Automating Workflows for Service Order Processing

Integrating AI and Database Technologies

Munindar P. Singh and Michael N. Huhns, Microelectronics and Computer Technology Corporation

THIS ARCHITECTURE MARRIES AI APPROACHES WITH STANDARD DATABASE TECHNIQUES TO MANAGE WORKFLOWS IN A DISTRIBUTED COMPUTING ENVIRONMENT WHOSE ACTIVITIES INVOLVE SEVERAL DATABASE SYSTEMS, USER INTERFACES, AND APPLICATION PROGRAMS.

MCC'S CARNOT PROJECT IS developing a suite of technologies for integrating heterogeneous information resources. One of the project's goals is an environment for developing complex multisystem applications that access information stored in existing heterogeneous systems and that maintain consistency constraints across those systems.

Integrating preexisting systems is generally harder than designing distributed systems afresh. Many systems, especially those based on older mainframe architectures, allow data to be accessed only through arcane interfaces of limited functionality. The systems and their interfaces cannot be easily modified, due to both the complexity of the programming effort required to achieve any modifications, and the constraint that older applications must continue to run as before, since they typically have a wide user base that relies heavily upon them. Thus, the integration must permit newly developed applications to coexist with previous applications.

A *workflow management* facility is an important component of the Carnot Project. Briefly, workflows are the structured activities or tasks that take place in typical business information systems. These activities frequently involve several database systems, user interfaces, and application programs. Unfortunately, traditional database systems do not support workflows well: People usually must intervene to ensure proper execution. If we could automate workflow processing, we could improve turnaround time, check initial input for errors, validate fields with respect to other fields and information in customer databases, streamline existing procedures (by removing redundant data gathering and processing), and gain the ability to modify the structure of the overall procedure easily.

We have developed an AI-based architecture that automatically manages workflows, and we have implemented a prototype that executes on top of a distributed computing environment to help a telecommunications company better provide a service that requires coordination among many operation support systems and network elements.

Database transactions

Classical transaction processing in databases deals with executing access and update tasks on a single database. Such tasks are traditionally assumed to have the so-called ACID properties:[1]

- *Atomicity*: All changes to the system state caused by a task happen, or none do.
- *Consistency*: A task takes the database from a consistent state to a consistent state.
- *Isolation*: The intermediate results of a task are not visible to another task.
- *Durability*: The changes committed by a task are persistent.

These properties simplify transaction management considerably, but they are too restrictive in loosely coupled heterogeneous

0885-9000/94/$4.00 © 1994 IEEE

An earlier version of this article appeared as "Automating Workflows for Service Provisioning: Integrating AI and Database Technologies" on pp. 405–411 of the *Proceedings of the 10th Conference on Artificial Intelligence for Applications* (CAIA '94), published in 1994 by the IEEE Computer Society Press, Los Alamitos, Calif.

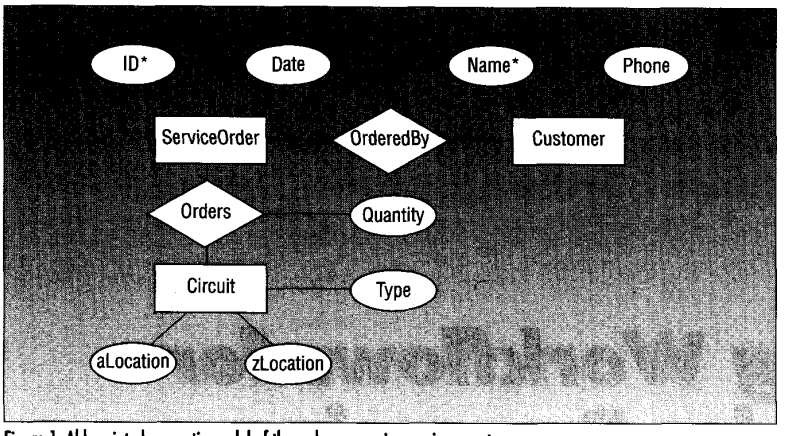

Figure 1. Abbreviated semantic model of the order processing environment.

environments. For example, ACID tasks can be coordinated through mutual commit protocols (which ensure that all of a given set of tasks commit, or none do), but such protocols are notoriously inefficient when executed over networks. Further, executing such a protocol requires access to the internal states of transactions, such as their precommit states (a transaction is in its precommit state when it is internally ready to commit, but is awaiting permission from the transaction manager to do so). Most commercial database systems do not provide access to such internal states, thereby making direct implementations of commit protocols extremely difficult.

