use of time-stamps. The time stamps are used to enforce serialization through a basic time-ordered scheduling algorithm (Bernstein, 1981). When an agent first participates in a transaction, it is assigned a time-stamp, called the agent time for that agent. The write time-stamp for each blackboard datum is the agent time of the write request being executed on that datum. The read time-stamp for each datum is the later of the datum's current read time-stamp and the agent time of the read request being executed on that datum. A datum maintains a separate read time-stamp for each transaction holding a read-lock. A read request for a datum is rejected if the agent time is earlier that the datum's write time-stamp. A write request is rejected if the agent time is earlier than either the read or write time stamp of the datum. The establishment of a serializable intra-transaction schedule and concomitant coordination of the participating agents is the responsibility of the agent controlling the transaction. The blackboard regards an intra-transaction time-order violation as a

fatal error and aborts the offending transaction. The blackboard's transaction manager checks for possible time-order violations before queuing a store or retrieve request, so the errors are detected immediately.

3. Persistence. Persistence ensures that the results of committed (and straddled and precommitted) transactions will survive system crashes. To implement persistence, copies of data are kept on devices with independent failure modes and recovery protocols are supported. We use a straightforward logging and checkpoint scheme to preserve copies on independent devices. The implementation might be expensive in both space and time. It is not practical to encumber all agents with this overhead, and thus logging can be deactivated for any transaction. If a transaction is not logged, crash recovery returns its data to some previous (archived) consistent state, instead of the most recent consistent state. The recovery protocols are not expensive to implement because they are driven by external agents, not the blackboard itself.

IV. CONCLUSIONS

Our transactional blackboard architecture supports the construction of expert systems for multiprocessor environments. The transactional interface allows asynchronous requests to be safely issued to the shared data of the blackboard. Clients of this transactional service must specify the boundaries of each transaction, and they must deal with aborted transactions. We feel that this marginal cost over a serial system is small and should not interfere with the business of building expert systems.

More importantly, we provide mechanisms to make use of this shared data in an intelligent way. Control decisions are based on domain knowledge and communication costs. The controller presumably tries to utilize the processors of the computing facility to effect good system performance or to model some system of interest. The controller remains knowledge-based and does not use scheduling to protect shared data. Knowledge sources are not required to provide explicit synchronization or protect the consistency of shared data. If several agents wish, they may participate in common transactions. To do so, they only need to pass transaction identifiers among themselves.

We are using this architecture to build some expert systems. Our experience indicates that this architecture is very helpful for large, multi-author projects, where each designer works rather independently to implement a small area of expertise. In addition to reaffirming the advantages of modularity in program structure, we would like to report on the performance advantages realized by executing our expert systems on multiprocessor computers. Unfortunately, we have not yet performed the necessary experiments.

REFERENCES

Balzer, R., Erman, L. D., London, P., and Williams, C, "Hearsay III: A Domain-Independent Framework for Expert Systems," in *Proc. 1st National Conference on Artificial Intelligence,* pp. 108-110, Stanford, CA., August 1980.

Bernstein, P. A., and Goodman, N., "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys,* 13:2, pp 185-221, (June 1981).

Dennis, J. B., "Data-flow Supercomputers," *Computer,* 13:11, (November 1980), pp. 48-56.

Erman, L. D., F. Hayes-Roth, V. Lesser, and D. Reddy, "The HEARSAY-II speech-understanding system: intergrating knowledge to solve uncertainty", *ACM Computing Surveys,* 12 (2), pp. 213-253, (June 1980).

Eswaran, K. P., et alia "The Notions of Consistency and Predicate Locks in a Database System", *Comm. ACM,* 19:11, pp. 624-633, (November 1976).

Lesser, V. R. and Corkill, D. D., "Cooperative distributed problem solving a new approach for structuring distributed systems", TR 78-7, Department of Computer and Information Science, Univ. of Massachusetts, Amherst, MA, 1978.

Nii, H. P., and Feigenbaum, E. A., "Rule-based Understanding of Signals," in *Pattern-Directed Inference Systems,* D. A. Waterman and R. Hayes-Roth (eds.), Academic Press, 1978.

Skeen, D., "Nonblocking Commit Protocols", Proceedings of ACM-SIGMOD International Conference on Management of Data, pp. 133-142 (April 1981).

Terry, A., "The CRYSALIS Project: Hierarchical Control of Production Systems," Stanford Heuristic Programming Project Memo HPP-83-19, Computer Science Department, Stanford University, Stanford, CA, 1983.

Weinreb, D., and Moon, D., *Lisp Machine Manual,* Symbolics Inc. Cambridge, MA, 1981.