The ACID properties are naturally realized when the correctness of database transactions is characterized through some purely syntactic or structural criterion, such as serializability.[2] However, serializability cannot be efficiently implemented in distributed systems whose component systems are autonomous. Instead of trying to specify correctness criteria purely syntactically, we characterize them semantically,[3] which allows us to specialize the correctness criteria to the application at the cost of building a deeper model of the application domain. This helps simplify several coordination requirements. For example, instead of executing mutual commit protocols, we can optimistically commit different tasks. If this action should prove erroneous, we undo the effects of incorrectly committed tasks by means of *compensating* transactions, whose definition depends on the semantics of the underlying domain.

Consequently, in heterogeneous environments, the unit of relevant activity is not a single database transaction, but rather a *workflow* that executes over a set of database and information resources. The constituent tasks of a workflow may be individually ACID, but the overall workflow usually is not. The problem is to ensure that no semantic constraint of the information model is violated despite this.

The activities that comprise a workflow of interest are typically already being carried out in the given organization. However, they are usually carried out by hand, with people intervening in several crucial stages to ensure that the necessary tasks are done and that organizationwide consistency constraints are enforced. The semantics is supplied by the people or is implicitly encoded in the business procedures. The canonical example of a workflow is a document flow through an organization. For instance, when an order is received, it must be entered into the system, and several decisions must be made to process it properly. These decisions would typically require information from several resources within an enterprise and possibly some outside of it. For example, to process a request to transfer money from one account to another, the authorization must be verified, the account numbers validated, and the source account tested to see if it has the required balance. External sources would be accessed for other requests, such as loan applications, where a credit bureau's databases may be consulted to determine an applicant's credit worthiness.

It is important to be able to handle the myriad error conditions that may arise in different workflows. The exception conditions are the hardest to automate. It is in identifying and resolving such conditions, and managing control and dataflow appropriately, that AI technology can contribute substantially.

Workflows for service order processing

The Carnot Project includes application partnerships with its sponsoring organizations — an arrangement that lets us test our research ideas in prototype systems that address real problems. One of these partners provides telecommunication services, and one of the company's services is to provide digital communication services between two specified points.

In the extant workflow, the company receives a set of paper forms that gives details about the service being ordered. It enters these forms into the system, and tests to determine if certain essential telecommunication equipment is already in place. If it is, the service can be provided quickly; otherwise, the processing must be delayed until the equipment is installed.

Providing the digital communication service using this workflow takes more than two weeks and involves 48 separate operations — 23 of which are manual — against 16 different database systems. In addition, configuring the operation-support systems so that they can perform such a task often takes several months. This is significant in our partner's business environment: Many of its competitors were formed in the last decade or so, and they typically have more modern computational facilities than our client's legacy systems.

To aid our partner, we sought to reduce this time to less than two hours and to provide a way in which new services could be introduced more easily. Our goals were to develop a prototype workflow management system that could apply to workflows in general, and that would let the company operate as efficiently as its competition without discarding its legacy systems. Our strategy for accomplishing these goals was to interconnect and interoperate among the previously independent systems, replace serial operations with parallel ones by using relaxed transaction processing,[4] and automate previously manual operations, thereby reducing errors and delays.

The entity-relationship diagram in Figure 1 shows the most relevant components of the problem's semantic model, while Figure 2 presents the basic structure of the workflow (the admissible executions when everything works correctly). Each node is a task, and the partial order reflects the dependencies among tasks. Tasks cannot be initiated until all their dependencies are met; ordinarily, they must be initiated if those dependencies are satisfied.

The Carnot solution

We defined a distributed agent architecture, shown in Figure 3, for intelligent workflow management that functions on top of Carnot's distributed execution environment. Our design was required to use existing procedures as much as possible so that it would not adversely affect other applications. As it turned out, our architecture accommodated this easily; indeed, we welcomed not having to worry about the details of the mainframe systems on which we ran various tasks. Since we assumed that the actual applications executed by the workflow were already defined, our goal was to manage the overall structure of the applications in as domain-independent a manner as possible.

Our system consists of four agents that interact to produce the desired behavior. The databases in Figure 3 are assumed to include the relevant data and the application programs that execute on them. The necessary applications are executed by the schedule processing agent; the user interface agent queries the systems to help users fill in order forms completely and correctly, and to provide feedback about progress. This enables the detection of data inconsistencies early in the process.

This architecture is made possible by our previous integration into Carnot of an expert system shell with forward- and backward-chaining capabilities, a type system, and truth maintenance. This environment provides the basic message-passing facility that our agents use to interact with other agents anywhere on the network. We used this facility to implement a scheme by which agents can exchange assertions, thereby triggering or disabling rules in each other. We augmented our scheme so that agents that are not expert systems can also participate, provided they satisfy a simple protocol. This allowed us to integrate transparently a graphical interaction agent, which is not an expert system shell.

Figure 4 describes the implementation of the transaction-scheduling agent at a high level as an entity-relationship diagram. The tasks that correspond to the nodes of Figure 2 are modeled as database transactions, each of which is initiated by an agent. Each task has an associated message type that essentially encodes the computation that the underlying IMS databases must execute. When an agent executes a task, it does so by passing along the relevant message, that is, the name of the file that contains it.

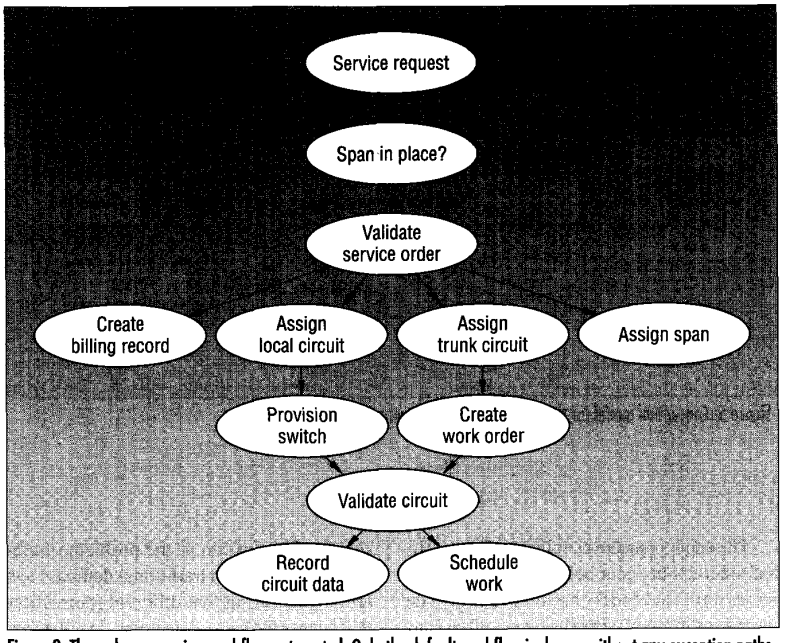

Figure 2. The order processing workflow automated. Only the default workflow is shown, without any exception paths.

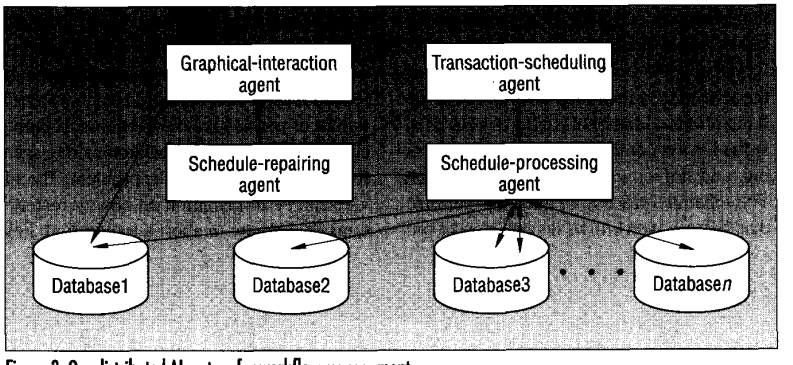

Figure 3. Our distributed AI system for workflow management.

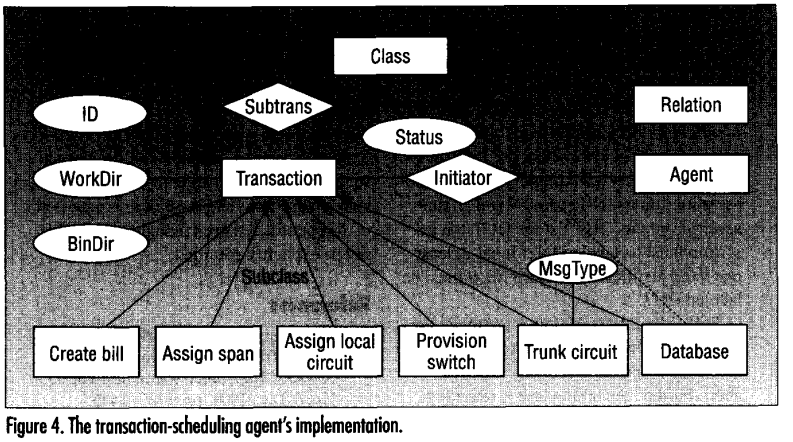

Figure 4. The transaction-scheduling agent's implementation.

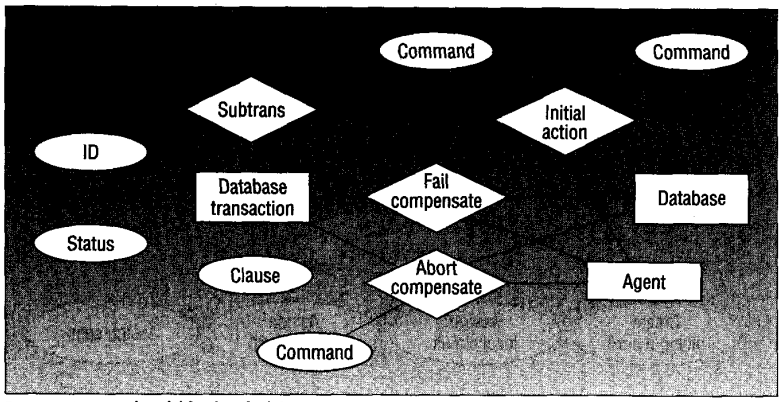

Figure 5. Conceptual model for the schedule-repairing agent.

The agents operate as follows. The graphical-interaction agent helps the user fill in an order form, and checks inventories to give the user an estimate of when the order will be completed. It also informs the user about the order's progress. The transaction-scheduling agent constructs the initial schedule for the request. The relevant subtasks are scheduled with the maximum concurrency possible, while still satisfying all precedence constraints.

The schedule-processing agent executes the schedule by invoking tasks as necessary. It maintains connections to the databases involved in telecommunication order processing, and implements transactions on them. The schedule-processing agent also ensures that different workflows do not interact spuriously. This is akin to the problem of concurrency control in traditional database systems — ensuring that different transactions that access the same data items do not access them in relative orders for which there are no equivalent serial executions. With a workflow, we need to ensure that subtasks on each database can be serialized in semantically consistent orders. This may require delaying some subtasks, or aborting and retrying them.

If the schedule-processing agent encounters an unexpected condition, such as a task failure, it notifies the transaction-scheduling agent, which asks the schedule-repairing agent for advice on how to fix the problem. The advice can be information on how to restart a transaction, how to abort a transaction, how to compensate for a previously committed transaction, or how to clean up a failed transaction. These actions are meant to restore semantic consistency across the system. For example, if the system cannot allocate a span to a given service request, it aborts the entire request; the billing task, if already committed, is compensated. On the other hand, if the billing task fails but the span allocation succeeds, the service order is allowed to proceed and the billing task is retried later. This example highlights the distinction between *vital* and *nonvital* subtasks. The failure of a vital subtask propagates to the global task; nonvital tasks can simply be retried. A conceptual model for the knowledge of the schedule-repairing agent is shown in Figure 5.

The initial schedule is constructed on the assumption that tasks will succeed as expected. This leads to a small, easily executable, schedule. If error conditions arise, they are accommodated at runtime by repairing the initial schedule. Some of this is automatic, since the undesirable and unexecuted parts of the schedule are disabled by the transaction-scheduling agent's truth maintenance system when their preconditions fail to hold (see the sidebar on truth maintenance).

The basic structure of this system is domain-independent, although the details of the messages are clearly domain-dependent. Certain parameters, such as the identifier of the service request, are known to the scheduler, but most of the data is passed through the file system. The files are uniquely named using the known identifier, thereby allowing different requests to execute concurrently. The other two domain-dependent components of the system are the resource constraints, which guide the scheduling and repairing processes, and translation routines invoked by the schedule-processing agent to convert data formats from those produced by one application to those expected by the next; these routines were written using the tools Lex and Yacc.

WE ARE REIMPLEMENTING OUR prototype implementation for installation in a restricted production environment (one switching center). If it is successful, our client will deploy it in all switching centers.

Certain desired features will call for AI technology in the final implementation, including schedule repair and other semantic aspects of the domain. Because of business constraints, we do not expect to use our present Lisp-based system for these, although

Truth maintenance systems

A truth maintenance system (TMS) provides a simple, built-in, generic way to manage dependencies, such as in a workflow schedule.[1] Justification-based TMSs assign a belief status of IN or OUT to each represented assertion. IN means *believed* and OUT means *not believed*. A *justification* for an assertion is a pair of lists of assertions: the IN-list and the OUT-list. A justification is *valid* if and only if all the assertions on its IN-list are IN and all the assertions on its OUT-list are OUT. An assertion must be labeled IN if it has at least one valid justification; otherwise, it must be labeled OUT.

TMSs simplify workflow scheduling. For example, the billing subtask proceeds on the assumption that the global task will not abort. Further, the billing task is retried on failure, but only if the global task does not abort in the meantime. The failure of the local circuit assignment subtask causes the global task to abort, thus removing the justification for proceeding with the billing and, if it already happened, adding the justification for proceeding with its compensation. Consequently, complicated but correct executions — such as when the billing subtask succeeds on the fifth attempt, the local circuit subtask fails, and the billing is canceled — can be realized even though they would not be explicitly specified.

References

1. M.N. Huhns and D.M. Bridgeland, "Multiagent Truth Maintenance," *IEEE Trans. Systems, Man, and Cybernetics*, Vol. 21, No. 6, Dec. 1991, pp. 1437–1445.

the ideas will be reimplemented in a C++ or Rosette-based constraint processor. Certain other features, notably those to do with schedule processing, do not really require AI approaches, even though AI approaches are useful for rapidly prototyping them.

An alternative approach for scheduling tasks is to use operations research techniques, such as MRP II. With this approach, however, it is difficult to handle contingencies, such as a task failure. An operations research approach would require new constraints to be added that reflect the failure, and then the MRP II system would have to be rerun to generate a new schedule. The new schedule might be quite different from the original one, which might cause additional problems, especially if the original schedule were already being executed.

It is safe to conclude that AI technology helped us sort out various issues and easily build a working system that could be tested. Having an implementation helps us understand the needed components and the interfaces among them, which aids in the design and testing of industrial-strength modules.

References

1. J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco, Calif., 1993.
2. P.A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading, Mass., 1987.
3. H. Garcia-Molina and K. Salem, "Sagas," *Proc. ACM SIG MOD Conf. Management of Data*, ACM Press, New York, 1987.
4. P.C. Attie et al., "Specifying and Enforcing Intertask Dependencies," *Proc. 19th Int'l Conf. Very Large Databases*, 1993.

Munindar P. Singh is a member of the Technical Staff in the R&D Division at Microelectronics and Computer Technology Corporation, where he has been conducting research on distributed AI, heterogeneous database systems, and relaxed transaction processing. His monograph on characterizing computational agents, *Multiagent Systems: A Theoretical Framework for Intentions, Know-How, and Communications*, was recently published by Springer Verlag. Singh received his PhD and MS in computer science in 1992 and 1988, respectively, from the University of Texas, Austin, and his BTech in computer science and engineering in 1986 from the Indian Institute of Technology, Delhi.

Michael N. Huhns is a senior member of the R&D Division at the Microelectronics and Computer Technology Corporation, where he has been conducting research on the Argo, Antares, Reasoning Architectures, and Carnot projects. He was previously an associate professor of electrical and computer engineering at the University of South Carolina, where he also directed the Center for Machine Intelligence. His research interests are in distributed AI, machine learning, enterprise modeling and integration, and computer vision.

Huhns is the author of more than 100 papers in machine intelligence and information systems, and is an editor of the books *Distributed Artificial Intelligence*, Volumes I and II. He received his PhD and MS in electrical engineering in 1975 and 1971, respectively, from the University of Southern California, and his BS in electrical engineering in 1969 from the University of Michigan, Ann Arbor. He is a member of Sigma Xi, Tau Beta Pi, Eta Kappa Nu, ACM, IEEE, and AAAI.

Readers can reach the authors at Microelectronics and Computer Technology Corp., R&D Division, 3500 West Balcones Center Drive, Austin, TX, 78759-5398; Internet: msingh or huhns@mcc.com